BIG CPU, BIG DATA

Solving the World's Toughest Computational Problems with Parallel Computing

Alan Kaminsky

BIG CPU, BIG DATA

Solving the World's Toughest Computational Problems with Parallel Computing

Alan Kaminsky

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology

ISBN 000-0-0000-0000-0

Front cover image: The IBM Blue Gene/P supercomputer installation at the Argonne Leadership Angela Yang Computing Facility located in the Argonne National Laboratory, in Lemont, Illinois, USA. Courtesy of Argonne National Laboratory. http://commons.wikimedia.org/wiki/File:IBM_Blue_Gene_P_supercomputer.jpg

Alan Kaminsky
Professor
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
ark@cs.rit.edu
http://www.cs.rit.edu/~ark/

August 2015 edition

Preface

With the book *BIG CPU, BIG DATA*, my goal is to teach you how to write parallel programs that take full advantage of the vast processing power of modern multicore computers, compute clusters, and graphics processing unit (GPU) accelerators. The book is free, Creative Commons licensed, and is available from my web site (http://www.cs.rit.edu/~ark/bcbd/).

I'm not going to teach you parallel programming using popular parallel libraries like MPI, OpenMP, and OpenCL. (If you're interested in learning those, plenty of other books are available.) Why? Two reasons:

- I prefer to program in Java. The aforementioned libraries do not, and in my belief never will, support Java.
- In my experience, teaching and learning parallel programming with the aforementioned libraries is more difficult than with Java.

Instead, I'm going to use my *Parallel Java 2 Library* (PJ2) in this book. PJ2 is free, GNU GPL licensed software available from my web site (http://www.cs.rit.edu/~ark/pj2.shtml). You can download the complete source files, compiled class files, and Javadoc documentation. PJ2 requires Java Development Kit (JDK) 1.7 or higher. Installation instructions are included in the Javadoc.

PJ2 is suitable both for teaching and learning parallel programming and for real-world parallel program development. I use PJ2 and its predecessor, the Parallel Java Library (PJ), in my cryptography research. Others have used PJ to do page rank calculations, ocean ecosystem modeling, salmon population modeling and analysis, medication scheduling for patients in long term care facilities, three-dimensional complex-valued fast Fourier transforms for electronic structure analysis and X-ray crystallography, and Monte Carlo simulation of electricity and gas markets. PJ was also incorporated into the IQM open source Java image processing application.

I am happy to answer general questions about PJ2, receive bug reports, and entertain requests for additional features. Please contact me by email at ark@cs.rit.edu. I regret that I am unable to provide technical support, specific installation instructions for your system, or advice about configuring your parallel computer hardware.

More fundamental than the language or library, however, are parallel pro-

gramming concepts and patterns, such as work sharing parallel loops, parallel reduction, and communication and coordination. Whether you use OpenMP's compiler directives, MPI's message passing subroutines, or PJ2's Java classes, the concepts and patterns are the same. Only the syntax differs. Once you've learned parallel programming in Java with PJ2, you'll be able to apply the same concepts and patterns in C, Fortran, or other languages with OpenMP, MPI, or other libraries.

To study parallel programming with this book, you'll need the following prerequisite knowledge: Java programming; C programming (for GPU programs); computer organization concepts (CPU, memory, cache, and so on); operating system concepts (threads, thread synchronization).

My pedagogical style is to teach by example. Accordingly, this book consists of a series of complete parallel program examples that illustrate various aspects of parallel programming. The example programs' source code is listed on the right-hand pages, and explanatory narrative is on the left-hand pages. The example source code is also included in the PJ2 download. To write programs well, you must first learn to read programs; so please avoid the temptation to gloss over the source code listings, and carefully study both the source code and the explanations.

Also study the PJ2 Javadoc documentation for the various classes used in the example programs. The Javadoc includes comprehensive descriptions of each class and method. Space does not permit describing all the classes in detail in this book; read the Javadoc for further information.

The book consists of these parts:

- Part I covers introductory concepts.
- Part II covers parallel programming for multicore computers.
- Part III covers parallel programming for compute clusters.
- Part IV covers parallel programming on GPUs.
- Part V covers big data parallel programming using map-reduce.

Instructors: There are no PowerPoint slides to go with this book. Slide shows have their place, but the classroom is not it. Nothing is guaranteed to put students to sleep faster than a PowerPoint lecture. An archive containing all the book's illustrations in PNG format is available from the book's web site; please feel free to use these to develop your own instructional materials.

Alan Kaminsky
August 2015

Table of Contents

PART I
PRELIMINARIES

Titan, the U.S.A.’s fastest supercomputer
Photograph courtesy Oak Ridge National Laboratory

Chapter 1
The Parallel Landscape

Parallel computing is concerned with designing computer programs having two characteristics: they run on *multiple processors*, or *cores*, and all the cores cooperate with each other to solve a *single problem.*

Both characteristics are necessary for a program to be considered a parallel program. A web server, like the one that runs Google's web site, typically runs on a multicore server machine, or even a group of several multicore server machines, so as to process hundreds or thousands of web page requests each second simultaneously with high throughput. The web server thus displays the first characteristic of parallel computing. However, the web server's cores are working on *different* problems, namely the different user's web page requests. The web server's cores are not cooperating with each other to solve the *same* problem. So the web server does not display the second characteristic of parallel computing. I would call a web server an example of *distributed* computing, not parallel computing. It's a subtle distinction, but a crucial one. In this book I am going to be discussing parallel computing, not distributed computing.

Google does, however, do other computations that *are* examples of parallel computing—namely, the big data analytics that convert the raw web pages swept up by Google's web crawlers into the query indexes that let Google speedily retrieve web pages that best match users' searches. Here, Google uses *multiple cores* cooperating to solve the *single problem* of converting enormous quantities of web page text into enormous query indexes. In fact, Google was the leader in popularizing the *map-reduce* paradigm of parallel computing on massive data sets. Google, and many other major Internet sites, also use map-reduce parallel computing to convert enormous quantities of information gathered from us users—the queries we do, the web pages we visit, the contents of our Gmail emails, Facebook posts, and Twitter tweets—into advertisements targeted specifically at each of us.

Other modern-day applications that use parallel computing include:

- Computational mathematics: numerical linear algebra, numerical solution of ordinary and partial differential equations, optimization, combinatorics, graph theory, . . .
- Scientific computation: weather prediction, hurricane forecasting, climate modeling, astrophysics (galaxy and star cluster simulation, black hole gravitational wave prediction), chemistry (molecular dynamics, *ab initio* molecular structure prediction), bioinformatics (DNA sequence matching, protein sequence matching, protein structure prediction, pharmaceutical drug design), geology (seismic data analysis, oil and mineral prospecting), . . .
- Engineering computation: computational fluid dynamics, simulated wind tunnels, finite element analysis, circuit verification, integrated circuit chip component and wiring placement, . . .

- Computational finance: asset pricing, derivative pricing, market modeling, algorithmic trading, . . .
- Big data analytics (as already mentioned): data mining, web indexing, user characterization, targeted advertising, . . .
- Security and cryptography: password cracking, cipher attacks, bitcoin mining and transaction processing, . . .
- Entertainment: computer games, computer generated imagery (CGI) for visual effects, computer animated films, . . .
- Fringe: Mersenne prime searching, Search for Extraterrestrial Intelligence (SETI), computer chess, . . .
- And the list goes on.

It wasn't always this way. In the beginning, up through the 1980s, parallel computers were esoteric and expensive. Each parallel computer vendor had its own proprietary hardware architectures, its own proprietary parallel programming languages (often nonstandard extensions to existing languages like Fortran and Lisp), and its own proprietary parallel software libraries. For the most part, the only ones with budgets big enough to afford parallel computers were large industrial organizations and government funded research laboratories. Consequently, parallel computing was used mostly for scientific and engineering applications.

A paradigm shift in parallel computing took place near the end of the twentieth century. Setting the stage was the development during the 1980s and early 1990s of inexpensive PC hardware, cheap Ethernet local area network hardware, standard TCP/IP network protocols, and the free Linux operating system. A 1995 paper* brought all these together and described how to build "Beowulf," an example of what is now called a cluster parallel computer, from standard, off-the-shelf hardware and software, for a mere fraction of the cost of a proprietary supercomputer. This publication ushered in an era of parallel computing done with commodity chips and hardware rather than proprietary machines.

In addition, parallel programming shifted to using standard languages like Fortran, C, and later C++ with standard parallel programming libraries. Message Passing Interface (MPI), introduced in 1994, became the *de facto* standard for parallel programming on cluster parallel computers. OpenMP, introduced in 1997, became the *de facto* standard for parallel programming on multicore parallel computers.

However, at that time multicore machines were still not common; if you wanted lots of cores, you usually had to build a cluster of single-core ma-

* T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer. Beowulf: A parallel workstation for scientific computation. *Proceedings of the 24th International Conference on Parallel Processing (ICPP 1995)*, 1995, volume 1, pages 11–14.

chines. Consequently, parallel computing was still concentrated in industrial and government research settings and was still dominated by scientific and engineering applications.

A second paradigm shift in parallel computing took place in 2004. Until then, processor chip manufacturers had exploited Moore's Law to steadily increase both the number of transistors on a chip and the chip speed, doubling the clock frequency about every two years. But by 2004, clock frequencies had gotten fast enough—around 3 GHz—that any further increases would have caused the chips to melt from the heat they generated (unless outfitted with cooling systems like Indianapolis 500 race cars). So while the manufacturers continued to increase the number of transistors per chip, they no longer increased the clock frequencies. Instead, they started putting multiple processor cores on the chip. A chip with two cores operating at 3 GHz is, theoretically, equivalent to a chip with one core operating at 6 GHz. The number of cores per chip continued to increase, until in 2015 as I am writing, *everyone's* computer is a multicore parallel machine with two, four, eight, or more cores.

At the same time, memory chip densities continued to increase, until now everyone's computer has 4 GB, 8 GB, 16 GB, or more of main memory. Furthermore, with the rise in popularity of computer games, both on desktop PCs and on gaming consoles, everyone's computer has a *graphics processing unit (GPU)* with dozens or hundreds of cores. While originally intended for real-time 3-D video rendering, GPUs can do general-purpose calculations as well. Today's PC is a powerful parallel computer with capabilities undreamed of in the 1980s, with computing power equivalent to a 1990s-era cluster requiring a whole rack of equipment.

The modern era of ubiquitous multicore machines opened parallel computing to a much broader range of applications than just scientific and engineering computation, as mentioned previously. Many of these newer applications eschew Fortran and C and MPI and OpenMP, which arose during the era when scientific and engineering applications dominated parallel computing. Modern applications also use newer languages like Java and newer programming paradigms like map-reduce. A prime example is Apache's Hadoop, a map-reduce library written in Java, designed for parallel computation on massive data sets.

To use your modern computer—your modern *parallel* computer—to its full potential, it's not enough to write plain old programs. You have to write *parallel* programs. These parallel programs have to exploit all the cores in the machine. If possible, these parallel programs also ought to utilize all the cores in the GPU. This book, *BIG CPU, BIG DATA*, is intended to teach you how to write parallel programs for multiple-core, GPU-equipped parallel computers.

Despite the vast increase in the PC's computing power, there still are computational problems too big for a single node to handle. The problems

require more memory than can fit in a single node, or the problems require so much computation that it would take too long for them to finish when run on a single node (even a multicore node), or both. Such problems need to run on a *cluster* parallel computer, one with multiple nodes.

An individual user could afford to set up a small-scale cluster with perhaps two or three or four PC-class nodes. A small company or academic department could afford to set up a medium-scale cluster with perhaps one or two dozen nodes. A large-scale cluster with hundreds or thousands of nodes requires the budgetary resources of a large company or a government.

The United States government, for example, funds a number of supercomputer centers at its national laboratories. These supercomputers are available to researchers who have government funding. The most powerful supercomputer in the U.S., according to the June 2015 Top500 List of the world's fastest computers (www.top500.org), is the Titan computer at the Oak Ridge National Laboratory in Tennessee. Titan has 35,040 nodes, each with 16 CPU cores and 2,688 GPU cores, for a total of 560,640 CPU cores and 94,187,520 GPU cores. Titan is able to execute the Linpack linear algebra parallel processing benchmark at a rate of 17.6 petaflops (17.6×10^{15} floating point operations per second).

Yet even Titan is only number two on the June 2015 Top500 List. First place goes to the Tianhe-2 computer at the National University of Defense Technology in Changsha, China. Tianhe-2 has 16,000 nodes, each with 24 CPU cores and 171 accelerator cores, for a total of 3,120,000 cores. Tianhe-2 executes Linpack at 33.9 petaflops, nearly twice as fast as Titan.

By the way, there's nothing special about supercomputer hardware. Supercomputers nowadays use the same commodity chips as desktop PCs. Tianhe-2 uses Intel Xeon E5 CPU chips (2.2 GHz clock) and Intel Xeon Phi manycore accelerators. Titan uses AMD Opteron CPU chips (2.2 GHz clock) along with Nvidia Tesla K20x GPU accelerators. Supercomputers get their massive computational power by having large numbers of nodes and cores, not from superior chips or enormous clock frequencies.

What about the poor individual or department who needs large-scale computing power but doesn't have large-scale funding? *Cloud computing* is an interesting and viable alternative. A cloud computing service, such as Amazon's EC2 (aws.amazon.com/ec2/), will rent you the use of as many compute nodes as you want, for as long as you want, and charge you only for the actual CPU time you use. Better still, you don't have to buy and maintain the computers, air conditioners, uninterruptible power supplies, network hardware, racks, cables, and so on needed to run a cluster parallel computer. The cloud service provider takes care of all that. Best of all, you don't need a difficult-to-get government research grant. The cloud service provider is happy to charge your credit card.

It's important to know how to write parallel programs for cluster parallel computers, to solve problems that are just too large for a single node. *BIG CPU, BIG DATA* is intended to teach you how to write parallel programs for multiple-node cluster parallel computers, including clusters in the cloud.

To make sense of the bewilderingly diverse landscape that is modern parallel computing, I'm going to characterize parallel computing hardware, software, and applications along several dimensions. Figure 1.1 shows the eight dimensions. The next few sections discuss each dimension. Any particular parallel computer, parallel program, or parallel application can be pinpointed along each dimension, illuminating its place in the overall scheme of things.

Hardware Dimensions

A *node* refers to an independent computer with its own CPU chip or chips, its own main memory, and its own network interface. A parallel computer can consist of a single node, or of multiple nodes. A multinode parallel computer is called a *cluster.*

A *core* refers to the hardware on a node that executes a stream of machine instructions. A node can have a single core, or multiple cores. Each core can also be hyperthreaded, meaning that the core can execute more than one stream of machine instructions.

In addition to the CPU, a node can have zero or more *accelerators.* These are separate processors that can perform computations alongside the CPU. There are several kinds:

- A *graphics processing unit (GPU) accelerator* typically has a large number of cores that are mainly intended for graphics rendering but that can do general calculations.
- A *manycore accelerator* is similar to a general purpose CPU, except it typically has more cores than a CPU, and it omits all the peripheral interface circuitry (disk, network, display, USB, and so on) not needed for doing pure calculation.
- A *field programmable gate array (FPGA) accelerator* is a digital chip whose logic circuitry can be reconfigured during operation, letting you create customized high-speed processors.
- An *application specific integrated circuit (ASIC) accelerator* uses specialized chips hardwired to perform specific computations at very high speed. One example is bitcoin mining (bitcoin.org). You can buy ASICs

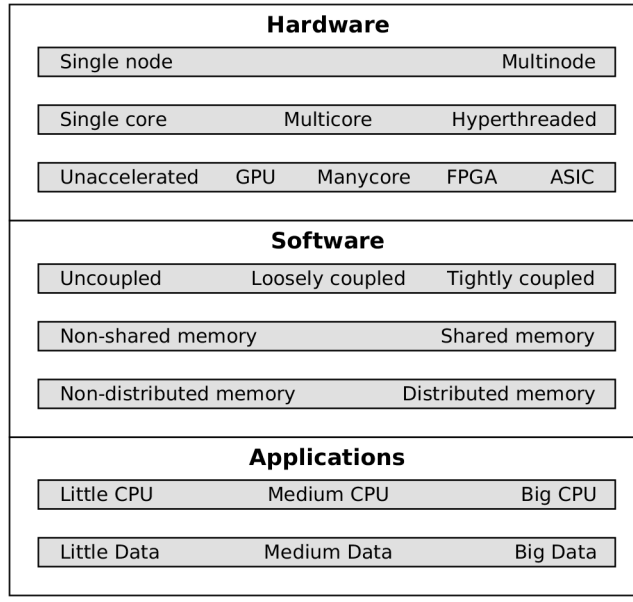
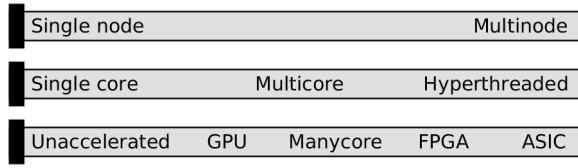


Figure 1.1. Dimensions of the parallel landscape

that compute the SHA-256 cryptographic hash function, part of the bitcoin protocol, at very high speed.

Having examined each hardware dimension by itself, let's look at examples of parallel computers and see where they fall along the dimensions.

Single-Core Computer



Most computers back in the twentieth century were single node single core unaccelerated (Figure 1.2). In such a computer, the core has an *instruction unit* that reads the program's machine instructions from memory, decodes them, and executes them; a number of *functional units*, such as adders, shifters, multipliers, and so on that carry out the machine instructions; and a number of high-speed *registers* that hold intermediate results. With one instruction unit, the core can execute only one stream of instructions—one thread—at a time. To execute more than one thread, the computer's operating

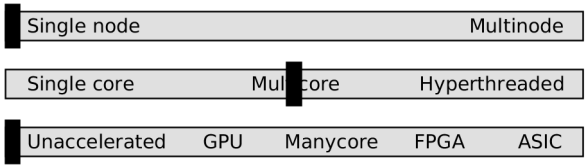
system must do a *context switch* from one thread to another every so often.

The computer has a *main memory* that stores program instructions and data. Because the main memory's circuitry is much slower than the core's circuitry, a *level 2 (L2) cache* sits between the core and the main memory. The L2 cache is faster than the main memory, but smaller (typically a few megabytes). When the core reads a memory location into the instruction unit or into a register, an entire *cache line*, typically 64 or 128 bytes, is fetched from main memory and stored in the L2 cache; from there, the requested data goes into the core. If the core reads an adjacent memory location, the data is already in the L2 cache (a *cache hit*) and can be read quickly without waiting for the slow main memory.

Still, the L2 cache's circuitry is not as fast as the core's circuitry; so another cache, the *level 1 (L1) cache*, sits between the core and the L2 cache. The L1 cache is nearly as fast as the core but is smaller than the L2 cache. On its way to the core, data read from main memory is cached in L2 and in L1. If the instructions and data the program is accessing can fit entirely in L2, or better still L1, the core can run at nearly full speed. Otherwise, the core will have to pause while new data is fetched from the slow main memory, reducing the processing speed.

So as to keep the core running as fast as possible, the L1 and L2 caches are made as large as possible while still fitting on the CPU chip along with the rest of the core circuitry. Often, the majority of a CPU chip's area is devoted to the caches.

Multicore Computer



To increase the processing power while not increasing the clock frequency, computers switched from single core to *multicore* (Figure 1.3). Now there are two or more cores, each with its own instruction unit, functional units, registers, and L1 cache. The cores all share the L2 cache and the main memory. The operating system can run multiple threads simultaneously, one in each core, without needing to context switch. The threads are running truly in parallel. Theoretically, a *K*-core computer ought to run *K* times faster than a single-core computer. (This does not always happen in practice, though.)

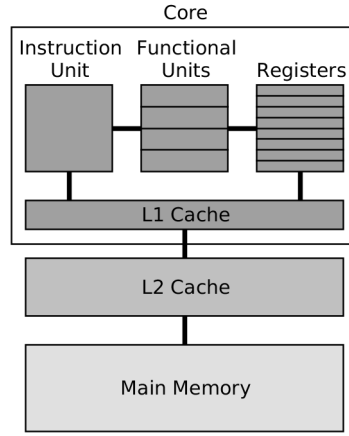


Figure 1.2. Single node single core computer

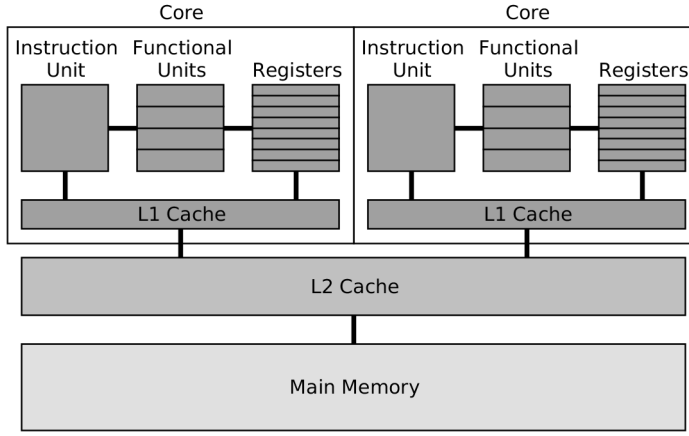
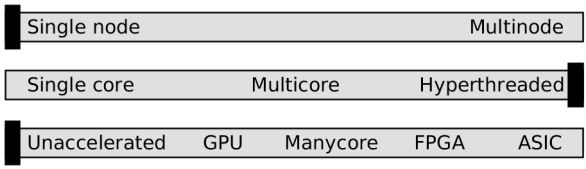


Figure 1.3. Single node multicore computer

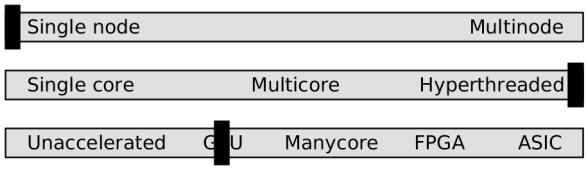
Hyperthreaded Computer



Replicating the instruction unit, functional units, registers, and L1 cache to get multiple cores requires a lot of transistors and chip area. To run more threads without needing quite so much area, multicore computers became *hyperthreaded* (Figure 1.4). In a hyperthreaded core, the instruction units are replicated, but not the functional units, registers, or L1 cache. A dual-hyperthreaded core can run two threads simultaneously without needing to context switch.

There's a catch, though. As long as the two threads are using different functional units and registers, both threads can run at full speed. But if the threads both need to use the same functional unit or register at the same time, the hardware makes the threads take turns. While one thread is accessing the functional unit or register, the other thread stalls. This reduces the effective rate at which the threads can execute instructions. In my experience, a dual-hyperthreaded core typically does not run as fast as two regular cores; but it does run faster than one regular core.

Multicore Accelerated Computer



Lately, parallel computers are including *accelerators* alongside the multicore hyperthreaded CPUs. A *GPU accelerator* (Figure 1.5) repurposes a graphics card to do general calculations. A GPU card has numerous cores, anywhere from dozens to thousands of them, as well as its own main memory. The GPU's main memory is linked with the CPU's main memory over a high-speed bus, allowing data to be transferred between the CPU and GPU. Standard GPU programming libraries, such as Nvidia Corporation's CUDA and the vendor-independent OpenCL, make it almost as easy to write GPU programs as it is to write CPU programs.

Both the CPU and the GPU have to deal with their main memory's large latency (access time) relative to high speed of their cores. The CPU deals with it by interposing L1 and L2 caches between the cores and the memory, devoting the bulk of the chip area to cache, leaving room for only a few cores. The GPU deals with it by reducing or even eliminating the cache, de-

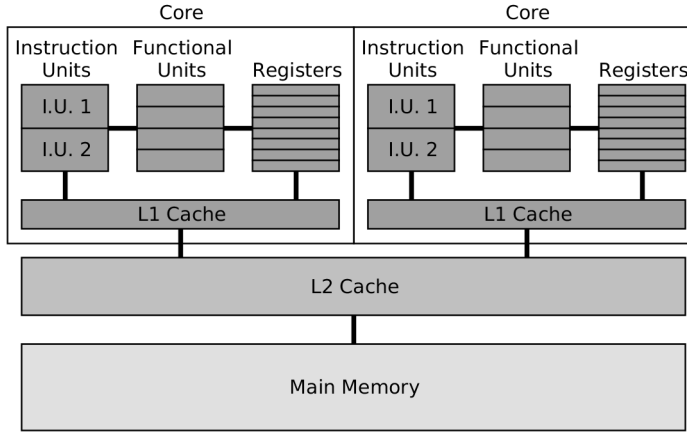


Figure 1.4. Single node multicore hyperthreaded computer

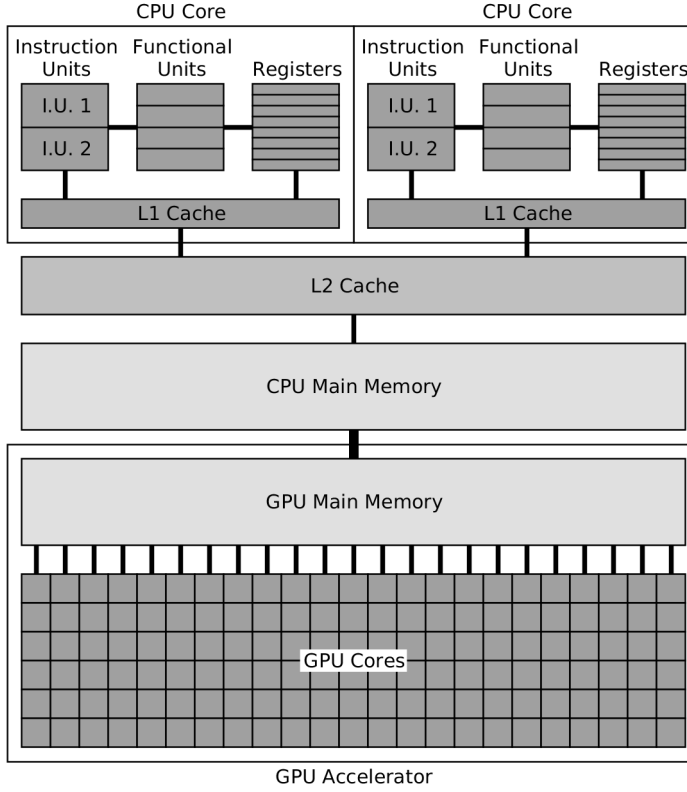
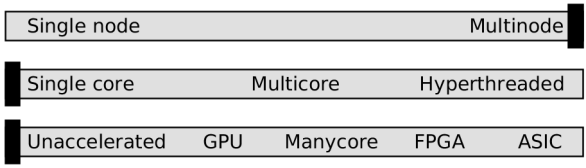


Figure 1.5. Single node multicore hyperthreaded GPU accelerated computer

voting the bulk of the chip area to a large number of cores. The GPU then runs large numbers of threads simultaneously on its cores. When one thread stalls waiting to access main memory, another thread instantly takes its place. There are enough threads so that whenever a core goes idle, a thread is available that has completed its previous memory access and is ready to run again. In this way, the cores stay busy all the time. This technique is called *latency hiding.*

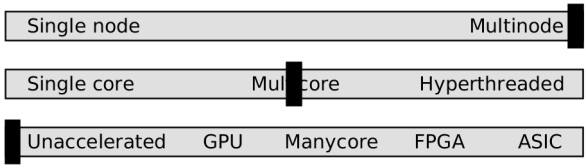
Depending on the nature of the problem, a GPU can perform calculations dozens or hundreds of times faster than a CPU. With GPUs incorporated into just about every modern computer, it's important to know how to write GPU parallel programs, to take advantage of the GPU's additional processing power. *BIG CPU, BIG DATA* is intended to teach you how to write parallel programs for GPU accelerated parallel computers. (I'm not going to cover the more esoteric manycore, FPGA, and ASIC accelerators in this book.)

Single-Core Cluster Computer



Turning away from single-node parallel computers, we come to multinode parallel computers, or *clusters* (Figure 1.6). The cluster has some number of *backend nodes* that carry out computations. Each node has a single core plus its own main memory. We say the cluster has a *distributed memory;* the cluster's memory is distributed across the nodes instead of concentrated in a single node. As we will see, the distributed memory has profound implications for the design of cluster parallel programs. The cluster has a dedicated high-speed *backend network* that allows the backend nodes to communicate with each other. The backend network may use commodity Ethernet hardware, or it may use specialized faster technology such as Infiniband, Myrinet, or Scalable Coherent Interface (SCI), or it may use a proprietary interconnect. The cluster usually also has a *frontend node,* connected to the Internet to let users log in and run parallel programs on the cluster, and connected to the backend network to control and monitor the backend nodes.

Multicore Cluster Computer



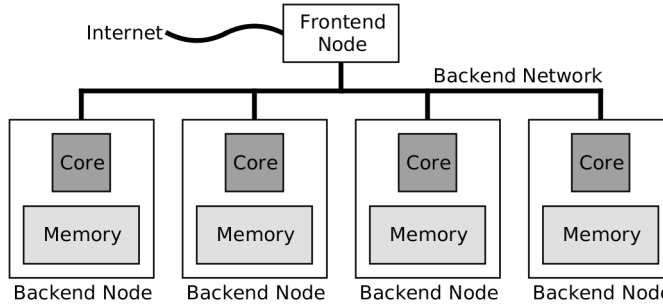


Figure 1.6. Multinode single core cluster computer

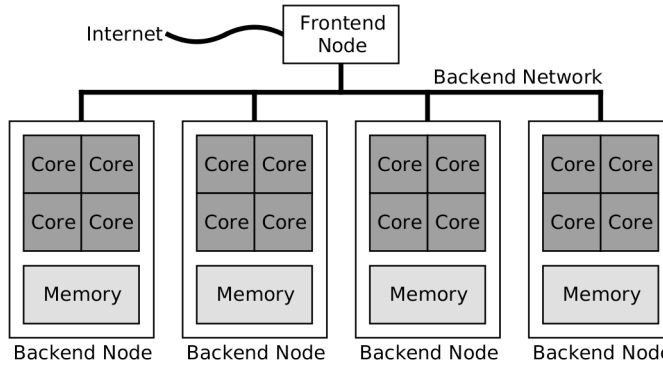


Figure 1.7. Multinode multicore cluster computer

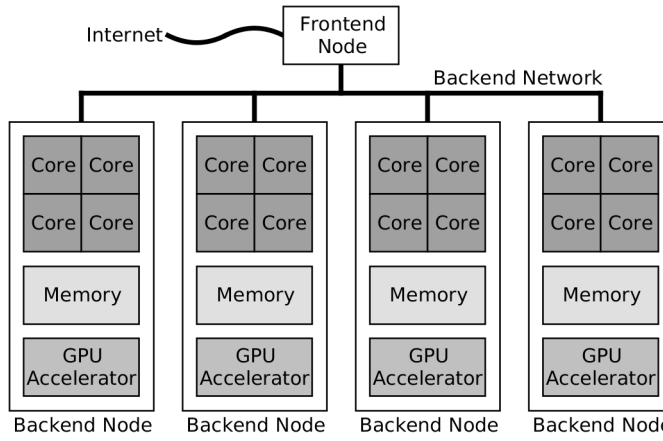
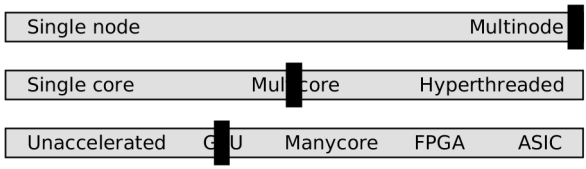


Figure 1.8. Multinode multicore GPU accelerated cluster computer

However, nowadays it's virtually impossible to buy a single-core node. So a modern cluster parallel computer consists of multiple multicore backend nodes (Figure 1.7). The cores might or might not be hyperthreaded.

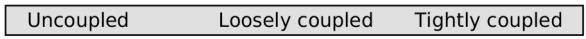
Multicore Accelerated Cluster Computer



In addition, the nodes of a modern cluster parallel computer might include accelerators (Figure 1.8). The Titan supercomputer looks like Figure 1.8; it has 35,040 nodes, each with 16 CPU cores and a GPU accelerator, and it uses a high-speed proprietary interconnect for its backend network.

Software Dimensions

A parallel program consists of multiple threads performing computations simultaneously. In a single-node multicore parallel computer, the threads run on the cores of the node. In a cluster parallel computer, the threads run on the cores of all the nodes.



The computations performed by the threads can be *uncoupled*, *loosely coupled*, or *tightly coupled*. In an uncoupled computation, the threads do not communicate or coordinate with each other at all; each thread runs and produces its results independently of all the other threads. In a loosely coupled computation, the threads communicate with each other, but only infrequently; for example, the threads compute results independently of each other, but at the end of the program the threads communicate their individual results to each other and combine them into one overall result. In a tightly coupled computation, the threads communicate with each other frequently; for example, each thread executes a loop, and at the end of every loop iteration, the threads communicate the results of that iteration to the other threads before proceeding with the next iteration.

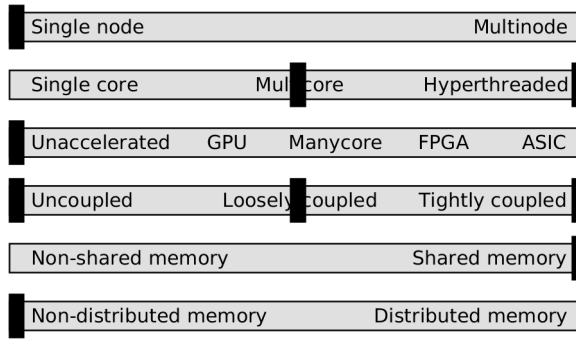
Coupling also refers to the quantity of data communicated between the threads. In an uncoupled computation, no data is exchanged between the threads. In a loosely coupled computation, a small amount of data is exchanged between the threads. In a tightly coupled computation, a large amount of data is exchanged between the threads. A particular parallel program might fall anywhere along the spectrum from uncoupled to tightly coupled.

In a *shared memory* parallel program, the data items the threads are accessing as they perform their computations—input parameters, intermediate values, output results—are stored in a single memory region that is shared by all the threads. Thus, any thread can get any other thread's results simply by reading the appropriate locations in the shared memory. In a non-shared memory parallel program, the threads do not have access to a common shared memory.

In a *distributed memory* parallel program, the data items the threads are accessing are stored in multiple memory regions. Each thread can directly read and write locations in one of the memory regions, and the thread stores its own data in that memory region. Each thread can also access the contents of the other memory regions, but not by directly reading and writing them. Rather, one thread accesses another thread's memory region via some form of explicit communication. The communication can take the form of *message passing;* the thread that owns the data sends a message that is received by the thread that needs to use the data. (MPI is a library of message passing subroutines of this sort.) Other possible communication mechanisms include *remote procedure call* (RPC), *remote method invocation* (RMI), and *tuple space.* In a non-distributed memory parallel program, the threads do not have any access to other threads' memory regions.

Having examined each software dimension by itself, let's look at examples of parallel computing software and see where they fall along all the dimensions.

Multicore Parallel Program



A parallel program intended to run on a single multicore node (including hyperthreaded cores) typically uses a shared memory model (Figure 1.9). The program runs in a single process with multiple threads, each thread executing on a different core. The program's data is located in the computer's memory.

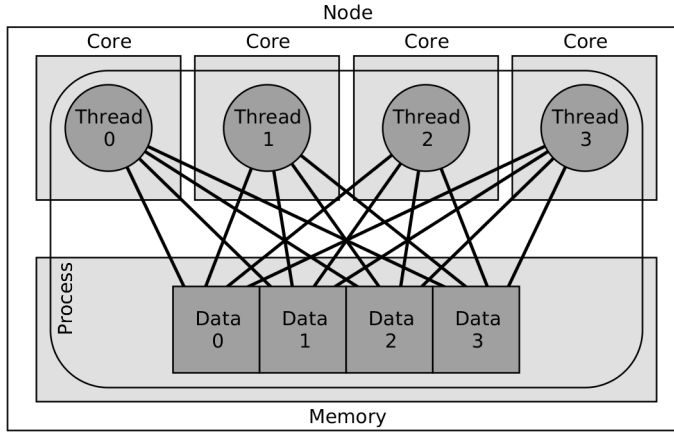


Figure 1.9. Shared memory parallel program running on a multicore node

Because all the threads are part of the same process, and the process consists of a single address space, each thread can access all the program's data; this is how the shared memory is achieved.

Typically, the data is partitioned into as many pieces as there are threads. Each thread computes and writes its own piece of the data. Each thread also reads the other pieces of the data as necessary. Thus, the threads communicate with each other by writing and reading the same shared memory locations. The threads can also coordinate with each other using synchronization primitives supported by most operating systems, such as *semaphores*, *locks*, and *barriers.*

Shared memory parallel programs can be tightly coupled, loosely coupled, or uncoupled. A loosely coupled or uncoupled program still looks like Figure 1.9; the only difference is the frequency with which one thread accesses another thread's data, or the amount of data accessed.

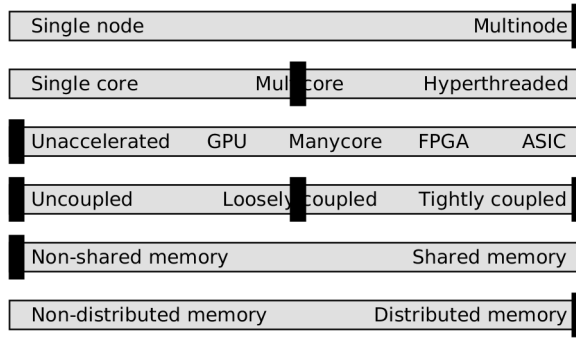
Shared memory parallel programs are just multithreaded programs, no more and no less. You can write a shared memory parallel program using the built-in threading constructs of a programming language or an operating system, such as Java threads or Unix pthreads. However, folks who need to do parallel computing often are not experts in writing threaded code. They want to write high-level application programs to solve their computational problems; they don't want to have to write low-level code to create threads, acquire and release semaphores and locks, destroy threads, and so on. Consequently, most folks use a *parallel programming library*, or *application programming interface (API)*, to write shared memory parallel programs. The API exposes high-level application-oriented parallel programming constructs

to the programmers, and the API handles all the low-level threading details under the hood.

OpenMP (www.openmp.org) is a widely used API for shared memory parallel programming. First released in 1997, and now in its fourth revision (version 4.0 of the OpenMP specification was released in July 2013), OpenMP supports parallel programming in the Fortran, C, and C++ languages.

I prefer to program in Java. So I prefer not to use OpenMP, which does not—and, in my belief, never will—support Java. Instead, I'm going to use the *Parallel Java 2 Library,* which I have developed, to teach you shared memory parallel programming in Java.

Cluster Parallel Program



A parallel program intended to run on a cluster of single-core or multicore nodes typically uses a distributed memory model (Figure 1.10). The program runs in multiple processes, one process for each core of each backend node. Each process has one thread running on the core plus data located in the node's memory. Typically, the data is partitioned into as many pieces as there are threads. But because the threads and data pieces are in different processes with different address spaces, the memory is not shared. Each thread can access its own data directly. But if one thread needs to use a data item located in another thread's memory region, *message passing* has to take place.

For example, suppose thread 7 needs a data item located in thread 3's memory region. Thread 3 has to retrieve the data item from its memory and load the data into a message of some kind. Because the threads are running on different nodes, thread 3 must send the message over the cluster's backend network to thread 7—an *inter-node* message. Thread 7 has to receive the message and extract the data. Or suppose thread 6 needs a data item located in thread 4's memory region. Although the threads are running on the same node, because the threads are running in different processes (different address spaces), a message still has to go from thread 4 to thread 6—an *intra-node* message. The threads' programs need to be coded to invoke message passing operations explicitly; this increases the complexity and programming effort for cluster parallel programs.

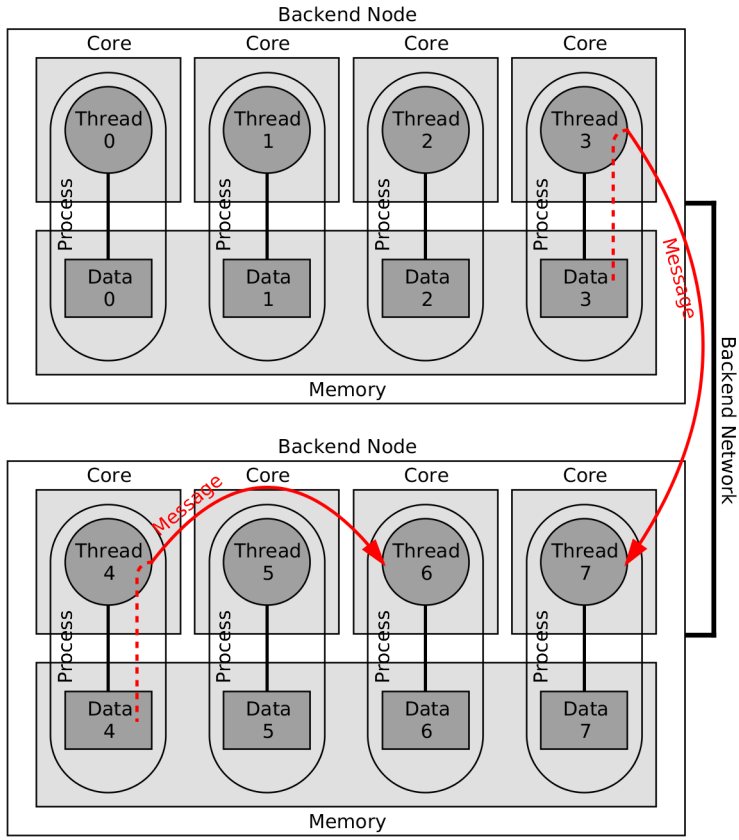


Figure 1.10. Distributed memory parallel program running on a cluster of multicore nodes

Message passing can, of course, be more complicated than these simple examples. One owner thread might need to send data items to several recipient threads. One recipient thread might need to gather data items from several owner threads. Every thread might need to send data items to and receive data items from every other thread.

Cluster parallel programs can be tightly coupled, loosely coupled, or uncoupled. An uncoupled program still looks like Figure 1.10, except there is no message passing. A loosely coupled program looks like Figure 1.10, but does fewer or less frequent message passing operations, or sends less data, than a tightly coupled program.

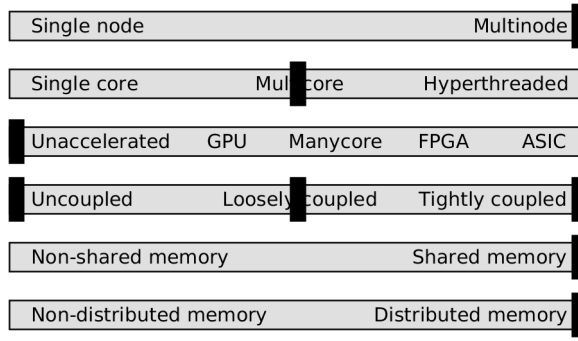
You can write a cluster parallel program using the *interprocess communication (IPC)* constructs of an operating system or using networking soft-

ware like TCP sockets. However, folks who need to do parallel computing often are not experts in writing IPC or networking code. They want to write high-level application programs to solve their computational problems; they don't want to have to write low-level code to open and close sockets, format data into and out of messages using some protocol, and so on. Consequently, as with shared memory parallel programming, most folks use a parallel programming library or API to write cluster parallel programs. The API exposes high-level application-oriented parallel programming constructs to the programmers, and handles all the low-level networking details under the hood.

To achieve acceptable performance, a tightly coupled cluster parallel program needs to use a fast, low-overhead message passing library. *Message Passing Interface (MPI)* (mpi-forum.org) is a widely used API for cluster parallel programming. First released in 1994, and updated to Version 3.0 in September 2012, MPI supports parallel programming in the Fortran, C, and C++ languages. MPI is a library of message passing subroutines; programs written to call these subroutines can run on any machine that has an MPI library installed. Often, a platform-specific MPI implementation is used to wring the fastest possible performance out of the hardware. Platform-independent MPI implementations are also available but tend to be slower.

Because I prefer to program in Java, I prefer not to use MPI, which does not—and, in my belief, never will—support Java. Also, I'm not fond of MPI's huge and complicated API. (It fills an 852-page book!) The Parallel Java 2 Library includes message passing capabilities via a simple API called *tuple space.* However, the Parallel Java 2 Library is intended mainly for uncoupled and loosely coupled cluster parallel programs. While tightly coupled cluster parallel programs can be written with Parallel Java 2, the tuple space's platform independent implementation is not designed to achieve the highest possible speed. If you need extremely fast message passing, use MPI.

Multicore Cluster Parallel Program



A tightly coupled parallel program running on a cluster of multicore nodes requires frequent exchange of copious data both between processes running on the same node and between processes running on different nodes,

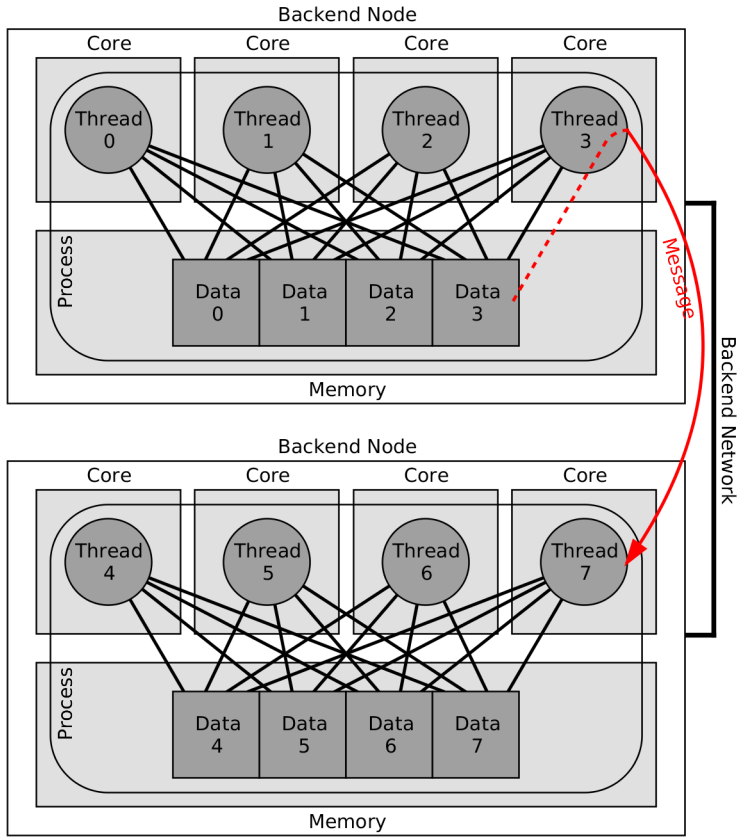
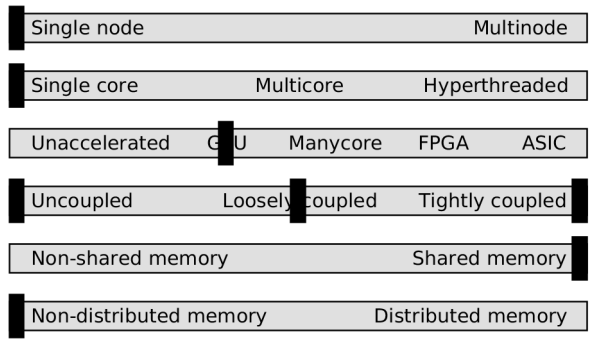


Figure 1.11. Hybrid shared/distributed memory parallel program running on a cluster of multicore nodes

as shown in Figure 1.10. But it doesn't make sense to do message passing between processes running on the same node. Sending data between processes on a node is typically much slower than accessing the data directly. What does make sense is to use the shared memory model within each node and the distributed memory model between nodes—a *hybrid shared/distributed memory* model (Figure 1.11). On each node there is one process with multiple threads, one thread per core, with the threads directly accessing each other's data in shared memory. When data has to go from one process (node) to another, message passing is used. By eliminating the bulk of the messages needed in the pure distributed memory model (Figure 1.10), the program's performance is improved.

Some parallel programs have more than one level of parallelism. A program might perform many separate, uncoupled computations, which can therefore be done in parallel. Each of these might itself be a tightly coupled parallel computation. Such a program is ideally suited to run on a multicore cluster parallel computer using the hybrid shared/distributed memory model. The computations run in separate parallel processes on separate nodes, with no message passing between computations. Each computation runs in multiple parallel threads on separate cores in the same node, all the threads accessing the computation's data in shared memory.

GPU Accelerated Parallel Program



All the previous parallel software options involved unaccelerated nodes. Figure 1.12 depicts a parallel program that uses a GPU accelerator. The figure shows the simplest kind of GPU parallel program: running in a single CPU thread, on a single CPU core, with all the parallelism in the GPU. The red arrows show what I like to call the *GPU computational arc:*

- The CPU sets up input data for the computation in the CPU memory. Often the data is an array or matrix consisting of many, many elements.
- The CPU sends the input data from the CPU memory to the GPU memory.
- The CPU launches a large number of *GPU threads.* Each thread will execute a *kernel function* (denoted by "K" in Figure 1.12). The whole assemblage of GPU threads executing kernel functions is called the *computational kernel,* or just *kernel.* The CPU waits for the kernel to finish.
- Each GPU thread executes the kernel function on a GPU core. Often, each individual data element is computed by its own separate GPU thread. The computation's output data is stored back in the GPU memory.
- When the kernel finishes, the CPU wakes up and sucks the output data from the GPU memory back to the CPU memory.
- The CPU outputs the computation's results.

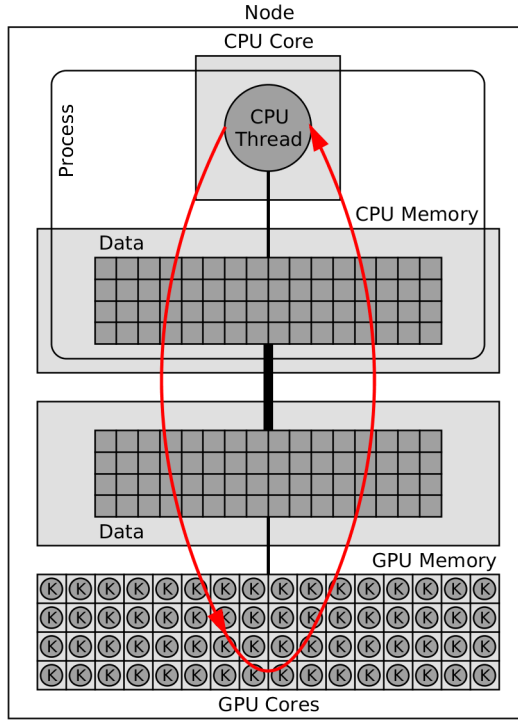


Figure 1.12. GPU accelerated parallel program

The GPU cores achieve their best performance when they all execute the exact same stream of machine instructions in lockstep, each on different data items—what is called *single instruction stream multiple data stream (SIMD)* parallelism. The GPU cores also achieve their best performance when the data they are accessing is stored in a regular pattern in memory, such as array or matrix elements in contiguous memory locations. A program that has a lot of data-dependent branching, with different instruction sequences being executed depending on the data values, or a program that has irregular data access patterns, such as pointer chasing through linked data structures, will not perform well on a GPU. Thus, typically only a portion of a GPU parallel program runs on the actual GPU—namely, the SIMD, regular-data-access, computational kernel. The rest of the parallel program runs on the CPU.

A GPU accelerated parallel program might run on more than one CPU core: the computational kernel runs in parallel on the GPU cores, and the non-kernel portion runs in parallel on the CPU cores. The CPU threads might run in parallel with the GPU threads, rather than waiting for the kernel to finish. Multiple kernels might run on the GPU at the same time. And with a

GPU accelerated cluster, all this could be happening in parallel repeatedly on multiple nodes. The possibilities for parallelism are endless.

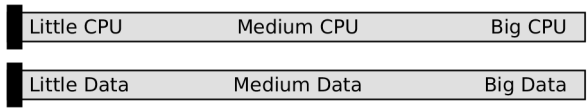
Nvidia Corporation pioneered general purpose computing on GPUs with their proprietary *Compute Unified Device Architecture (CUDA)* and the programming API that goes with it. CUDA supports writing GPU kernel functions in Fortran, C, and C++. The CPU main programs are written in the same languages. *OpenCL* (www.khronos.org/opencl) is a more recent, vendor neutral API for GPU programming. First released in 2009, and last updated in November 2014, OpenCL uses its own programming language based on C.

The Parallel Java 2 Library supports GPU parallel programming via a combination of CUDA and Java. The GPU kernel functions are written in C or C++ using CUDA. (OpenCL support is a planned enhancement.) The main programs running on the CPU are written in Java, using classes that provide high level abstractions of the GPU. Under the hood, these classes access the GPU via Java's native interface capability. (At this time, I don't know of a way to write GPU kernel functions directly in Java. A Java compiler targeting GPUs would make for a very interesting project!)

Application Dimensions

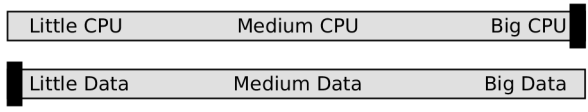
Parallel computing applications are characterized along two orthogonal dimensions: little CPU—big CPU, and little data—big data.

Little CPU Little Data Application



A little CPU little data application works with only a small amount of data, and does only a few calculations (CPU cycles) with each data item. Still, the calculations are or can be done in parallel. A spreadsheet is an example. Compared to a supercomputer program, a spreadsheet works with very little data (the cell values) and does very little computation (the cell formulas). However, the cells can be calculated in parallel, as long as any data dependencies between cells are obeyed.

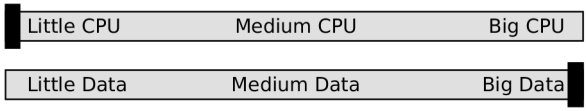
Big CPU Little Data Application



A big CPU little data application also works with only a small amount of data, but it spends a large amount of CPU time doing calculations with that data. Doing the calculations in parallel can speed up the application. Crypto-

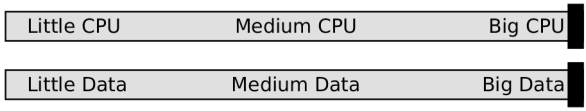
graphic applications are often of this kind. Bitcoin mining is one example. A bitcoin "block" is a piece of digital cash. It occupies just a few kilobytes of data. But determining the value of a certain field of the block—"mining" the bitcoin, as it is called—requires calculating the SHA-256 cryptographic hash function many, many times. These calculations can be, and usually are, performed in parallel. Also, bitcoin miners can mine multiple blocks in parallel.

Little CPU Big Data Application



A little CPU big data application devotes only a little bit of CPU time to each data item, but works with an enormous number of data items. Consequently the application can take a long time to run, and processing the data in parallel can speed it up. Map-reduce, pioneered by Google and implemented in the popular Apache Hadoop, is a widely used paradigm for parallel big-data applications. Apache's "Powered by Hadoop" web page* shows that many major Internet players—Amazon, eBay, Facebook, Hulu, LinkedIn, Spotify, Twitter, Yahoo, and others—use Hadoop running on multicore clusters for their big data analytics. Google also does big data analytics with their own map-reduce software. The Parallel Java 2 Library includes *Parallel Java Map Reduce (PJMR)*, a lightweight map-reduce framework built on top of the Library's cluster parallel programming capability.

Big CPU Big Data Application



Finally, a big CPU big data application works with lots and lots of data *and* does lots and lots of calculations with each data item. Scientific and engineering calculations on supercomputers are of this kind. As an example of the extreme scale of these applications, consider the LAMMPS molecular dynamics program, which simulates the motion of atoms from first principles of physics, running on the Keeneland supercomputer at the Oak Ridge National Laboratory. Keeneland is a medium-size cluster of 120 nodes, with two CPU cores and three GPU accelerators per node. LAMMPS running a one billion atom benchmark for 100 time steps requires about half a terabyte (5×10^{11} bytes) of data and would take the better part of an hour (2,350 seconds) on one core of Keeneland. Running on the entire cluster, the same benchmark takes just 17.7 seconds.†

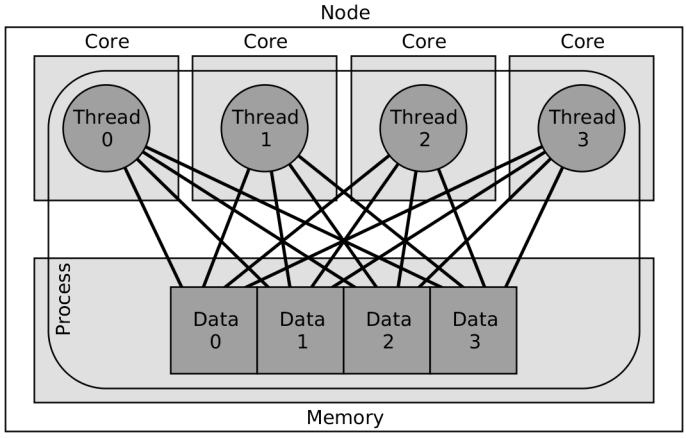
* http://wiki.apache.org/hadoop/PoweredBy

† http://lammps.sandia.gov/bench.html

Points to Remember

- A parallel program runs on *multiple cores*, with all the cores cooperating with each other to solve a *single problem.*
- Nowadays, improved computing performance comes from parallel programs, not increased CPU clock speeds.
- Applications in every area of computing now run on parallel computers.
- Parallel computing hardware can be characterized along three dimensions: single node—multinode, single core—multicore—hyperthreaded, and unaccelerated—accelerated.
- Parallel computing software can be characterized along three dimensions: uncoupled—loosely coupled—tightly coupled, non-shared memory—shared memory, and non-distributed memory—distributed memory.
- Parallel computing applications can be characterized along two dimensions: little CPU—big CPU, and little data—big data.
- The modern computer programmer must know how to write multicore, cluster, and GPU accelerated parallel programs.
- Parallel programs are written using a library, such as OpenMP, MPI, CUDA, OpenCL, or the Parallel Java 2 Library.

PART II
TIGHTLY COUPLED MULTICORE



Parallel program running on a multicore node

Chapter 2
Parallel Loops

We begin our study of tightly coupled multicore parallel programming with a simple Parallel Java 2 program to test numbers for primality. Recall that a number is prime if it is divisible only by itself and 1. PrimeSeq (Listing 2.1) is a sequential (non-parallel) program to test the numbers specified on the command line. The program illustrates several features that I'll include in all the parallel programs we study:

- The program is implemented as a subclass of class Task (in package edu.rit.pj2), and the program's code is in the body of the task's `main()` method.
- Unlike a typical Java program, the `main()` method is an instance method, not a static method. The Parallel Java 2 middleware expects this.
- The `main()` method is declared to throw any exception (line 9). If an exception is thrown anywhere in the program, this lets the exception propagate out of the `main()` method, which will terminate the program and print an error message with an exception stack trace. I do this because I'm lazy and I don't want to write a handler for every exception.
- At line 12, the program makes sure the proper command line arguments are present; if not, the program prints an error message and exits. If I run the program with no arguments, this reminds me what the arguments should be.
- The program exits by throwing an IllegalArgumentException on line 41, which propagates out of the `main()` method, causing the program to terminate. *The program must not call* `System.exit();` doing so interferes with the Parallel Java 2 middleware.
- The static `coresRequired()` method (lines 45–48) has been overridden to return the value 1, indicating that this program will use one core. This is standard for a non-parallel program. If the `coresRequired()` method is not overridden, the Parallel Java 2 middleware will assume that the program will use all the cores on the node.

The loop on lines 15–17 is the heart of the program. The loop iterates over the command line argument array, converts each number from a String to a `long`, calls the `isPrime()` method, and prints the number if `isPrime()` says it's prime. The `isPrime()` method uses *trial division* to test whether the number x is prime. The method tries to divide x by 2 and by every odd number p up through the square root of x. If any remainder is 0, then p is a factor of x, so x is not prime. If none of the remainders are 0, then x is prime. (There's no point in trying factors greater than the square root of x, because if there were such a factor, x would have another factor less than the square root of x, and we would have found that other factor already.) Trial division is not a very efficient algorithm for primality testing, but it suffices for this example.

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Task;
3  public class PrimeSeq
4     extends Task
5     {
6     // Main program.
7     public void main
8        (String[] args)
9        throws Exception
10       {
11       // Validate command line arguments.
12       if (args.length < 1) usage();
13
14       // Test numbers for primality.
15       for (int i = 0; i < args.length; ++ i)
16          if (isPrime (Long.parseLong (args[i])))
17             System.out.printf ("%s%n", args[i]);
18       }
19
20    // Test the given number for primality.
21    private static boolean isPrime
22       (long x)
23       {
24       if (x % 2 == 0) return false;
25       long p = 3;
26       long psqr = p*p;
27       while (psqr <= x)
28          {
29          if (x % p == 0) return false;
30          p += 2;
31          psqr = p*p;
32          }
33       return true;
34       }
35
36    // Print a usage message and exit.
37    private static void usage()
38       {
39       System.err.println ("Usage: java pj2 " +
40         "edu.rit.pj2.example.PrimeSeq <number> ...");
41       throw new IllegalArgumentException();
42       }
43
44    // Specify that this task requires one core.
45    protected static int coresRequired()
46       {
47       return 1;
48       }
49    }
```

Listing 2.1. PrimeSeq.java

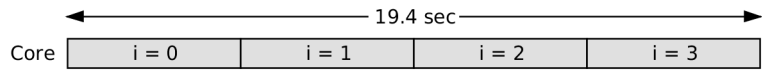
I ran the PrimeSeq program on `tardis`, a cluster parallel computer with ten nodes, each node having four cores at a clock speed of 2.6 GHz. For now I'll confine myself to running multicore parallel programs on just one node of the cluster. (Later we will develop cluster parallel programs and run them on all the cluster nodes.) Because PrimeSeq is not a parallel program, though, it ran on only one core. I gave it four very large numbers to test, all of which happened to be prime. Here is the command and the program's output on one of the `tardis` nodes:

```
$ java pj2 debug=makespan edu.rit.pj2.example.PrimeSeq \
  100000000000000003 100000000000000013 100000000000000019 \
  100000000000000021
100000000000000003
100000000000000013
100000000000000019
100000000000000021
Job 1 makespan 19422 msec
```

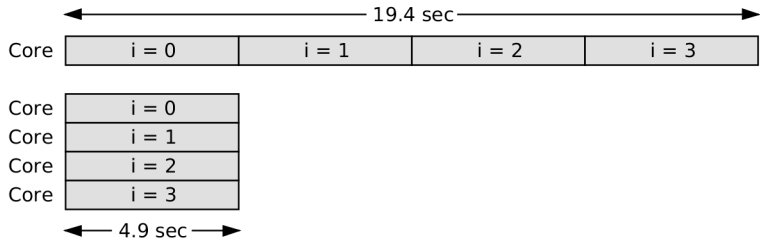
Note that I ran the program by typing the command "`java pj2`". `pj2` is the actual Java main program; it is a *launcher* for Parallel Java 2 programs. `pj2` creates an instance of the specified class (class edu.rit.pj2.example.PrimeSeq in this case), which must be a subclass of class Task. `pj2` then calls the task's `main()` method, passing in an array of the command line argument strings.

I also included the option "`debug=makespan`" before the task class name. This sets the `pj2` program's `debug` parameter to include the "makespan" debug printout. Makespan is the elapsed wall clock time from when the task starts running to when the task finishes running. With that option, the `pj2` program measures the makespan and prints it as the final line of output. (You can turn on other debug printouts and set other `pj2` parameters as well. Refer to the Javadoc documentation for the `pj2` program.)

The loop iterations executed sequentially one after another on a single core. The running time measurement says that the whole program took 19.4 seconds. From this I infer that each loop iteration—each execution of the `isPrime()` method—took 4.9 seconds.



However, for this program there's no need to do the loop iterations in sequence. Because no iteration depends on the results of any prior iteration, we say that the loop does not have any *sequential dependencies*. (Later we will study loops that do have sequential dependencies.) Therefore, we can execute all the loop iterations in parallel, each on a separate core. Doing so, we hope the program will finish in less time.



PrimeSmp (Listing 2.2) is a parallel version of the primality testing program. It starts out the same as program PrimeSeq. But I replaced the normal, sequential for loop in the original program with a *work sharing parallel for loop* in the new program (lines 16–23). The pattern for writing a parallel for loop is

```
parallelFor (lb, ub) .exec (new Loop()
   {
   public void run (int i)
      {
      Loop body code for iteration i
      }
   });
```

The parallel for loop begins with `parallelFor` instead of just `for`. (`parallelFor()` is actually a method of class Task.) Then come the lower and upper bounds of the loop index. The loop goes from *`lb`* to *`ub` inclusive*, so at line 16 I specified bounds of `0` through `args.length-1` to loop over all the command line arguments. The statement so far creates a *parallel loop object.* Then I called the parallel loop's `exec()` method, passing in another object, namely the loop body. The loop body is an instance of a subclass of class Loop (in package edu.rit.pj2), which I created using Java's anonymous inner class syntax. I put the code for one loop iteration in the Loop class's `run()` method, whose argument is the loop index `i`; this is the same code as the loop body in the sequential version. The rest of the program is the same as the sequential version.

I did not override the `coresRequired()` method in the PrimeSmp program. Thus, by default, the Parallel Java 2 middleware will assume that the program will use all the cores on the node.

When the PrimeSmp program runs, the parallel for loop object that is created contains a hidden *parallel thread team.* There is one thread in the team for each core of the machine where the program is executing. The loop iterations are partitioned among the team threads, and each team thread calls the loop body's `run()` method repeatedly for a different subset of the loop indexes, concurrently with the other team threads. In other words, the *work* of the parallel for loop is *shared* among all the threads, rather than being executed by a single thread as in the sequential version. When the program runs on a multicore parallel computer, the Java Virtual Machine and the operating

system schedule each team thread to run on a separate core, resulting in parallel execution of the loop iterations. At the end of the parallel for loop, each team thread waits until all the team threads have finished executing their subsets of the loop iterations, then the program proceeds to execute whatever comes after the parallel loop. This implicit thread synchronization is called a *barrier.*

I ran the PrimeSmp program on `tardis`, with the same arguments as the previous example. A `tardis` node has four cores, so the parallel for loop has four team threads. The loop's iterations are divided among the team threads; thus, each team thread does one iteration. Here is the result:

```
$ java pj2 debug=makespan edu.rit.pj2.example.PrimeSmp \
  100000000000000003 100000000000000013 100000000000000019 \
  100000000000000021
100000000000000013
100000000000000019
100000000000000003
100000000000000021
Job 2 makespan 4907 msec
```

Notice two things about the parallel program. First, while the sequential program finished in 19.4 seconds, the parallel program finished in only 4.9 seconds. From this I infer that the four loop iterations were indeed performed simultaneously on four cores instead of one after another on a single core. That is, the parallel program yielded a *speedup* over the sequential program. To be precise, the speedup factor was 19422 ÷ 4907 = 3.958. (Later we will study more about parallel program performance and the metrics with which we measure performance.)

Second, while the parallel program did determine correctly that every number on the command line was prime, the parallel program reported the prime numbers in a different order from the sequential program. This happened because each number was printed by a separate team thread, and there was nothing in the program to force the team threads to print the numbers in any particular order. (In this example program, there's no need to synchronize the threads so as to make them do their printouts in a certain order.)

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Loop;
3  import edu.rit.pj2.Task;
4  public class PrimeSmp
5     extends Task
6     {
7     // Main program.
8     public void main
9        (final String[] args)
10       throws Exception
11       {
12       // Validate command line arguments.
13       if (args.length < 1) usage();
14
15       // Test numbers for primality.
16       parallelFor (0, args.length - 1) .exec (new Loop()
17          {
18          public void run (int i)
19             {
20             if (isPrime (Long.parseLong (args[i])))
21                System.out.printf ("%s%n", args[i]);
22             }
23          });
24
25    // Test the given number for primality.
26    private static boolean isPrime
27       (long x)
28       {
29       if (x % 2 == 0) return false;
30       long p = 3;
31       long psqr = p*p;
32       while (psqr <= x)
33          {
34          if (x % p == 0) return false;
35          p += 2;
36          psqr = p*p;
37          }
38       return true;
39       }
40
41    // Print a usage message and exit.
42    private static void usage()
43       {
44       System.err.println ("Usage: java pj2 " +
56          "edu.rit.pj2.example.PrimeSmp <number> ...");
57       throw new IllegalArgumentException();
58       }
59    }
```

Listing 2.2. PrimeSmp.java

Under the Hood

Figure 2.1 shows in more detail what happens when the PrimeSmp program executes the parallel for loop code,

```
parallelFor (0, args.length – 1) .exec (new Loop()
   {
   public void run (int i)
      {
      if (isPrime (Long.parseLong (args[i])))
         System.out.printf ("%s%n", args[i]);
      }
   });
```

on a parallel computer with four cores. Keep in mind that all this happens automatically. I only had to write the code above. Still, it's helpful to understand what's going on under the hood.

In the figure, the various objects are arranged from top to bottom, and time flows from left to right. Methods called on each object are depicted as gray boxes. The threads calling the methods are shown as thick lines. A solid line means the thread is executing; a dashed line means the thread is blocked waiting for something to happen.

Execution begins with the main program thread calling the `main()` method on the Task object, an instance of the PrimeSmp subclass. The main thread calls the task's `parallelFor()` method with a lower bound of 0 and an upper bound of 3, which are the index bounds of the `args` array with four command line arguments. The `parallelFor()` method creates a parallel for loop object, an instance of class IntParallelForLoop. Hidden inside the parallel for loop is a team of threads, one thread for each core. Each team thread has a different *rank* in the range 0 through 3. Initially, the team threads are blocked until the program is ready to execute the parallel for loop.

The main thread creates a loop object, which is an instance of an anonymous inner subclass of class Loop. The main thread calls the parallel for loop's `exec()` method, passing in the loop object. The `exec()` method creates three additional copies of the loop object by calling the loop object's `clone()` method. The main thread unblocks the team threads, then blocks itself inside the `exec()` method until the team threads finish.

At this point the parallel for loop begins executing. Each team thread calls the `run()` method on a different one of the loop objects. Because each team thread is executing on a different core, the `run()` method calls proceed in parallel. Each team thread passes a different index as the `run()` method's argument. However, across all the team threads, every loop index from 0 to 3 gets passed to some loop object's `run()` method. Each team thread, executing the code in its own `run()` method, tests one of the numbers on the command line for primality and prints the number if it's prime.

The Parallel Java 2 middleware automatically collects the characters

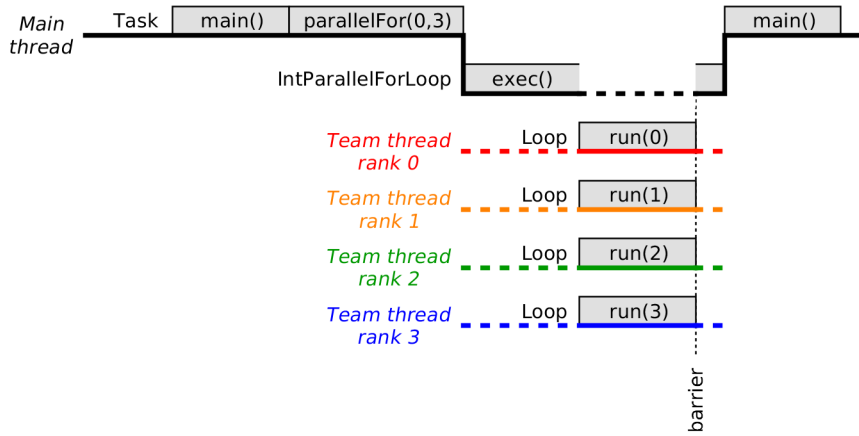


Figure 2.1. Parallel for loop execution flow

each thread prints on `System.out` (or `System.err`) in separate per-thread internal buffers. When the program terminates, the buffers' contents are written to the program's standard output stream (or standard error stream), one buffer at a time. Thus, characters printed by different threads are not commingled. To emit a printout before the end of the program, after printing the characters, call `System.out.flush()` (or `System.err.flush()`).

After returning from the loop object's `run()` method, each team thread waits at a barrier. When all the team threads have arrived at the barrier, the team threads block themselves, and the main thread is unblocked. The main thread resumes executing and returns from the parallel for loop's `exec()` method. The main thread continues executing the code in the task's `main()` method after the parallel for loop.

Thus, the parallel program follows this pattern of execution:

- Sequential section (single main program thread)
- Parallel section (multiple team threads)
- Sequential section (single main program thread)

This pattern, of one or more parallel sections interspersed within a sequential section, is found in almost every multicore parallel program. Only a portion of the program is executed in parallel; the rest of the program is executed sequentially. We will return to this observation when we study parallel program performance.

What if we run the PrimeSmp program with four command line arguments on a parallel computer with more than four cores? The parallel team will have more threads than needed to handle all the loop indexes. In that

case, team threads rank 0 through 3 call the loop objects' `run()` methods as described above, and team threads rank 4 and higher merely proceed directly to the barrier without calling the `run()` method.

What if we do a parallel for loop with more loop indexes than there are cores? In that case, each team thread will call its loop object's `run()` method *more than once.* We'll study how that works in the next chapter.

Points to Remember

- Write a Parallel Java 2 program as a subclass of class Task.
- Put the program code in the task's `main()` method.
- The task's `main()` method must be an instance method, not a static method.
- You can parallelize a loop if there are no sequential dependencies among the loop iterations.
- Use the `parallelFor()` pattern to parallelize a for loop.
- The parallel for loop index goes from the lower bound to the upper bound *inclusive.*
- Put the loop body code in the `run()` method of the inner Loop subclass.
- To terminate the program, don't call `System.exit()`; instead, throw an exception.
- To emit a printout before the end of the program, call `System.out.flush()` or `System.err.flush()`.
- In a non-parallel program, override the static `coresRequired()` method to return the number of cores the program will use, namely 1.
- Use the "`java pj2`" command to run your Parallel Java 2 program.
- Use the "`debug=makespan`" option to measure the program's running time.

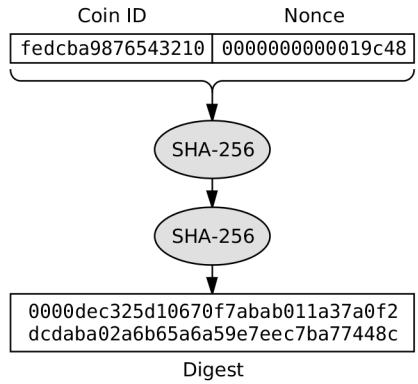
Chapter 3
Parallel Loop Schedules

Bitcoin (bitcoin.org) is an open-source, peer-to-peer, digital currency system. Bitcoins are created by "mining," reminiscent of how the 1849 Gold Rush prospectors panned for gold. Our next program example implements a simplified version of bitcoin mining. You specify a *coin ID*, which is just a string of hexadecimal digits, along with a number *N*. The program then finds a 64-bit *nonce* such that the *digest* has *N* leading zero bits. The digest is computed by concatenating the coin ID and the nonce, applying the SHA-256 cryptographic hash function to that, and applying SHA-256 to the result:

$$\text{digest} = \text{SHA-256}(\text{SHA-256}(\text{coin ID} \parallel \text{nonce}))$$

The digest is a 256-bit number. An example will clarify what the program does:

```
$ java pj2 edu.rit.pj2.example.MineCoinSeq fedcba9876543210 16
Coin ID = fedcba9876543210
Nonce   = 0000000000019c48
Digest  = 0000dec325d10670f7abab011a37a0f2dcdaba02a6b65a6a59e7
eec7ba77448c
```



For the coin ID `fedcba9876543210`, the program found that the nonce `19c48` yielded a digest with 16 leading zero bits (4 leading zero hexadecimal digits). How did it find this nonce? By brute force search. The program simply started with nonce 0, computed the digest, incremented the nonce, and repeated until the digest had *N* leading zero bits. Like a gold miner, the program had to pan through 105,544 worthless nonces until it hit that one golden nonce.

How many nonces must the program examine in general before finding the golden nonce? The output of a cryptographic hash function looks like a random number; each bit is a zero or a one with probability 1/2, independent of all the other bits. Therefore, the probability that the *N* leading bits are all zeroes is $1/2^N$, and we expect to examine 2^N nonces on the average before we find the golden nonce. Even for small *N* values like 24 or 32, that's a lot of nonces. A parallel program should let us find the golden nonce more quickly.

```
1  package edu.rit.pj2.example;
2  import edu.rit.crypto.SHA256;
3  import edu.rit.pj2.Task;
4  import edu.rit.util.Hex;
5  import edu.rit.util.Packing;
6  public class MineCoinSeq
7     extends Task
8     {
9     // Command line arguments.
10    byte[] coinId;
11    int N;
12
13    // Mask for leading zeroes.
14    long mask;
15
16    // For computing hash digests.
17    byte[] coinIdPlusNonce;
18    SHA256 sha256;
19    byte[] digest;
20
21    // Main program.
22    public void main
23       (String[] args)
24       throws Exception
25       {
26       // Validate command line arguments.
27       if (args.length != 2) usage();
28       coinId = Hex.toByteArray (args[0]);
29       N = Integer.parseInt (args[1]);
30       if (1 > N || N > 63) usage();
31
32       // Set up mask for leading zeroes.
33       mask = ~((1L << (64 - N)) - 1L);
34
35       // Set up for computing hash digests.
36       coinIdPlusNonce = new byte [coinId.length + 8];
37       System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
38          coinId.length);
39       sha256 = new SHA256();
40       digest = new byte [sha256.digestSize()];
41
42       // Try all nonces until the digest has N leading zero bits.
43       for (long nonce = 0L; nonce <= 0x7FFFFFFFFFFFFFFFL; ++ nonce)
44          {
45          // Test nonce.
46          Packing.unpackLongBigEndian (nonce, coinIdPlusNonce,
47             coinId.length);
48          sha256.hash (coinIdPlusNonce);
49          sha256.digest (digest);
50          sha256.hash (digest);
51          sha256.digest (digest);
52          if ((Packing.packLongBigEndian (digest, 0) & mask) == 0L)
53             {
54             // Print results.
55             System.out.printf ("Coin ID = %s%n",
56                Hex.toString (coinId));
57             System.out.printf ("Nonce   = %s%n",
58                Hex.toString (nonce));
```

Listing 3.1. MineCoinSeq.java (part 1)

Program MineCoinSeq (Listing 3.1) is a sequential program that solves the preceding problem. It begins by declaring several global variables on lines 9–19. These are actually fields of the Task subclass, so they are in scope for all of the task's methods as well as any inner classes. Lines 27–30 extract the coin ID and *N* from the command line. Line 33 computes a mask with the *N* most significant bits set to 1 and the other bits set to 0; the bits that are 1 in the mask correspond to the bit positions that must be 0 in the digest. Lines 36–38 set up a byte array that will be input to the hash function, with the coin ID at the beginning plus eight bytes at the end to hold the nonce. Line 39 creates an object to compute the SHA-256 hash function, using class edu.rit.crypto.SHA256. Line 40 sets up a byte array to hold the digest.

Lines 43–63 are the heart of the computation. The loop iterates over all nonces from 0 to $2^{63} - 1$. The loop body unpacks the 64-bit nonce into the final eight bytes of the byte array, computes the digest of the coin ID and nonce, extracts the first eight bytes of the digest as a `long`, and bitwise-ands that with the mask. If the result is zero, then the *N* most significant bits of the digest are all 0s. (The *N* most significant bits of the digest were anded with 1s in the mask, so they stayed intact. It doesn't matter what the other bits of the digest are; they were anded with 0s in the mask, so they became 0s.) When the program finds the golden nonce, it prints the results and breaks out of the loop without testing any further nonces—an *early loop exit.*

Here's a run of the sequential program on a `tardis` node, with a random coin ID and a larger *N:*

```
$ java pj2 debug=makespan edu.rit.pj2.example.MineCoinSeq \
  b3e5da601135706f 24
Coin ID = b3e5da601135706f
Nonce   = 0000000000f6fa72
Digest  = 000000f4a186e571af5e0910fc48a0ed87078f7337016d4d406cf
c3a05b328e8
Job 3 makespan 33698 msec
```

Now let's go parallel with program MineCoinSmp (Listing 3.2). As I did with the primality testing program in Chapter 2, I'll change the for loop to a parallel for loop. But notice that the loop body code refers to the global variables `coinIdPlusNonce`, `sha256`, and `digest`. In program MineCoinSmp, there will be multiple threads executing copies of the loop body concurrently. If the threads all manipulate the same global variables, the threads will interfere with each other; when one thread changes the state of a shared global variable, that will wipe out another thread's changes, and the program will not find the correct answer. To avoid this, I want each thread to have *its own separate copy* of each of these variables. These are called *thread-local variables.* However, the other variables—`coinId`, `N`, and `mask`—can remain shared global variables, because the threads will only read them, not change them. These are called *shared global WORM variables*—variables that are

```
59             System.out.printf ("Digest  = %s%n",
60                Hex.toString (digest));
61             break;
62             }
63          }
64       }
65
66    // Print a usage message and exit.
67    private static void usage()
68       {
69       System.err.println ("Usage: java pj2 " +
70          "edu.rit.pj2.example.MineCoinSeq <coinid> <N>");
71       System.err.println ("<coinid> = Coin ID (hexadecimal)");
72       System.err.println ("<N> = Number of leading zero bits " +
73          "(1 .. 63)");
74       throw new IllegalArgumentException();
75       }
76
77    // Specify that this task requires one core.
78    protected static int coresRequired()
79       {
80       return 1;
81       }
82    }
```

Listing 3.1. MineCoinSeq.java (part 2)

```
1  package edu.rit.pj2.example;
2  import edu.rit.crypto.SHA256;
3  import edu.rit.pj2.LongLoop;
4  import edu.rit.pj2.Schedule;
5  import edu.rit.pj2.Task;
6  import edu.rit.util.Hex;
7  import edu.rit.util.Packing;
8  public class MineCoinSmp
9     extends Task
10    {
11    // Command line arguments.
12    byte[] coinId;
13    int N;
14
15    // Mask for leading zeroes.
16    long mask;
17
18    // Main program.
19    public void main
20       (String[] args)
21       throws Exception
22       {
23       // Validate command line arguments.
24       if (args.length != 2) usage();
25       coinId = Hex.toByteArray (args[0]);
26       N = Integer.parseInt (args[1]);
27       if (1 > N || N > 63) usage();
28
29       // Set up mask for leading zeroes.
30       mask = ~((1L << (64 - N)) - 1L);
```

Listing 3.2. MineCoinSmp.java (part 1)

shared by all the threads, Written Once, and Read Many times.

Program MineCoinSmp starts out the same as program MineCoinSeq, with declarations for the shared global variables `coinId`, `N`, and `mask`. The main program code is the same, up to the loop at line 32, which becomes a parallel for loop. There's a call to the parallel for loop's `schedule()` method, which I'll gloss over for now. The loop body is in the inner LongLoop subclass. (The loop body uses class LongLoop, which has a loop index of type `long`, rather than class Loop, which has a loop index of type `int`.) This time, I declare the `coinIdPlusNonce`, `sha256`, and `digest` variables as fields of the inner LongLoop subclass, which makes them thread-local variables. Why? Recall that each thread in the parallel for loop's team works with its own separate copy of the loop body object; thus, each team thread ends up with its own separate copies of the fields (thread-local variables).

The thread-local variables are *not* initialized as part of their declarations. Rather, the thread-local variables *must be initialized in the loop body's `start()` method* (lines 40–48). Why? Because when each team thread clones the loop body object, the `clone()` method creates a copy of the loop body object itself, but then *assigns* the fields of the original to the fields of the copy. At this point, every team thread's loop body's fields refer to the same objects as the original loop body's fields. To ensure that each team thread gets its own separate copies of the thread-local variables, the fields must be re-initialized *after* cloning the loop body object. That is the purpose of the `start()` method. After cloning the loop body object, each team thread calls the `start()` method, once only, to initialize its own loop body object's fields. Each team thread then proceeds to call the `run()` method to perform the loop iterations.

The code for one loop iteration is in the loop body object's `run()` method, with the `nonce` loop index passed in as an argument. But this time the variables `coinIdPlusNonce`, `sha256`, and `digest` are thread-local variables. Each team thread can read and update these variables without needing to synchronize with the other team threads, because each team thread has its own copies.

After one team thread finds the golden nonce and prints its results, I want to do an early loop exit and terminate the program. But I can't do a `break` statement as in the sequential program, because there is no `for` statement in the parallel program. Instead, I call the `stop()` method (line 69). This tells the parallel for loop to stop iterating *in all the team threads*. After finishing its current iteration, each team thread immediately proceeds to the end-of-loop barrier, and then the program goes on to execute the code after the parallel for loop.

Let's go back to the `schedule()` method call on line 33 and explain what it does. Every parallel for loop has a *schedule* that determines the loop indexes that each team thread will perform. In other words, the schedule deter-

```
31          // Try all nonces until the digest has N leading zero bits.
32          parallelFor (0L, 0x7FFFFFFFFFFFFFFFL)
33             .schedule (leapfrog) .exec (new LongLoop()
34             {
35             // For computing hash digests.
36             byte[] coinIdPlusNonce;
37             SHA256 sha256;
38             byte[] digest;
39
40             public void start() throws Exception
41                {
42                // Set up for computing hash digests.
43                coinIdPlusNonce = new byte [coinId.length + 8];
44                System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
45                   coinId.length);
46                sha256 = new SHA256();
47                digest = new byte [sha256.digestSize()];
48                }
49
50             public void run (long nonce)
51                {
52                // Test nonce.
53                Packing.unpackLongBigEndian (nonce, coinIdPlusNonce,
54                   coinId.length);
55                sha256.hash (coinIdPlusNonce);
56                sha256.digest (digest);
57                sha256.hash (digest);
58                sha256.digest (digest);
59                if ((Packing.packLongBigEndian (digest, 0) & mask)
60                      == 0L)
61                   {
62                   // Print results.
63                   System.out.printf ("Coin ID = %s%n",
64                      Hex.toString (coinId));
65                   System.out.printf ("Nonce   = %s%n",
66                      Hex.toString (nonce));
67                   System.out.printf ("Digest  = %s%n",
68                      Hex.toString (digest));
69                   stop();
70                   }
71                }
72             });
73
74       // Print a usage message and exit.
75       private static void usage()
76          {
77          System.err.println ("Usage: java pj2 "+
78             "edu.rit.pj2.example.MineCoinSmp <coinid> <N>");
79          System.err.println ("<coinid> = Coin ID (hexadecimal)");
80          System.err.println ("<N> = Number of leading zero bits " +
81             "(1 .. 63)");
82          throw new IllegalArgumentException();
83          }
84       }
```

Listing 3.2. MineCoinSmp.java (part 2)

mines how the work of the for loop is divided among the team threads.

If you don't specify a schedule for a parallel for loop, the default is to use a *fixed schedule*. A fixed schedule divides the loop index range into *K* subranges, where *K* is the number of team threads. Each subrange is the same size, except possibly the last. (If the loop index range is not evenly divisible by *K*, the last subrange will include only the leftover indexes.) For example, suppose the loop index goes from 1 to 100. Here's how a fixed schedule would divide the loop index range among different numbers of team threads:

- $K = 1$
  - Team thread rank 0 does indexes 1, 2, 3, . . . , 100 (100 indexes)
- $K = 2$
  - Team thread rank 0 does indexes 1, 2, 3, . . . , 50 (50 indexes)
  - Team thread rank 1 does indexes 51, 52, 53, . . . , 100 (50 indexes)
- $K = 3$
  - Team thread rank 0 does indexes 1, 2, 3, . . . , 34 (34 indexes)
  - Team thread rank 1 does indexes 35, 36, . . . , 68 (34 indexes)
  - Team thread rank 2 does indexes 69, 70, 71, . . . , 100 (32 indexes)
- $K = 4$
  - Team thread rank 0 does indexes 1, 2, 3, . . . , 25 (25 indexes)
  - Team thread rank 1 does indexes 26, 27, 28, . . . , 50 (25 indexes)
  - Team thread rank 2 does indexes 51, 52, 53, . . . , 75 (25 indexes)
  - Team thread rank 3 does indexes 76, 77, 78, . . . , 100 (25 indexes)

Thus, with a fixed schedule, the work of the parallel for loop is divided as equally as possible among the team threads, and each team thread does a contiguous range of loop indexes.

Program MineCoinSmp's parallel for loop goes through 2^{63} indexes (disregarding the early loop exit). If I were to use the default fixed schedule, and if I were to run the program on a `tardis` node with four cores, look at the indexes each team thread would do (in hexadecimal):

- Rank 0 does 0000000000000000 – 1FFFFFFFFFFFFFFF
- Rank 1 does 2000000000000000 – 3FFFFFFFFFFFFFFF
- Rank 2 does 4000000000000000 – 5FFFFFFFFFFFFFFF
- Rank 3 does 6000000000000000 – 7FFFFFFFFFFFFFFF

This is *not* what I want. In effect, each team thread would be doing a separate golden nonce search within a different index range, so I would expect each team thread to do an average of 2^N iterations before finding a golden nonce, so the parallel program would take the same amount of time as the sequential program. I would not experience any speedup!

Instead, I want the team threads to do loop indexes this way:

- Rank 0 does 0, 4, 8, 12, . . .
- Rank 1 does 1, 5, 9, 13, . . .
- Rank 2 does 2, 6, 10, 14, . . .
- Rank 3 does 3, 7, 11, 15, . . .

Now all the team threads are searching the same index range starting at 0, and as soon as any team thread finds the first golden nonce, the program terminates. The work of searching this one index range is divided equally among the team threads, so ideally the parallel program should yield a speedup factor of 4.

This parallel for loop schedule is called a *leapfrog schedule.* The loop indexes are allocated in a round robin fashion among the *K* team threads. Each team thread does a series of *noncontiguous* loop indexes, with each index equaling the previous index plus *K.*

To get a leapfrog schedule in the MineCoinSmp program, I called the parallel for loop's `schedule()` method on line 33, passing `leapfrog` as the argument, before actually executing the loop. The possible schedules, listed in enum edu.rit.pj2.Schedule, are `fixed` (the default), `leapfrog`, `dynamic`, `proportional`, and `guided`. Program PrimeSmp in Chapter 2 uses a fixed schedule; program MineCoinSmp uses a leapfrog schedule; we will see examples of dynamic, proportional, and guided schedules in later chapters.

Here's a run of the parallel program on on a four-core `tardis` node, with the same arguments as the previous sequential program run.

```
$ java pj2 debug=makespan edu.rit.pj2.example.MineCoinSmp \
  b3e5da601135706f 24
Coin ID = b3e5da601135706f
Nonce   = 0000000000f6fa72
Digest  = 000000f4a186e571af5e0910fc48a0ed87078f7337016d4d406cf
c3a05b328e8
Job 4 makespan 8646 msec
```

The parallel program found the same golden nonce as the sequential program, but it took only 8.6 seconds instead of 33.7 seconds—a speedup factor of 33698 ÷ 8646 = 3.898.

Under the Hood

Figure 3.1 shows in more detail what happens when the MineCoinSmp program executes the parallel for loop on a parallel computer with four cores. (The `main()` method is not shown.) When the main thread calls the parallel for loop's `exec()` method, the main thread blocks and the team threads unblock. Each of the team threads creates its own copy of the loop object, calls the loop object's `start()` method which initializes the thread-local variables, and calls the loop object's `run()` method multiple times with the loop indexes dictated by the parallel for loop's schedule. When one of the team

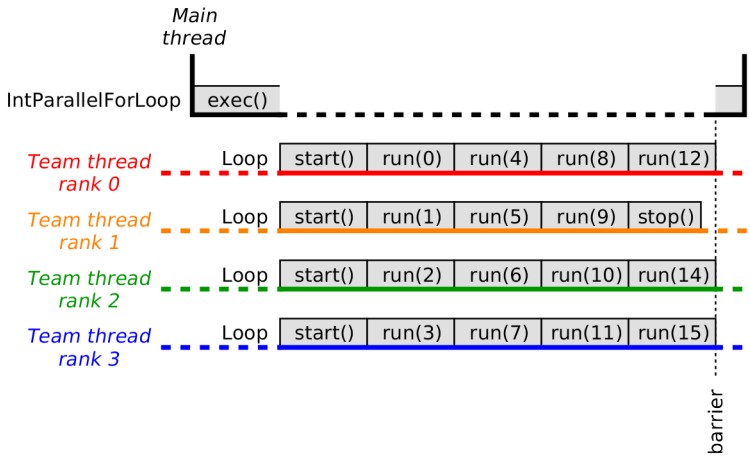


Figure 3.1. Parallel for loop execution flow, leapfrog schedule

threads doing one of the `run()` invocations—team thread rank 1 calling `run(9)` in this example—finds the golden nonce, that team thread calls the loop object's `stop()` method and proceeds to the barrier. After the loop has been stopped, as each team thread finishes its current invocation of the `run()` method, each team thread proceeds to the barrier without calling the `run()` method further. When all the team threads have arrived at the barrier, the team threads block, the main thread unblocks, and the main thread resumes executing the main program.

Points to Remember

- Declare variables as shared global WORM variables (fields of the Task subclass) if the parallel for loop team threads will read them but not alter them.
- Thread synchronization is not needed when reading shared global WORM variables.
- Declare variables as thread-local variables (fields of the loop body subclass) if the parallel for loop team threads will read and alter them.
- *Do not initialize thread-local variables as part of their declarations.* Initialize thread-local variables in the `start()` method.
- Thread synchronization is not needed when reading and writing thread-local variables.
- Decide which kind of schedule a parallel for loop needs, and specify it by calling the `schedule()` method if necessary.
- Use the `stop()` method to exit early from a parallel for loop.

Chapter 4
Parallel Reduction

Imagine a square dartboard (Figure 4.1) with sides of length 1 and with a quadrant of a circle drawn in it. Suppose we throw a large number N of darts at the dartboard, and suppose C of them land inside the circle quadrant. Assuming the darts land at random positions, the ratio C/N should be approximately the same as the ratio of the circle quadrant's area = $\pi/4$ to the dartboard's area = 1:

$$C/N \approx \pi/4, \text{ or } \pi \approx 4C/N.$$

This suggests an algorithm for approximating π: Generate N random (x, y) points in the unit square; count how many lie at a distance of 1 or less from the origin ($x^2 + y^2 \le 1$) yielding C; report $4C/N$ as the estimate for π. However, to get an accurate estimate, N needs to be very large, so we want to do the computation in parallel.

As shown in Figure 4.2, a sequential program simply throws N darts, counts C, and calculates π. On the other hand, a parallel program running on, say, four cores partitions the computation among four threads. Each thread throws $N/4$ darts and counts how many of its own darts land within the circle quadrant. The program then has to add these per-thread counts, C_0 through C_3, together to get the total count C, from which the program calculates π.

The process of combining multiple parallel threads' results into one overall result is called *reduction.* Here the *reduction operation* is addition, or sum, and we refer to the reduction as a *sum-reduce.* (Other programs would use other reduction operations as part of the same reduction pattern.)

Listing 4.1 is the sequential π program. It is mostly self-explanatory. On line 28 I created a *pseudorandom number generator (PRNG),* an instance of class edu.rit.util.Random, to generate the random (x, y) dart coordinates. The PRNG's *seed* (constructor argument) determines the sequence of random numbers the PRNG will produce. Two PRNGs initialized with the same seed will generate the same sequence; two PRNGs initialized with different seeds will generate different sequences. The user specifies the seed and the number of darts N on the command line.

I used the Parallel Java 2 Library class edu.rit.util.Random, rather than the standard Java class java.util.Random, for two reasons: my PRNG class is faster than Java's PRNG class; and my PRNG class has features useful for parallel programming that Java's PRNG class lacks (we will look at these features later).

Listing 4.2 is the parallel π program. It illustrates how to set up the reduction pattern. Instead of a single counter variable, I need a per-thread counter in each parallel team thread (C_0 through C_3 in Figure 4.2) plus a global counter variable to hold the result of the reduction (C in Figure 4.2). In order to do the reduction, the global counter variable can no longer be type `long`. Instead, it is an instance of class edu.rit.pj2.vbl.LongVbl (line 14); this class performs the reduction automatically. I initialized the global counter

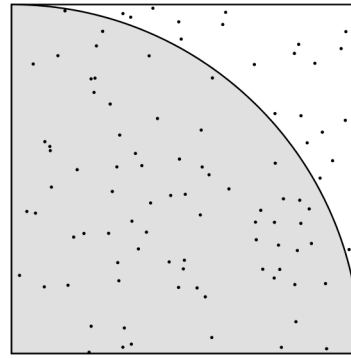


Figure 4.1. A dartboard for estimating π

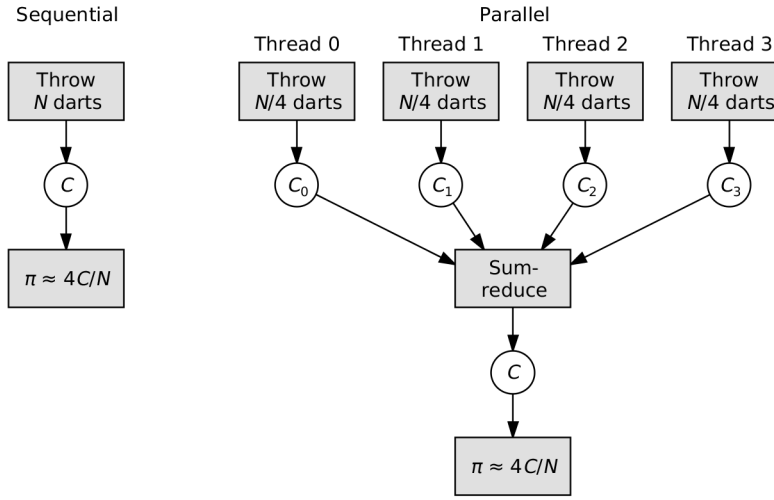


Figure 4.2. Estimating π sequentially and in parallel

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Task;
3  import edu.rit.util.Random;
4  public class PiSeq
5     extends Task
6     {
7     // Command line arguments.
8     long seed;
9     long N;
10
11    // Pseudorandom number generator.
12    Random prng;
```

Listing 4.1. PiSeq.java (part 1)

variable to an instance of subclass LongVbl.Sum (line 28); this subclass does a sum-reduce. (Other subclasses do other reduction operations, and you can define your own subclasses with your own reduction operations.) The per-thread counter variable is declared on line 32 and is initialized—as all per-thread variables must be—in the parallel loop's `start()` method on line 36. I obtained the per-thread object by calling the parallel loop's `threadLocal()` method, specifying the global variable. The `threadLocal()` method calls "link" the per-thread counter variables to the global counter variable and set everything up to do the reduction automatically under the hood, as I will describe shortly.

I also did away with the global PRNG variable and made it a per-thread variable, declared on line 31 and initialized on line 35. Why? Because if I left it as a global variable, all the parallel team threads would use the same PRNG object, and the threads would interfere with each other as they all accessed this object and altered its state. Making the PRNG a per-thread variable lets each thread work with its own separate PRNG object without interference from the other threads and without needing to synchronize with the other threads.

However, I had to change the way the per-thread PRNGs are seeded. Instead of initializing each one with just the seed, I initialized each one with the seed plus the thread's rank (as returned by the `rank()` method). Why? Because if I seeded all the PRNGs with the same value, all the PRNGs would produce the same sequence of random numbers, *and all the threads would generate the same (x, y) dart coordinates.* This is not what I want. I want each thread to generate *different* random dart coordinates. I get that by seeding the PRNGs with different values, making them produce different sequences of random numbers.

The parallel for loop starting at line 29 divides the *N* dart throws among the parallel team threads. I went with the default fixed schedule. Each thread increments the `item` field of its own `thrCount` per-thread variable. When the parallel loop finishes, before returning back to the main program, the contents of all the per-thread `thrCount` variables' `item` fields are automatically added together under the hood (because, remember, these variables are of type LongVbl.Sum, which does a sum-reduce), and the result is automatically stored in the global `count` variable's `item` field. The main program then uses `count.item` to estimate π.

Here are runs of the sequential program and the parallel program on a four-core `tardis` node with *N* = four billion darts:

```
$ java pj2 debug=makespan edu.rit.pj2.example.PiSeq 142857 \
  4000000000
pi = 4*3141556293/4000000000 = 3.141556293
Job 5 makespan 76051 msec
```

```
13
14      // Number of points within the unit circle.
15      long count;
16
17      // Main program.
18      public void main
19         (String[] args)
20         throws Exception
21         {
22         // Validate command line arguments.
23         if (args.length != 2) usage();
24         seed = Long.parseLong (args[0]);
25         N = Long.parseLong (args[1]);
26
27         // Set up PRNG.
28         prng = new Random (seed);
29
30         // Generate n random points in the unit square, count how many
31         // are in the unit circle.
32         count = 0;
33         for (long i = 0; i < N; ++ i)
34            {
35            double x = prng.nextDouble();
36            double y = prng.nextDouble();
37            if (x*x + y*y <= 1.0) ++ count;
38            }
39
40         // Print results.
41         System.out.printf ("pi = 4*%d/%d = %.9f%n",
42            count, N, 4.0*count/N);
43         }
44
45      // Print a usage message and exit.
46      private static void usage()
47         {
48         System.err.println ("Usage: java pj2 "+
49            "edu.rit.pj2.example.PiSeq <seed> <N>");
50         System.err.println ("<seed> = Random seed");
51         System.err.println ("<N> = Number of random points");
52         throw new IllegalArgumentException();
53         }
54
55      // Specify that this task requires one core.
56      protected static int coresRequired()
57         {
58         return 1;
59         }
60      }
```

Listing 4.1. PiSeq.java (part 2)

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.LongLoop;
3  import edu.rit.pj2.Task;
4  import edu.rit.pj2.vbl.LongVbl;
5  import edu.rit.util.Random;
```

Listing 4.2. PiSmp.java (part 1)

```
$ java debug=makespan pj2 edu.rit.pj2.example.PiSmp 142857 \
  4000000000
pi = 4*3141567239/4000000000 = 3.141567239
Job 6 makespan 20345 msec
```

The speedup was 76051 ÷ 20345 = 3.738.

Notice that, although each program's estimate for π was within 0.001 percent of the true value, the two programs computed different estimates. This is because the two programs generated different random dart coordinates. PiSeq's darts came from one random sequence of length four billion from a seed of 142857. PiSmp's darts came from four random sequences, each of length one billion, each from a different seed of 142857 through 142860. For the same reason, PiSmp will compute different answers when run with different numbers of cores (threads).

For the π estimating problem, it doesn't much matter if the sequential and parallel programs compute different answers. But for another problem involving random numbers, there might be a requirement for the parallel program to compute exactly the same answer, no matter how many cores it runs on. Such a program would have to generate the same random numbers regardless of the number of cores. This can be tricky when the random numbers are not all being generated by the same thread. We'll look at this in the next chapter.

Under the Hood

Figure 4.3 shows how the automatic sum-reduce of the per-thread `thrCount` variables into the shared global `count` variable happens under the hood when running with eight threads. The boxes at the top show the contents of the `thrCount` variables' `item` fields at the end of each parallel team thread's iterations.

The reduction proceeds in a series of *rounds.* Here's what each thread does in the first round:

- Thread 0 waits for thread 4 to finish its iterations, then thread 0 adds thread 4's `item` field to its own `item` field.
- Thread 1 waits for thread 5 to finish its iterations, then thread 1 adds thread 5's `item` field to its own `item` field.
- Thread 2 waits for thread 6 to finish its iterations, then thread 2 adds thread 6's `item` field to its own `item` field.
- Thread 3 waits for thread 7 to finish its iterations, then thread 3 adds thread 7's `item` field to its own `item` field.
- Threads 4 through 7 do nothing.

The above operations proceed concurrently. Notice that each thread only synchronizes with one other thread; this results in less overhead than if all the

```
6  public class PiSmp
7     extends Task
8     {
9     // Command line arguments.
10    long seed;
11    long N;
12
13    // Number of points within the unit circle.
14    LongVbl count;
15
16    // Main program.
17    public void main
18       (String[] args)
19       throws Exception
20       {
21       // Validate command line arguments.
22       if (args.length != 2) usage();
23       seed = Long.parseLong (args[0]);
24       N = Long.parseLong (args[1]);
25
26       // Generate n random points in the unit square, count how many
27       // are in the unit circle.
28       count = new LongVbl.Sum (0);
29       parallelFor (0, N - 1) .exec (new LongLoop()
30          {
31          Random prng;
32          LongVbl thrCount;
33          public void start()
34             {
35             prng = new Random (seed + rank());
36             thrCount = threadLocal (count);
37             }
38          public void run (long i)
39             {
40             double x = prng.nextDouble();
41             double y = prng.nextDouble();
42             if (x*x + y*y <= 1.0) ++ thrCount.item;
43             }
44          });
45
46       // Print results.
47       System.out.printf ("pi = 4*%d/%d = %.9f%n",
48          count.item, N, 4.0*count.item/N);
49       }
50
51    // Print a usage message and exit.
52    private static void usage()
53       {
54       System.err.println ("Usage: java pj2 "+
55          "edu.rit.pj2.example.PiSmp <seed> <N>");
56       System.err.println ("<seed> = Random seed");
57       System.err.println ("<N> = Number of random points");
58       throw new IllegalArgumentException();
59       }
60    }
```

Listing 4.2. PiSmp.java (part 2)

threads had to synchronize with each other.

The second round is similar to the first round, except only threads 0 and 1 synchronize with threads 2 and 3. The third round is similar, except only thread 0 synchronizes with thread 1. After all the rounds, thread 0's `item` field holds the sum of all the threads' original `item` fields. The sum is then stored in the global variable's `item` field. Once the reduction is complete, all the threads proceed to the end-of-loop barrier as usual.

This pattern is called a *parallel reduction tree.* The tree can be extended upwards to accommodate any number of threads. At each round of the reduction, half of the threads' `item` fields are combined in parallel with the other half of the threads' `item` fields. Thus, with K threads, the number of reduction rounds is $\log_2 K$. The reduction tree is efficient even with large numbers of threads.

The Parallel Java 2 Library has classes in package edu.rit.pj2.vbl for doing parallel reduction on primitive types—`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`. Each of these classes implements interface Vbl in package edu.rit.pj2, which specifies the interface for a reduction variable class. You can create reduction variables for arbitrary data types, such as objects and arrays, by defining your own classes that implement interface Vbl; we'll see an example in the next chapter.

In the PiSmp program, I declared the reduction variables to be type LongVbl, so their `item` fields were type `long`. I initialized the reduction variables with instances of class LongVbl.Sum, a subclass of LongVbl whose reduction operation is summation. I initialized the global `count` variable with a new instance of class LongVbl.Sum, and I initialized the per-thread `thrCount` variables by calling the parallel loop's `threadLocal()` method. Initializing the global and per-thread variables this way links the pre-thread variables to the global variable and causes the parallel reduction to happen automatically as the parallel team threads finish the parallel for loop.

Why use the parallel reduction pattern? Chiefly to minimize the amount of thread synchronization the parallel program has to do. Each thread can manipulate its own per-thread variables without needing to synchronize at all with the other threads. At the end of the parallel loop, only a few pairwise thread synchronizations have to happen in the parallel reduction tree. Minimizing the amount of thread synchronization, and combining the per-thread results together in parallel, reduces the parallel program's running time and improves its performance.

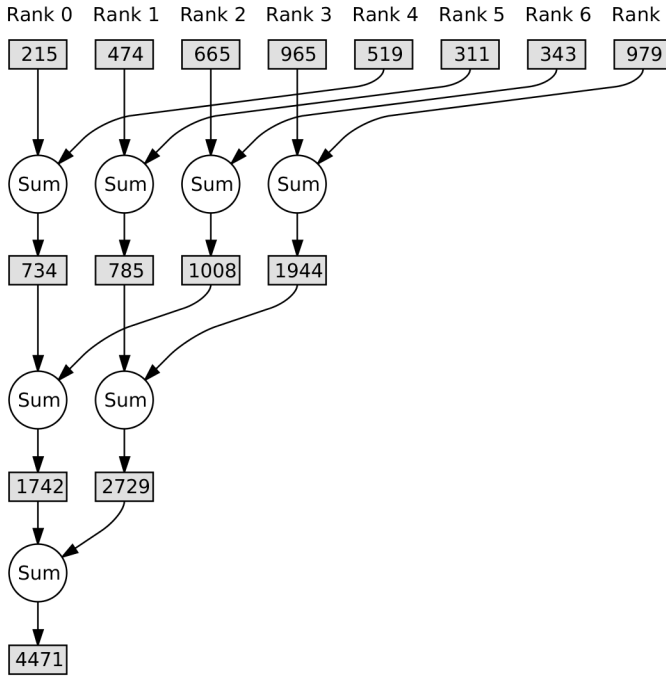


Figure 4.3. Sum-reduce parallel reduction tree

Points to Remember

- Use the parallel reduction pattern to combine per-thread results together into one global result.
- Make the reduction variables instances of classes that implement interface Vbl, ones having the desired data types and reduction operations.
- Initialize a parallel loop's per-thread reduction variables by calling the `threadLocal()` method.
- When generating random numbers in a parallel loop, declare the PRNG as a per-thread variable.
- Initialize each thread's per-thread PRNG with a different seed.

Chapter 5
Reduction Variables

The π estimating programs in Chapter 4 used class edu.rit.util.Random to generate random (x, y) points. Each call of the `nextDouble()` method produced a new random number in the range 0.0 (inclusive) through 1.0 (exclusive). But this is paradoxical. How can a completely *deterministic* algorithm (in the `nextDouble()` method) produce *random* numbers?

The answer is that, for all practical purposes, the sequence of numbers produced by class Random is *indistinguishable* from the sequence of random numbers produced by a "true" random source, like flipping coins or rolling dice. To emphasize that the numbers did not come from a true random source, we call them *pseudorandom* numbers, and we call the algorithm that produced them a *pseudorandom number generator* (PRNG).

But how can we tell if the output of a PRNG is indistinguishable from the output of a true random number generator? The generally accepted criterion is to use *statistical tests of randomness.* If the PRNG passes many such tests, then for all practical purposes we can treat the PRNG's output as being random. Several random number generator statistical test suites exist, such as Diehard* and TestU01.†

A statistical test of a sequence of allegedly random numbers goes like this: Formulate a *null hypothesis*, which is a statement about the probability distribution we expect the random numbers to obey. For example, the null hypothesis might be that the numbers are uniformly distributed between 0.0 and 1.0. Next, compute a *statistic* from the sequence of numbers. By convention, a large value of the statistic signals poor agreement between the numbers and the null hypothesis; a small statistic signals good agreement; and a statistic of 0.0 signals perfect agreement. Finally, compute the *p-value* of the statistic. The *p*-value is the probability that a value greater than or equal to the calculated statistic would be observed, even if the null hypothesis were true; that is, even if the numbers were in fact random. The smaller the *p*-value, the less likely that the null hypothesis is true; that is, the less likely that the numbers are random. If the *p*-value falls below a *significance* threshold, such as 0.05 or 0.01, then the null hypothesis is disproved (at that significance), and we conclude that the numbers are in fact not random.

One widely-used statistical test is the *chi-square test.* Suppose I am dealing with numbers x in the range $0.0 \le x < 1.0$, and suppose the null hypothesis is that the numbers are uniformly distributed; this is what I expect from class Random's `nextDouble()` method. I divide the interval from 0.0 to 1.0 into some number of equally-sized *bins;* say, ten bins. Bin 0 corresponds to the subinterval $0.0 \le x < 0.1$, bin 0 to subinterval $0.1 \le x < 0.2$, . . . bin 9 to subinterval $0.9 \le x < 1.0$. Each bin has an associated counter. Now I do a bunch of *trials*, say N of them. For each trial, I generate a random number,

* http://www.stat.fsu.edu/pub/diehard/

† P. L'Ecuyer and R. Simard. TestU01: a C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software,* 33(4):22, 2007.

and I increment the proper bin's counter depending on the subinterval in which the random number falls.

When I'm done with the trials, I compute the chi-square statistic, χ^2, from the bin counters. If the null hypothesis is true, that the numbers are uniformly distributed, then ideally the count in each bin should be the same, namely N divided by the number of bins. The formula for χ^2 in this case is

$$\chi^2 = \sum_{0}^{B-1} \frac{(n_i - N/B)^2}{N/B} \tag{5.1}$$

where B is the number of bins, N is the number of trials, and n_i is the count in bin i. Finally, I compute the p-value of χ^2. (I'll discuss how later.)

If the count in every bin is exactly equal to the expected value N/B, then χ^2 is 0.0 and the p-value is 1.0. As the bin counts start to deviate from the expected value, either up or down, χ^2 increases and the p-value decreases. If the bin counts get too far from the expected value, χ^2 gets too large, the p-value gets too small and falls below the significance threshold, and the statistical test fails.

Here are examples of a six-bin chi-square test with 6000 trials on two different PRNGs. The bin counts and the χ^2 statistics are

Bin	*PRNG 1 Count*	*PRNG 2 Count*
0	976	955
1	971	950
2	989	968
3	989	968
4	1039	1018
5	1036	1141
χ^2	4.4760	26.778

Here is a plot of the p-value versus χ^2 for a six-bin chi-square test:

PRNG 1 passes the test at a significance of 0.01; for $\chi^2 = 4.4760$, the p-value is above the significance threshold. PRNG 2 fails the test; for $\chi^2 = 26.778$, the p-value is below the significance threshold.

I want to write computer programs that carry out the chi-square test on the PRNG in class edu.rit.util.Random, both a sequential version and a parallel version. The programs' designs are similar to those of the π estimating programs in Chapter 4: generate a bunch of pseudorandom numbers, count things, compute the answer from the counts. But this time, instead of a single counter, I'm dealing with multiple counters, one for each bin. Following the principles of object oriented design, I'll encapsulate the bin counters inside an object; this object is called a *histogram.*

The sequential statistical test program will generate N random numbers, accumulate them into a histogram, and compute χ^2 (Figure 5.1). The parallel statistical test program will use the parallel reduction pattern, just like the parallel π estimating program. The parallel program will partition the N trials among the K parallel team threads. Each thread will do N/K trials and will accumulate its random numbers into its own per-thread histogram.

Before computing χ^2, the per-thread histograms have to be reduced together into one overall histogram. Here is an example of a six-bin chi-square test with 6000 trials, where the trials were done in parallel by four threads (1500 trials in each thread). The per-thread histograms, and the overall histogram after the reduction, are

Bin	*Thread 0 Histogram*	*Thread 1 Histogram*	*Thread 2 Histogram*	*Thread 3 Histogram*	*Overall Histogram*
0	244	249	227	256	976
1	239	241	248	243	971
2	248	259	239	243	989
3	261	241	260	227	989
4	245	260	259	275	1039
5	263	250	267	256	1036
				χ^2	4.4760

Each bin count in the overall histogram is the sum of the corresponding bin counts in the per-thread histograms. Thus, the reduction operation that combines two histograms together is to add each bin in one histogram to its counterpart in the other histogram.

I want the parallel loop classes in the Parallel Java 2 Library to do the histogram reduction automatically, just as they did with the LongVbl reduction variable in the parallel π estimating program. This means that the histogram class must be suitable for use as a reduction variable. Specifically, the histogram reduction variable class must implement interface edu.rit.pj2.Vbl and must provide implementations for the methods declared in that interface.

The Parallel Java 2 Library provides two classes, one for a histogram, the other for a histogram reduction variable. The classes are separate so that a program that needs a histogram, but not a histogram reduction variable, can just use the former class.

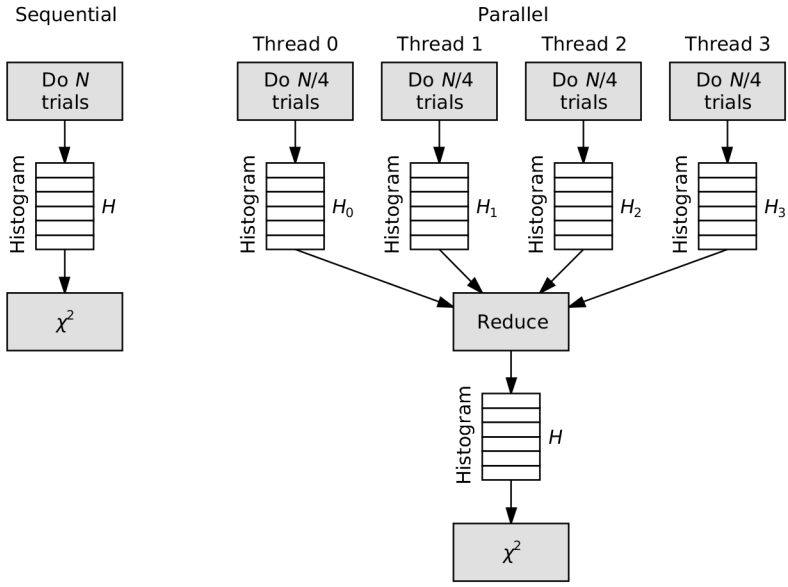


Figure 5.1. Chi-square test done sequentially and in parallel

```
1  package edu.rit.numeric;
2  public class Histogram
3     implements Cloneable
4     {
5     private int B;        // Number of bins
6     private long[] count; // Count in each bin
7     private long total;   // Total count in all bins
8
9     // Construct a new histogram with the given number of bins.
10    public Histogram
11       (int B)
12       {
13       if (B < 2)
14          throw new IllegalArgumentException (String.format
15             ("Histogram(): B = %d illegal", B));
16       this.B = B;
17       this.count = new long [B];
18       this.total = 0;
19       }
20
21    // Make this histogram be a deep copy of the given histogram.
22    public Histogram copy
23       (Histogram hist)
24       {
25       this.B = hist.B;
26       this.count = hist.count == null ? null :
27          (long[]) hist.count.clone();
28       this.total = hist.total;
29       return this;
30       }
```

Listing 5.1. Histogram.java (part 1)

Listing 5.1 is the code for the histogram, class edu.rit.numeric.Histogram. (To save space, I omitted some of the methods that are not used by the statistical test program.) The code is mostly self-explanatory. There are three hidden fields: the number of bins, an array of `long` bin counters, and the total of all the bin counters (lines 5–7). There is a constructor for a given number of bins (line 10). There is a method to copy one histogram into another (line 22). This method makes a *deep copy* by setting this histogram's bin counter array to a new array whose contents are copied from the other histogram's bin counter array (lines 26–27); afterwards, changes to one histogram will not affect the other. There are methods to clone a histogram (line 33), to get the number of histogram bins (line 48), to accumulate a `double` value into the proper bin (line 54), to do the actual bin update (line 61), to return a given bin counter (line 69), to compute χ^2 (line 90), and to compute the *p*-value of χ^2 (line 104) by calling a method in class edu.rit.numeric.Statistics. There is also a method to add each bin counter in one histogram to the corresponding bin counter in another histogram (line 111); this method will be used by the histogram reduction variable class.

Now that I have the Histogram class, I can write a program to do the actual chi-square test. Class edu.rit.pj2.example.StatTestSeq (Listing 5.2) is the sequential version. The command line arguments are the seed for the PRNG, the number of histogram bins *B*, and the number of trials *N*. The code is mostly straightforward. It sets up a PRNG object and a histogram object; performs *N* trials in a loop, where each trial generates the next random number from the PRNG and accumulates the random number into the histogram; and finally prints the χ^2 statistic and the *p*-value. The sequential version does not do a reduction and so does not use the histogram reduction variable class.

The only thing not quite straightforward in the StatTestSeq program is the way the histogram variable is initialized (lines 34–40). I designed class Histogram with flexibility in mind. One aspect of a histogram that could change from program to program is the manner in which data values are mapped to histogram bins. For data values of type `double`, the mapping must be specified by overriding the `accumulate()` method. The StatTestSeq program does this by assigning an instance of a subclass of class Histogram to the `hist` variable. The subclass is specified using Java's anonymous inner class syntax, like the body of a parallel for loop. When the subclass's `accumulate()` method is called, the argument is the output of the PRNG's `nextDouble()` method, namely a value in the range 0.0 (inclusive) through 1.0 (exclusive). This is multiplied by the number of histogram bins (`size()`) and truncated to an integer, yielding a value in the range 0 (inclusive) through *B* (exclusive); that is, an integer bin index in the range 0 through $B - 1$, where *B* is the number of bins. Calling the protected `increment()` method then increments the count in that bin.

```
31    // Create a clone of this histogram.
32    public Object clone()
33       {
34       try
35          {
36          Histogram hist = (Histogram) super.clone();
37          hist.copy (this);
38          return hist;
39          }
40       catch (CloneNotSupportedException exc)
41          {
42          throw new RuntimeException ("Shouldn't happen", exc);
43          }
44       }
45
46    // Returns the number of bins in this histogram.
47    public int size()
48       {
49       return B;
50       }
51
52    // Accumulate the given value of type double into this histogram.
53    public void accumulate
54       (double x)
55       {
56       throw new UnsupportedOperationException();
57       }
58
59    // Increment the given bin in this histogram.
60    protected void increment
61       (int i)
62       {
63       ++ count[i];
64       ++ total;
65       }
66
67    // Returns the count in the given bin of this histogram.
68    public long count
69       (int i)
70       {
71       return count[i];
72       }
73
74    // Determine the expected count in the given bin for a chi-square
75    // test.
76    public double expectedCount
77       (int i)
78       {
79       return expectedProb(i)*total;
80       }
81
82    // Determine the expected probability of incrementing the given
83    // bin for a chi-square test.
84    public double expectedProb
85       (int i)
86       {
87       return 1.0/B;
88       }
```

Listing 5.1. Histogram.java (part 2)

Turning to the parallel statistical test program, I need a reduction variable class for histograms. This is class edu.rit.pj2.vbl.HistogramVbl in the Parallel Java 2 Library (Listing 5.3). (Again, I have omitted some of the methods to save space.) The class implements interface Vbl (line 5). The class "wraps" an instance of class Histogram, which is specified as the constructor argument (line 12) and is stored in the public `hist` field (line 8). The remaining methods are the ones declared in interface Vbl, that make the HistogramVbl class suitable for use as a reduction variable. *These methods must be implemented as described below,* otherwise parallel reduction will not work.

- The reduction variable's `clone()` method (line 19) must create a new reduction variable object that is a *deep copy* of this reduction variable object (the one being cloned). Class HistogramVbl's `clone()` method does so by invoking the superclass's `clone()` method (line 21). This creates a *shallow copy* of the reduction variable. To make the new reduction variable be a deep copy of the original reduction variable, the histogram in the `hist` field is itself cloned, creating a deep copy of the original histogram (see Listing 5.1 lines 32–44); the deep copy then replaces the original histogram (lines 22–23). This follows the standard pattern for cloning an object in Java.
- The reduction variable's `set()` method (line 28) must change this reduction variable object to be a deep copy of the given reduction variable object. Class HistogramVbl's `set()` method does so by invoking the histogram's `copy()` method (see Listing 5.1 lines 22–30) to copy the given reduction variable's histogram into this reduction variable's histogram.
- The reduction variable's `reduce()` method (line 35) performs the reduction operation. It must apply the reduction operation to this reduction variable object and the given reduction variable object, and store the result back into this reduction variable object. Class HistogramVbl's `reduce()` method accomplishes this by calling the `add()` method on this reduction variable's histogram, passing in the other reduction variable's histogram. The `add()` method then does the work (see Listing 5.1 lines 111–121).

I want the parallel version of the program to compute exactly the same histogram as the sequential version, no matter how many cores (threads) the program runs on. Strictly speaking, the program doesn't need to do this, but I want to illustrate how to code it.

As in the parallel π estimating program, each thread in the parallel chi-square program will have its own per-thread PRNG. Unlike the parallel π estimating program which initialized each per-thread PRNG with a different seed, in the parallel chi-square program I will initialize each per-thread PRNG with the *same* seed. Doing so would normally cause each per-thread PRNG to generate the same sequence of random numbers. But this time, af-

```
89     // Returns the chi-square statistic for this histogram.
90     public double chisqr()
91        {
92        double chisqr = 0.0;
93        for (int i = 0; i < B; ++ i)
94           {
95           double expected = expectedCount (i);
96           double d = expected - count[i];
97           chisqr += d*d/expected;
98           }
99        return chisqr;
100       }
101
102    // Returns the p-value of the given chi-square statistic for this
103    // histogram.
104    public double pvalue
105       (double chisqr)
106       {
107       return Statistics.chiSquarePvalue (B - 1, chisqr);
108       }
109
110    // Add the given histogram to this histogram.
111    public void add
112       (Histogram hist)
113       {
114       if (hist.B != this.B)
115          throw new IllegalArgumentException
116             ("Histogram.add(): Histograms are different sizes");
117       for (int i = 0; i < B; ++ i)
118          this.count[i] += hist.count[i];
119       this.total += hist.total;
120       }
121    }
```

Listing 5.1. Histogram.java (part 3)

```
1  package edu.rit.pj2.example;
2  import edu.rit.numeric.Histogram;
3  import edu.rit.pj2.Task;
4  import edu.rit.util.Random;
5  public class StatTestSeq
6     extends Task
7     {
8     // Command line arguments.
9     long seed;
10    int B;
11    long N;
12
13    // Pseudorandom number generator.
14    Random prng;
15
16    // Histogram of random numbers.
17    Histogram hist;
18
```

Listing 5.2. StatTestSeq.java (part 1)

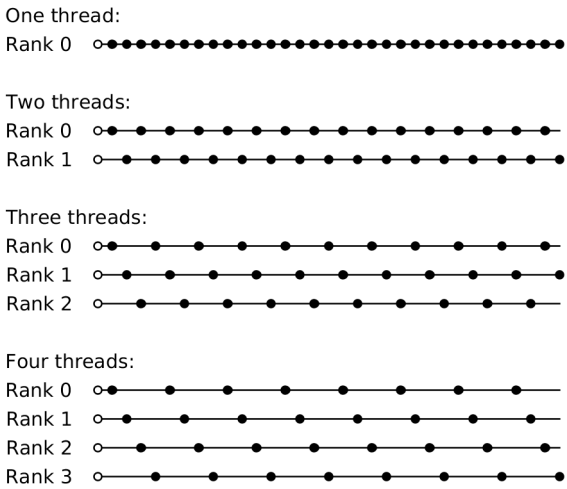


Figure 5.2. Generating random numbers in a round robin fashion

ter initializing the PRNG, parallel team thread 1 will *skip over* one random number, namely the random number generated by thread 0; thread 2 will *skip over* two random numbers, namely those generated by threads 0 and 1; thread 3 will *skip over* three random numbers, namely those generated by threads 0, 1, and 2; and so on. Later, after generating each random number, each thread will skip over $K - 1$ random numbers, namely those generated by the other $K - 1$ threads in the parallel team of K threads. In this way, the threads in the parallel program will generate the same random numbers as the single thread in the sequential program, except that the random numbers will be generated in a round robin fashion among all the threads.

Figure 5.2 shows how the parallel program generates the same sequence of random numbers, no matter how many threads are running. The lines stand for the per-thread PRNGs, the white circles stand for the initial seed values (the same in all the threads), and the black circles stand for the random numbers generated by each thread.

Class edu.rit.pj2.example.StatTestSmp (Listing 5.4) is the multicore parallel chi-square test program. It starts by declaring the `histvbl` variable (line 16); this is the global reduction variable. In the `main()` method, the loop over the trials has become a parallel loop (line 39). Inside each thread's parallel loop object are the per-thread `thrHistvbl` variable, the per-thread PRNG, and a variable `leap` (lines 41–43). The parallel loop's `start()` method links the per-thread histogram reduction variable to the global histogram reduction variable, so that the automatic parallel reduction will take place (line 46). The

```
19    // Main program.
20    public void main
21       (String[] args)
22       throws Exception
23       {
24       // Validate command line arguments.
25       if (args.length != 3) usage();
26       seed = Long.parseLong (args[0]);
27       B = Integer.parseInt (args[1]);
28       N = Long.parseLong (args[2]);
29
30       // Set up PRNG.
31       prng = new Random (seed);
32
33       // Set up histogram.
34       hist = new Histogram (B)
35          {
36          public void accumulate (double x)
37             {
38             increment ((int)(x*size()));
39             }
40          };
41
42       // Do N trials.
43       for (long i = 0; i < N; ++ i)
44          hist.accumulate (prng.nextDouble());
45
46       // Print results.
47       System.out.printf ("Bin\tCount%n");
48       for (int i = 0; i < B; ++ i)
49          System.out.printf ("%d\t%d%n", i, hist.count (i));
50       double chisqr = hist.chisqr();
51       System.out.printf ("Chisqr = %.5g%n", chisqr);
52       System.out.printf ("Pvalue = %.5g%n", hist.pvalue (chisqr));
53       }
54
55    // Print a usage message and exit.
56    private static void usage()
57       {
58       System.err.println ("Usage: java pj2 " +
59          "edu.rit.pj2.example.StatTestSeq <seed> <B> <N>");
60       System.err.println ("<seed> = Random seed");
61       System.err.println ("<B> = Number of histogram bins");
62       System.err.println ("<N> = Number of trials");
63       throw new IllegalArgumentException();
64       }
65
66    // Specify that this task requires one core.
67    protected static int coresRequired()
68       {
69       return 1;
70       }
71    }
```

Listing 5.2. StatTestSeq.java (part 2)

`start()` method initializes the per-thread PRNG with the same seed in each thread (line 47), then skips over some random numbers; the quantity skipped is the thread's rank in the parallel team (line 48). Thus, thread 0 skips over none; thread 1 skips over one; thread 2 skips over two; and so on. The `start()` method initializes `leap` to the number of team threads, minus one (line 49).

On each loop iteration, the parallel loop's `run()` method generates a random number from the per-thread PRNG, accumulates it into the per-thread histogram, and skips over `leap` (that is, $K - 1$) random numbers. To skip the PRNG ahead, I call the PRNG's `skip()` method (line 54). The `skip()` method is very fast; it advances the PRNG without bothering to generate the intervening random numbers. (The standard Java PRNG class has no such capability, which is one reason why I prefer to use my own class.)

Once the parallel loop iterations have finished, the per-thread histograms are automatically reduced into the global histogram under the hood. The reductions are performed using class HistogramVbl's `reduce()` method. The program then uses the global histogram to compute and print the χ^2 statistic and the p-value.

Here are runs of the sequential program and the parallel program on a four-core `tardis` node with B = ten bins and N = two billion trials:

```
$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSeq \
  142857 10 2000000000
Bin     Count
0       200007677
1       199996144
2       199981249
3       199998299
4       199998146
5       200010690
6       200019197
7       200003302
8       199983897
9       200001399
Chisqr = 5.9335
Pvalue = 0.74655
Job 160 makespan 48461 msec
$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSmp \
  142857 10 2000000000
Bin     Count
0       200007677
1       199996144
2       199981249
3       199998299
4       199998146
5       200010690
6       200019197
7       200003302
8       199983897
```

```
1  package edu.rit.pj2.vbl;
2  import edu.rit.pj2.Vbl;
3  import edu.rit.numeric.Histogram;
4  public class HistogramVbl
5     implements Vbl
6     {
7     // The histogram itself.
8     public Histogram hist;
9
10    // Construct a new histogram reduction variable wrapping the
11    // given histogram.
12    public HistogramVbl
13       (Histogram hist)
14       {
15       this.hist = hist;
16       }
17
18    // Create a clone of this shared variable.
19    public Object clone()
20       {
21       HistogramVbl vbl = (HistogramVbl) super.clone();
22       if (this.hist != null)
23          vbl.hist = (Histogram) this.hist.clone();
24       return vbl;
25       }
26
27    // Set this shared variable to the given shared variable.
28    public void set
29       (Vbl vbl)
30       {
31       this.hist.copy (((HistogramVbl)vbl).hist);
32       }
33
34    // Reduce the given shared variable into this shared variable.
35    public void reduce
36       (Vbl vbl)
37       {
38       this.hist.add (((HistogramVbl)vbl).hist);
39       }
40    }
```

Listing 5.3. HistogramVbl.java

```
1  package edu.rit.pj2.example;
2  import edu.rit.numeric.Histogram;
3  import edu.rit.pj2.LongLoop;
4  import edu.rit.pj2.Task;
5  import edu.rit.pj2.vbl.HistogramVbl;
6  import edu.rit.util.Random;
7  public class StatTestSmp
8     extends Task
9     {
10    // Command line arguments.
11    long seed;
12    int B;
13    long N;
14
```

Listing 5.4. StatTestSmp.java (part 1)

```
9       200001399
Chisqr = 5.9335
Pvalue = 0.74655
Job 165 makespan 13608 msec
```

The speedup was 48461 ÷ 13608 = 3.561.

Note that both programs did indeed compute exactly the same histogram. The *p*-value was 0.74655, so the PRNG passed the chi-square test at a significance of 0.01. (If it had failed the test, I'd be worried!)

Under the Hood

When dealing with a reduction variable, the Parallel Java 2 middleware utilizes the reduction variable class's methods that are declared in interface edu.rit.pj2.Vbl. Here's how the middleware uses each method:

- When a parallel team thread calls a parallel loop's `start()` method, and the `start()` method calls the `threadLocal()` method passing in a global reduction variable, the `threadLocal()` method calls the global reduction variable's `clone()` method to create a new reduction variable that is a deep copy of the global reduction variable. A reference to this new reduction variable is returned, and this new reduction variable becomes the per-thread variable.
- When a parallel loop finishes, the middleware performs a reduction tree (Figure 4.3) for each reduction variable that was specified earlier in a `threadLocal()` method call. All of the team threads' per-thread variables feed into the top of the reduction tree. As execution proceeds down the reduction tree, the intermediate results are stored back into the team threads' per-thread variables. When the reduction tree has finished, team thread 0's per-thread variable ends up containing the result of the reduction. This result now has to be stored in the global reduction variable for use outside the parallel loop. The middleware does this by calling the global reduction variable's `set()` method, passing in team thread 0's per-thread variable. This sets the global reduction variable to be a deep copy of the final reduced result.
- While executing the parallel reduction tree, at multiple points the middleware has to combine two intermediate results together by performing the reduction operation. The middleware does this by calling the `reduce()` method. This combines two per-thread variables together in the desired manner and stores the result back in one of the per-thread variables. The middleware itself handles all the necessary thread synchronization, so `reduce()` does not need to be a synchronized method.

As stated previously, when defining your own reduction variable class, it's crucial to implement the `clone()`, `set()`, and `reduce()` methods exactly as

```
15    // Global histogram of random numbers.
16    HistogramVbl histvbl;
17
18    // Main program.
19    public void main
20       (String[] args)
21       throws Exception
22       {
23       // Validate command line arguments.
24       if (args.length != 3) usage();
25       seed = Long.parseLong (args[0]);
26       B = Integer.parseInt (args[1]);
27       N = Long.parseLong (args[2]);
28
29       // Set up global histogram.
30       histvbl = new HistogramVbl (new Histogram (B)
31          {
32          public void accumulate (double x)
33             {
34             increment ((int)(x*size()));
35             }
36          });
37
38       // Do N trials.
39       parallelFor (0, N - 1) .exec (new LongLoop()
40          {
41          HistogramVbl thrHistvbl;
42          Random prng;
43          int leap;
44          public void start()
45             {
46             thrHistvbl = threadLocal (histvbl);
47             prng = new Random (seed);
48             prng.skip (rank());
49             leap = threads() - 1;
50             }
51          public void run (long i)
52             {
53             thrHistvbl.hist.accumulate (prng.nextDouble());
54             prng.skip (leap);
55             }
56          });
57
58       // Print results.
59       System.out.printf ("Bin\tCount%n");
60       for (int i = 0; i < B; ++ i)
61          System.out.printf ("%d\t%d%n", i, histvbl.hist.count (i));
62       double chisqr = histvbl.hist.chisqr();
63       System.out.printf ("Chisqr = %.5g%n", chisqr);
64       System.out.printf ("Pvalue = %.5g%n",
65          histvbl.hist.pvalue (chisqr));
66       }
67
68    // Print a usage message and exit.
69    private static void usage()
70       {
71       System.err.println ("Usage: java pj2 " +
72          "edu.rit.pj2.example.StatTestSmp <seed> <B> <N>");
```

Listing 5.4. StatTestSmp.java (part 2)

specified in interface edu.rit.pj2.Vbl. If you don't, the Parallel Java 2 middleware will not do the right thing when it performs the reduction, and the program will compute the wrong answer.

Here's how the *p*-value of the χ^2 statistic is calculated. In a chi-square test, if the null hypothesis is true, and if the count in each histogram bin is large, then the χ^2 statistic is a random variable that obeys the *chi-square distribution* with $B - 1$ *degrees of freedom*, where B is the number of bins. The *p*-value is given by the formula

$$p\text{-value} = 1 - P\left(\frac{B-1}{2}, \frac{\chi^2}{2}\right) \tag{5-2}$$

where $P()$ is the "incomplete gamma function." The incomplete gamma function is included in most numerical software libraries as well as the Parallel Java 2 Library. For further information about the chi-square test, refer to a statistics textbook or to a numerical software textbook like *Numerical Recipes.**

The Parallel Java 2 Library's class edu.rit.numeric.Histogram lets you compute histograms for data values of type `int`, `long`, `float`, and `double`, as well as object types. As illustrated earlier, you have to override the `accumulate()` method to map a data value to the proper histogram bin and increment the corresponding bin counter.

Class Histogram also lets you customize the formula for the expected probability of each bin; that is, the probability that a given bin will be incremented, given a randomly chosen data value. The default expected probability distribution is a uniform distribution, where each bin has an equal chance $1/B$ of being incremented; but you can override this if the expected probability distribution is nonuniform.

Class edu.rit.pj2.vbl.HistogramVbl wraps an instance of class Histogram or a subclass thereof, to provide a reduction variable for any kind of histogram.

Class edu.rit.util.Random generates a sequence of pseudorandom numbers, like any PRNG class. Unlike most PRNG classes, class Random can efficiently skip ahead in the sequence without actually generating the intervening pseudorandom numbers. How? Class Random maintains a hidden 64-bit counter. The seed supplied as the constructor argument is used to initialize the counter. To generate the next pseudorandom number in the sequence, class Random increments the counter, feeds the new counter value through a *hash function*, and returns the hash function's result. The hash function is designed so that its sequence of outputs looks random—that is, its sequence of outputs passes the Diehard and TestU01 statistical test suites—when its in-

* W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 2007.

```
73              System.err.println ("<seed> = Random seed");
74              System.err.println ("<B> = Number of histogram bins");
75              System.err.println ("<N> = Number of trials");
76              throw new IllegalArgumentException();
77              }
78      }
```

Listing 5.4. StatTestSmp.java (part 3)

puts are a sequence of consecutive counter values. (Class Random does not use a cryptographic hash function like SHA-256; its hash function is much simpler.) To skip ahead *n* positions in the pseudorandom sequence, one merely increases the counter by *n*. As the statistical test program in this chapter illustrates, this skipping capability is often useful in parallel programs that work with random numbers.

Points to Remember

- Make your own class a reduction variable class by implementing interface edu.rit.pj2.Vbl.
- Define the `clone()` method to create a new reduction variable that is a deep copy of the given reduction variable.
- Define the `set()` method to make this reduction variable be a deep copy of the given reduction variable.
- Define the `reduce()` method to combine this reduction variable with the given reduction variable using the desired reduction operation and store the result back in this reduction variable.
- To generate the same sequence of random numbers in a parallel program no matter how many threads are running, initialize all the threads' per-thread PRNGs with the same seed, and use the `skip()` method to skip past the other threads' random numbers.
- The chi-square test is often used to test whether a sequence of numbers obeys a certain probability distribution.
- In your programs, consider using the Parallel Java 2 Library's histogram class and histogram reduction variable class.

Chapter 6
Load Balancing

The *Euler totient function* of a number *n*, denoted $\Phi(n)$, is the number of numbers in the range 1 through $n - 1$ that are relatively prime to *n*. Two numbers are said to be relatively prime if they have no factors in common other than 1. The Euler totient function is the foundation upon which the security of the RSA public key cryptosystem rests. An RSA public key consists of a modulus *n* and an exponent *e*. The modulus is the product of two large prime numbers. The prime numbers are typically 300 or more digits long, so *n* is typically 600 or more digits long. If you could compute the Euler totient of an RSA public key modulus, you would be able to decrypt messages encrypted with that key. This would let you hack into the connection between a web browser and a secure web site, steal passwords and credit card numbers, and generally bring Internet commerce to a standstill. So far, however, no one has managed to invent an algorithm that computes the Euler totient of the product of two 300-digit prime numbers in a practical amount of time.

Listing 6.1 is a sequential program to compute $\Phi(n)$. The program simply loops through every number *i* from 1 through $n - 1$, determines whether *i* and *n* are relatively prime, and increments a counter if so. The program decides whether *i* and *n* are relatively prime by computing a list of the prime factors of each number and comparing the lists to see if they have any entries in common. The program factorizes a number using trial division, similar to the algorithm in the primality testing program in Chapter 2.

For example, suppose I am computing $\Phi(n)$ for $n = 100$. The prime factors of 100 are {2, 2, 5, 5}. Next I compute the prime factors of every number *i* from 1 to 99. For example, the prime factors of 35 are {5, 7}. Comparing the lists of factors, I see that 35 and 100 have a factor in common, namely 5. Therefore 35 and 100 are not relatively prime. On the other hand, the prime factors of 39 are {3, 13}; 39 and 100 have no factors in common (other than 1); so 39 and 100 are relatively prime. It turns out that the following numbers are relatively prime to 100: 1, 3, 7, 9, 11, 13, 17, 19, 21, 23, 27, 29, 31, 33, 37, 39, 41, 43, 47, 49, 51, 53, 57, 59, 61, 63, 67, 69, 71, 73, 77, 79, 81, 83, 87, 89, 91, 93, 97, and 99. There are 40 of them, so $\Phi(100) = 40$.

This is a horribly inefficient way to compute the Euler totient. I'm not trying to develop the world's finest Euler totient program; I'm using this example to make a point about parallel programming. Bear with me.

To parallelize this program, note that there are no sequential dependencies in the loop on lines 24–26. The computation for each number *i* can be done independently of every other number. So I can change the loop to a parallel for loop and use the parallel reduction pattern to add all the per-thread counters together. Listing 6.2 is the result.

Here is what each program printed when told to compute the Euler totient of $n = 10{,}000{,}019$ on a four-core `tardis` node. Because this *n* is prime, every number from 1 through $n - 1$ is relatively prime to *n*, so $\Phi(n) = n - 1$ in this case.

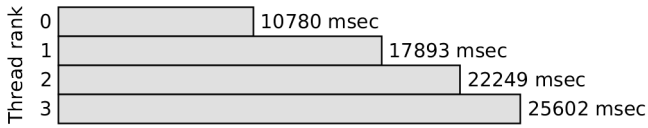
```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Task;
3  import edu.rit.util.LongList;
4  public class TotientSeq
5     extends Task
6     {
7     long n;
8     long phi;
9     LongList nFactors = new LongList();
10    LongList iFactors = new LongList();
11
12    // Main program.
13    public void main
14       (String[] args)
15       throws Exception
16       {
17       // Validate command line arguments.
18       if (args.length != 1) usage();
19       n = Long.parseLong (args[0]);
20
21       // Compute totient.
22       phi = 0;
23       factorize (n, nFactors);
24       for (long i = 2; i < n; ++ i)
25          if (relativelyPrime (factorize (i, iFactors), nFactors))
26             ++ phi;
27
28       // Print totient.
29       System.out.printf ("%d%n", phi + 1);
30       }
31
32    // Store a list of the prime factors of <I>x</I> in ascending
33    // order in the given list.
34    private static LongList factorize
35       (long x,
36        LongList list)
37       {
38       list.clear();
39       long p = 2;
40       long psqr = p*p;
41       while (psqr <= x)
42          {
43          if (x % p == 0)
44             {
45             list.addLast (p);
46             x /= p;
47             }
48          else
49             {
50             p = p == 2 ? 3 : p + 2;
51             psqr = p*p;
52             }
53          }
54       if (x != 1)
55          list.addLast (x);
56       return list;
57       }
58
```

Listing 6.1. TotientSeq.java (part 1)

```
$ java pj2 debug=makespan edu.rit.pj2.example.TotientSeq \
  10000019
10000018
Job 9 makespan 76015 msec
$ java pj2 debug=makespan edu.rit.pj2.example.TotientSmp \
  10000019
10000018
Job 10 makespan 25293 msec
```

The speedup was 76015 ÷ 25293 = 3.005. Unlike the previous parallel program's we've studied, the TotientSmp program's speedup is not very close to the ideal speedup of 4. Ideally, the TotientSmp program's running time on the four-core node should have been 76015 ÷ 4 = 19004 msec. What's going on?

To find out, we have to look at how much time each parallel team thread spends executing its portion of the parallel loop. I modified the TotientSmp program to measure and print each thread's running time. Look at the result:



Ideally, each thread should spend the same amount of time in the parallel loop. But we see they do not; some take more time than others, some take less. Because the parallel loop does not finish until the longest-running thread finishes, the program's running time ends up being larger and its speedup smaller than they should be.

This situation, where the parallel team threads take different amounts of time to execute, is called an *unbalanced load.* An unbalanced load is undesirable; it causes the parallel program to take more time than necessary. Rather, we want the program to have a *balanced load,* where every thread takes about the same amount of time.

But why is the TotientSmp program's load unbalanced? After all, the parallel for loop is using the default schedule, namely a fixed schedule, which gives each parallel team thread an equal portion of the loop iterations. However, in the TotientSmp program, the code executed in each loop iteration on lines 36–37 *takes a different amount of time in each loop iteration.* Why? Because as the number *i* increases, the `factorize()` method takes longer and longer to compute the prime factors of *i.* Thus, the higher-ranked threads—the ones that factorize the larger values of *i*—take longer than the lower-ranked threads. This leads to the unbalanced load. In the previous chapters' parallel programs, each parallel for loop iteration took the *same* amount of time; so the load was inherently balanced, and using the default fixed schedule resulted in close-to-ideal speedups.

How can we get a balanced load in the TotientSmp program? We have to use a different parallel for loop schedule. Instead of dividing the parallel for loop iterations into four large chunks (one chunk for each parallel team

```
59    // Determine whether two numbers are relatively prime, given
60    // their lists of factors.
61    private static boolean relativelyPrime
62       (LongList xFactors,
63        LongList yFactors)
64       {
65       int xSize = xFactors.size();
66       int ySize = yFactors.size();
67       int ix = 0;
68       int iy = 0;
69       long x, y;
70       while (ix < xSize && iy < ySize)
71          {
72          x = xFactors.get (ix);
73          y = yFactors.get (iy);
74          if (x == y) return false;
75          else if (x < y) ++ ix;
76          else ++ iy;
77          }
78       return true;
79       }
80
81    // Print a usage message and exit.
82    private static void usage()
83       {
84       System.err.println ("Usage: java pj2 " +
85          "edu.rit.pj2.example.TotientSeq <n>");
86       throw new IllegalArgumentException();
87       }
88
89    // Specify that this task requires one core.
90    protected static int coresRequired()
91       {
92       return 1;
93       }
94    }
```

Listing 6.1. TotientSeq.java (part 2)

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.LongLoop;
3  import edu.rit.pj2.Task;
4  import edu.rit.pj2.vbl.LongVbl;
5  import edu.rit.util.LongList;
6  public class TotientSmp
7     extends Task
8     {
9     long n;
10    LongVbl phi;
11    LongList nFactors = new LongList();
12
13    // Main program.
14    public void main
15       (String[] args)
16       throws Exception
```

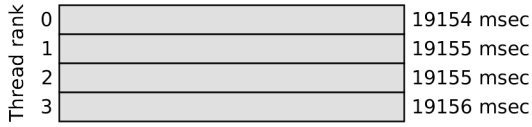
Listing 6.2. TotientSmp.java (part 1)

thread) as the fixed schedule does, let's divide the loop iterations into many small chunks with, say, 1000 iterations in each chunk. Then let the threads execute chunks in a dynamic fashion. Each thread starts by executing one chunk. When a thread finishes its chunk, it executes the next available chunk. When all chunks have been executed, the parallel for loop finishes. This way, some threads execute fewer longer-running chunks, other threads execute more shorter-running chunks, the threads finish at roughly the same time, and the load is balanced. This is called a *dynamic schedule.*

You can specify the parallel for loop schedule and chunk size on the `pj2` command line by including the `schedule` and `chunk` parameters. These override the default fixed schedule. Here is the same TotientSmp program run on `tardis`, this time with a dynamic schedule and a chunk size of 1000 iterations:

```
$ java pj2 debug=makespan schedule=dynamic chunk=1000 \
  edu.rit.pj2.example.TotientSmp 10000019
10000018
Job 11 makespan 19178 msec
```

This time the speedup was 76015 ÷ 19178 = 3.964, very close to an ideal speedup. The dynamic schedule has indeed balanced the load, as is evident from the parallel team threads' individual running times:



With a dynamic schedule, there's a tradeoff. If the chunk size is too large, the load can become unbalanced again, like a fixed schedule. The unbalanced load can result in a longer running time. However, if the chunk size is too small, there will be extra overhead in the program, as the parallel loop has to generate and feed more chunks to the parallel team threads. This extra overhead can also result in a longer running time. It's not always apparent what the best chunk size should be for a dynamic schedule.

As an alternative, you can specify a *proportional schedule.* Instead of specifying chunks of a certain *size,* you specify a certain *number* of chunks. The number of chunks is the number of threads times a *chunk factor.* The set of loop iterations is partitioned into that many equal-sized chunks, and the threads execute these chunks in a dynamic fashion. As the number of threads increases, the number of chunks also increases proportionally, and the chunks become smaller. Here is the same TotientSmp run on `tardis` with a proportional schedule:

```
$ java pj2 debug=makespan schedule=proportional \
  edu.rit.pj2.example.TotientSmp 10000019
10000018
Job 12 makespan 19775 msec
```

```
17          {
18          // Validate command line arguments.
19          if (args.length != 1) usage();
20          n = Long.parseLong (args[0]);
21
22          // Compute totient.
23          phi = new LongVbl.Sum (0);
24          factorize (n, nFactors);
25          parallelFor (2, n - 1) .exec (new LongLoop()
26             {
27             LongList iFactors;
28             LongVbl thrPhi;
29             public void start()
30                {
31                iFactors = new LongList();
32                thrPhi = (LongVbl) threadLocal (phi);
33                }
34             public void run (long i)
35                {
36                if (relativelyPrime (factorize (i, iFactors), nFactors))
37                   ++ thrPhi.item;
38                }
39             });
40
41          // Print totient.
42          System.out.printf ("%d%n", phi.item + 1);
43          }
44
45       // Store a list of the prime factors of <I>x</I> in ascending
46       // order in the given list.
47       private static LongList factorize
48          (long x,
49           LongList list)
50          {
51          list.clear();
52          long p = 2;
53          long psqr = p*p;
54          while (psqr <= x)
55             {
56             if (x % p == 0)
57                {
58                list.addLast (p);
59                x /= p;
60                }
61             else
62                {
63                p = p == 2 ? 3 : p + 2;
64                psqr = p*p;
65                }
66             }
67          if (x != 1)
68             list.addLast (x);
69          return list;
70          }
71
72       // Determine whether two numbers are relatively prime, given
73       // their lists of factors.
74       private static boolean relativelyPrime
```

Listing 6.2. TotientSmp.java (part 2)

For the above run on the four-core `tardis` node, the loop index range was partitioned into 40 chunks—4 threads times the default chunk factor of 10. The speedup was 3.844; better than the fixed schedule, not quite as good as the dynamic schedule.

As another alternative, you can specify a *guided schedule.* Like a dynamic schedule, a guided schedule divides the parallel for loop iterations into many smaller chunks. However, the chunks are not all the same size. Earlier chunks have more iterations; later chunks have fewer iterations. This tends to balance the load automatically without needing to specify the chunk size. Here is the same TotientSmp program run on `tardis` with a guided schedule:

```
$ java pj2 debug=makespan schedule=guided \
  edu.rit.pj2.example.TotientSmp 10000019
10000018
Job 13 makespan 18931 msec
```

The speedup was 4.015, even better than the dynamic schedule—in fact, essentially an ideal speedup. (That it is slightly greater than 4 might be due to random measurement error.) For a guided schedule, the `chunk` parameter gives the minimum chunk size; if omitted, the default is 1.

When a parallel program needs load balancing, you should experiment with different parallel for loop schedules, chunk sizes, and chunk factors on typical inputs to determine the schedule that yields the smallest running time. If you choose, you can then hard-code the schedule into the program; this overrides the default schedule and any schedule specified on the `pj2` command line. To get a dynamic schedule with a chunk size of 1000, write:

```
parallelFor(lb,ub) .schedule(dynamic) .chunk(1000) ...
```

To get a proportional schedule with a chunk factor of 100, write:

```
parallelFor(lb,ub) .schedule(proportional) .chunk(100) ...
```

To get a guided schedule with the default minimum chunk size, write:

```
parallelFor(lb,ub) .schedule(guided) ...
```

Under the Hood

Figure 6.1 shows how various schedules partition the iterations of a parallel for loop into chunks. The loop has $N = 100$ iterations and is being executed by a parallel team with $K = 4$ threads.

- A fixed schedule partitions the iterations into K chunks, each of size $N \div K$, and assigns one chunk to each team thread. If N is not evenly divisible by K, the final chunk has fewer iterations than the other chunks. A leapfrog schedule is similar, except each team thread increments the loop index by K on each iteration instead of by 1.

```
75          (LongList xFactors,
76           LongList yFactors)
77          {
78          int xSize = xFactors.size();
79          int ySize = yFactors.size();
80          int ix = 0;
81          int iy = 0;
82          long x, y;
83          while (ix < xSize && iy < ySize)
84             {
85             x = xFactors.get (ix);
86             y = yFactors.get (iy);
87             if (x == y) return false;
88             else if (x < y) ++ ix;
89             else ++ iy;
90             }
91          return true;
92          }
93
94       // Print a usage message and exit.
95       private static void usage()
96          {
97          System.err.println ("Usage: java pj2 " +
98             "edu.rit.pj2.example.TotientSmp <n>");
99          throw new IllegalArgumentException();
100         }
101      }
```

Listing 6.2. TotientSmp.java (part 3)

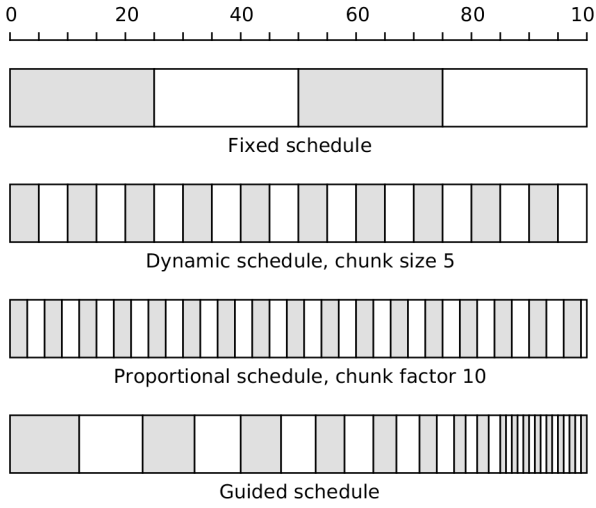


Figure 6.1. Chunk sizes for 100 iterations and four threads

- A dynamic schedule partitions the iterations into chunks of a fixed size (5 iterations in this example). The default chunk size is 1. If N is not evenly divisible by the chunk size, the final chunk has fewer iterations than the other chunks. Each chunk is assigned to a team thread at the beginning of the loop and whenever a team thread finishes its previous chunk.
- A proportional schedule is similar to a dynamic schedule, except it partitions the iterations into a fixed *number* of chunks rather than a fixed *size* of chunks. The number of chunks is equal to a *chunk factor* times K; the default chunk factor is 10. Thus, as K increases, the number of chunks also increases proportionally. The chunk size is $N \div (C \cdot K)$, where C is the chunk factor. If N is not evenly divisible by $C \cdot K$, the final chunk has fewer iterations than the other chunks.
- A guided schedule is similar to a dynamic schedule, except it determines the size of each chunk on the fly. Each chunk's size is half the number of remaining iterations divided by K. If this is less than the specified minimum chunk size (default one iteration), the chunk size is the minimum chunk size. Earlier chunks have more iterations, later chunks have fewer iterations. With $N = 100$ and $K = 4$, the guided schedule's chunk sizes are 12, 11, 9, 8, 7, 6, and so on.

Points to Remember

- If different iterations of a parallel for loop can take different amounts of time to execute, use a dynamic, proportional, or guided schedule to balance the load.
- Run the parallel program on typical inputs with various schedules, chunk sizes, and chunk factors to determine the schedule that yields the smallest overall running time.

Chapter 7
Overlapping

The *Mandelbrot Set* (Figure 7.1) is perhaps the most famous fractal object ever discovered. It is named in honor of mathematician Benoit Mandelbrot, who studied the set extensively in the 1980s. Without going into gory detail about the set's mathematical definition, the image in Figure 7.1 was generated by the following algorithm:

For each point (x, y):
  $i \leftarrow 0$
  $(a, b) \leftarrow (0, 0)$
  While $i \le maxiter$ and $a^2 + b^2 \le 4.0$:
    $i \leftarrow i + 1$
    $(a, b) \leftarrow (a^2 - b^2 + x, 2ab + y)$
  If $i > maxiter$:
    Color the pixel black
  Else:
    Color the pixel with hue $((i - 1)/maxiter)^\gamma$

The image in Figure 7.1 is 1200×1200 pixels. The image is centered at $(x, y) = (–0.75, 0.0)$. The pixel resolution is 450; that is, the distance between the centers of adjacent pixels is 1/450. Thus, the coordinates of the points range from $(x, y) = (–2.083, –1.333)$ at the lower left to (0.583, 1.333) at the upper right. By specifying all these parameters, you can zoom in on any portion of the Mandelbrot Set you want.

Points in the Mandelbrot Set are black. Points not in the set are colored, depending on the value of the counter i when the inner loop exits. The red band at the extreme left of the image corresponds to points where $i = 1$; the next dark orange band corresponds to $i = 2$; the next slightly lighter orange band corresponds to $i = 3$; and so on. Technically, a point is in the Mandelbrot Set only if $a^2 + b^2$ stays ≤ 4.0 forever, no matter how long the inner loop iterates. The algorithm gives up after *maxiter* iterations and decides that the point is (probably) in the set. Figure 7.1 used *maxiter* = 1000.

Each pixel's color is specified by a *hue*, a value between 0.0 and 1.0. A hue corresponds to a color from red to yellow to green to cyan to blue to magenta to red again (Figure 7.2). The counter value i and the parameters *maxiter* and γ determine the hue, via the formula $((i - 1)/maxiter)^\gamma$. Figure 7.1 used $\gamma = 0.4$.

In the above pseudocode, there are no sequential dependencies in the outer loop over the points of the image, so the points can all be calculated in parallel. However, the time required to perform each outer loop iteration is, in general, different for different points. For some points, the inner loop will exit after just a few iterations; for other points, the inner loop will not exit until *maxiter* (e.g., 1000) iterations have happened. Thus, the parallel program requires load balancing.

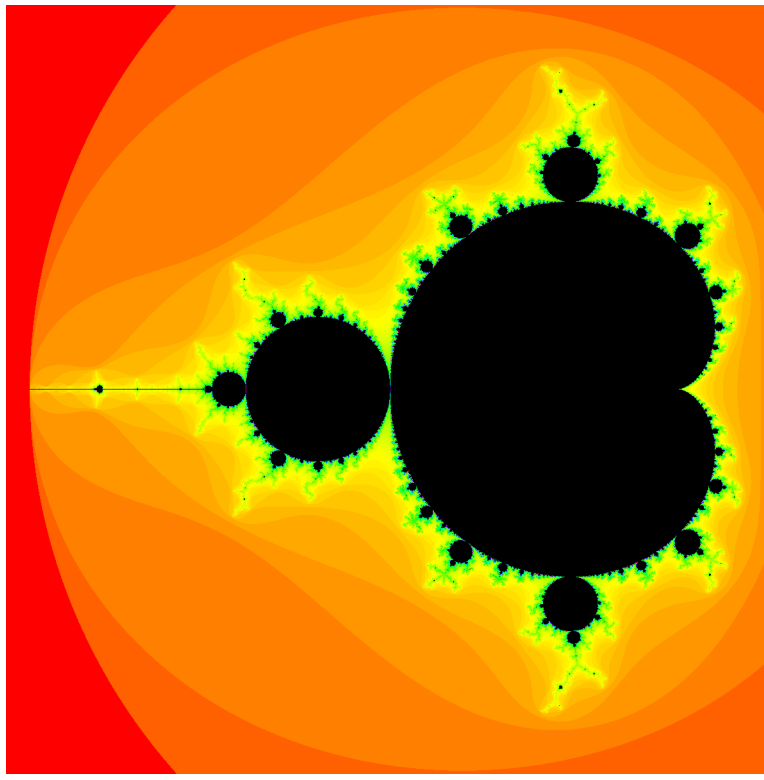


Figure 7.1. The Mandelbrot Set

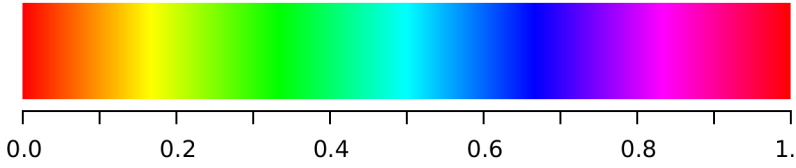


Figure 7.2. Mapping from hues to colors

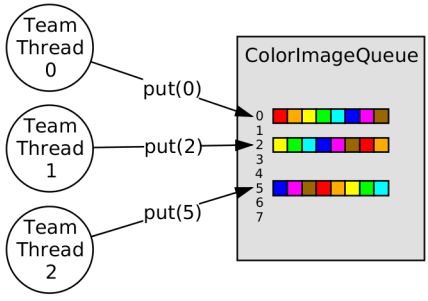
Listing 7.1 gives the parallel Mandelbrot Set program. (From now on, I'm going to dispense with showing the sequential version.) The command line arguments determine which part of the set to calculate, with the final argument naming the PNG file in which to store the image. Figure 7.1 was created with this command:

```
$ java pj2 edu.rit.pj2.example.MandelbrotSmp 1200 1200 \
  -0.75 0 450 1000 0.4 ms1200a.png
```

Lines 60–67 precompute a table of colors corresponding to the possible inner loop counter values. Later, when calculating the actual pixels, I can do a *table lookup* to convert the counter *i* to the appropriate color. This is faster than recomputing the hue formula for every pixel.

Line 80 commences a parallel for loop over the pixel rows. The loop uses a dynamic schedule (line 81) to balance the load, with the default chunk size of one iteration. Thus, each thread in the parallel thread team computes one row at a time. The loop has a per-thread ColorArray object (line 83) in which the color of each pixel in the row will be stored. Inside each row iteration is a regular loop over the pixel columns, starting at line 94. The row and column indexes determine the point's (x, y) coordinates (lines 92 and 96). Executing the innermost loop starting at line 105, each parallel team thread decides whether the point is in the Mandelbrot Set; when this loop exits, the thread stores the appropriate color in its color array. When all columns have been computed, the thread puts its color array into a shared global ColorImageQueue that was created back on line 75. The thread then goes on to compute another row as determined by the dynamic schedule.

As the parallel loop executes, the parallel team threads simultaneously compute various pixel rows, storing the pixel colors in the threads' own separate per-thread color arrays and putting the color arrays into the shared global image queue. The image queue's `put()` method makes a copy of the color array and stores the copy internally, so the thread can reuse the color array for the next pixel row. The `put()` method is multiple thread safe, so all the threads can store pixel rows concurrently without interfering with each other.

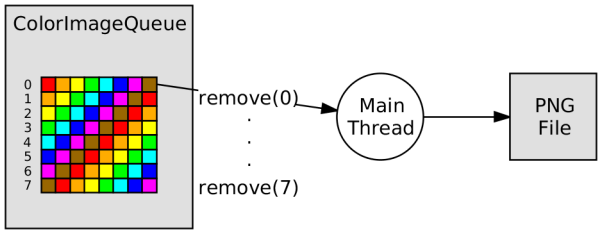


When the parallel loop finishes, the image queue contains the color of every pixel in every row and column of the image. The main program thread

```
1  package edu.rit.pj2.example;
2  import edu.rit.image.Color;
3  import edu.rit.image.ColorArray;
4  import edu.rit.image.ColorImageQueue;
5  import edu.rit.image.ColorPngWriter;
6  import edu.rit.pj2.Loop;
7  import edu.rit.pj2.Schedule;
8  import edu.rit.pj2.Task;
9  import java.io.BufferedOutputStream;
10 import java.io.File;
11 import java.io.FileOutputStream;
12 public class MandelbrotSmp
13    extends Task
14    {
15    // Command line arguments.
16    int width;
17    int height;
18    double xcenter;
19    double ycenter;
20    double resolution;
21    int maxiter;
22    double gamma;
23    File filename;
24
25    // Initial pixel offsets from center.
26    int xoffset;
27    int yoffset;
28
29    // Table of hues.
30    Color[] huetable;
31
32    // For writing PNG image file.
33    ColorPngWriter writer;
34    ColorImageQueue imageQueue;
35
36    // Mandelbrot Set main program.
37    public void main
38       (String[] args)
39       throws Exception
40       {
41       // Start timing.
42       long t1 = System.currentTimeMillis();
43
44       // Validate command line arguments.
45       if (args.length != 8) usage();
46       width = Integer.parseInt (args[0]);
47       height = Integer.parseInt (args[1]);
48       xcenter = Double.parseDouble (args[2]);
49       ycenter = Double.parseDouble (args[3]);
50       resolution = Double.parseDouble (args[4]);
51       maxiter = Integer.parseInt (args[5]);
52       gamma = Double.parseDouble (args[6]);
53       filename = new File (args[7]);
54
55       // Initial pixel offsets from center.
56       xoffset = -(width - 1) / 2;
57       yoffset = (height - 1) / 2;
58
```

Listing 7.1. MandelbrotSmp.java (part 1)

then calls the `write()` method (line 126) on the ColorPngWriter that was created back on line 70. The `write()` method in turn calls the image queue's `remove()` method to remove each pixel row from the image queue and writes each pixel row to the specified PNG file. Note that the file has to be written by a single thread (the main thread); if multiple threads tried to write the file simultaneously, the file's contents would be all jumbled up.



Here's what the sequential and parallel versions of the program printed, computing a 6400×6400-pixel image of the Mandelbrot Set, running on a four-core `tardis` node. Each program prints the time spent before doing the computation, the time spent computing the pixels, the time spent writing the PNG image file, and the total running time. (These times are measured by taking snapshots of the system clock at lines 42, 77, 123, and 129.)

```
$ java pj2 edu.rit.pj2.example.MandelbrotSeq 6400 6400 \
  -0.75 0 2400 1000 0.4 /var/tmp/ark/ms6400_seq.png
22 msec pre
64697 msec calc
2661 msec write
67380 msec total
$ java pj2 edu.rit.pj2.example.MandelbrotSmp 6400 6400 -0.75 0
2400 1000 0.4 /var/tmp/ark/ms6400_smp.png
23 msec pre
16265 msec calc
2649 msec write
18937 msec total
```

The speedup on four cores was 67380 ÷ 18937 = 3.558, not very close to ideal. We also saw a non-ideal speedup in the Euler totient program in Chapter 6. There, the non-ideal speedup was due to an unbalanced load. But here, we took care to balance the load by specifying a dynamic schedule for the parallel for loop, so an unbalanced load is not the problem. What's going on?

The answer becomes clear if we consider just the *calculation* times rather than the total times. The speedup for the calculation was 64697 ÷ 16265 = 3.978. This is very nearly an ideal speedup, as we would expect for a parallel loop with a balanced load. But this ignores the rest of the running time.

The problem is that the MandelbrotSmp program parallelizes, and therefore speeds up, *only a portion of the running time*, namely the time spent in the calculation section. The rest of the program—initialization, and writing the image file—is not done in parallel and so is not sped up. The program's

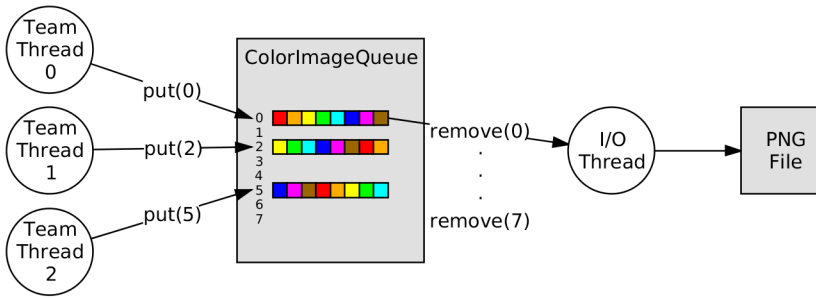
```
59          // Create table of hues for different iteration counts.
60          huetable = new Color [maxiter + 2];
61          for (int i = 1; i <= maxiter; ++ i)
62             huetable[i] = new Color().hsb
63                (/*hue*/ (float) Math.pow ((double)(i – 1)/maxiter,
64                                gamma),
65                 /*sat*/ 1.0f,
66                 /*bri*/ 1.0f);
67          huetable[maxiter + 1] = new Color().hsb (1.0f, 1.0f, 0.0f);
68
69          // For writing PNG image file.
70          writer = new ColorPngWriter (height, width,
71             new BufferedOutputStream
72                (new FileOutputStream (filename)));
73          filename.setReadable (true, false);
74          filename.setWritable (true, false);
75          imageQueue = writer.getImageQueue();
76
77          long t2 = System.currentTimeMillis();
78
79          // Compute all rows and columns.
80          parallelFor (0, height - 1)
81             .schedule (dynamic) .exec (new Loop()
82             {
83             ColorArray pixelData;
84
85             public void start()
86                {
87                pixelData = new ColorArray (width);
88                }
89
90             public void run (int r) throws Exception
91                {
92                double y = ycenter + (yoffset - r) / resolution;
93
94                for (int c = 0; c < width; ++ c)
95                   {
96                   double x = xcenter + (xoffset + c) / resolution;
97
98                   // Iterate until convergence.
99                   int i = 0;
100                  double aold = 0.0;
101                  double bold = 0.0;
102                  double a = 0.0;
103                  double b = 0.0;
104                  double zmagsqr = 0.0;
105                  while (i <= maxiter && zmagsqr <= 4.0)
106                     {
107                     ++ i;
108                     a = aold*aold - bold*bold + x;
109                     b = 2.0*aold*bold + y;
110                     zmagsqr = a*a + b*b;
111                     aold = a;
112                     bold = b;
113                     }
114
115                  // Record number of iterations for pixel.
116                  pixelData.color (c, huetable[i]);
```

Listing 7.1. MandelbrotSmp.java (part 2)

total running time is therefore longer, and its speedup is smaller, than they would be if we had parallelized the entire program.

The MandelbrotSmp program spent 2.7 seconds writing the 6400×6400-pixel image file. For much of that time, the main program thread doing the writing was blocked, waiting for the operating system to output bytes into the file on the disk. This is a waste. Instead of sitting blocked doing nothing a fair fraction of the time, the machine should be doing useful work all the time.

I can keep the machine busy 100 percent of the time by computing pixel colors and writing the image file *concurrently*, rather than waiting until all the pixels have been computed before starting to write the image file. How? By running an additional thread that writes the image file in parallel with the threads computing the pixel rows. This is called *overlapped computation and I/O*, or just *overlapping*. I want the program to operate this way:



Note that the file is still being written by a single thread. The I/O thread will steal some CPU cycles from the computational threads, but whenever the I/O thread is blocked waiting for the disk, the computational threads will have full use of all the cores on the machine.

Overlapping will work as long as I take care to hold off writing each pixel row until the computations have finished for that row—that is, as long as the computational threads and the I/O thread are properly synchronized. The ColorImageQueue object does the necessary synchronization. The image queue's methods are multiple thread safe, so the computational threads and the I/O thread can all be calling methods on the image queue simultaneously without interfering with each other. The `remove()` method does not return (it blocks the calling thread) until the requested row has been put into the image queue.

To get the program to do one section of code (the pixel computations) and another section of code (the file I/O) in parallel, I have to use a new pattern: the *parallel sections* pattern. The pattern consists of multiple *sections* of code. Each section is executed by its own thread, concurrently with all the other sections. Here is how to get a bunch of parallel sections in a program:

```
117                }
118
119             imageQueue.put (r, pixelData);
120             }
121          });
122
123       long t3 = System.currentTimeMillis();
124
125       // Write image to PNG file.
126       writer.write();
127
128       // Stop timing.
129       long t4 = System.currentTimeMillis();
130       System.out.println ((t2-t1) + " msec pre");
131       System.out.println ((t3-t2) + " msec calc");
132       System.out.println ((t4-t3) + " msec write");
133       System.out.println ((t4-t1) + " msec total");
134       }
135
136    // Print a usage message and exit.
137    private static void usage()
138       {
139       System.err.println ("Usage: java " +
140          "edu.rit.pj2.example.MandelbrotSmp <width> <height> " +
141          "<xcenter> <ycenter> <resolution> <maxiter> <gamma> " +
142          "<filename>");
143       System.err.println ("<width> = Image width (pixels)");
144       System.err.println ("<height> = Image height (pixels)");
145       System.err.println ("<xcenter> = X coordinate of center " +
146          "point");
147       System.err.println ("<ycenter> = Y coordinate of center " +
148          "point");
149       System.err.println ("<resolution> = Pixels per unit");
150       System.err.println ("<maxiter> = Maximum number of " +
151          "iterations");
152       System.err.println ("<gamma> = Used to calculate pixel hues");
153       System.err.println ("<filename> = PNG image file name");
154       throw new IllegalArgumentException();
155       }
156 }
```

Listing 7.1. MandelbrotSmp.java (part 3)

```
parallelDo (new Section()
   {
   public void run()
      {
      // Code for first section
      }
   },
new Section()
   {
   public void run()
      {
      // Code for second section
      }
   },
// Additional sections go here
);
```

Listing 7.2 shows the MandelbrotSmp2 program, which illustrates how to code the parallel sections pattern. MandelbrotSmp2 is the same as MandelbrotSmp, except for a piece in the middle. Listing 7.2 shows only what was added going from MandelbrotSmp to MandelbrotSmp2.

The parallel sections begin at line 4 with the `parallelDo()` method call. (This is a method in class Task.) The `parallelDo()` method's arguments are two Section objects, one created on line 4, the other created on line 16. (You can call `parallelDo()` with any number of sections.) Under the hood, the `parallelDo()` method creates a parallel team with two threads, one for each section. Each team thread calls its own section's `run()` method. When the `run()` method returns, the team thread waits at a barrier. When all the team threads have arrived at the barrier, the `parallelDo()` method returns, and the main program continues executing the code that comes afterwards.

I defined each section's `run()` method using Java's anonymous inner class syntax. Each section's `run()` method contains the code that is to be executed in parallel with all the other sections. The computation section's `run()` method does the parallel for loop over the pixel rows; this loop is identical to the one in the MandelbrotSmp program. The I/O section's `run()` method calls the ColorPngWriter's `write()` method to write the PNG file; this call is also identical to the one in the MandelbrotSmp program. As mentioned above, the ColorImageQueue takes care of synchronizing the computational threads and the I/O thread.

Here's what the MandelbrotSmp2 program printed, computing the same image as before, running on a four-core `tardis` node:

```
$ java pj2 edu.rit.pj2.example.MandelbrotSmp2 6400 6400 \
  -0.75 0 2400 1000 0.4 /var/tmp/ark/ms6400_smp2.png
22 msec pre
17145 msec calc+write
17167 msec total
```

```
      long t2 = System.currentTimeMillis();

      // Overlapped computation and I/O.
      parallelDo (new Section()
         {
         public void run()
            {
            // Compute all rows and columns.
            parallelFor (0, height - 1)
               .schedule (dynamic) .exec (new Loop()
               {
               ... same as MandelbrotSmp program ...
               });
            }
         },
      new Section()
         {
         public void run() throws Exception
            {
            // Write image to PNG file.
            writer.write();
            }
         });

      // Stop timing.
      long t3 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc+write");
      System.out.println ((t3-t1) + " msec total");
      }
```

Listing 7.2. MandelbrotSmp2.java (relevant portion)

Compared to the MandelbrotSmp program, the total running time decreased from 18937 msec to 17167 msec, and the speedup increased from 3.558 to 3.925. Adding overlapping markedly improved the program's running time and speedup. (The speedup is still not quite ideal because the additional I/O thread is taking some CPU time away from the computational threads.)

Under the Hood

Figure 7.3 shows in more detail what happens when the MandelbrotSmp2 program executes on a machine with four cores. Recall that there are two parallel sections, the computation section and the I/O section. Furthermore, the computation section contains a parallel for loop. This means the program has no fewer than six threads—the main program thread, two parallel section threads, and four team threads in the parallel for loop.

When the program starts, only the main thread is running. The main thread calls the task's `main()` method. After doing some initialization, the main thread creates two Section objects, namely the computation section and the I/O section. The main thread calls the task's `parallelDo()` method, passing in the Section objects. The `parallelDo()` method creates a hidden thread team with two threads, labeled "Section thread rank 0" and "Section thread rank 1" in the diagram. The main thread blocks inside the `parallelDo()` method until the parallel sections have finished.

At this point the two section threads start executing, each on a separate core. Section thread rank 0 calls the computation section's `run()` method. The `run()` method calls the `parallelFor()` method, which creates and returns an IntParallelForLoop object. Hidden inside the parallel for loop is *another* team of four threads, one for each core. Initially, the loop threads are blocked. Back in the `run()` method, section thread rank 0 creates a Loop object and passes it to the parallel for loop's `exec()` method. The `exec()` method clones three copies of the Loop object. Section thread rank 0 unblocks the loop threads, then blocks itself inside the `exec()` method until the loop threads finish.

Now the loop threads start executing, each on a separate core. Each loop thread calls its own Loop object's `start()` method to do per-thread initialization. Each loop thread then calls its own Loop object's `run()` method. Due to the parallel for loop's dynamic schedule, loop threads rank 0 through 3 call their `run()` methods with the first four available loop indexes, 0 through 3. Each `run()` method computes the given row of pixels, then calls the ColorImageQueue's `put()` method, passing in the row index and the pixel color array. However, because the `put()` method is multiple thread safe, each loop thread blocks until another loop thread's in-progress `put()` method call has returned. The ColorImageQueue stores a copy of each color array internally, associated with the row index. Each loop thread continues to call its own

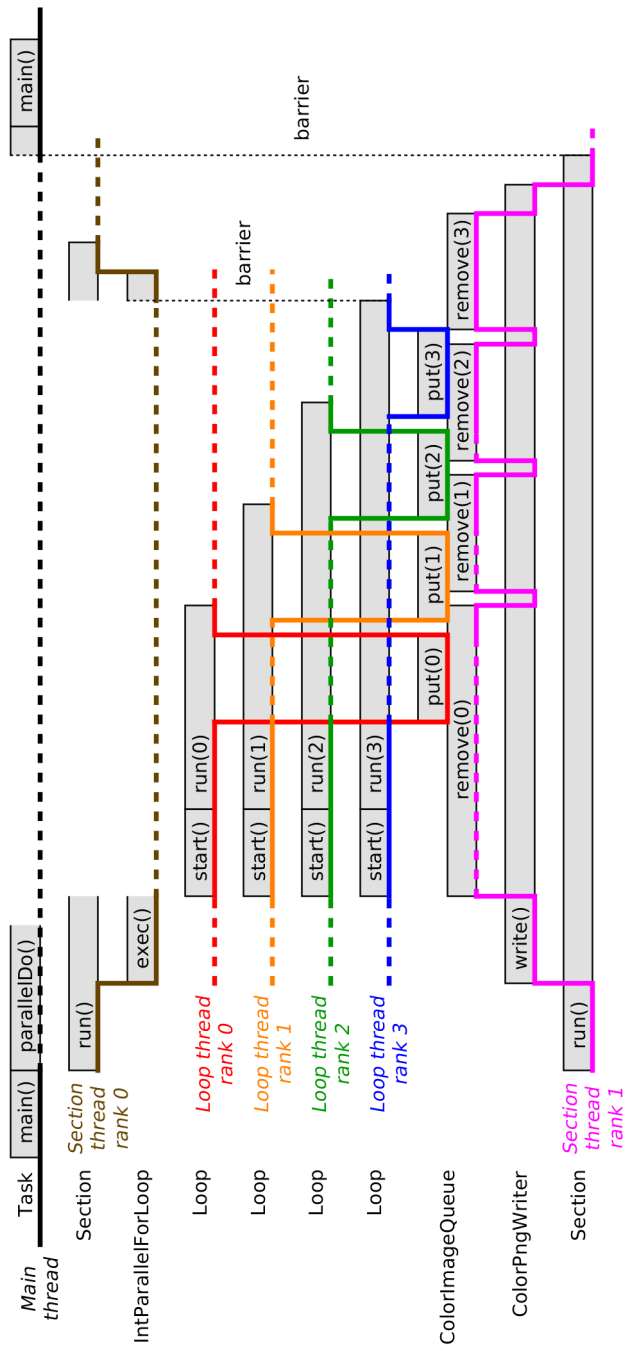


Figure 7.3. Parallel sections execution flow

Loop object's `run()` method, passing in the next available loop index from the dynamic schedule. (The diagram does not show these additional `run()` method calls.) When all loop indexes have been executed, each loop thread arrives at the end-of-loop barrier and blocks.

When all the loop threads have arrived at the end-of-loop barrier—that is, when the parallel for loop is finished—section thread rank 0 unblocks and returns from the parallel for loop's `exec()` method back to the computation section's `run()` method. With no further code in the computation section, the `run()` method returns. Section thread rank 0 arrives at the end-of-sections barrier and blocks.

Meanwhile, section thread rank 1 has been executing in parallel with all the above. Section thread rank 1 calls the I/O section's `run()` method. The `run()` method calls the ColorPngWriter's `write()` method. The `write()` method calls the ColorImageQueue's `remove()` method to get the pixel color array for row 0. The ColorImage queue blocks section thread rank 1 inside the `remove()` method until another thread (one of the loop threads) has finished calling the `put()` method for row 0. The `remove()` method then unblocks section thread rank 1 and returns the row 0 color array back to the `write()` method, which writes the pixels to the PNG file. The `write()` method continues calling the `remove()` method for rows 1, 2, 3, and so on until all the pixel rows have been written to the PNG file. The `write()` method then returns back to the I/O section's `run()` method. With no further code in the I/O section, the `run()` method returns. Section thread rank 1 arrives at the end-of-sections barrier and blocks.

When both section threads have arrived at the end-of-sections barrier—that is, when the parallel sections have finished—the main thread unblocks and returns from the `parallelDo()` method back to the task's `main()` method. The main thread prints the running times, and the program terminates.

Points to Remember

- Use overlapping to reduce the running time of a parallel program that does both computations and I/O.
- Use the `parallelDo()` pattern to execute multiple sections of code in parallel.
- Be sure to synchronize the computational threads and the I/O threads properly, so the I/O doesn't get ahead of the computation or vice versa.
- Consider using a multiple thread safe, shared global object to perform the synchronization.
- Consider creating a table of precalculated items and doing table lookups, rather than recalculating the same items many times.

Chapter 8
Sequential Dependencies

So-called *N-body problems* pop up now and then when studying natural phenomena. The behavior of flocks of birds, schools of fish, insect swarms, and stars in a star cluster or galaxy can be modeled as *N*-body problems and computed with *N*-body programs. Our next parallel program is an *N*-body program where the bodies literally are corpses, or more precisely, zombies. I could have based this example on birds or stars, but zombies are cooler. (The zombie model in this chapter is actually based on a model of locust swarms.*)

Initially, *N* zombies are scattered at random locations within a certain area; this is the *initial state*. The zombies want to get together, so they all start moving towards each other. However, the zombies don't want to be too close to one another. Each zombie continues moving until its attraction to the other zombies is exactly counterbalanced by its repulsion from nearby zombies. Thereafter, the zombies stay put; this is the *equilibrium state*. I want to write a program that, given the zombies' initial state and certain other parameters, computes the zombies' positions as a function of time (Figure 8.1).

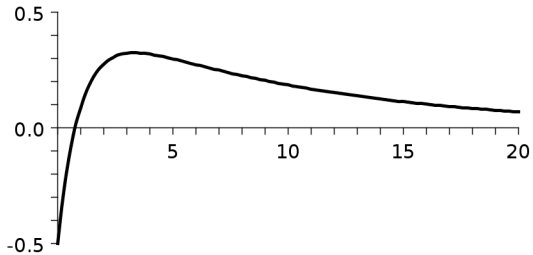
The zombies are indexed from 0 to $N-1$. The zombies move in a two-dimensional plane. At any given instant in time, zombie i's position is (x_i, y_i). Consider just two zombies at indexes i and j. The distance between zombie i and zombie j is the Euclidean distance:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \ . \tag{8.1}$$

Zombie i moves toward or away from zombie j at a velocity given by

$$v_{ij} = G \cdot \exp(-d_{ij}/L) - \exp(-d_{ij}) \ . \tag{8.2}$$

A positive velocity means zombie i moves toward zombie j (attaction), a negative velocity means zombie i moves away from zombie j (repulsion). The first term in equation (8.2) models attraction, the second term models repulsion. G and L are constants that determine the strength of the attraction. Here is a plot of velocity versus distance for $G = 0.5$ and $L = 10$:



As the two zombies move closer together, the attractive velocity increases. But as they move still closer, the attractive velocity diminishes. If they get

* A. Bernoff and C. Topaz. Nonlocal aggregation models: a primer of swarm equilibria. *SIAM Review*, 55(4):709–747, December 2013.

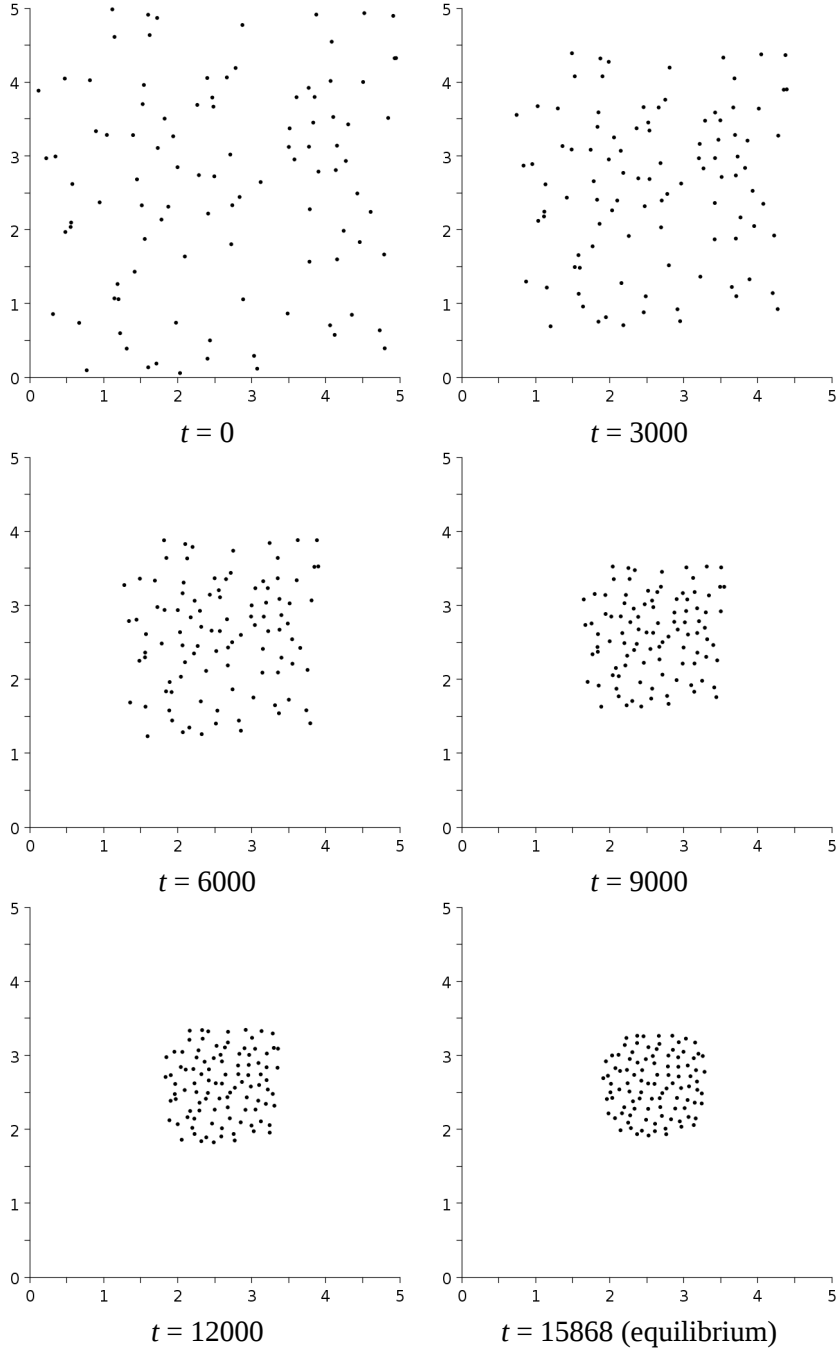


Figure 8.1. Snapshots of zombie positions, $N = 100$

too close, the velocity reverses and becomes repulsive.

The velocity from zombie *i* towards zombie *j* is actually a vector quantity (vx_{ij}, vy_{ij}) whose magnitude is v_{ij}. The velocity vector's X and Y components are

$$vx_{ij} = v_{ij} \cdot \frac{x_j - x_i}{d_{ij}} \quad , \qquad vy_{ij} = v_{ij} \cdot \frac{y_j - y_i}{d_{ij}} \quad . \tag{8.3}$$

Now consider all *N* zombies. The net velocity of zombie *i* (vx_i, vx_j) is the vector sum of the velocities with respect to all the other zombies:

$$vx_i = \sum_{\substack{j=0 \\ j \neq i}}^{N-1} vx_{ij} \quad , \qquad vy_i = \sum_{\substack{j=0 \\ j \neq i}}^{N-1} vy_{ij} \quad . \tag{8.4}$$

Given zombie *i*'s position (x_i, y_i) at a certain instant, zombie *i*'s position a small amount of time *dt* later ($nextx_i$, $nexty_i$) is obtained by multiplying zombie *i*'s velocity times *dt* and adding that to zombie *i*'s current position:

$$nextx_i = x_i + vx_i \cdot dt \quad , \qquad nexty_i = y_i + vy_i \cdot dt \quad . \tag{8.5}$$

Calculating the next positions of all the zombies using the preceding formulas, and then replacing the current positions with the next positions, is called "taking a time step." By taking a series of time steps, I can plot out the zombies' positions as a function of time. I'll stop taking time steps when the zombies reach the equilibrium state; that is, when the zombies are no longer changing their positions. I'll detect this by adding up the absolute values of the distances the zombies move:

$$\Delta = \sum_{i=0}^{N-1} |vx_i \cdot dt| + |vy_i \cdot dt| \quad . \tag{8.6}$$

When Δ becomes less than a certain tolerance ε, I'll stop. (Δ will never become exactly 0, so I'll stop when Δ becomes too small to notice.)

Putting it all together, here is the pseudocode for the zombie program.

```
Initialize zombie positions
Repeat: (time step loop)
   For i = 0 to N – 1:
      (vxi, vyi) ← (0, 0)
      For j = 0 to N – 1, j ≠ i:
         Compute (vxij, vyij) using equations (8.1), (8.2), and (8.3)
         (vxi, vyi) ← (vxi + vxij, vyi + vyij)
      (nextxi, nextyi) ← (xi + vxi · dt, yi + vyi · dt)
   Replace current zombie positions with next zombie positions
```

If $\Delta < \varepsilon$:
  Exit time step loop

Note the doubly-nested loop over the zombies within each time step. The outer loop has N iterations, the inner loop also has N iterations (one of which does no calculations), so each time step requires $O(N^2)$ computations. To get accurate results, the time step size *dt* must be very, very small. Consequently, the program will have to take many, many time steps to reach equilibrium. The N-body program can easily require enormous amounts of computation. It is a prime candidate for parallelization.

The zombie N-body algorithm consists of three nested loops: the outer loop over the time steps, the middle loop to calculate each zombie's next position, and the inner loop to calculate each zombie's velocity with respect to every other zombie. The Mandelbrot Set algorithm in Chapter 7 also had nested loops, the outer loop over the pixel rows and the inner loop over the pixel columns. We were able to make the Mandelbrot Set's outer loop a parallel loop because the outer loop had no sequential dependencies—each pixel could be calculated independently of all the other pixels, so the pixel rows could all be calculated in parallel. That is not the case with the N-body algorithm. This time, the outer loop *does* have sequential dependencies. It is not possible to calculate each time step independently of all the other time steps. Rather, *we must wait for the preceding time step to finish before commencing the next time step*, because we need the zombie positions from the previous time step to calculate the zombie positions for the next time step. So the outer loop must remain a regular, non-parallel loop.

The middle loop over the zombies, however, does *not* have sequential dependencies; the next position of each zombie *can* be calculated independently of all the other zombies. So to parallelize the algorithm, I can make the middle loop a parallel loop:

Initialize zombie positions
Repeat: (time step loop)
  Parallel for $i = 0$ to $N - 1$:
    $(vx_i, vy_i) \leftarrow (0, 0)$
    For $j = 0$ to $N - 1$, $j \neq i$:
      Compute (vx_{ij}, vy_{ij}) using equations (8.1), (8.2), and (8.3)
      $(vx_i, vy_i) \leftarrow (vx_i + vx_{ij}, vy_i + vy_{ij})$
    $(nextx_i, nexty_i) \leftarrow (x_i + vx_i \cdot dt, y_i + vy_i \cdot dt)$
  Replace current zombie positions with next zombie positions
  If $\Delta < \varepsilon$:
    Exit time step loop

Once I've parallelized the middle loop, there's no point in parallelizing the inner loop. So the inner loop remains a regular loop. Each parallel team thread performs a subset of the iterations over i and all the iterations over j.

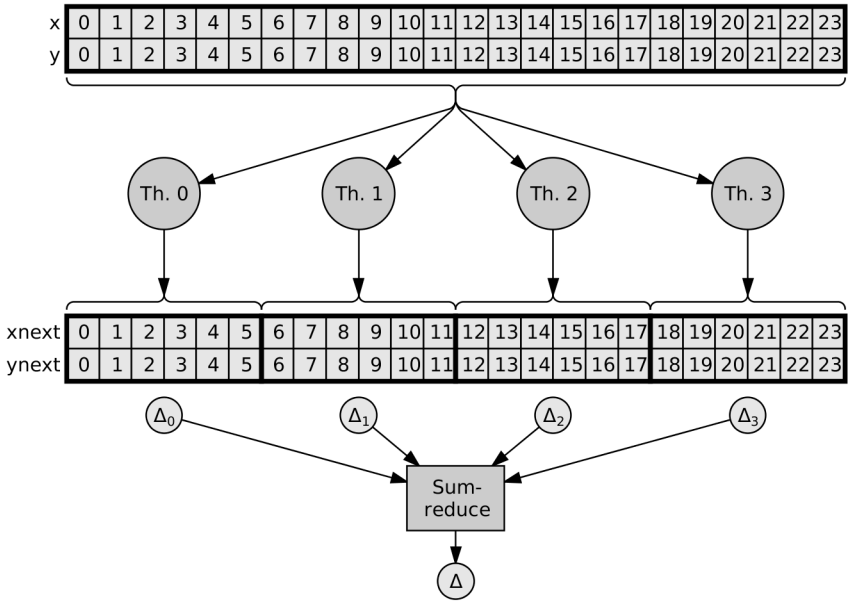


Figure 8.2. One time step performed in parallel

Figure 8.2 depicts what goes on in one time step of the parallel zombie program. In this example, there are 24 zombies and four parallel team threads (cores). The 24 zombies are partitioned into four equal *slices* of six zombies each. During each time step, each team thread loops over the zombies in the thread's slice (the middle loop), computing the zombies' next positions, *xnext* and *ynext.* During each middle loop iteration, the team thread loops over all the zombies (the inner loop), retrieving the zombies' current positions, *x* and *y.* The threads do not need to synchronize with each other while reading the elements of *x* and *y,* because the threads are not changing these values. The threads do not need to synchronize with each other while writing the elements of *xnext* and *ynext,* because each element is written by only one thread.

Also during the middle loop, each team thread accumulates the total distance moved by the zombies in the thread's slice, Δ_0 through Δ_3. As the middle loop exits, these per-thread values are added together via a sum reduction to yield the total distance moved by all the zombies, Δ. The total Δ then controls whether the outer time step loop terminates.

Listing 8.1 gives the parallel zombie program. It is a straightforward translation of the parallel pseudocode into Java. The program's command line arguments are

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Loop;
3  import edu.rit.pj2.Task;
4  import edu.rit.pj2.vbl.DoubleVbl;
5  import edu.rit.util.Random;
6  import static java.lang.Math.*;
7  public class ZombieSmp
8     extends Task
9     {
10    // Command line arguments.
11    long seed;
12    int N;
13    double W;
14    double G;
15    double L;
16    double dt;
17    double eps;
18    int steps;
19    int snap;
20
21    // Current body positions.
22    double[] x;
23    double[] y;
24
25    // Next body positions.
26    double[] xnext;
27    double[] ynext;
28
29    // For detecting convergence.
30    DoubleVbl delta = new DoubleVbl.Sum();
31
32    // Task main program.
33    public void main
34       (String[] args)
35       {
36       // Parse command line arguments.
37       if (args.length != 9) usage();
38       seed = Long.parseLong (args[0]);
39       N = Integer.parseInt (args[1]);
40       W = Double.parseDouble (args[2]);
41       G = Double.parseDouble (args[3]);
42       L = Double.parseDouble (args[4]);
43       dt = Double.parseDouble (args[5]);
44       eps = Double.parseDouble (args[6]);
45       steps = Integer.parseInt (args[7]);
46       snap = Integer.parseInt (args[8]);
47
48       // Set up N bodies' initial (x,y) coordinates at random in a
49       // WxW square region.
50       x = new double [N];
51       y = new double [N];
52       xnext = new double [N];
53       ynext = new double [N];
54       Random prng = new Random (seed);
55       for (int i = 0; i < N; ++ i)
56          {
57          x[i] = prng.nextDouble()*W;
58          y[i] = prng.nextDouble()*W;
```

Listing 8.1. ZombieSmp.java (part 1)

- *seed*—Seed for a pseudorandom number generator for initializing the zombies' positions
- *N*—Number of zombies
- *W*—Size of the initial region in which to place the zombies
- *G*—Attraction factor used in equation (8.2)
- *L*—Attraction length scale used in equation (8.2)
- *dt*—Time step size
- *eps*—Convergence threshold ε
- *steps*—Number of time steps (0 means iterate until convergence)
- *snap*—Snapshot interval (0 means no snapshots)

If a nonzero *steps* parameter is specified on the command line, the program computes the specified number of time steps, whether or not the zombie positions have converged. Otherwise, the program iterates for as many time steps as necessary to achieve convergence (lines 120–122).

The program prints a snapshot of the zombies' initial positions (line 63) and final positions (line 129). If a nonzero *snap* parameter is specified on the command line, the program also prints a snapshot of the zombies' positions every *snap* time steps (line 125). The zombie positions plotted in Figure 8.1 were computed by this program run:

```
$ java pj2 edu.rit.pj2.example.ZombieSeq 142857 100 5 0.5 10 \
  0.00001 0.001 0 3000
```

Each iteration of the middle loop, which calculates the next position of one zombie, does exactly the same number of computations as every other iteration. Thus, each middle loop iteration takes the same amount of time, so the load is inherently balanced, so the parallel for loop can use the default fixed schedule.

The `delta` variable (line 30), which holds the quantity Δ used to detect convergence, is a global reduction variable of type DoubleVbl. It is initialized to an instance of class DoubleVbl.Sum, which does a sum-reduce. Following the parallel reduction pattern, each parallel team thread has a thread-local `thrDelta` variable that accumulates Δ for that thread's subset of the iterations. The thread-local Δ values are automatically summed together into the global reduction variable as the middle parallel for loop finishes.

The outer loop body (lines 66–126), which computes one time step, consists of an initial section executed by a single thread (line 68), a parallel for loop executed by multiple threads (lines 71–107), and a final section executed by a single thread again (lines 110–125). The final section is not executed until all the parallel team threads have arrived at the implicit barrier at the end of the parallel for loop, signaling that the zombies' next position calculations are all complete. This barrier synchronization is essential if the program is to compute the correct results. It is also essential that the final section

```
59          }
60
61       // Snapshot all bodies' initial positions.
62       int t = 0;
63       snapshot (t);
64
65       // Do time steps.
66       for (;;)
67          {
68          delta.item = 0.0;
69
70          // Do each body i.
71          parallelFor (0, N - 1) .exec (new Loop()
72             {
73             DoubleVbl thrDelta;
74
75             public void start()
76                {
77                thrDelta = threadLocal (delta);
78                }
79
80             public void run (int i)
81                {
82                double vx = 0.0;
83                double vy = 0.0;
84                double dx, dy, d, v;
85
86                // Accumulate velocity due to every other body j.
87                for (int j = 0; j < N; ++ j)
88                   {
89                   if (j == i) continue;
90                   dx = x[j] - x[i];
91                   dy = y[j] - y[i];
92                   d = sqrt(dx*dx + dy*dy);
93                   v = G*exp(-d/L) - exp(-d);
94                   vx += v*dx/d;
95                   vy += v*dy/d;
96                   }
97
98                // Move body i in the direction of its velocity.
99                dx = vx*dt;
100               dy = vy*dt;
101               xnext[i] = x[i] + dx;
102               ynext[i] = y[i] + dy;
103
104               // Accumulate position delta.
105               thrDelta.item += abs(dx) + abs(dy);
106               }
107            });
108
109         // Advance to next time step.
110         ++ t;
111
112         // Update positions.
113         double[] tmp;
114         tmp = x; x = xnext; xnext = tmp;
115         tmp = y; y = ynext; ynext = tmp;
116
```

Listing 8.1. ZombieSmp.java (part 2)

—checking for convergence, updating the zombies' positions, and printing a snapshot—be done outside the parallel for loop in a single thread.

The sections of the outer loop body executed by a single thread don't get sped up when running on multiple cores. Thus, it's important to minimize the time spent in these single-threaded sections. For this reason, the program updates the zombie positions (lines 113–115) simply by swapping the array references, so that `x` and `y` refer to the arrays that used to be `nextx` and `nexty`, and vice versa. This is faster than copying the array elements, especially if N is large.

One detail about the snapshots: The `snapshot()` method, after printing all the zombies' positions, calls `System.out.flush()` (line 138). Why? Because if it didn't call `flush()`, the snapshot printouts would get stored in an internal buffer and would be printed all at once at the end of the program. This is not what I want. I want each snapshot to appear as soon as its time step has finished, so I can tell that the program is making progress. Calling `flush()` makes the printout appear on the standard output stream immediately.

Here are the sequential and parallel zombie programs' running times on a four-core `tardis` node for $N = 200$ zombies:

```
$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSeq \
  142857 200 5 0.5 10 0.00001 0.001 0 0 \
  > /var/tmp/ark/zom200_0.txt
Job 14 makespan 84670 msec
$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSmp \
  142857 200 5 0.5 10 0.00001 0.001 0 0 \
  > /var/tmp/ark/zom200_4.txt
Job 15 makespan 23566 msec
```

The program required 10653 time steps to reach the equilibrium state. Thus, the program had to do 10653×200^2 = over 426 million calculations (executions of the inner loop body).

The speedup factor was $84640 \div 23566 = 3.592$, not that good. Both the Mandelbrot Set program and the zombie program experienced less-than-ideal speedups when run on four cores. We'll explore the reasons for this in the next two chapters.

Under the Hood

The outer non-parallel for loop at line 66 has a nested parallel for loop at line 71. This means that *on every outer loop iteration,* the program needs a multiple-thread team to execute the parallel for loop. If these threads were created and destroyed each time through the outer loop, the thread creation and destruction overhead would severely reduce the program's performance.

Instead, the Parallel Java 2 Library uses a *thread pool* under the hood. Initially, the pool is empty. When a thread team is needed, the program first

```
117         // Stop when position delta is less than convergence
118         // threshold or when the specified number of time steps
119         // have occurred.
120         if ((steps == 0 && delta.item < eps) ||
121            (steps != 0 && t == steps))
122               break;
123
124         // Snapshot all bodies' positions every <snap> time steps.
125         if (snap > 0 && (t % snap) == 0) snapshot (t);
126         }
127
128      // Snapshot all bodies' final positions.
129      snapshot (t);
130      }
131
132   // Snapshot all bodies' positions.
133   private void snapshot
134      (int t)
135      {
136      for (int i = 0; i < N; ++ i)
137         System.out.printf ("%d\t%d\t%g\t%g%n", t, i, x[i], y[i]);
138      System.out.flush();
139      }
140
141   // Print a usage message and exit.
142   private static void usage()
143      {
144      System.err.println ("Usage: java pj2 " +
145         "edu.rit.pj2.example.ZombieSmp <seed> <N> <W> <G> <L> " +
146         "<dt> <eps> <steps> <snap>");
147      System.err.println ("<seed> = Random seed");
148      System.err.println ("<N> = Number of bodies");
149      System.err.println ("<W> = Region size");
150      System.err.println ("<G> = Attraction factor");
151      System.err.println ("<L> = Attraction length scale");
152      System.err.println ("<dt> = Time step size");
153      System.err.println ("<eps> = Convergence threshold");
154      System.err.println ("<steps> = Number of time steps (0 = " +
155          "until convergence");
156      System.err.println ("<snap> = Snapshot interval (0 = none)");
157      throw new IllegalArgumentException();
158      }
159   }
```

Listing 8.1. ZombieSmp.java (part 3)

tries to find a team in the pool that has the required number of threads. If such a team is in the pool, the program reuses that team. Only if there is no suitable team in the pool does the program create a new team with new threads. When a team finishes executing a group of parallel sections or a parallel for loop, the program puts the team back in the pool so the team can be reused later. This minimizes the overhead when repeatedly executing parallel sections or parallel for loops.

Points to Remember

- Carefully consider whether each loop in the program has sequential dependencies.
- A sequential dependency exists when a later loop iteration needs to use results computed in an earlier loop iteration.
- If a loop has sequential dependencies, the loop must remain a regular non-parallel loop.
- If a loop does not have sequential dependencies, the loop can be converted into a parallel loop.
- Parallel sections, regular loops, and parallel loops can be nested together, following the algorithm's natural structure.
- The implicit barrier at the end of each group of parallel sections and at the end of each parallel for loop ensures synchronization between the team threads executing the parallel code and the thread executing the enclosing sequential code.

Chapter 9
Strong Scaling

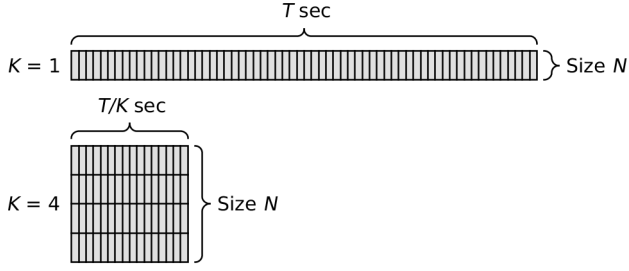
We've been informally observing the preceding chapters' parallel programs' running times and calculating their speedups. The time has come to formally define several *performance metrics* for parallel programs and discover what these metrics tell us about the parallel programs. First, some terminology.

A program's *problem size N* denotes the amount of computation the program has to do. The problem size depends on the nature of the problem. For the bitcoin mining program, the problem size is the expected number of nonces examined, $N = 2^n$, where n is the number of leading zero bits in the digest. For the π program, the problem size is the number of darts thrown at the dartboard. For the Mandelbrot Set program, the problem size is the total number of inner loop iterations needed to calculate all the pixels. For the zombie program, the problem size is $N = s \cdot n^2$, where n is the number of zombies and s is the number of time steps (n^2 is the number of inner loop iterations in each time step).

The program is run on a certain number of processor *cores K*. The sequential version of the program is, of course, run with $K = 1$. The parallel version can be run with K = any number of cores, from one up to the number of cores in the machine.

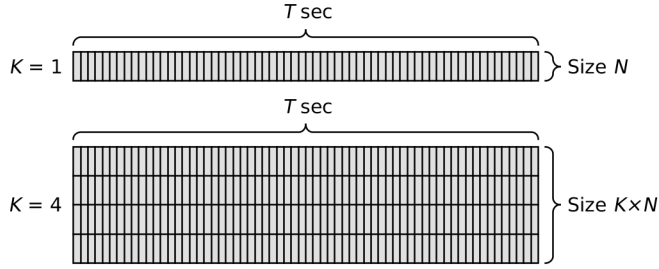
The program's *running time T* is the wall clock time the program takes to run from start to finish. We write $T(N,K)$ to emphasize that the running time depends on the problem size and the number of cores. We write $T_{\text{seq}}(N,K)$ for the sequential version's running time and $T_{\text{par}}(N,K)$ for the parallel version's running time. The problem size is supposed to be defined so that T is directly proportional to N.

Scaling refers to running the program on increasing numbers of cores. There are two ways to scale up a parallel program onto more cores: *strong scaling* and *weak scaling*. With strong scaling, as the number of cores increases, the problem size stays the same. This means that the program should ideally take $1/K$ the amount of time to compute the answer for the same problem—a *speedup*.



With weak scaling, as the number of cores increases, the problem size is also increased in direct proportion to the number of cores. This means that the program should ideally take the same amount of time to compute the an-

swer for a K times larger problem—a *sizeup*.



So far, I've been doing strong scaling on the `tardis` machine. I ran the sequential version of each program on one core, and I ran the parallel version on four cores, for the same problem size. In this chapter we'll take an in-depth look at strong scaling. We'll look at weak scaling in the next chapter.

The fundamental quantity of interest for assessing a parallel program's performance, whether doing strong scaling or weak scaling, is the *computation rate* (or *computation speed*), denoted $R(N,K)$. This is the amount of computation per second the program performs:

$$R(N,K) = \frac{N}{T(N,K)} \quad . \tag{9.1}$$

Speedup is the chief metric for measuring strong scaling. Speedup is the ratio of the parallel program's speed to the sequential program's speed:

$$Speedup(N,K) = \frac{R_{\text{par}}(N,K)}{R_{\text{seq}}(N,1)} \quad . \tag{9.2}$$

Substituting equation (9.1) into equation (9.2), and noting that with strong scaling N is the same for both the sequential and the parallel program, speedup turns out to be the sequential program's running time on one core divided by the parallel program's running time on K cores:

$$Speedup(N,K) = \frac{T_{\text{seq}}(N,1)}{T_{\text{par}}(N,K)} \quad . \tag{9.3}$$

Note that the numerator is the *sequential* version's running time on one core, not the parallel version's. When running on one core, we'd run the sequential version to avoid the parallel version's extra overhead.

Ideally, the speedup should be equal to K, the number of cores. If the parallel program takes more time than the ideal, the speedup will be less than K. *Efficiency* is a metric that tells how close to ideal the speedup is:

$$Eff(N,K) = \frac{Speedup(N,K)}{K} \quad . \tag{9.4}$$

If the speedup is ideal, the efficiency is 1, no matter how many cores the program is running on. If the speedup is less than ideal, the efficiency is less than 1.

I measured the running time of the MandelbrotSeq and MandelbrotSmp programs from Chapter 7 on the `tardis` machine. (The parallel version is the one *without* overlapping.) I measured the program for five image dimensions from 3,200×3,200 pixels to 12,800×12,800 pixels. For each image size, I measured the sequential version running on one core and the parallel version running on one through four cores. Here is one of the commands:

```
$ java pj2 threads=2 edu.rit.pj2.example.MandelbrotSmp \
  3200 3200 -0.75 0 1200 1000 0.4 ms3200.png
```

The `pj2` program's `threads=2` option specifies that parallel for loops in the program should run on two cores (with a team of two threads). This overrides the default, which is to run on all the cores in the machine.

To reduce the effect of random timing fluctuations on the metrics, I ran the program three times for each N and K, and I took the program's running time T to be the *smallest* of the three measured values. Why take the smallest, rather than the average? Because the larger running times are typically due to other processes on the machine stealing some CPU cycles away from the parallel program, causing the parallel program's wall clock time to increase. The best estimate of the parallel program's true running time would then be the smallest measured time, the one where the fewest CPU cycles were stolen by other processes.

Figure 9.1 shows the running time data I measured, along with the calculated speedup and efficiency metrics. (K = "seq" refers to the sequential version.) Figure 9.1 also plots the parallel version's running time and efficiency versus the number of cores. The running times are shown on a *log-log plot*—both axes are on a logarithmic scale instead of the usual linear scale. Why? Because under strong scaling the running time is ideally supposed to be proportional to K^{-1}. On a log-log plot, K^{-1} shows up as a straight line with a slope of –1. Eyeballing the log-log running time plot gives a quick indication of whether the program is experiencing ideal speedups.

The efficiency plots show that the parallel Mandelbrot Set program's performance progressively degrades as we scale up to more cores. On four cores, the speedup is only about 3.5, and the efficiency is only about 0.89. Why is this happening? The answer will require some analysis.

In a 1967 paper,* Gene Amdahl pointed out that usually only a portion of a program can be parallelized. The rest of the program has to run on a single core, no matter how many cores the machine has available. As a result, there

* G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pages 483–485.

Image	*K*	*T (msec)*	*Speedup*	*Eff*
3200	seq	16979		
	1	16817	1.010	1.010
	2	8832	1.922	0.961
	3	6180	2.747	0.916
	4	4854	3.498	0.874
4525	seq	33813		
	1	33494	1.010	1.010
	2	17545	1.927	0.964
	3	12230	2.765	0.922
	4	9557	3.538	0.885
6400	seq	67418		
	1	66857	1.008	1.008
	2	34880	1.933	0.966
	3	24275	2.777	0.926
	4	18943	3.559	0.890
9051	seq	134401		
	1	133371	1.008	1.008
	2	69569	1.932	0.966
	3	48289	2.783	0.928
	4	37684	3.567	0.892
12800	seq	269634		
	1	267580	1.008	1.008
	2	139984	1.926	0.963
	3	97495	2.766	0.922
	4	76209	3.538	0.885

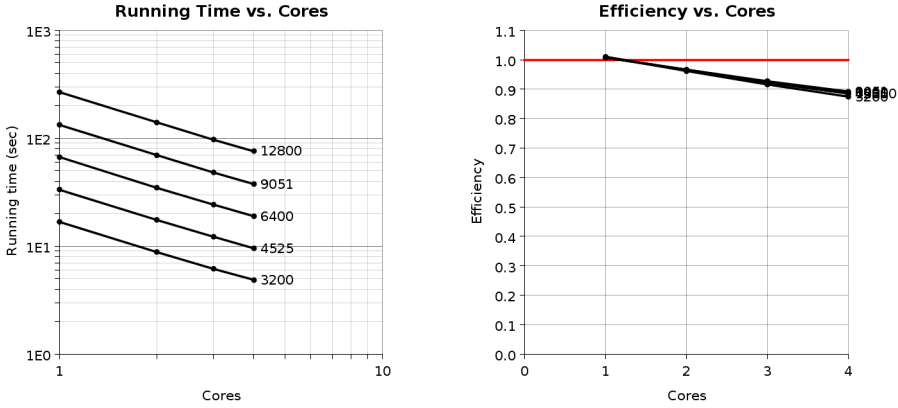


Figure 9.1. MandelbrotSmp strong scaling performance metrics

is a limit on the achievable speedup as we do strong scaling.

Figure 9.2 shows what happens. The program's *sequential fraction F* is the fraction of the total running time that must be performed in a single core. When run on one core, the program's total running time is *T*, the program's *sequential portion* takes time *FT*, and the program's *parallelizable portion* takes time $(1 - F)T$. When run on *K* cores, the parallelizable portion experiences an ideal speedup and takes time $(1 - F)T/K$, but the sequential portion still takes time *FT*. The formula for the program's total running time on *K* cores is therefore

$$T(N,K) = F\,T(N,1) + \frac{1-F}{K}\,T(N,1) \ . \tag{9.5}$$

Equation (9.5) is called *Amdahl's Law*, in honor of Gene Amdahl.

From Amdahl's Law and equations (9.3) and (9.4) we can derive formulas for a program's speedup and efficiency under strong scaling:

$$Speedup(N,K) = \frac{1}{F + \dfrac{1-F}{K}} \ ; \tag{9.6}$$

$$Eff(N,K) = \frac{1}{KF + 1 - F} \ . \tag{9.7}$$

Now consider what happens to the speedup and efficiency as we do strong scaling. In the limit as *K* goes to infinity, the speedup goes to $1/F$. It's not possible to achieve a speedup larger than that under strong scaling. Also, in the limit as *K* goes to infinity, the efficiency goes to zero. As *K* increases, the efficiency just keeps decreasing.

But how quickly do the program's speedup and efficiency degrade as we add cores? That depends on the program's sequential fraction. Figure 9.3 shows the speedups and efficiencies predicted by Amdahl's Law for various sequential fractions. Even with a sequential fraction of just 1 or 2 percent, the degradation is significant. A parallel program will not experience good speedups under strong scaling unless its sequential fraction is very small. *It's critical to design a parallel program so that F is as small as possible.*

Compare the Mandelbrot Set program's measured performance in Figure 9.1 with the theoretically predicted performance in Figure 9.3. You can see that Amdahl's Law explains the observed performance: the degradation is due to the program's sequential fraction. What is its sequential fraction? That will require some further analysis.

While Amdahl's Law gives insight into why a parallel program's performance degrades under strong scaling, I have not found Amdahl's Law to be of much use for analyzing a parallel program's actual measured running time data. A more complicated model is needed. I use this one:

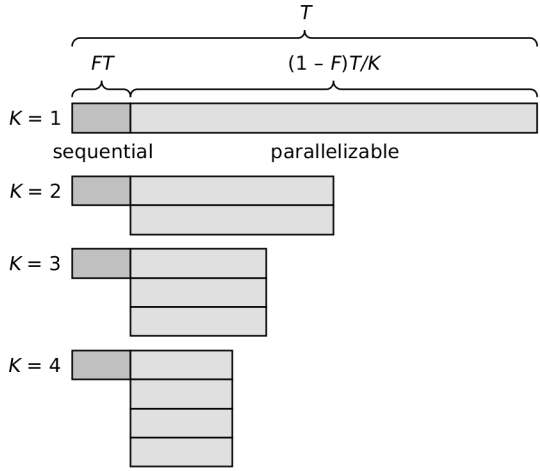


Figure 9.2. Strong scaling with a sequential fraction

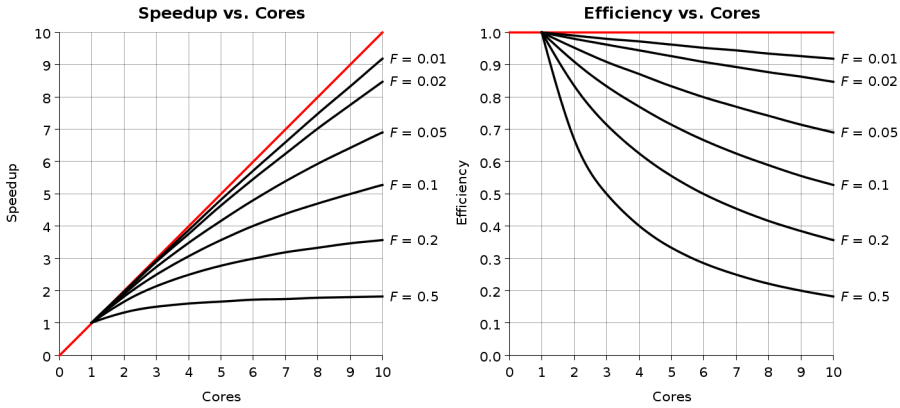


Figure 9.3. Speedup and efficiency predicted by Amdahl's Law

$$T(N,K) = (a + bN) + (c + dN)K + (e + fN)\frac{1}{K} \quad . \qquad (9.8)$$

The first term in equation (9.8) represents the portion of the running time that stays constant as K increases; this is the sequential portion of Amdahl's Law. The second term represents the portion of the running time that increases as K increases, which is not part of Amdahl's Law; this portion could be due to per-thread overhead. The third term represents the portion of the running time that decreases (speeds up) as K increases; this is the parallelizable portion of Amdahl's Law. Each of the three terms is directly proportional to N, plus a constant. The quantities a, b, c, d, e, and f are the *model parameters* that characterize the parallel program.

We are given a number of N, K, and T data samples for the parallel program (such as in Figure 9.1). Our task is to *fit the model to the data*—that is, to find values for the model parameters a through f such that the T values predicted by the model match the measured T values as closely as possible. A program in the Parallel Java 2 Library implements this fitting procedure.

To carry out the fitting procedure on the Mandelbrot Set program's data, I need to know the problem size N for each data sample. N is the number of times the innermost loop is executed while calculating all the pixels in the image. I modified the Mandelbrot Set program to count and print out the number of inner loop iterations; these were the results:

Image	*N*
3200	2.23×10^{9}
4525	4.47×10^{9}
6400	8.93×10^{9}
9051	1.79×10^{10}
12800	3.57×10^{10}

Then running the Mandelbrot Set program's data through the fitting procedure yields these model parameters:

$a = 0$	$c = 0$	$e = 0$
$b = 3.41\times10^{-10}$	$d = 4.39\times10^{-13}$	$f = 7.14\times10^{-9}$

In other words, the Mandelbrot Set program's running time (in seconds) is predicted to be

$$T = 3.41\times10^{-10}\,N + 4.39\times10^{-13}\,N{\cdot}K + 7.14\times10^{-9}\,N{\div}K\,. \qquad (9.9)$$

Finally, we can determine the Mandelbrot Set program's sequential fraction, F. It is the time taken by the first term in equation (9.9), $3.41\times10^{-10}\,N$, divided by the total running time on one core. For image size 3200×3200 pixels, plugging in $N = 2.23\times10^{9}$ and $T = 16.818$ seconds yields $F = 0.045$. (The other image sizes yield the same F.) That's a fairly substantial sequential fraction. Where does it come from? It comes from the code outside the

parallel loop—the initialization code at the beginning, and the PNG file writing code at the end—which is executed by just one thread, no matter how many cores the program is running on.

I improved the Mandelbrot Set program's efficiency by overlapping the computation with the file I/O. This had the effect of reducing the sequential fraction: the program spent less time in the sequential I/O section relative to the parallel computation section. When F went down, the efficiency went up, as predicted by Amdahl's Law.

The Mandelbrot Set program's running time model gives additional insights into the program's behavior. The parameter $d = 4.39\times10^{-13}$ shows that there is a very slight but nonzero increase in the running time as the number of cores increases. This is likely due to thread synchronization overhead: the more cores there are, the more threads there are, and the greater the overhead when these threads synchronize with each other as they access the shared image queue. Still, this increases the running time by only a fraction of a second. The parameter $f = 7.14\times10^{-9}$ shows that one iteration of the inner loop takes 7.14 nanoseconds.

In contrast to the Mandelbrot Set program which has a substantial sequential fraction due to file I/O, the π estimating program from Chapter 4 ought to have a very small sequential fraction. The sequential portion consists of reading the command line arguments, setting up the parallel loop, doing the reduction at the end of the parallel loop, and printing the answer. These should not take a lot of time relative to the time needed to calculate billions and billions of random dart coordinates.

I measured the running time of the PiSeq and PiSmp programs from Chapter 4 on the `tardis` machine for five problem sizes, from 4 billion to 64 billion darts. Figure 9.4 shows the data and the calculated speedups and efficiencies. The speedups and efficiencies, although less than ideal, show almost no degradation as the number of cores increases, evincing a very small sequential fraction. The parallel π program's running time model parameters are

$a = 0$	$c = 0$	$e = 0$
$b = 1.17\times10^{-10}$	$d = 0$	$f = 1.99\times10^{-8}$

From the model, the parallel π program's sequential fraction is determined to be $F = 0.006$.

Why were the π estimating program's speedups and efficiencies less than ideal? I don't have a good answer. I suspect that the Java Virtual Machine's just in time (JIT) compiler is responsible. On some programs, the JIT compiler seemingly does a better job of optimizing the sequential version, so the sequential version's computation rate ends up faster than the parallel version's, so the speedups and efficiencies end up less than ideal. On other programs, it's the other way around, so the speedups and efficiencies end up bet-

ter than ideal. But I have no insight into what makes the JIT compiler optimize one version better than another—assuming the JIT compiler is in fact responsible.

Amdahl's Law places a limit on the speedup a parallel program can achieve under strong scaling, where we run the same problem on more cores. If the sequential fraction is substantial, the performance limit is severe. But strong scaling is not the only way to do scaling. In the next chapter we'll study the alternative, weak scaling.

Under the Hood

The plots in Figures 9.1 and 9.4 were generated by the ScalePlot program in the Parallel Java 2 Library. The ScalePlot program uses class edu.rit.numeric.Plot in the Parallel Java 2 Library, which can generate plots of all kinds directly from a Java program.

The ScalePlot program uses class edu.rit.numeric.NonNegativeLeastSquares to fit the running time model parameters in equation (9.8) to the running time data. This class implements a "non-negative least squares" fitting procedure. This procedure finds the set of model parameter values that minimizes the sum of the squares of the errors between the model's predicted T values and the actual measured T values. However, the procedure will not let any parameter's value become negative. If a parameter "wants" to go negative, the procedure freezes that parameter at 0 and finds the best fit for the remaining parameters.

Points to Remember

- Problem size N is the amount of computation a parallel program has to do to solve a certain problem.
- The program's running time T depends on the problem size N and on the number of cores K.
- Strong scaling is where you keep N the same while increasing K.
- Amdahl's Law, equation (9.5), predicts a parallel program's performance under strong scaling, given its running time on one core $T(N,1)$, its sequential fraction F, and the number of cores K.
- The maximum achievable speedup under strong scaling is 1/F.
- Design a parallel program so that F is as small as possible.
- Measure a parallel program's running time T for a range of problem sizes N and numbers of cores K.
- For each N and K, run the program several times and use the smallest T value.
- Calculate speedup and efficiency using equations (9.4) and (9.5).

N	*K*	*T (msec)*	*Speedup*	*Eff*
4 B	seq	76285		
	1	79930	0.954	0.954
	2	40062	1.904	0.952
	3	26784	2.848	0.949
	4	20564	3.710	0.927
8 B	seq	152528		
	1	159827	0.954	0.954
	2	80112	1.904	0.952
	3	55779	2.735	0.912
	4	40302	3.785	0.946
16 B	seq	305009		
	1	319612	0.954	0.954
	2	160956	1.895	0.947
	3	108029	2.823	0.941
	4	80443	3.792	0.948
32 B	seq	609980		
	1	639162	0.954	0.954
	2	321885	1.895	0.948
	3	213853	2.852	0.951
	4	160964	3.790	0.947
64 B	seq	1220150		
	1	1278245	0.955	0.955
	2	646776	1.887	0.943
	3	427829	2.852	0.951
	4	327522	3.725	0.931

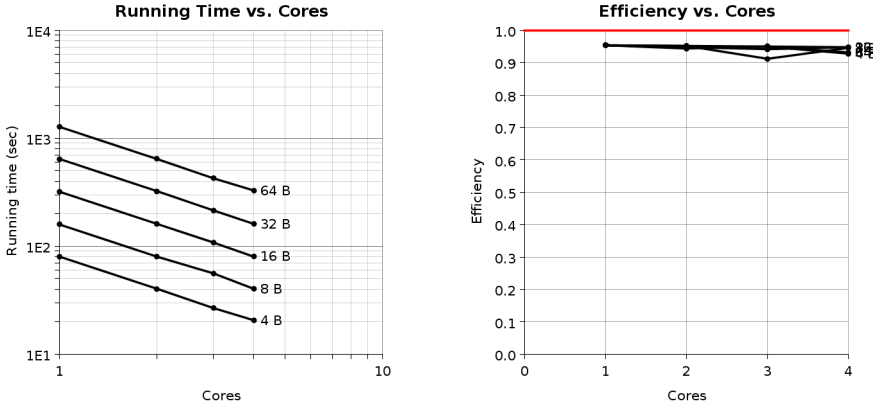


Figure 9.4. PiSmp strong scaling performance metrics

- Use the running time model in equation (9.8) to gain insight into a parallel program's performance based on its measured running time data, and to determine the program's sequential fraction.

Chapter 10
Weak Scaling

Gene Amdahl pointed out the limit on a parallel program's speedup as it runs on more cores while solving the same problem—1/*F,* the reciprocal of the program's sequential fraction. Twenty years later, in a 1988 paper,* John Gustafson pointed out that there's another way to scale up a parallel program to run on more cores: increase the size of the problem being solved as you increase the number of cores. This is known as *weak scaling.* The idea is that often you don't want to solve the same problem more quickly; rather, you want to solve a larger problem in the same time.

For example, suppose a weather prediction program covers a 100×100 kilometer region of the earth's surface at a resolution of 0.1 kilometer, requiring calculations on a 1000×1000 grid. Suppose the program takes 16 hours on one core. All of a sudden, you get a parallel computer with 16 cores. With the larger machine:

- You can solve the same 100×100-kilometer region at the same 0.1-kilometer resolution in one hour (strong scaling).
- You can solve a larger 400×400-kilometer region at the same 0.1-kilometer resolution in the same 16 hours (weak scaling to a 4000×4000 grid, one with 16 times as many grid elements).
- You can solve the same 100×100-kilometer region at a finer 0.025-kilometer resolution in the same 16 hours (also weak scaling to a 4000×4000 grid).

The last alternative, yielding a more accurate (higher resolution) forecast in the same time, might be preferable.

Gustafson asserted that as the problem size increases, it is the program's parallelizable portion's running time that increases; the program's sequential portion's running time typically remains the same. Consequently, as the number of cores increases and the problem size also increases proportionately, the sequential portion occupies less and less of the program's total running time, so the program's speedup continually increases without hitting the limit imposed by Amdahl's Law (which applies only to strong scaling).

However, in my experience, the program's sequential portion's running time does *not* always stay the same. Rather, the running time of at least some of the sequential portion also increases as the problem size increases. This is often the case when the sequential portion consists of I/O. As we will see, a formula that tries to predict weak scaling performance has to take this into account.

The fundamental quantity of interest for assessing a parallel program's performance, whether doing strong scaling or weak scaling, is the *computation rate,* the ratio of problem size to running time. But now under weak scaling, the problem size is not going to stay the same; rather, as the number of

* J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM,* 31(5):532–533, May 1988.

cores increases, the problem size will also increase. We will write $N(K)$ to emphasize this dependence. Then the computation rate is

$$R(N,K) = \frac{N(K)}{T(N(K),K)} \ . \tag{10.1}$$

Sizeup is the chief metric for measuring weak scaling. Sizeup is the ratio of the parallel program's computation rate to the sequential program's computation rate:

$$Sizeup(N,K) = \frac{R_{\text{par}}(N,K)}{R_{\text{seq}}(N,K)} \ . \tag{10.2}$$

Substituting equation (10.1) into equation (10.2) yields an alternate formula for sizeup:

$$Sizeup(N,K) = \frac{N(K)}{N(1)} \times \frac{T_{\text{seq}}(N(1),1)}{T_{\text{par}}(N(K),K)} \ . \tag{10.3}$$

Note that the numerator involves the *sequential* version's running time on one core, not the parallel version's. When running on one core, we'd run the sequential version to avoid the parallel version's extra overhead.

Compare equation (10.3) for sizeup under weak scaling with equation (9.3) for speedup under strong scaling. If the parallel program's problem size is the same as the sequential program's problem size, which is the case under strong scaling, then the first factor in equation (10.3) becomes unity, and the sizeup formula reduces to the speedup formula. Sizeup and speedup are measuring essentially the same thing—the ratio of computation rates—but in different contexts.

Suppose the parallel program's problem size running on K cores is K times larger than the sequential program's problem size running on one core. Ideally, then, the parallel and sequential running times should be the same, and the sizeup should be equal to K, the number of cores. *Efficiency* is a metric that tell how close to ideal the sizeup is:

$$Eff(N,K) = \frac{Sizeup(N,K)}{K} \ . \tag{10.4}$$

(This is the same as the strong scaling efficiency metric.) If the sizeup is ideal, the efficiency is 1, no matter how many cores the program is running on. If the sizeup is less than ideal, the efficiency is less than 1.

Let's derive a formula to predict a program's running time under weak scaling. We'll assume that N increases in direct proportion to K: $N(K) = K \cdot N(1)$. Figure 10.1 shows what happens when we do weak scaling on a program that has a sequential portion. The program's *sequential fraction F* is the fraction of the total running time that must be performed in a single core *and*

that remains constant as the problem size N increases. The program's *weak sequential fraction* G is the fraction of the total running time that must be performed in a single core *and* that increases in proportion to N. When run on one core, the program's total running time is T, the program's *sequential portion* takes time $FT + GT$, and the program's *parallelizable portion* takes time $(1 - F - G)T$. When run on K cores, N increases by a factor of K, but the parallelizable portion speeds up by a factor of K, so the parallelizable portion takes the same time as before, $(1 - F - G)T$. The constant piece of the sequential portion also takes the same time as before, FT. However, the other piece of the sequential portion increases in proportion to N, and N increased by a factor of K, so this piece takes time KGT. The parallel program's running time is therefore

$$\begin{aligned} T(KN, K) &= (F + KG + 1 - F - G)\, T(N, 1) \\ &= (KG + 1 - G)\, T(N, 1) \end{aligned} . \tag{10.5}$$

Equation (10.5) is called the *Weak Scaling Law* (not in honor of anyone).

From the Weak Scaling Law and Equations (10.3) and (10.4) we can derive formulas for a program's sizeup and efficiency under weak scaling:

$$Sizeup(N, K) = \frac{1}{G + \dfrac{1 - G}{K}} ; \tag{10.6}$$

$$Eff(N, K) = \frac{1}{KG + 1 - G} . \tag{10.7}$$

These formulas are the same as the ones for speedup and efficiency under strong scaling, except these formulas depend on G, the weak sequential fraction—namely, just the piece of the sequential portion whose running time increases in proportion to N.

If $G = 0$, then the sizeup is always equal to K and the efficiency is always 1, no matter how many cores K the program runs on. (This is what Gustafson assumed.) But if $G > 0$, then the sizeup approaches a limit of $1/G$ and the efficiency approaches a limit of 0 as K increases—the same as the speedup and efficiency limits under strong scaling. Again, i*t's critical to design a parallel program so that the sequential fraction is as small as possible.*

For an example of weak scaling, I'll use the N-body zombie program from Chapter 8. It's easy to do a strong scaling study; just pick a problem size, then run the same problem with increasing numbers of cores. But a weak scaling study takes a bit more thought to decide how to increase the problem size in proportion to the number of cores.

In the zombie program, the problem size—the amount of computation—depends on the number of zombies n and the number of time steps s. At each time step there are n^2 computations (iterations of the inner loop), leading to

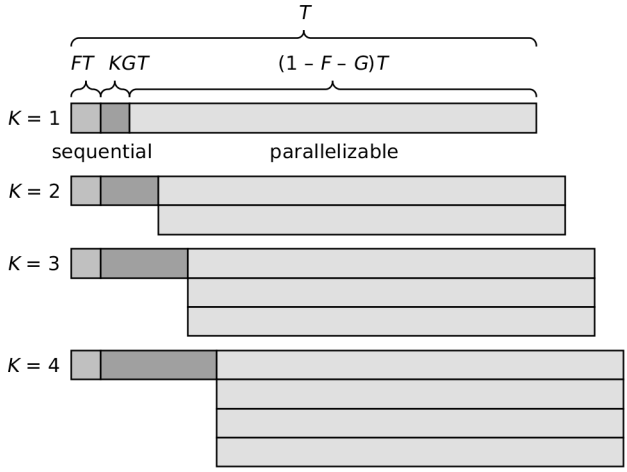


Figure 10.1. Weak scaling with a sequential fraction

the formula $N = s \cdot n^2$. While I can't specify the number of time steps to reach equilibrium, I can specify the number of zombies. I decided to make the number of zombies depend on the number of cores: $n = n(K)$. I decided to pick $n(K)$ such that the amount of computation in one time step when running on K cores is K times the amount of computation in one time step when running on one core: $n(K)^2 = K \cdot n(1)^2$; that is, $n(K) = K^{1/2} \cdot n(1)$. In other words, as the number of cores increases, the number of zombies goes up as the square root of K. To determine the problem size, I still needed s, the number of time steps; I discovered that by running the program.

I also had to pick W, which specifies the initial region within which the zombies are randomly placed. The initial region is a square of side W, so the initial area is W^2. I decided to increase the initial area in proportion to the number of zombies; this yields the same average initial zombie density in all cases. Mathematically, $W(K)^2 \div W(1)^2 = n(K) \div n(1)$, or $W(K) = W(1) \cdot n(K)^{1/2} \div n(1)^{1/2}$. For example, here are the parameters for $n(1) = 200$, $W(1) = 5$, and $K = 1$ to 4:

K	$n(K)$	$W(K)$
1	200	5.00
2	283	5.95
3	346	6.58
4	400	7.07

I measured the running time of the ZombieSeq and ZombieSmp pro-

grams from Chapter 8 on one node of the `tardis` machine. I measured the program for five problem sizes: $n(1)$ = 200, 300, 400, 600, and 800. When I ran the parallel version on K cores, I increased n and W as described previously. Here is one of the commands:

```
$ java pj2 debug=makespan threads=2 \
  edu.rit.pj2.example.ZombieSmp 142857 283 5.95 0.5 10.0 \
  0.00001 0.001 0
```

Figure 10.2 shows the problem size and running time data I measured, along with the calculated sizeup and efficiency metrics. (K = "seq" refers to the sequential version.) Figure 10.2 also plots the parallel version's running times and efficiencies versus the number of cores. The running times for each problem size are more or less constant, as they should be under weak scaling. The efficiency plot's vertical scale is exaggerated, so as to show a slight degradation in the efficiencies (and the sizeups too) as the number of cores increases. What is causing the degradation?

Recall the parallel program running time model from Chapter 9:

$$T(N,K) = (a + bN) + (c + dN)K + (e + fN)\frac{1}{K} . \qquad (10.8)$$

I fit the model to the data to determine the zombie program's model parameters, which turned out to be

$a = 0$	$c = 0.412$	$e = 0.615$
$b = 0$	$d = 4.84\times10^{-11}$	$f = 2.01\times10^{-7}$

The zombie program's efficiency degradation is not coming from a sequential fraction; a and b are both zero. Rather, the degradation is coming from the second term in equation (10.8), $(c + dN)K$. As the number of cores K increases, and as the problem size N increases along with K, that second term causes the running time to increase; hence, the sizeup and efficiency diminish. What in the program is responsible for causing the running time to increase with the number of cores? Most likely, it is the thread synchronization at the beginning and end of the middle parallel for loop, including the parallel reduction. As the number of threads (cores) increases, so does the time spent in thread synchronization. Still, the effect is very slight.

Notice also the third term in equation (10.8). This represents the bulk of the computation (the parallelizable portion). The third term's coefficient f is several orders of magnitude larger than the second term's coefficient d. Therefore, as the problem size N increases, the third term grows much faster than the second term; the second term accounts for a decreasing fraction of the total running time; and the efficiency improves. On four cores, the efficiency is 0.955 for n = 200 zombies, but the efficiency is 0.987 for n = 800 zombies. Weak scaling often has this beneficial effect; as the program scales up to more cores, increasing the amount of computation (the problem size)

n(1)	*K*	*n*	*s*	*N*	*T (msec)*	*Sizeup*	*Eff*
200	seq	200	10653	4.26×10^{8}	83587		
	1	200	10653	4.26×10^{8}	85614	0.976	0.976
	2	283	9665	7.74×10^{8}	78442	1.936	0.968
	3	346	9329	1.12×10^{9}	76175	2.876	0.959
	4	400	9048	1.45×10^{9}	74362	3.819	0.955
300	seq	300	9488	8.54×10^{8}	169369		
	1	300	9488	8.54×10^{8}	171264	0.989	0.989
	2	424	8911	1.60×10^{9}	161231	1.971	0.985
	3	520	8438	2.28×10^{9}	153812	2.942	0.981
	4	600	8116	2.92×10^{9}	148766	3.895	0.974
400	seq	400	9048	1.45×10^{9}	288834		
	1	400	9048	1.45×10^{9}	291031	0.992	0.992
	2	566	8255	2.64×10^{9}	267765	1.970	0.985
	3	693	8016	3.85×10^{9}	258672	2.969	0.990
	4	800	7912	5.06×10^{9}	256894	3.933	0.983
600	seq	600	8116	2.92×10^{9}	583214		
	1	600	8116	2.92×10^{9}	588350	0.991	0.991
	2	849	7897	5.69×10^{9}	573857	1.980	0.990
	3	1039	7983	8.92×10^{9}	581143	2.960	0.987
	4	1200	7986	1.15×10^{10}	581139	3.950	0.987
800	seq	800	7912	5.06×10^{9}	1008586		
	1	800	7912	5.06×10^{9}	1018024	0.991	0.991
	2	1131	7934	1.01×10^{10}	1017075	1.988	0.994
	3	1386	8010	1.54×10^{10}	1029579	2.977	0.992
	4	1600	8016	2.05×10^{10}	1035085	3.949	0.987

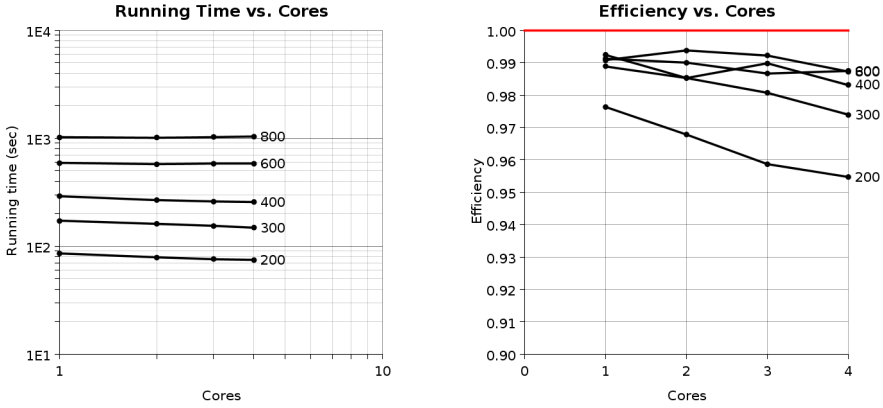


Figure 10.2. ZombieSmp weak scaling performance metrics

often results in higher efficiencies than doing the same amount of computation (strong scaling).

I also studied the performance of the π estimating program under weak scaling. I measured the running time of the PiSeq and PiSmp programs from Chapter 4 on the `tardis` machine for five problem sizes from 4 billion to 64 billion darts on one core. When I ran the parallel version on K cores, I increased the number of darts by a factor of K. Figure 10.3 gives the measured running time data and the calculated performance metrics, along with plots of the running time and efficiency versus the number of cores.

The π program's running time model parameters are

$$a = 0 \qquad c = 0 \qquad e = 0.339$$
$$b = 0 \qquad d = 2.87\times10^{-11} \qquad f = 1.99\times10^{-8}$$

Again, there is a slight degradation in the sizeups and efficiencies due to the second term in the model, $(c + dN)K$. Again, this is most likely due to the thread synchronization at the beginning and end of the parallel for loop, including the parallel reduction.

Under the Hood

The plots in Figures 10.2 and 10.3 and the running time model parameters were generated by the ScalePlot program in the Parallel Java 2 Library. The ScalePlot program uses classes edu.rit.numeric.Plot and edu.rit.numeric.NonNegativeLeastSquares in the Parallel Java 2 Library.

Points to Remember

- Problem size N is the amount of computation a parallel program has to do to solve a certain problem.
- The program's running time T depends on the problem size N and on the number of cores K.
- Weak scaling is where you increase both N and K proportionally.
- The Weak Scaling Law, equation (10.5), predicts a parallel program's performance under weak scaling, given its running time on one core $T(N,1)$, its weak sequential fraction G, and the number of cores K.
- The maximum achievable sizeup under weak scaling is $1/G$.
- Design a parallel program so that the sequential fraction is as small as possible.
- Calculate sizeup and sizeup efficiency using equations (10.3) and (10.4).
- Use the running time model in equation (10.8) to gain insight into a parallel program's performance based on its measured running time data, and to determine the program's sequential fraction.

N(1)	*N*	*K*	*T (msec)*	*Sizeup*	*Eff*
4 billion	4×10^9	seq	76284		
	4×10^9	1	79939	0.954	0.954
	8×10^9	2	80888	1.886	0.943
	12×10^9	3	80249	2.852	0.951
	16×10^9	4	80394	3.796	0.949
8 billion	8×10^9	seq	152543		
	8×10^9	1	159826	0.954	0.954
	16×10^9	2	160136	1.905	0.953
	24×10^9	3	163484	2.799	0.933
	32×10^9	4	160964	3.791	0.948
16 billion	16×10^9	seq	305069		
	16×10^9	1	319612	0.954	0.954
	32×10^9	2	320059	1.906	0.953
	48×10^9	3	320818	2.853	0.951
	64×10^9	4	322548	3.783	0.946
32 billion	32×10^9	seq	608417		
	32×10^9	1	639232	0.952	0.952
	64×10^9	2	643963	1.890	0.945
	96×10^9	3	641472	2.845	0.948
	128×10^9	4	647956	3.756	0.939
64 billion	64×10^9	seq	1220115		
	64×10^9	1	1278302	0.954	0.954
	128×10^9	2	1280819	1.905	0.953
	192×10^9	3	1287311	2.843	0.948
	256×10^9	4	1309942	3.726	0.931

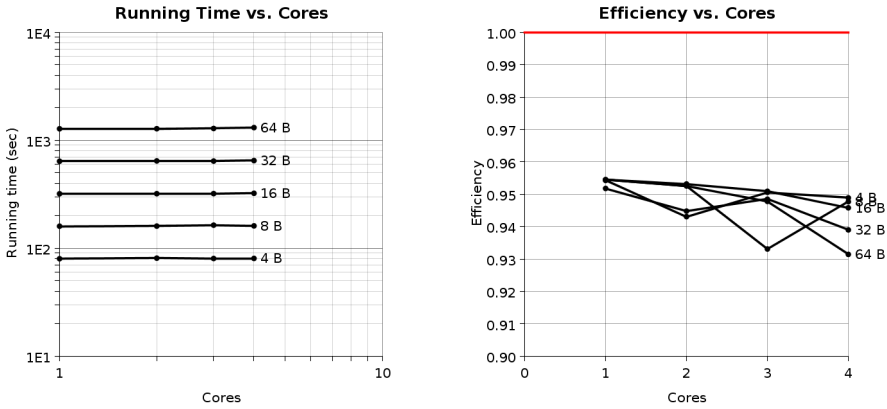
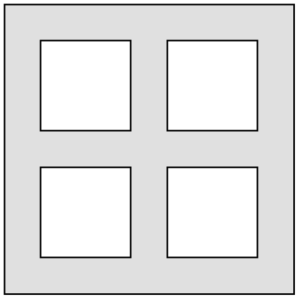


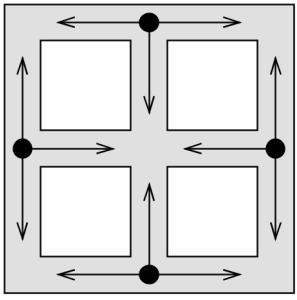
Figure 10.3. PiSmp weak scaling performance metrics

Chapter 11
Exhaustive Search

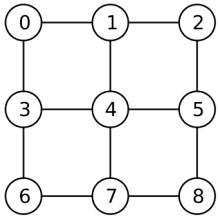
Imagine a building with a number of intersecting straight corridors. To improve security, you want to install surveillance cameras—the black hemispherical ceiling-mounted kind that can scan in all directions—so that you can monitor every corridor. But to save money, you want to install as few cameras as possible while still covering every corridor. Here's the building's floor plan. How many cameras do you need, and where do you put them?



A little thought should convince you that one, two, or even three cameras can't cover all the corridors. But four cameras can do the job, if properly placed:



This is an example of the *Minimum Vertex Cover Problem*, a well-known problem in graph theory. A *graph* is a set of *vertices* plus a set of *edges* connecting pairs of vertices. A graph is often drawn with circles depicting the vertices and lines depicting the edges. Here is a graph representing the building floor plan; the vertices are the corridor intersections, the edges are the corridor segments:



A *vertex cover* of a graph is a subset of the vertices such that every edge in the graph is attached to at least one vertex in the subset. A *minimum vertex cover* is a vertex cover with as few vertices as possible. For the above graph, a minimum vertex cover is the subset {1, 3, 5, 7}.

In this chapter and the next, I'm going to build some parallel programs to solve the Minimum Vertex Cover Problem. These programs will further illustrate features of the Parallel Java 2 Library, including parallel loops and reduction variables. These programs will also illustrate *bitsets*, a technique that utilizes machine instructions, rather than multiple threads, to achieve parallelism.

The first program finds a minimum vertex cover via *exhaustive search.* The program considers every possible subset of the set of V vertices as potential candidates. For each candidate, the program checks whether the candidate is in fact a cover. Of the candidates that are covers, the program keeps whichever one has the fewest elements. At the end, the candidate left standing is a minimum vertex cover. (I say "a" minimum vertex cover because there might be more than one vertex cover with the same minimum size. In that case, I don't care which one the program finds.)

This exhaustive search process is guaranteed to find a minimum vertex cover. However, the program has to look at all subsets of the set of vertices, and there are 2^V possible subsets. For all but the smallest graphs, the program is going to take a long time to run. I'll need to have a speedy algorithm, and then I'll want to run it on a multicore parallel computer. Even so, for graphs with more than around 40 or so vertices, the program is going to take too much time to be useful. Solving the minimum vertex cover problem for larger graphs will require a different approach. (See the next chapter.)

I need a data structure to represent a graph in the program. Like many abstract data types, there are several ways to implement a graph data structure. The appropriate data structure depends on the problem; there is no one-size-fits-all graph data structure. Graph data structures include:

- **Adjacency matrix.** This is a $V \times V$ matrix of 0s and 1s, with rows and columns indexed by the vertices. Matrix element $[r, c]$ is 1 if vertices r and c are adjacent (if there is an edge between vertex r and vertex c); element $[r, c]$ is 0 otherwise.
- **Adjacency list.** This is an array of V lists, indexed by the vertices. Array element $[i]$ is a list of the vertices adjacent to vertex i.
- **Edge list.** This is simply a list of the edges, each edge consisting of a pair of vertices.

As will become apparent shortly, the best data structure for the minimum vertex cover exhaustive search program is the adjacency matrix. Here is the adjacency matrix corresponding to the building floor plan graph:

8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	0	0
0	0	0	0	1	0	1	0	1	1
0	0	0	1	0	0	0	1	0	2
0	0	1	0	1	0	0	0	1	3
0	1	0	1	0	1	0	1	0	4
1	0	0	0	1	0	1	0	0	5
0	1	0	0	0	1	0	0	0	6
1	0	1	0	1	0	0	0	0	7
0	1	0	1	0	0	0	0	0	8

Note that the matrix is symmetric; an edge between vertex *r* and vertex *c* appears both as element [*r, c*] and element [*c, r*]. (The reason I've numbered the columns from right to left will also become apparent shortly.)

Now I need a way to decide if a particular candidate set of vertices is a cover. For example, consider the set {4, 6, 8}. I shade in rows 4, 6, and 8 and columns 4, 6, and 8 of the adjacency matrix:

8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	0	0
0	0	0	0	1	0	1	0	1	1
0	0	0	1	0	0	0	1	0	2
0	0	1	0	1	0	0	0	1	3
0	1	0	1	0	1	0	1	0	4
1	0	0	0	1	0	1	0	0	5
0	1	0	0	0	1	0	0	0	6
1	0	1	0	1	0	0	0	0	7
0	1	0	1	0	0	0	0	0	8

All the 1s in the shaded cells represent edges that are attached to at least one of the vertices in the set {4, 6, 8}. However, there are several 1s in cells that are not shaded; these represent edges that are attached to none of the vertices in the set. Therefore, {4, 6, 8} is not a cover.

On the other hand, the candidate vertex set {1, 3, 5, 7} is a cover, as is apparent from the adjacency matrix—all the 1s are in shaded cells:

8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	0	0
0	0	0	0	1	0	1	0	1	1
0	0	0	1	0	0	0	1	0	2
0	0	1	0	1	0	0	0	1	3
0	1	0	1	0	1	0	1	0	4
1	0	0	0	1	0	1	0	0	5
0	1	0	0	0	1	0	0	0	6
1	0	1	0	1	0	0	0	0	7
0	1	0	1	0	0	0	0	0	8

I need to express this procedure in a way that can be coded in a computer program. I don't need to look at the matrix rows corresponding to vertices in the candidate set (shaded rows); all 1s in those rows are automatically covered. I only need to look at the matrix rows corresponding to vertices *not* in the candidate set (*unshaded* rows). I view each unshaded row as itself a vertex set, with 1s indicating which vertices are in the row set. All the 1s in an unshaded row are covered if the row set is a subset of the candidate set. For example, consider row 0. The row set {1, 3} is a subset of the candidate set {1, 3, 5, 7}; therefore the 1s in row 0 are covered. The same is true of rows 2, 4, 6, and 8. Therefore, {1, 3, 5, 7} is a cover for the whole graph.

So in the program code, I don't want to implement the adjacency matrix as an actual matrix. Rather, I want to implement it as an array of rows, where each row is a vertex set. And I can see that I'll need at least two operations on a vertex set: an operation to determine if a vertex is or is not a member of a vertex set (to decide if a given row is not in the candidate set); and an operation to determine if one vertex set is a subset of another (to decide if a given row set is a subset of the candidate set).

Now I need a way to represent a vertex set in the program. For sets with a limited number of possible elements, a *bitset* implementation is attractive. What's a bitset? Consider the first row of the above adjacency matrix:

0	0	0	0	0	1	0	1	0

Now take away the cell borders:

000001010

This looks suspiciously like a binary number, which is how a computer stores an integer. A bitset uses an integer to indicate which elements are members of the set. Suppose the set can contain n different elements. The elements are numbered from 0 to $n - 1$. The bit positions of the integer are likewise numbered from 0 to $n - 1$, with bit position 0 being the least significant (rightmost) bit and bit position $n - 1$ being the most significant (leftmost) bit. In the integer, bit i is on (1) if element i is a member of the set; bit i is off (0) if element i is not a member of the set. Using type `int`, a bitset can represent a set with up to 32 elements; using type `long`, up to 64 elements.

With a bitset representation, I can do set operations *on all the elements simultaneously* with just one or two integer operations. These usually involve the bitwise Boolean operations on integers—bitwise and (`&`), bitwise or (`|`), bitwise exclusive-or (`^`), bitwise complement (`~`), and bitwise shifts (`<<`, `>>`, `>>>`). In effect, bitset operations utilize the CPU's integer functional unit to manipulate all 32 or 64 set elements in parallel with just a few machine instructions. Consequently, operations on a bitset are quite fast.

The Parallel Java 2 Library includes bitset classes in package edu.rit.util. Class BitSet32 holds up to 32 elements; class BitSet64, up to 64 elements.

The Library also includes bitset reduction variable classes in package edu.rit.pj2.vbl, namely classes BitSet32Vbl and BitSet64Vbl, as well as several subclassess that do various reduction operations on bitsets. For further information about how the bitset classes work, see the "Under the Hood" section below.

To represent a vertex set, my program will use class BitSet64. Thus, the program can handle graphs with as many as $V = 64$ vertices. This is more than large enough to accommodate an exhaustive search program that has to look at 2^V subsets. My program will also use class BitSet64Vbl to do parallel reductions.

Finally, I can start writing some code. Listing 11.1 is the first multicore parallel minimum vertex cover program, MinVerCovSmp. The program begins by reading the graph from a file in this format: The first pair of numbers gives *V,* the number of vertices, and *E,* the number of edges. Each subsequent pair of numbers gives two vertex numbers each in the range 0 through $V - 1$, defining an edge between those vertices. For example, here is the file for the building graph:

```
9 12
0 1 1 2 0 3 1 4
2 5 3 4 4 5 3 6
4 7 5 8 6 7 7 8
```

The program reads the file given on the command line using a Scanner (line 32). After reading *V* and *E,* the program initializes the adjacency matrix to an array of *V* vertex sets (bitsets), all initially empty (lines 36–38). Upon reading each subsequent pair of vertices *a* and *b,* the program turns on elements [*a, b*] and [*b, a*] in the adjacency matrix, thus ensuring the matrix is symmetric (lines 39–45).

The program is now ready to do a parallel loop over all possible subsets of the set of vertices, with each parallel team thread examining a different portion of the subsets. The program will also use parallel reduction. Each team thread will find its own minimum vertex cover among the vertex subsets the team thread examines. When the parallel loop finishes, these per-thread minimum vertex covers will be reduced together into the overall minimum vertex cover.

Following the parallel reduction pattern, the program creates a global reduction variable `minCover` of type BitSet64Vbl.MinSize (line 49). This subclass's reduction operation combines two bitsets (covers) by keeping whichever cover has fewer elements (vertices), which is what I need to do to find a *minimum* cover. The `add(0,V)` method initializes `minCover` to contain all the vertices, from 0 through $V - 1$ (line 50). The set of all vertices is obviously a cover. Any other cover the program finds will have fewer vertices and will replace this initial cover when the reduction happens. The program sets the variable `full` to the bitmap corresponding to this set of all vertices (line 51).

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.LongLoop;
3  import edu.rit.pj2.Task;
4  import edu.rit.pj2.vbl.BitSet64Vbl;
5  import edu.rit.util.BitSet64;
6  import java.io.File;
7  import java.util.Scanner;
8  public class MinVerCovSmp
9     extends Task
10    {
11    // Number of vertices and edges.
12    int V;
13    int E;
14
15    // The graph's adjacency matrix. adjacent[i] is the set of
16    // vertices adjacent to vertex i.
17    BitSet64[] adjacent;
18
19    // Minimum vertex cover.
20    BitSet64Vbl minCover;
21
22    // Main program.
23    public void main
24       (String[] args)
25       throws Exception
26       {
27       // Parse command line arguments.
28       if (args.length != 1) usage();
29       File file = new File (args[0]);
30
31       // Read input file, set up adjacency matrix.
32       Scanner s = new Scanner (file);
33       V = s.nextInt();
34       E = s.nextInt();
35       if (V < 1 || V > 63) usage ("V must be >= 1 and <= 63");
36       adjacent = new BitSet64 [V];
37       for (int i = 0; i < V; ++ i)
38          adjacent[i] = new BitSet64();
39       for (int i = 0; i < E; ++ i)
40          {
41          int a = s.nextInt();
42          int b = s.nextInt();
43          adjacent[a].add (b);
44          adjacent[b].add (a);
45          }
46       s.close();
47
48       // Check all candidate covers (sets of vertices).
49       minCover = new BitSet64Vbl.MinSize();
50       mincover.bitset.add (0, V);
51       long full = minCover.bitset.bitmap();
52       parallelFor (0L, full) .exec (new LongLoop()
53          {
54          BitSet64Vbl thrMinCover;
55          public void start()
56             {
57             thrMinCover = threadLocal (minCover);
58             }
```

Listing 11.1. MinVerCovSmp.java (part 1)

Next the program does a parallel loop over all the bitsets from the empty set (`0L`) to the full set (`full`). Along the way, the loop index visits every possible subset of the set of vertices. For example, with $V = 4$, here are the bitsets the loop visits (in binary):

```
0000    0100    1000    1100
0001    0101    1001    1101
0010    0110    1010    1110
0011    0111    1011    1111
```

This is another reason to use bitsets; it makes looping over every possible subset a simple matter of incrementing an integer loop index. However, I have to be careful. The maximum positive value for a loop index of type `long` is $2^{63} - 1$. For this reason, I had to restrict the number of vertices V to be 63 or less (line 35).

Inside the parallel loop, the program declares a thread-local variable `thrMinCover` of type BitSet64Vbl (line 54) linked to the global reduction variable (line 57). `thrMinCover` will hold the best (smallest-size) cover the parallel team thread has seen so far. In the loop body, the program creates a candidate vertex set from the loop index bitset (line 61). If the candidate is smaller than (has fewer vertices than) the best cover seen so far, and if the candidate is in fact a cover, the program copies the candidate into the thread's minimum cover variable (lines 62–64). Otherwise, the thread's minimum cover variable remains as it was.

Note that on line 62, because of the "short-circuit" semantics of the logical and operator `&&`, if the candidate vertex set is not smaller than the best cover seen so far, the program will not even bother to check whether the candidate is a cover. This saves time.

The program checks whether the candidate is a cover using the procedure described earlier, embodied in the `isCover()` method (lines 78–86). The method uses the bitset's `contains()` and `isSubsetOf()` methods to do its check. If the method discovers the candidate is not a cover, the method stops immediately, which again saves time.

When the parallel loop finishes, each parallel team thread's thread-local minimum cover variable contains the smallest cover among the vertex subsets the team thread examined. The thread-local minimum cover variables are automatically reduced together into the global minimum cover variable. As mentioned before, the reduction operation is to keep the cover that has the fewer elements. The global minimum cover variable ends up holding the smallest cover among all the possible vertex subsets. Finally, the program prints this minimum vertex cover (lines 69–74).

To test the MinVerCovSmp program and measure its performance, I need graphs stored in files to use as input. I wrote a program to create a random graph with a given number of vertices and edges (Listing 11.2). The program is self-explanatory.

```
59          public void run (long elems)
60             {
61             BitSet64 candidate = new BitSet64 (elems);
62             if (candidate.size() < thrMinCover.bitset.size() &&
63                isCover (candidate))
64                thrMinCover.bitset.copy (candidate);
65             }
66          });
67
68       // Print results.
69       System.out.printf ("Cover =");
70       for (int i = 0; i < V; ++ i)
71          if (minCover.bitset.contains (i))
72             System.out.printf (" %d", i);
73       System.out.println();
74       System.out.printf ("Size = %d%n", minCover.bitset.size());
75       }
76
77    // Returns true if the given candidate vertex set is a cover.
78    private boolean isCover
79       (BitSet64 candidate)
80       {
81       boolean covered = true;
82       for (int i = 0; covered && i < V; ++ i)
83          if (! candidate.contains (i))
84             covered = adjacent[i].isSubsetOf (candidate);
85       return covered;
86       }
87
88    // Print an error message and exit.
89    private static void usage
90       (String msg)
91       {
92       System.err.printf ("MinVerCovSmp: %s%n", msg);
93       usage();
94       }
95
96    // Print a usage message and exit.
97    private static void usage()
98       {
99       System.err.println ("Usage: java pj2 " +
100        "edu.rit.pj2.example.MinVerCovSmp <file>");
101      System.err.println ("<file> = Graph file");
102      throw new IllegalArgumentException();
103      }
104 }
```

Listing 11.1. MinVerCovSmp.java (part 2)

```
1  package edu.rit.pj2.example;
2  import edu.rit.util.Random;
3  import edu.rit.util.RandomSubset;
4  public class RandomGraph
5     {
```

Listing 11.2. RandomGraph.java (part 1)

I ran the sequential MinVerCovSeq program on one core and the multicore parallel MinVerCovSmp program on one to four cores of a `tardis` node (strong scaling) and measured the running times. I ran the programs on five random graphs, with $V = 31$ to 35 and $E = 310$ to 350. Here are examples of the minimum vertex cover the programs found for the $V = 31$ test case:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSeq \
  g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23
24 25 26 27 28
Size = 26
Job 3 makespan 289506 msec
$ java pj2 debug=makespan cores=4 \
  edu.rit.pj2.example.MinVerCovSmp g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23
24 25 26 27 28
Size = 26
Job 17 makespan 111606 msec
```

Figure 11.1 plots the program's running times and efficiencies. Fitting the running time model to the measurements gives this formula for the running time T as a function of the problem size N ($= 2^V$) and the number of cores K:

$$T = 2.61\times10^{-8}\,N + (25.8 + 1.10\times10^{-7}\,N)\div K\,. \qquad (11.1)$$

For a given problem size, the first term in the formula represents the program's sequential portion, and the second term represents the parallelizable portion. The sequential fraction is quite large—17 to 19 percent for the runs I measured. This large sequential fraction causes the speedups and efficiencies to droop drastically, as is apparent in Figure 11.1.

This sequential fraction comes from the way the program uses vertex set objects. The first statement in the parallel loop body (line 60) constructs a new instance of class BitSet64 and assigns its reference to the `candidate` local variable. The first term in formula (11.1) is proportional to the number of candidate vertex sets examined, N—because each candidate causes a new object to be constructed. Constructing an instance requires allocating storage from the JVM's heap and setting the new object's fields to their default initial values. Continually constructing new objects takes time. For a graph with 31 vertices, 2^{31}, or over two billion, vertex set objects have to be constructed. Even if the constructor takes only a few nanoseconds, the time spent in the constructor adds up to a noticeable amount.

Furthermore, once the parallel loop body's `run()` method returns, the local `candidate` reference goes away, and the vertex set object becomes garbage. Eventually the heap fills up with garbage objects, and the JVM has to run the garbage collector. This takes more time. Worse, when the JVM runs the garbage collector, the JVM typically suspends execution of other

```
6     // Main program.
7     public static void main
8        (String[] args)
9        {
10       // Parse command line arguments.
11       if (args.length != 3) usage();
12       int V = Integer.parseInt (args[0]);
13       int E = Integer.parseInt (args[1]);
14       long seed = Long.parseLong (args[2]);
15
16       // Validate command line arguments.
17       if (V < 1)
18          usage ("V must be >= 1");
19       int N = V*(V - 1)/2;
20       if (E < 0 || E > N)
21          usage (String.format ("E must be >= 0 and <= %d", N));
22
23       // Set up array of all possible edges on V vertices.
24       long[] edges = new long [N];
25       int i = 0;
26       for (int a = 0; a < V - 1; ++ a)
27          for (int b = a + 1; b < V; ++ b)
28             edges[i++] = ((long)a << 32) | ((long)b);
29
30       // Print out a random subset of the possible edges.
31       Random prng = new Random (seed);
32       RandomSubset subset = new RandomSubset (prng, N, true);
33       System.out.printf ("%d %d%n", V, E);
34       for (i = 0; i < E; ++ i)
35          {
36          long edge = edges[subset.next()];
37          int a = (int)(edge >> 32);
38          int b = (int)(edge);
39          System.out.printf ("%d %d%n", a, b);
40          }
41       }
42
43    // Print an error message and exit.
44    private static void usage
45       (String msg)
46       {
47       System.err.printf ("RandomGraph: %s%n", msg);
48       usage();
49       }
50
51    // Print a usage message and exit.
52    private static void usage()
53       {
54       System.err.println ("Usage: java " +
55          "edu.rit.pj2.example.RandomGraph <V> <E> <seed>");
56       System.err.println ("<V> = Number of vertices, V >= 1");
57       System.err.println ("<E> = Number of edges, 0 <= E <= " +
58          "V(V-1)/2");
59       System.err.println ("<seed> = Random seed");
60       System.exit (1);
61       }
62    }
```

Listing 11.1. RandomGraph.java (part 2)

threads in the program; thus, the time spent collecting garbage is typically part of the program's sequential fraction. (I suspect garbage collection is the chief reason for MinVerCovSmp's large sequential fraction.) It would be better all around if the program didn't create and discard an object on every loop iteration.

To address this problem, I wrote another version of the program. MinVerCovSmp2 is the same as MinVerCovSmp, except I changed the parallel loop body slightly; the differences are highlighted:

```
parallelFor (0L, full) .exec (new LongLoop()
   {
   BitSet64Vbl thrMinCover;
   BitSet64 candidate;
   public void start()
       {
       thrMinCover = threadLocal (minCover);
       candidate = new BitSet64();
       }
   public void run (long elems)
       {
       candidate.bitmap (elems);
       . . .
```

This time, the `candidate` variable is a field of the loop body subclass rather than a local variable of the `run()` method. A new vertex set object is created, once only, in the `start()` method and is assigned to the `candidate` variable. In the `run()` method, each loop iteration *reuses the existing vertex set object* by calling the `candidate` variable's `bitmap()` method, rather than constructing a new object. The `bitmap()` method replaces the candidate variable's elements with those of the loop index, `elems`. Coding the loop this way eliminates the repeated object creation, thus eliminating the garbage collection as well.

I ran the sequential MinVerCovSeq2 program on one core and the multicore parallel MinVerCovSmp2 program on one to four cores of a `tardis` node and measured the running times. I ran the programs on five random graphs, with $V = 31$ to 35 and $E = 310$ to 350—the same graphs as the original program version. Here are examples of the minimum vertex cover the programs found for the $V = 31$ test case:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSeq2 \
  g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23
24 25 26 27 28
Size = 26
Job 78 makespan 52265 msec
$ java pj2 debug=makespan cores=4 \
  edu.rit.pj2.example.MinVerCovSmp2 g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23
24 25 26 27 28
```

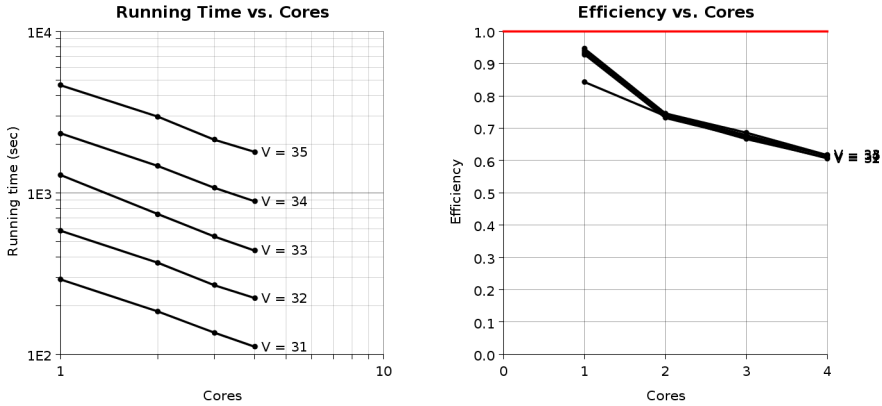


Figure 11.1. MinVerCovSmp strong scaling performance metrics

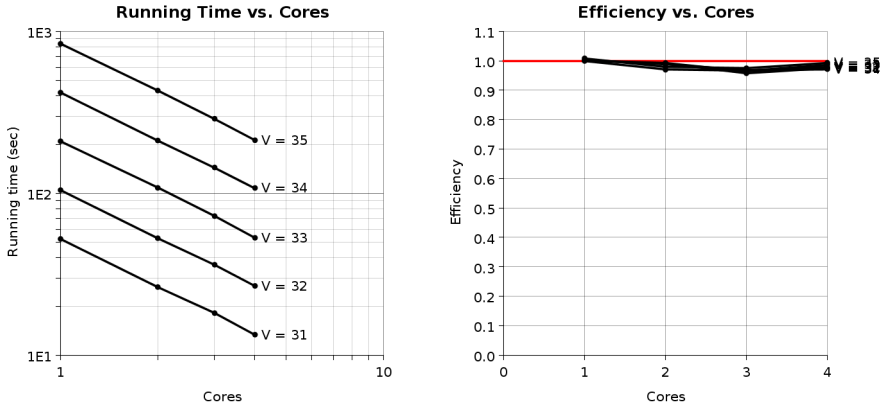


Figure 11.2. MinVerCovSmp2 strong scaling performance metrics

```
Size = 26
Job 90 makespan 13383 msec
```

The new version's running times are significantly—about 81 percent—smaller than the original version's, due to eliminating all the object creation and garbage collection. Fitting the running time model to the measurements gives this formula for the new version's running time:

$$T = 3.13\times10^{-10}\,N + 2.41\times10^{-8}\,N\div K\,. \tag{11.2}$$

Now the sequential fraction is only 1.3 percent for the problem sizes I studied. The new version's plots (Figure 11.2) show that the efficiencies degrade

hardly at all as the number of cores increases.

The moral? It's okay to use objects in Java parallel programs. However, you have to be careful, and you have to be aware of the consequences. Avoid repeatedly constructing and discarding objects. Rather, if at all possible, reuse existing objects by changing their state as necessary. I realize this might go counter to what you've been taught or how you're accustomed to designing Java programs. However, better program performance trumps conventional design wisdom.

Another moral is that investigating the parallel program's scalability, as I did in this chapter, can yield insights that lead to design changes that improve the program's performance. I would not have realized what was going on with object creation and garbage collection if I had not measured the program's performance.

Under the Hood

Let's look more closely at how bitsets are implemented. Class edu.rit.util.BitSet32 provides a set with elements from 0 to 31. The set elements are stored in a value of type `int`, which has 32 bits. Each bit position of the integer corresponds to a different set element: bit position 0 (the rightmost, or least significant bit) to element 0, bit position 1 to element 1, and so on. Each bit's value is a 1 if the set contains the corresponding element; each bit's value is a 0 if the set does not contain the corresponding element.

This data structure can also be viewed as a *mapping* from elements (0 through 31) to Boolean values (0 or 1). Each mapping occupies one bit, and thirty-two such mappings are crammed into an `int`. It is a *map* composed of *bits*, or a *bitmap*.

Here is the bitmap representation of the set {1, 3, 5, 7}:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Or, expressed as a 32-bit binary integer:

```
00000000000000000000000010101010
```

Class BitSet32's `contains()` method checks whether a bitset contains a given element *e*. It does this by forming a *mask* for *e*, namely an integer with a 1 at bit position *e* and 0s elsewhere. The mask is generated by the expression `(1 << e)`. The value `1` has a 1 at bit position 0 and 0s elsewhere; left-shifting the value `1` by *e* bit positions moves the 1 to bit position *e* and leaves 0s elsewhere. The method then does a bitwise Boolean "and" operation between the bitmap and the mask. The resulting value has a 0 bit wherever the mask has a 0 bit, namely in all bit positions except *e*. The resulting value is the same as the bitmap in bit position *e*, where the mask has a 1 bit. If the re-

sulting value is 0, namely all 0 bits, then the bitmap at bit position *e* is 0, meaning the set does not contain element *e*. If the resulting value is not 0, then the bitmap at bit position *e* is 1, meaning the set does contain element *e*. Therefore, the expression `((bitmap & (1 << e)) != 0)` is true if the set contains element *e*. Here is an example of the `contains(7)` method called on the set {1, 3, 5, 7}:

```
00000000000000000000000010101010  bitmap
00000000000000000000000000000001  1
00000000000000000000000010000000  1 << 7
00000000000000000000000010000000  bitmap & (1 << 7)
```

The resulting value is not equal to 0, so the method returns true, signifying that {1, 3, 5, 7} does contain 7. On the other hand, here is the `contains(9)` method called on the set {1, 3, 5, 7}:

```
00000000000000000000000010101010  bitmap
00000000000000000000000000000001  1
00000000000000000000001000000000  1 << 9
00000000000000000000000000000000  bitmap & (1 << 9)
```

The resulting value is equal to 0, so the method returns false, signifying that {1, 3, 5, 7} does not contain 9.

Class BitSet32's `add(e)` method works in a similar fashion. It forms a mask for *e*, then it does a bitwise Boolean "or" operation between the bitmap and the mask. The resulting value has a 1 at bit position *e*, where the mask has a 1 bit, regardless of what was in the bitmap before. The resulting value's other bit positions, where the mask has 0 bits, are the same as those of the original bitmap. This new value replaces the bitmap's original value. The bitmap ends up the same as before, except bit position *e* has been set to 1; that is, element *e* has been added to the set.

Class BitSet32's `isSubsetOf()` method forms the bitwise Boolean "and" of the two bitmaps. If the result is equal to the first bitmap, then every bit position that is a 1 in the first map is also a 1 in the second bitmap; that is, every element of the first set is also an element of the second set; that is, the first set is a subset of the second set. Otherwise, the first set is not a subset of the second set.

Class BitSet64 is the same as class BitSet32, except the bitmap uses a 64-bit long integer (type `long`) to store the bitmap.

All the methods in both bitset classes are implemented like those described above, with just a few operations on the integer bitmaps. As mentioned before, each such operation manipulates all the bits of the bitmap at the same time—that is, in parallel. The parallelism comes, not from multiple threads, but from the CPU's integer functional unit's inherent parallelism. Multithreading is not the only way to get a bunch of things to happen at the same time!

Points to Remember

- An exhaustive search solves a problem by looking at every possible candidate solution and keeping the best one.
- However, some problems have exponentially many candidate solutions. Consequently, an exhaustive search will take too long, unless the problem size is small.
- Consider implementing a set of elements using a bitset representation. Bitsets are fast and compact.
- Use the Parallel Java 2 Library's bitset classes in package edu.rit.util.
- Use the Parallel Java 2 Library's bitset reduction variable classes in package edu.rit.pj2.vbl.
- Avoid repeatedly creating and discarding objects. Reuse existing objects wherever possible.
- Measure the parallel program's performance and scalability. Derive the program's running time formula. If necessary, use the insights gained to change the program's design to improve its performance.

Chapter 12
Heuristic Search

The exhaustive search program for the minimum vertex cover problem in Chapter 11 is guaranteed to find a minimum cover. But because the problem size is an exponential function of the number of vertices—$N = 2^V$—the program's running time is too long to be practical for larger graphs, even on a parallel computer. If I want to tackle larger problem sizes, I'm going to need a different approach.

An alternative to exhaustive search is *heuristic search.* Rather than looking at every possible candidate solution, a heuristic search program looks at only a selected number of candidate solutions. The candidates are chosen by some rule, or *heuristic.* The heuristic attempts to select candidates that have a higher likelihood of solving the problem. Of the selected candidates, the program chooses the one that yields the best solution. Now the problem size N is the number of candidates selected rather than an exponential function. You can specify the number of candidates to yield a running time that you are willing to tolerate.

There are several general approaches for heuristic search programs aimed at solving problems with exponentially large problem spaces, such as *simulated annealing*, *genetic algorithms*, and *tabu search.* I've worked with all of these approaches. In my experience, they tend to require rather intricate data structures and algorithms. They also tend to depend on parameters that have to be tuned to obtain good solutions, and it's often not clear what the optimum parameter settings are. Furthermore, these are inherently sequential algorithms, and it's difficult to parallelize them.

To solve exponentially hard problems on parallel computers, I prefer a different approach, which I call *massively parallel randomized approximation (MPRA).* An MPRA program generates a very large (but not exponentially large) number of candidate solutions, chosen *at random* using a simple heuristic. The program then evaluates all the candidates and reports the one that yields the best solution.

Because the candidates can all be generated and evaluated independently of each other, an MPRA program is trivial to parallelize. It is just a parallel loop to generate all the candidates, combined with a parallel reduction to choose the best candidate. An MPRA program also exhibits near-ideal weak scaling. If you increase the number of cores by some factor, you can increase the number of candidates by the same factor while keeping the running time the same. Increasing the number of candidates examined might increase the likelihood of finding a better solution.

There's a catch, though. Because an MPRA program searches only some of the candidate solutions, not all possible candidate solutions, an MPRA program is not *guaranteed* to find the absolutely best solution. However, an MPRA program might be able to find an *approximate* solution that's only a little worse than the absolutely best solution. This approximate solution might still be useful for practical purposes—and it can be found in a practical

amount of time.

Let's apply these considerations to the minimum vertex cover problem. I need a heuristic for generating a random candidate solution, namely a random vertex cover, that I hope will be close to a minimum vertex cover. Many such heuristics could be envisioned. Here's one that's particularly simple: Start with an empty subset of vertices; repeatedly add a vertex chosen at random from those not yet in the subset; stop as soon as the subset is a cover; and use that as the candidate solution. Generate a large number of such candidates in parallel, and keep the one with the fewest vertices. (If multiple candidates are tied for the fewest vertices, any of them will do.) This heuristic doesn't guarantee that a candidate cover will be a true minimum cover; depending on the order in which vertices are added, the candidate might end up with more vertices than necessary to cover all the edges. Still, by stopping as soon as the subset becomes a cover, the hope is that the candidate will be close to a minimum cover, if not actually a minimum cover.

How effective is this heuristic at finding a minimum vertex cover? It's difficult to say in general. I wrote a program to study the question for smaller graphs (class edu.rit.pj2.example.MinVerCovDist in the Parallel Java 2 Library). The program generated a given number of random graphs; I used 100 random graphs in my study. Each random graph had a given number of vertices V; I used 20, 22, 24, 26, 28, 30, and 32 vertices. Each random graph had a given number of edges E; I chose E to yield graphs with densities of 0.2, 0.4, 0.6, and 0.8. A graph's *density* D is the ratio of the actual number of edges to the total possible number of edges between V vertices,

$$D = \frac{E}{V(V-1)/2} \quad . \tag{12.1}$$

For each random graph, the program did an exhaustive search to find the size of a minimum vertex cover; counted the total number of covers and the number of minimum covers; and computed the minimum cover fraction F,

$$F = \frac{\text{Number of minimum covers}}{\text{Total number of covers}} \quad . \tag{12.2}$$

Figure 12.1 plots the median F over all 100 random graphs for each value of V and D.

Why look at the minimum cover fraction F? Assume that the heuristic procedure yields a cover chosen at random from all the possible covers. Then roughly speaking, F is the probability that the heuristic procedure will land on a minimum cover. This in turn tells me that the expected number of trials before landing on a minimum cover is $1/F$. So if I set the number of candidates at $10/F$, or $100/F$, there ought to be a reasonable chance that the MPRA program will find an actual minimum cover.

Unfortunately, because the program I used for my study does an exhaus-

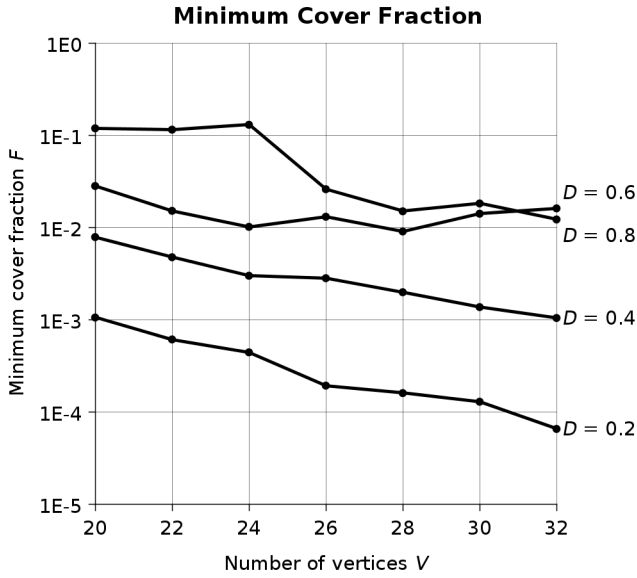


Figure 12.1. Minimum cover fraction versus vertices and density

tive search, I was not able to study graphs with more than 32 vertices. Still, some trends are apparent in Figure 12.1. When I increased the number of vertices by 12 (from 20 to 32), F went down by about a factor of 10. This means that F is proportional to $10^{-V/12}$. I can then extrapolate the curves to get a rough F value for larger V values. For example, consider the $D = 0.4$ curve. Going from $V = 20$ to $V = 50$ multiplies $F = 1\times10^{-2}$ by a factor of $10^{-(50-20)/12}$, yielding $F = 3.16\times10^{-5}$. So doing $100/F$ = about three million candidates, the MPRA program ought to find a vertex cover pretty close to a minimum vertex cover for a 50-vertex random graph of density 0.4. Three million candidates is a lot fewer than the 2^{50}, or about one quadrillion, candidates the exhaustive search program would have to examine.

However, this trend also shows that the MPRA program's running time still increases exponentially—if I want a decent chance of it finding an actual minimum vertex cover. Each time the number of vertices increases by 12, the number of candidates has to increase by a factor of 10. At some point I'll have to stop increasing the number of candidates to keep the running time reasonable; but then the program might not be able to find a true minimum vertex cover. How close can it come? There's no way to know for sure, other than by comparing the MPRA program's results to the exhaustive search program's results.

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.LongLoop;
3  import edu.rit.pj2.Task;
4  import edu.rit.pj2.vbl.BitSetVbl;
5  import edu.rit.util.BitSet;
6  import edu.rit.util.Random;
7  import edu.rit.util.RandomSubset;
8  import java.io.File;
9  import java.util.Scanner;
10 public class MinVerCovSmp3
11    extends Task
12    {
13    // Number of vertices and edges.
14    int V;
15    int E;
16
17    // The graph's adjacency matrix. adjacent[i] is the set of
18    // vertices adjacent to vertex i.
19    BitSet[] adjacent;
20
21    // Minimum vertex cover.
22    BitSetVbl minCover;
23
24    // Main program.
25    public void main
26       (String[] args)
27       throws Exception
28       {
29       if (args.length != 3) usage();
30       final File file = new File (args[0]);
31       final long seed = Long.parseLong (args[1]);
32       final long N = Long.parseLong (args[2]);
33
34       // Read input file, set up adjacency matrix.
35       Scanner s = new Scanner (file);
36       V = s.nextInt();
37       E = s.nextInt();
38       if (V < 1) usage ("V must be >= 1");
39       adjacent = new BitSet [V];
40       for (int i = 0; i < V; ++ i)
41          adjacent[i] = new BitSet (V);
42       for (int i = 0; i < E; ++ i)
43          {
44          int a = s.nextInt();
45          int b = s.nextInt();
46          adjacent[a].add (b);
47          adjacent[b].add (a);
48          }
49       s.close();
50
51       // Check N randomly chosen candidate covers.
52       minCover = new BitSetVbl.MinSize (new BitSet (V));
53       minCover.bitset.add (0, V);
54       parallelFor (0L, N - 1) .exec (new LongLoop()
55          {
56          BitSetVbl thrMinCover;
57          BitSet candidate;
58          Random prng;
```

Listing 12.1. MinVerCovSmp3.java (part 1)

Setting aside questions about how close the program can come to finding a minimum vertex cover, let's examine the code for the minimum vertex cover MPRA program, class edu.rit.pj2.example.MinVerCovSmp3 (Listing 12.1).

Like the programs in Chapter 11, I need a class to represent a vertex set. This time, however, I don't want to be limited to at most 63 vertices, because I'm no longer doing an exhaustive search. Instead, I want a vertex set that can support an arbitrary number of vertices. I still want to use a bitset data structure. Instead of using class edu.rit.util.BitSet64, which can only hold 64 elements, I'll use class edu.rit.util.BitSet, which can accommodate an arbitrary number of elements. For doing parallel reduction, I'll use class edu.rit-.pj2.vbl.BitSetVbl.

The MinVerCovSmp3 main program's command line arguments are the graph file, the pseudorandom number generator seed, and *N*, the number of random candidate covers to generate. The program starts by reading the graph file and setting up the graph's adjacency matrix, as in the previous programs. This time the parallel loop iterates over the *N* candidates (line 54). As in the previous programs, each parallel team thread has a per-thread minimum vertex cover variable (line 56) that is linked to the global minimum vertex cover reduction variable (line 62). Taking to heart the lesson from Chapter 11, each team thread also has a candidate vertex cover variable (line 57) that the program will reuse on each parallel loop iteration.

Each parallel team thread needs to generate its own series of random candidate covers. So each thread gets its own per-thread pseudorandom number generator, seeded differently in each thread (line 64). To generate a random candidate, each team thread needs to generate a series of vertices, chosen *without replacement* from the set of all vertices 0 through $V-1$—that is, a *random subset* of the set of all vertices. To do so, the program uses a *random subset generator,* an instance of class edu.rit.util.RandomSubset in the Parallel Java 2 Library, layered on top of the per-thread pseudorandom number generator (line 65).

Each parallel loop iteration (lines 67–75) performs the heuristic procedure for generating a random cover: Clear the candidate back to an empty set; restart the random subset generator to obtain a new random subset; as long as the candidate is not a cover, get a random vertex from the random subset generator and add the vertex to the candidate. As soon as the candidate becomes a cover, stop; and if the candidate is smaller than the per-thread minimum vertex cover variable, copy the candidate there, thus retaining the smallest cover seen. When the parallel loop finishes, the per-thread minimum covers are automatically reduced under the hood into the global minimum cover, which the program prints.

I ran the exhaustive search MinVerCovSmp2 program on four cores of a `tardis` node on a graph with 40 vertices and 312 edges (density 0.4). I ran

```
59          RandomSubset rsg;
60          public void start()
61             {
62             thrMinCover = threadLocal (minCover);
63             candidate = new BitSet (V);
64             prng = new Random (seed + rank());
65             rsg = new RandomSubset (prng, V, true);
66             }
67          public void run (long i)
68             {
69             candidate.clear();
70             rsg.restart();
71             while (! isCover (candidate))
72                candidate.add (rsg.next());
73             if (candidate.size() < thrMinCover.size())
74                thrMinCover.bitset.copy (candidate);
75             }
76          });
77
78       // Print results.
79       System.out.printf ("Cover =");
80       for (int i = 0; i < V; ++ i)
81          if (minCover.bitset.contains (i))
82             System.out.printf (" %d", i);
83       System.out.println();
84       System.out.printf ("Size = %d%n", minCover.bitset.size());
85       }
86
87    // Returns true if the given candidate vertex set is a cover.
88    private boolean isCover
89       (BitSet candidate)
90       {
91       boolean covered = true;
92       for (int i = 0; covered && i < V; ++ i)
93          if (! candidate.contains (i))
94             covered = adjacent[i].isSubsetOf (candidate);
95       return covered;
96       }
97
98    // Print an error message and exit.
99    private static void usage
100      (String msg)
101      {
102      System.err.printf ("MinVerCovSmp3: %s%n", msg);
103      usage();
104      }
105
106   // Print a usage message and exit.
107   private static void usage()
108      {
109      System.err.println ("Usage: java pj2 " +
110         "edu.rit.pj2.example.MinVerCovSmp3 <file> <seed> <N>");
111      System.err.println ("<file> = Graph file");
112      System.err.println ("<seed> = Random seed");
113      System.err.println ("<N> = Number of trials");
114      throw new IllegalArgumentException();
115      }
116   }
```

Listing 12.1. MinVerCovSmp3.java (part 2)

the heuristic search MinVerCovSmp3 program on four cores of a tardis node on the same graph, generating 100 million random candidate covers. Here is what the programs printed:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp2 \
  g40.txt
Cover = 0 2 3 4 6 8 9 11 12 13 14 16 18 19 20 21 22 23 24 25 26
27 30 31 32 33 34 36 37 38 39
Size = 31
Job 62 makespan 6113078 msec
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp3 \
  g40.txt 23576879 100000000
Cover = 0 2 3 4 5 6 8 9 10 11 12 13 14 16 19 20 21 22 23 24 25
26 27 30 31 32 33 34 36 37 38 39
Size = 32
Job 64 makespan 76399 msec
```

The heuristic search program found a cover that was almost, but not quite, a true minimum vertex cover; 32 vertices instead of 31. But the heuristic search program's running time was a little over a minute rather than nearly two hours. These results are typical of heuristic search programs; the running times are more practical, but the solutions are only approximate, although close to the true solutions.

To see if the heuristic search program could find a true minimum cover, I increased the number of candidates from 100 million to one billion. Here is what the program printed:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp3 \
  g40.txt 23576879 1000000000
Cover = 0 2 3 4 6 8 9 11 12 13 14 16 18 19 20 21 22 23 24 25 26
27 30 31 32 33 34 36 37 38 39
Size = 31
Job 63 makespan 771547 msec
```

This time the heuristic search program found a true minimum cover in about 13 minutes.

To study the heuristic search program's performance under weak scaling, I ran the sequential MinVerCovSeq3 program on one core and the multicore parallel MinVerCovSmp3 program on one to four cores of a `tardis` node and measured the running times. I ran the programs on five random graphs, with V = 50, 100, 150, 200, and 250; and E = 500, 1000, 1500, 2000, and 2500. I did 10 million candidate covers on one core, 20 million on two cores, 30 million on three cores, and 40 million on four cores. Figure 12.2 plots the program's running times and efficiencies. Although slightly less than ideal, the efficiencies droop very little as the number of cores increases, evincing good weak scaling behavior.

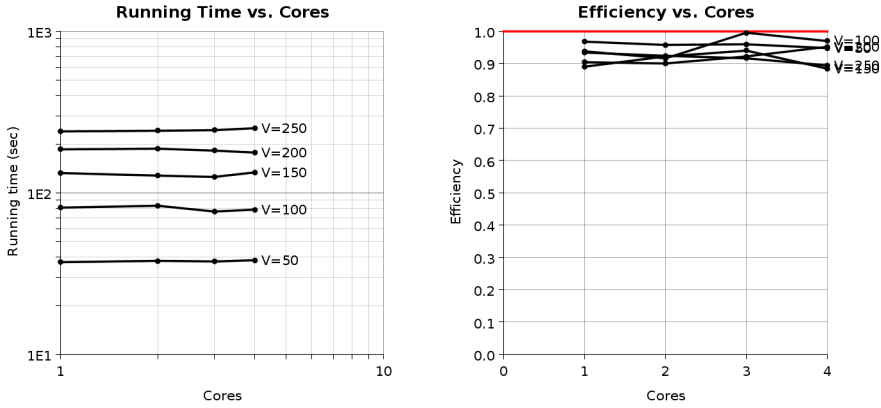


Figure 12.2. MinVerCovSmp3 weak scaling performance metrics

Under the Hood

Class edu.rit.util.BitSet32 uses a single value of type `int` (a private field) to hold the bitmap. Class BitSet64 uses a single value of type `long` to hold the bitmap. To accommodate an arbitrary number of elements, class BitSet uses an array of one or more `int`s to hold the bitmap. The required number of set elements is specified as an argument to the BitSet constructor, which allocates an `int` array large enough to hold that many bits. Class BitSet's methods are implemented in the same way as class BitSet32 and BitSet64, except the methods loop over all the bitmap array elements.

The minimum vertex cover program needs to create a random subset of the set of vertices in the graph. This amounts to creating a random subset of the integers 0 through $V-1$. To do so, the program uses class RandomSubset in package edu.rit.util in the Parallel Java 2 Library. To generate a random subset, this class has to pick integers at random, with equal probability, from the set $\{0, 1, 2, \ldots V-1\}$, *without replacement.* It's the "without replacement" requirement that makes this process a bit tricky.

Here's the *wrong* way to generate a random subset with k elements:

```
1  Subset ← {}
2  Repeat k times:
3     Repeat:
4        n ← Random integer in the range 0 through V – 1
5     Until n is not in the subset
6     Add n to the subset
```

This is the wrong way to do it because of the open-ended loop on lines 3–5. That loop might have to iterate several times before finding a random integer that is not in the subset; and the more elements there are in the subset, the more iterations the loop might have to do. This wastes CPU time, which is especially problematic in a program that has to generate an enormous number of random subsets.

Here's the *right* way to generate a random subset with k elements. The algorithm revolves around an array S with V elements.

```
1  For i = 0 to V – 1:
2     S[i] ← i
3  For i = 0 to k – 1:
4     j ← Random integer in the range 0 through V – 1 – i
5     Swap S[i] and S[i + j]
6  Subset ← S[0] through S[k – 1]
```

Why does this procedure work? The S array starts out containing each possible integer from 0 through $V – 1$ (lines 1–2). At each iteration of the loop on lines 3–5, the next element of the array is swapped with one of the available integers from the remainder of the array, chosen at random. Each available integer is chosen with equal probability. After k iterations, the first k elements of the array end up containing a randomly chosen subset of the integers 0 through $V – 1$.

Here's an example of choosing a random four-element subset of the integers 0 through 9:

```
S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]   i = 0   j = 3
S = [3, 1, 2, 0, 4, 5, 6, 7, 8, 9]   i = 1   j = 5
S = [3, 6, 2, 0, 4, 5, 1, 7, 8, 9]   i = 2   j = 0
S = [3, 6, 2, 0, 4, 5, 1, 7, 8, 9]   i = 3   j = 2
S = [3, 6, 2, 5, 4, 0, 1, 7, 8, 9]
```

The random subset is {3, 6, 2, 5}, or {2, 3, 5, 6}. (For a set, the order of the elements doesn't matter.)

If you perform this procedure and iterate through the entire array (by setting k equal to V), the array elements—taken in order—comprise a *random permutation* of the integers 0 through $V – 1$. So class RandomSubset is useful both for generating random subsets and for generating random permutations.

Why is this procedure the right way to generate a random subset or a random permutation? Because the loop body (lines 4–5) is executed precisely once for each of the k elements in the subset or permutation. The procedure no longer has an open-ended loop like the previous version. This minimizes the CPU time required.

Class RandomSubset implements the above procedure. The constructor initializes the internal S array (lines 1–2) and initializes i to 0. The `next()`

method performs the loop body (lines 4–5) and returns $S[i]$. (You write the enclosing loop yourself.) The `restart()` method reinitializes S and i; this lets you generate multiple random subsets from the same RandomSubset object.

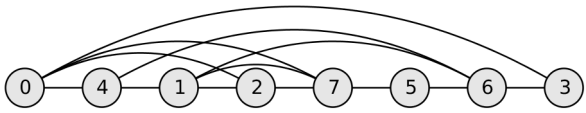
Points to Remember

- A heuristic search solves a problem by looking at a limited number of candidate solutions, generating using a heuristic, and keeping the best solution.
- The heuristic attempts to generate solutions that are close to the optimum solution.
- A massively parallel randomized approximation (MPRA) program generates and evaluates a large number of random candidate solutions, in parallel, using a heuristic.
- Because it does not consider all possible solutions, a heuristic search is not guaranteed to find a true optimum solution. However, it might find an approximate solution that is close enough to the true optimum solution for practical purposes.
- An MPRA program is easy to write, is trivial to parallelize, and exhibits good weak scaling behavior.
- Scaling an MPRA program up to more cores, thereby examining more candidate solutions, increases the likelihood of finding a better solution.
- Use class edu.rit.util.RandomSubset in the Parallel Java 2 Library to generate a random subset or a random permutation of a set of integers.

Chapter 13
Parallel Work Queues

Consider a computer network with eight nodes, numbered 0 through 7. Certain nodes are connected to each other via the following bidirectional communication links: 1–4, 3–6, 1–7, 5–6, 1–2, 5–7, 0–4, 4–6, 0–7, 2–7, 0–2, 1–6, and 0–3. You want to broadcast a message to all the nodes. You start by sending the message to one of the nodes. That node forwards the message to another node to which it is connected. Each node continues to forward the message to one further node. The last node in the chain sends the message back to the original node as an acknowledgment that the broadcast is complete. Two questions: Is it even possible to send the message from one node to another such that the message reaches every node and returns to the origin? If so, to what sequence of nodes must the message be sent?

Just looking at the list of links, it might take you quite a while to answer those questions. But if I draw the computer network in the form of a graph, like so:



it becomes apparent that the message can in fact be sent to all the nodes in the order 0, 4, 1, 2, 7, 5, 6, 3, and back to 0 again.

This is an example of the *Hamiltonian cycle problem*, another well-known graph theory problem like the minimum vertex cover problem. A *cycle* in a graph is a sequence of adjacent vertices (vertices connected by an edge) that starts and ends with the same vertex. A *Hamiltonian cycle* is a cycle that includes each vertex in the graph exactly once. Hamiltonian cycles are named after nineteenth century Irish physicist Sir William Rowan Hamilton, who invented a puzzle called the "Icosian Game" that involved finding a path that visited all the vertices of a dodecahedron and returned to the starting point.

The Hamiltonian cycle problem is an instance of a class of problems called *nondeterministic polynomial-time (NP)* problems. An NP problem is a *decision problem*, one whose answer is "yes" or "no". In our case, the decision problem is "Does a Hamiltonian cycle exist in a given graph?" If the answer to an NP problem is yes, the solution includes a *proof* that the answer is correct; furthermore, the proof can be verified by an algorithm whose asymptotic running time is a polynomial function of the problem size. In our case, the proof is simply a list of the vertices in the Hamiltonian cycle; and the proof can be verified in linear time (proportional to the problem size, that is, the number of vertices) by checking that each vertex in the purported Hamiltonian cycle is adjacent to its predecessor.

Although the solution to an NP problem can be *verified* in polynomial time, this says nothing about the time required to *find* the solution. There might be an efficient (polynomial-time) algorithm to find the solution, or

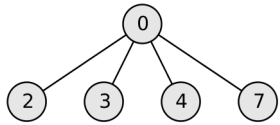
there might not.

It's always possible to solve an NP problem by exhaustive search. In our case, this involves generating every possible path in the graph and checking to see whether any of these paths is a Hamiltonian cycle. But in general, the number of possible paths in a graph, as a function of the number of vertices, grows much faster than any polynomial function. No one knows an efficient algorithm for finding a Hamiltonian cycle in an arbitrary graph.

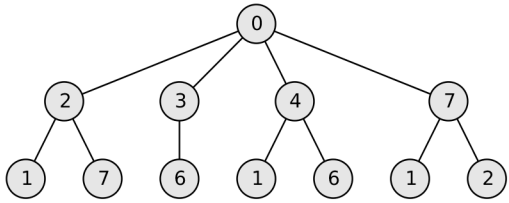
Let's develop a parallel program to solve the Hamiltonian cycle problem by exhaustive search. This requires generating and checking each possible path in the graph. There are two approaches for generating all paths: *breadth first search* and *depth first search.*

A breadth first search starts with one vertex of the graph. Because all the vertices have to appear in a Hamiltonian cycle, it doesn't matter where the search starts; I'll start with vertex 0:

I now have one *subproblem:* "Find all paths starting with vertex 0." The breadth first search continues by examining all vertices adjacent to vertex 0:



Now I have four subproblems: "Find all paths starting with vertices 0–2; 0–3; 0–4; and 0–7." For each of these subproblems, the breadth first search continues by examining all vertices adjacent to the last vertex in the partial path, that do not already appear in the partial path:



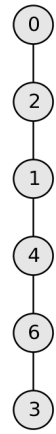
Now I have seven subproblems: "Find all paths starting with vertices 0–2–1; 0–2–7; 0–3–6; 0–4–1; 0–4–6; 0–7–1; and 0–7–2." The breadth first search continues in this fashion, exploring the *search tree* one level at a time, until every possible path has been visited. At any stage, a path from the root to a leaf of the search tree corresponds to a path in the original graph. If the search reaches a path that contains all the vertices, and the last vertex is adjacent to the first vertex, it's a Hamiltonian cycle.

A breadth first search is easy to parallelize. Each subproblem can be processed in parallel with all the other subproblems. For the typical case where

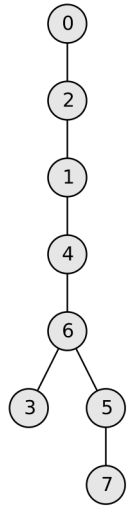
there are more subproblems than there are cores, a parallel program can maintain a queue of pending subproblems. Each thread takes a subproblem (partial path) off the queue, extends the partial path by each possible adjacent vertex, and adds these new subproblems back into the queue. Whenever a thread finishes processing a subproblem, it goes to work on the next subproblem from the queue.

However, in a breadth first search, the number of subproblems—hence, the amount of storage required for the subproblem queue—becomes enormously large as the search proceeds to each successive level in the search tree. If the average vertex degree (average number of vertices adjacent to each vertex) is K, then each level of the search tree has about K times as many nodes as the previous level, the number of nodes at level L is about K^L, and the number of leaf nodes at level V is about K^V, where V is the number of vertices in the original graph—an exponential function of V. No supercomputer in the world has that much storage, even for a moderately sized graph.

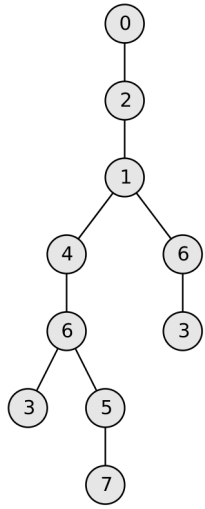
The other technique, a depth first search, also starts with one vertex of the graph, such as vertex 0. The search then finds one complete path by visiting a sequence of adjacent vertices (that have not appeared in the path yet) until it can go no farther; for example:



From vertex 3, edges lead to vertices 0 and 6; but vertices 0 and 6 already appear in the path; so the path cannot be extended to any new vertices. Because the path does not include all the vertices, it is not a solution. The depth first search therefore *backtracks* one or more levels in the search tree until it returns to a vertex that has an edge to an as yet unvisited vertex. In this example, vertex 6 has edges to vertices 1, 3, 4, and 5. The search has already tried vertex 3. Vertices 1 and 4 are already in the path. That leaves vertex 5 as an alternative. The search branches off vertex 6 to vertex 5, and proceeds from there:



Another brick wall; the path cannot be extended past vertex 7, and the path does not include all the vertices, so this path is not a solution either. Now the depth first search has to backtrack all the way back to vertex 1 before proceeding along an alternate path:



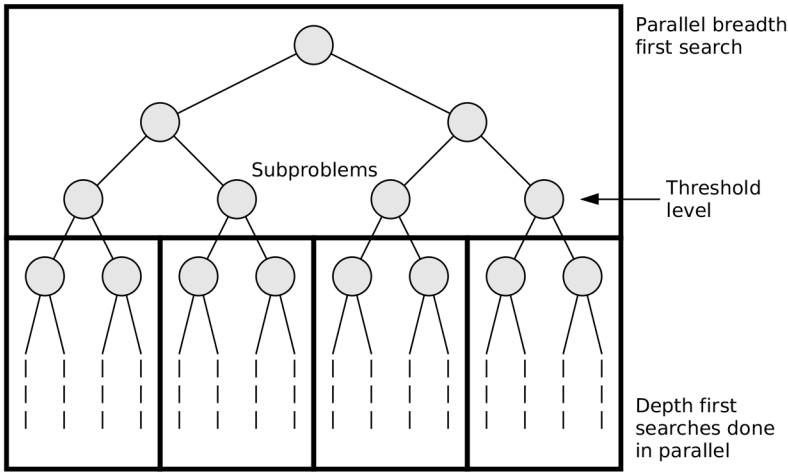
The depth first search continues in this fashion, proceeding as deep as it can go along each branch of the search tree, then backtracking to a different branch, until it has visited every possible path or it has found a Hamiltonian cycle.

Unlike a breadth first search, a depth first search's storage is not an exponential function of *V,* the number of vertices. Rather, the depth first search requires storage proportional just to *V,* to hold the current partial path—a linear function of *V.* However, a depth first search is difficult to parallelize.

Whether the search is implemented recursively or iteratively, the search has to proceed in sequence down each branch of the tree and back up again.

A breadth first search is easy to parallelize but requires too much storage. A depth first search requires just a little storage but is difficult to parallelize. What's a poor parallel program designer to do?

The answer is to use both strategies. A parallel program that must examine potentially all branches of a search tree should start by doing a breadth first search in parallel until it reaches a certain *threshold level* in the search tree. At this point the program will have accumulated some number of subproblems in a queue. The program should then switch strategies and do a depth first search on each subproblem. Because the subproblems can be searched independently, the program can do the depth first searches in parallel, each depth first search being done sequentially in a separate thread.



This way, we get the best of both worlds. Stopping the breadth first search at the threshold level ensures that the subproblem queue does not consume an excessive amount of storage. Having multiple subproblems in the queue ensures that all the machine's cores can be occupied doing depth first searches simultaneously.

What should the threshold level be? That depends on the particular problem. In general, the subproblems' depth first searches will take different amounts of time. The subproblems should therefore be partitioned among the parallel threads (cores) in a dynamic fashion. The threshold level should be set so that after the breadth first search phase, there are enough subproblems to balance the load across the cores in the machine for the depth first search phase. You should experiment with various threshold levels until you find one that yields the smallest running time on the parallel machine.

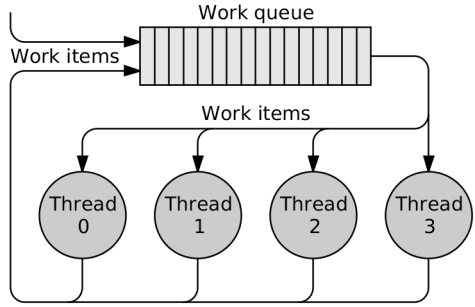
What I've described is a general pattern that can be applied to any problem involving a search tree, not just the Hamiltonian cycle problem. Many

NP problems are like this; the best known algorithm for finding the exact solution requires searching an exponentially large search tree. Doing the search in parallel can speed it up. (Keep in mind that we are talking about an exhaustive search program that is guaranteed to find the exact solution, not a heuristic search program that typically finds only an approximate solution.)

Let's consider how to code a parallel program that does breadth first and depth first searches over a search tree.

In all the parallel programs we've studied so far, the number of loop iterations was known before the loop started. I therefore could iterate using a parallel for loop, and I could specify the known lower and upper bound loop indexes in the `parallelFor()` statement.

When iterating over the subproblems in a search tree, however, *the number of iterations is not known before the loop starts* (in general). Rather, the loop must iterate *while* there are still subproblems to look at. Furthermore, the act of processing one subproblem might create one or more additional subproblems that have to be processed. To do this kind of iteration, I'll use a new programming pattern: the *parallel work queue.*



In this pattern, there is a *work queue* containing *work items*. Typically, one or more work items are added to the queue before the parallel processing starts. A parallel thread team then commences operating. Each team thread repeatedly takes one work item out of the queue and does whatever computation is needed for that item. As part of the computation, the thread might create one or more new work items, which the thread adds back into the queue. Work items can also be added to the queue from outside the parallel team. Processing the work items one at a time automatically balances the load, like a dynamic schedule. The threads continue in this fashion until the queue is empty and all the threads have finished computing their final work items.

I'll use the parallel work queue pattern for the parallel Hamiltonian cycle program. Each work item will be a subproblem in the search, that is, a partial path. The first work item will be the initial subproblem, namely a partial path consisting of just vertex 0. A parallel team thread will process a work item by doing a breadth first search if the partial path is below the threshold level, or by doing a depth first search if the partial path is at the threshold level.

The Parallel Java 2 Library supports parallel loops over the work items in a work queue. The pattern for writing this kind of parallel loop is

```
WorkQueue<W> queue = new WorkQueue<W>();
queue.add (new W (...)); // First work item
parallelFor (queue) .exec (new ObjectLoop<W>()
   {
   public void run (W workitem)
      {
      Loop body code for workitem
      queue().add (new W (...)); // (optional)
      }
   });
```

The work queue itself is an instance of the generic class edu.rit.pj2.WorkQueue. Replace the generic type parameter `<W>` with the data type of the work items; this can be any object type. Add one or more work items to the queue before commencing the parallel loop. (If you don't, the parallel loop won't do anything.) The argument of the `parallelFor()` method is the work queue, rather than the index bounds. The argument of the `exec()` method is the loop object itself, an instance of the generic class edu.rit.pj2.ObjectLoop. The loop object's `run()` method's argument is a work item; the method body contains the loop body code for processing that work item. The `run()` method may optionally add more work items to the work queue; to access the work queue from inside the loop object, call the `queue()` method. Class ObjectLoop has the same capabilities as all the other parallel loops, including per-thread variables (fields), the `start()` method, parallel reduction with the `threadLocal()` method, the `finish()` method, and so on.

Finally, I can write the code for the multicore parallel Hamiltonian cycle finding program, class edu.rit.pj2.example.HamCycSmp (Listing 13.1). The program's command line arguments are the name of the file containing the graph to be solved and the threshold level.

The program begins by declaring some global variables, including the graph's adjacency matrix, implemented as an array of bitsets like the minimum vertex cover programs (line 22); a flag for terminating the search in all the parallel team threads as soon as a solution is found (line 25); and the parallel work queue (line 28). By the way, the `found` flag is declared with the keyword `volatile`. There's a reason for that. I'll discuss the reason in the "Under the Hood" section below. For now, just ignore the `volatile` keyword.

The work item class is class State (line 31). An instance of class State encapsulates the state of the search over all possible paths in the graph. The search state is represented by the two fields `path` and `level` (lines 34–37). At all times, the elements `path[0]` through `path[level]` contain the vertices in the partial path at which the search is positioned. The remaining elements of `path` contain the vertices that are not in the partial path, in no particular or-

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.ObjectLoop;
3  import edu.rit.pj2.Task;
4  import edu.rit.pj2.WorkQueue;
5  import edu.rit.util.BitSet;
6  import java.io.File;
7  import java.util.Formatter;
8  import java.util.Scanner;
9  public class HamCycSmp
10    extends Task
11    {
12    // Command line arguments.
13    File file;
14    int threshold;
15
16    // Number of vertices and edges.
17    int V;
18    int E;
19
20    // The graph's adjacency matrix. adjacent[i] is the set of
21    // vertices adjacent to vertex i.
22    BitSet[] adjacent;
23
24    // For early loop exit.
25    volatile boolean found;
26
27    // Parallel work queue.
28    WorkQueue<State> queue;
29
30    // Class for the search state.
31    private class State
32       {
33       // Vertices in the path.
34       private int[] path;
35
36       // Search level = index of last vertex in the path.
37       private int level;
38
39       // Construct a new search state object.
40       public State
41          (int V)
42          {
43          path = new int [V];
44          for (int i = 0; i < V; ++ i)
45             path[i] = i;
46          level = 0;
47          }
48
49       // Construct a new search state object that is a copy of the
50       // given search state object.
51       public State
52          (State state)
53          {
54          this.path = (int[]) state.path.clone();
55          this.level = state.level;
56          }
57
```

Listing 13.1. HamCycSmp.java (part 1)

der. For example, when searching the eight-vertex graph from the beginning of the chapter, the state might be `path` = {0, 3, 6, 1, 4, 5, 2, 7} and `level` = 2; this represents the partial path 0–3–6.

The State constructor (line 40) initializes `path` to contain all the vertices 0 through $V - 1$ and initializes `level` to 0; this represents the partial path consisting just of vertex 0. There is also a copy constructor (line 51) that makes a deep copy of a state object.

The `search()` method (line 59) either does one step of a breadth first search or does a complete depth first search, starting from the state object's current search state, depending on whether the search level is below or above the threshold level. The search is carried out by a separate method. The `search()` method either returns a state object containing a Hamiltonian cycle or returns null if no Hamiltonian cycle was found.

The breadth first search, in the `bfs()` method (line 68), iterates over the vertices not already in the partial path, namely, those stored in `path[level+1]` through `path[V-1]` (line 72). If one of these vertices is adjacent to the last vertex in the partial path (line 73), the adjacent vertex is appended to the partial path (lines 75–76), forming a new subproblem. A copy of this new subproblem is added to the work queue (line 77), to be processed later. The partial path is put back the way it was (line 78), and the loop goes on to try the next available vertex. The `bfs()` method returns null, signifying that it did not find a Hamiltonian cycle. (The program assumes that the threshold level is smaller than the number of vertices, so the breadth first search will never go far enough to find a Hamiltonian cycle.)

The depth first search, in the `dfs()` method (line 84), is implemented recursively. As in any recursive algorithm, there are two cases, the base case and the recursive case. The base case (lines 88–92) happens when all the vertices are in the path. If the last vertex, `path[V-1]`, is adjacent to the first vertex, `path[0]`, then the path is a Hamiltonian cycle, and a reference to this state object (which contains the Hamiltonian cycle) is returned. Otherwise, the recursive case happens (lines 96–107). The code is nearly the same as for the breadth first search, except when a new vertex is appended to the path, `dfs()` is called recursively to handle the new subproblem. If the recursive call finds a solution, again, a reference to this state object is returned; otherwise the depth first search continues. If neither the base case nor the recursive case finds a solution, the `dfs()` method returns null. In addition, the loop over the adjacent vertices exits early if a solution is found, namely if the `found` flag is set to true (line 98). This ensures that if one parallel team thread finds a solution, the program stops immediately without needing to wait for the other threads to finish their searches, which could take quite a while.

The rest of class State consists of a couple subroutines used by the `bfs()` and `dfs()` methods, as well as a `toString()` method for displaying the search state (path) as a string.

```
58        // Search the graph from this state.
59        public State search()
60           {
61           if (level < threshold)
62              return bfs();
63           else
64              return dfs();
65           }
66
67        // Do a breadth first search of the graph from this state.
68        private State bfs()
69           {
70           // Try extending the path to each vertex adjacent to the
71           // current vertex.
72           for (int i = level + 1; i < V; ++ i)
73              if (adjacent (i))
74                 {
75                 ++ level;
76                 swap (level, i);
77                 queue.add (new State (this));
78                 -- level;
79                 }
80           return null;
81           }
82
83        // Do a depth first search of the graph from this state.
84        private State dfs()
85           {
86           // Base case: Check if there is an edge from the last
87           // vertex to the first vertex.
88           if (level == V - 1)
89              {
90              if (adjacent (0))
91                 return this;
92              }
93
94           // Recursive case: Try extending the path to each vertex
95           // adjacent to the current vertex.
96           else
97              {
98              for (int i = level + 1; i < V && ! found; ++ i)
99                 if (adjacent (i))
100                   {
101                   ++ level;
102                   swap (level, i);
103                   if (dfs() != null)
104                      return this;
105                   -- level;
106                   }
107             }
108
109          return null;
110          }
111
```

Listing 13.1. HamCycSmp.java (part 2)

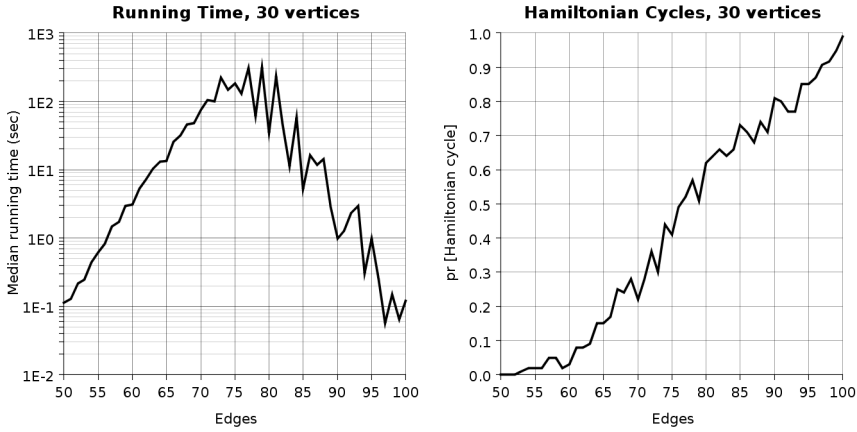


Figure 13.1. Program results for 30-vertex random graphs

The main program (line 142) is where all the parallel processing takes place. After reading the input file and setting up the graph's adjacency matrix (lines 145–166), the program sets up the parallel work queue (line 169) and adds the first work item to the queue, namely a new state object with a partial path of just vertex 0 (line 172). The program then does a parallel loop over the work items in the work queue (line 173). For each work item (search state object), the program calls the `search()` method to do a breadth first or depth first search from that search state (line 177). If a Hamiltonian cycle is found, the program sets the `found` flag to true, which causes all the parallel team threads to stop their searches; the program calls the `stop()` method, which causes the parallel loop to stop processing the work queue; and the program prints the Hamiltonian cycle that was found (lines 178–184). When the parallel loop finishes, if a Hamiltonian cycle was not found, the program prints a message to that effect (lines 189–190).

To study the program's scalability, I want to find some input graphs for which the program takes a long time to find a Hamiltonian cycle. Finding such graphs turns out to be a bit difficult. I considered random graphs with V = 30 vertices and E = from 50 to 100 edges. For each E, I generated 100 different random graphs; ran each graph through a sequential version of the Hamiltonian cycle program; measured the running times; and counted how many of the 100 graphs had a Hamiltonian cycle. Figure 13.1 plots the program's median running time versus E, as well as the probability that a 30-vertex E-edge random graph has a Hamiltonian cycle versus E.

As E increases, the running time increases, plateaus, and then decreases. Also, the probability of a Hamiltonian cycle stays low until about E = 60,

```
112         // Determine if the given path element is adjacent to the
113         // current path element.
114         private boolean adjacent
115            (int a)
116            {
117            return adjacent[path[level]].contains (path[a]);
118            }
119
120         // Swap the given path elements.
121         private void swap
122            (int a,
123             int b)
124            {
125            int t = path[a];
126            path[a] = path[b];
127            path[b] = t;
128            }
129
130         // Returns a string version of this search state object.
131         public String toString()
132            {
133            StringBuilder b = new StringBuilder();
134            Formatter f = new Formatter (b);
135            for (int i = 0; i <= level; ++ i)
136               f.format (" %d", path[i]);
137            return b.toString();
138            }
139         }
140
141      // Main program.
142      public void main
143         (String[] args)
144         throws Exception
145         {
146         // Parse command line arguments.
147         if (args.length != 2) usage();
148         file = new File (args[0]);
149         threshold = Integer.parseInt (args[1]);
150
151         // Read input file, set up adjacency matrix.
152         Scanner s = new Scanner (file);
153         V = s.nextInt();
154         E = s.nextInt();
155         if (V < 1) usage ("V must be >= 1");
156         adjacent = new BitSet [V];
157         for (int i = 0; i < V; ++ i)
158            adjacent[i] = new BitSet (V);
159         for (int i = 0; i < E; ++ i)
160            {
161            int a = s.nextInt();
162            int b = s.nextInt();
163            adjacent[a].add (b);
164            adjacent[b].add (a);
165            }
166         s.close();
167
168         // Set up parallel work queue.
169         queue = new WorkQueue<State>();
```

Listing 13.1. HamCycSmp.java (part 3)

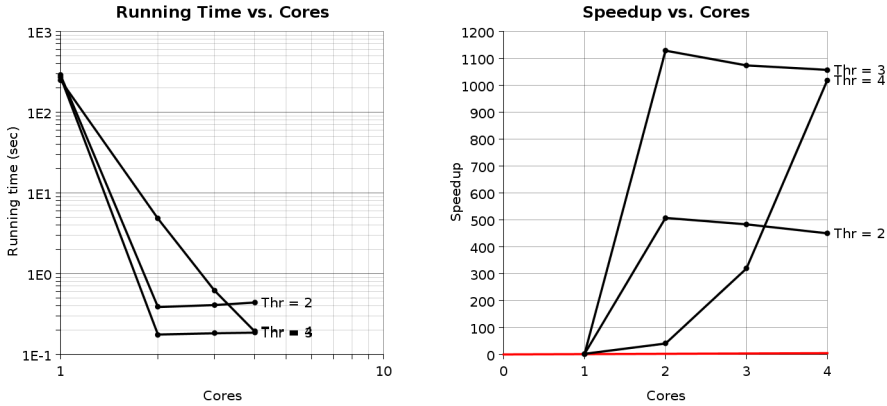


Figure 13.2. HamCycSmp running time metrics

then increases as E increases. Why? When E is small, there aren't many paths starting from vertex 0, so the program doesn't take very long to go through them all and either find a solution or conclude none are Hamiltonian cycles. When E is large, there are many, many paths starting from vertex 0, and most of them are Hamiltonian cycles, so the program doesn't take very long to find a solution. When E is somewhere in the middle, however, there are still many, many paths starting from vertex 0, but few if any of them are Hamiltonian cycles, so the program has to look at most or all of them before finding a solution or deciding there is no solution. This is what's taking a long time.

I picked a particular random graph with 30 vertices and 72 edges. I ran the HamCycSmp program on this graph, with the threshold level set to 2, 3, and 4. For each threshold level, I ran the program on one through four cores of a `tardis` node. Figure 13.2 plots the running times and speedups I measured. On one core, the program takes somewhat over 200 seconds to find a Hamiltonian cycle. But on two through four cores, the program takes drastically less time to find a solution, leading to ridiculous speedups of 500, 1000, or more. What's going on?

On one core, the program processes the subproblems one by one, from the beginning of the search tree to the end. But on more than one core, the program looks at several subproblems in the middle of the search tree at the same time in multiple threads. With this graph, it just so happens that one of the threads goes to work on a subproblem where a Hamiltonian cycle is found right away, so the running time is small. On one core, the one thread has to do complete searches on all the subproblems preceding the subproblem that contains a solution, and this takes a long time.

```
170
171         // Search the graph in parallel.
172         queue.add (new State (V));
173         parallelFor (queue) .exec (new ObjectLoop<State>()
174            {
175            public void run (State state)
176               {
177               State hamCycle = state.search();
178               if (hamCycle != null)
179                  {
180                  stop();
181                  found = true;
182                  System.out.printf ("Hamiltonian cycle =%s%n",
183                     hamCycle);
184                  }
185               }
186            });
187
188         // Print negative result.
189         if (! found)
190            System.out.printf ("No Hamiltonian cycle%n");
191         }
192
193      // Print an error message and exit.
194      private static void usage
195         (String msg)
196         {
197         System.err.printf ("HamCycSmp: %s%n", msg);
198         usage();
199         }
200
201      // Print a usage message and exit.
202      private static void usage()
203         {
204         System.err.println ("Usage: java pj2 " +
205            "edu.rit.pj2.example.HamCycSmp <file> <threshold>");
206         System.err.println ("<file> = Graph file");
207         System.err.println ("<threshold> = Parallel search " +
208            "threshold level");
209         throw new IllegalArgumentException();
210         }
211      }
```

Listing 13.1. HamCycSmp.java (part 4)

In general, a parallel program that searches a search tree will exhibit speedups that vary depending on the particular problem being solved. The speedups might be roughly the same as the number of cores, or they might be larger or smaller. I just happened to luck out when I picked this graph.

Under the Hood

As previously mentioned, when one of the parallel team threads in the HamCycSmp program finds a Hamiltonian cycle, the depth first searches in the other team threads must stop immediately. To signal this, the global `found` flag (line 25) is set to true. This variable is *shared* by the team threads. All the team threads *read* the `found` flag (line 98) and exit the depth first search loop if the flag is true. One of the team threads *writes* the `found` flag (line 181) when it finds a Hamiltonian cycle.

Two issues with the `found` flag now arise. The first issue has to do with thread synchronization. Normally, when some threads read a shared variable and other threads write a shared variable, the threads have to *synchronize* with each other. This ensures that one thread's operation on the variable does not interfere with another thread's operation on the variable. But there is no synchronization code around the `found` flag; the threads just read or write the variable directly. This works because the JVM ensures that all reads and writes of a Boolean variable are *atomic*. That is, while a read or a write is in progress on a Boolean variable, no other read or write will be performed on that variable until after the first read or write finishes. This atomic behavior implicitly synchronizes the multiple threads doing reads and writes of the `found` flag.

In fact, read and write operations on the primitive types `boolean`, `byte`, `char`, `short`, `int`, and `float` are guaranteed to be atomic (but not reads or writes of types `long` or `double`). Also, read and write operations on references to objects and arrays are guaranteed to be atomic (but operations on the objects' fields or the arrays' elements might or might not be atomic, depending on the types of the fields or elements). On the other hand, operations that involve both reading and writing a variable, such as incrementing an integer (`x++`), are *not* atomic and must be explicitly synchronized.

The second issue has to do with how the JVM deals with an object shared by multiple threads, like the HamCycSmp instance the `pj2` launcher creates when it runs the program. An object's fields are located in a block of storage that is allocated when the object is constructed. This block of storage resides in the computer's main memory. Suppose multiple threads have references to the object; that is, to the object's block of storage. When a thread reads a field from the shared object, the thread typically gets the field's value from main memory and puts the value in a register inside the CPU. Thereafter, the JVM is permitted to get the value *from the register* rather than from

the main memory. Furthermore, the JVM is permitted to store a new value *in the register* without writing the new value back to the main memory right away. The JVM will eventually put the register and the main memory back in sync, but there is no guarantee on how quickly this will happen. Thus, if one thread updates a field in a shared object, other threads might not see the new value immediately.

Consequently, in the HamCycSmp program, when one parallel team thread sets the `found` variable (field) to true, the other team threads might not see this new value right away, and the team threads might not exit their loops and terminate right away. This would cause the program to run longer than it should.

Here's where the `volatile` keyword comes in. When the program reads a `volatile` field, the JVM is required always to obtain the field's value from main memory, and not use a previous value that might be in a CPU register. When the program writes a `volatile` field, the JVM is required always to store the field's new value into main memory immediately, and not just update a CPU register.

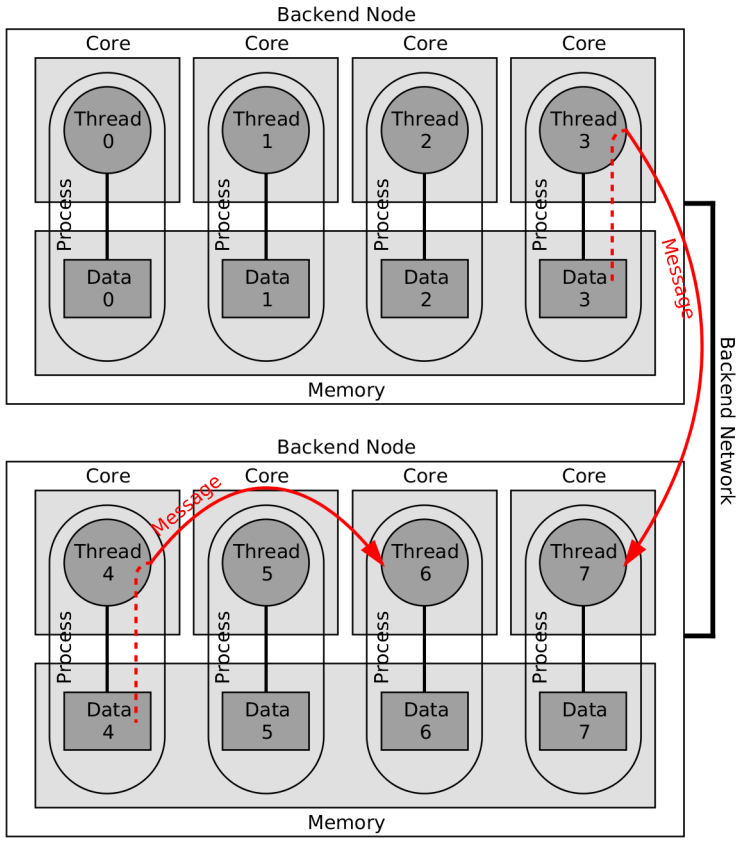
Thus, declaring the `found` field to be `volatile` ensures that all reads and writes of the `found` flag go to or from main memory, where all the parallel team threads can see the flag's value. This in turn ensures that the team threads terminate as soon as the flag is set to true; that is, as soon as a Hamiltonian cycle is found.

Points to Remember

- A *nondeterministic polynomial-time (NP) problem* is a decision problem for which a proof of a solution can be verified efficiently (with a polynomial-time algorithm).
- An efficient (polynomial-time) algorithm might or might not be known for finding the solution to an NP problem.
- An NP problem can always be solved by exhaustive search of all possible solutions; but the number of possible solutions typically grows much faster than any polynomial function as the problem size increases.
- Some NP problems can be solved by traversing a *search tree* consisting of all possible solutions.
- A search tree can be traversed by a *breadth first search* or a *depth first search.*
- A parallel program traversing a search tree should do a parallel breadth first search down to a certain threshold level, then should do multiple depth first searches in parallel thereafter.
- Use the *parallel work queue* pattern to do a parallel loop when the number of iterations is not known before the loop starts.

- The JVM guarantees that reads and writes of the primitive types `boolean`, `byte`, `char`, `short`, `int`, and `float`, as well as reads and writes of object and array references, are atomic.
- When a shared object has a field of one of the above primitive types that is read and written by multiple threads, declare the field to be `volatile`.

PART III
LOOSELY COUPLED CLUSTER



Parallel program running on a multi-node cluster

Chapter 14
Massively Parallel

Turning away from multicore parallel programming, we come to cluster parallel programming. A cluster parallel computer can have more—perhaps *many* more—cores than a single multicore node, hence can offer a higher degree of scalability than a single node. That's good. But in a cluster, the memory is distributed and cannot be shared by all the cores, requiring the program to do interprocess communication to move data from node to node and increasing the program's overhead. That's bad. But because there isn't a limit on the amount of memory in the cluster, as there is with a single node—if you need more memory, just add more nodes to the cluster—a memory-hungry parallel program can scale up to much larger problem sizes on a cluster. That's good. On balance, extreme-scale parallel computing has to be done on a cluster. Most of the world's fastest supercomputers on the Top500 List are cluster parallel computers.

An easy way to utilize the parallelism of a cluster is to run multiple independent instances of a sequential program. As an example, consider the bitcoin mining program from Chapter 3. The sequential program in that chapter mined (computed the golden nonce for) just one bitcoin. But suppose I want to mine *many* bitcoins, *K* of them. All I have to do is execute *K* processes, each process running one thread on one core of the cluster, each process running the sequential bitcoin mining program on a different coin ID (Figure 14.1). No thread needs to access the memory of another thread, or communicate with another thread. Each thread mines its own bitcoin and prints the result.

A parallel program like this, where each thread runs without interacting with any other thread, is called a *massively parallel program.* The lack of interaction minimizes the overhead and leads to ideal or near-ideal weak scaling. If I had a 1,000-core cluster, I could mine 1,000 bitcoins simultaneously, and the program would finish in the same time as one bitcoin. Such a program is also called *embarrassingly parallel*—there's so much parallelism, it's embarrassing!

I could mine a bunch of bitcoins in a massively parallel fashion by logging into the cluster and typing in a bunch of commands to run multiple instances of the sequential program. However, it would be far better to automate this and let the parallel program itself fire up all the necessary processes and threads. That's where the cluster parallel programming features of the Parallel Java 2 Library come in.

Listing 14.1 is the MineCoinClu program, a massively parallel cluster version of the bitcoin mining program. Let's take it one line at a time.

Whereas a multicore parallel program consists of a single task, a cluster parallel program is expressed as a *job* consisting of one or more tasks. Accordingly, class MineCoinClu extends class edu.rit.pj2.Job (line 8) rather than class edu.rit.pj2.Task. The job defines the tasks that will run, but the job itself does not do any computations. The tasks do all the work.

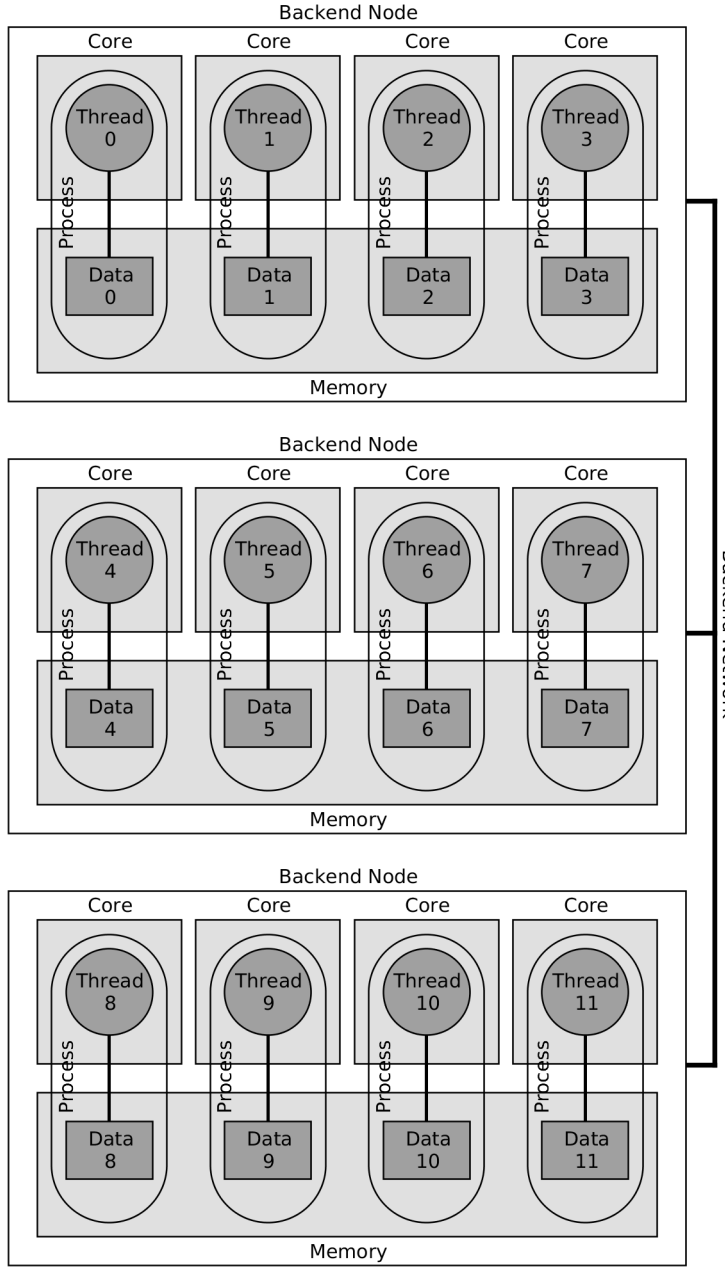


Figure 14.1. Massively parallel program running on a cluster

To run a cluster parallel program, run the `pj2` launcher program, giving it the job class name and any command line arguments. MineCoinClu's first argument is *N*, the number of most significant zero bits needed in the digest; the remaining arguments are one or more coin IDs to be mined. For example:

```
$ java pj2 edu.rit.pj2.example.MineCoinClu \
  16 0123456789abcdef 3141592653589793
```

`pj2` creates an instance of the job class, then calls the job's `main()` method, passing in an array of the command line argument strings. The code in the `main()` method sets up the job's tasks but does not actually execute the tasks. Once the `main()` method returns, `pj2` proceeds to create and execute the tasks, running each task on some node in the cluster.

Let's examine how the job's tasks are set up. Line 20 loops over the job's second and following command line arguments, setting up one task for each coin ID. A task is specified by calling the `rule()` method (line 21) to define a *rule*. The rule is an instance of class edu.rit.pj2.Rule. The rule object specifies several things:

- The rule specifies when the task is to be started. By default, the task is started when the job commences executing. (Other possibilities are to start the task later during the job's execution, or to start the task at the end of the job after all the other tasks have finished; we will see examples of such rules later.)
- The rule specifies the task that is to be executed. This is done by calling the rule's `task()` method, passing in the task's class (line 21). In this case, the task will be an instance of class MineCoinTask, which appears further down in the listing (line 38).

The `task()` method returns a *task spec*, an instance of class edu.rit.pj2.TaskSpec. The task spec specifies several things:

- The task spec specifies the task's command line arguments if any. This is done by calling the task spec's `args()` method (line 21). In this case the task's two command line arguments are the coin ID (which is the job's command line argument `args[i]`) and *N* (which is the job's `args[0]`).
- The task's default parallel loop properties—`threads`, `schedule`, and `chunk`—can be specified. (The MineCoinClu program doesn't need to do this.)
- Other attributes of the task can also be specified; we will look at some of these later.

Class Rule and class TaskSpec are separate classes because a rule can also specify a *task group* consisting of multiple tasks—one rule with many task specs. MineCoinClu does not need this capability, but some of the cluster parallel programs in later chapters will.

```
1  package edu.rit.pj2.example;
2  import edu.rit.crypto.SHA256;
3  import edu.rit.pj2.Job;
4  import edu.rit.pj2.Task;
5  import edu.rit.util.Hex;
6  import edu.rit.util.Packing;
7  public class MineCoinClu
8     extends Job
9     {
10    // Job main program.
11    public void main
12       (String[] args)
13       {
14       // Parse command line arguments.
15       if (args.length < 2) usage();
16       int N = Integer.parseInt (args[0]);
17       if (1 > N || N > 63) usage();
18
19       // Set up one task for each coin ID.
20       for (int i = 1; i < args.length; ++ i)
21          rule() .task (MineCoinTask.class) .args (args[i], args[0]);
22       }
23
24    // Print a usage message and exit.
25    private static void usage()
26       {
27       System.err.println ("Usage: java pj2 " +
28          "edu.rit.pj2.example.MineCoinClu <N> <coinid> " +
29          "[<coinid> ...]");
30       System.err.println ("<N> = Number of leading zero bits " +
31          "(1 .. 63)");
32       System.err.println ("<coinid> = Coin ID (hexadecimal)");
33       throw new IllegalArgumentException();
34       }
35
36    // Class MineCoinTask provides the Task that computes one coin
37    // ID's nonce in the MineCoinClu program.
38    private static class MineCoinTask
39       extends Task
40       {
41       // Command line arguments.
42       byte[] coinId;
43       int N;
44
45       // Mask for leading zeroes.
46       long mask;
47
48       // For computing hash digests.
49       byte[] coinIdPlusNonce;
50       SHA256 sha256;
51       byte[] digest;
52
53       // Timestamps.
54       long t1, t2;
55
56       // Task main program.
57       public void main
58          (String[] args)
```

Listing 14.1. MineCoinClu.java (part 1)

The MineCoinTask class comes next. It is a nested class inside the outer MineCoinClu job class. (The task class doesn't have to be a nested class; I wrote it that way for convenience.) Does the code in this class look familiar? It should; this class is virtually identical to the sequential program that mines one BitCoin, namely class MineCoinSeq from Chapter 3. The only thing I added was code to measure and print the task's running time (lines 62 and 100). I want to know how long it took each task to find its own golden nonce.

Here are two separate runs of the MineCoinClu program. Each runs one coin mining task on one core of the `tardis` cluster.

```
$ java pj2 edu.rit.pj2.example.MineCoinClu 28 0123456789abcdef
Job 65 launched Tue Jun 24 18:29:25 EDT 2014
Job 65 started Tue Jun 24 18:29:26 EDT 2014
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc
4c1e3329ca4
25702 msec
Job 65 finished Tue Jun 24 18:29:51 EDT 2014 time 26223 msec
$ java pj2 edu.rit.pj2.example.MineCoinClu 28 3141592653589793
Job 69 launched Tue Jun 24 18:33:43 EDT 2014
Job 69 started Tue Jun 24 18:33:43 EDT 2014
Coin ID = 3141592653589793
Nonce   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
4314 msec
Job 69 finished Tue Jun 24 18:33:48 EDT 2014 time 4697 msec
```

When running a cluster job, the `pj2` program automatically prints timestamps when the job launches, starts, and finishes. You can turn these and other debugging printouts on and off; refer to the `pj2` Javadoc for more information.

Here are the same two coin IDs mined by a single run of the MineCoinClu program in parallel on two cores of the `tardis` cluster.

```
$ java pj2 edu.rit.pj2.example.MineCoinClu \
  28 0123456789abcdef 3141592653589793
Job 70 launched Tue Jun 24 18:35:18 EDT 2014
Job 70 started Tue Jun 24 18:35:18 EDT 2014
Coin ID = 3141592653589793
Nonce   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
4286 msec
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc
4c1e3329ca4
25874 msec
Job 70 finished Tue Jun 24 18:35:44 EDT 2014 time 26239 msec
```

```
59          throws Exception
60          {
61          // Start timing.
62          t1 = System.currentTimeMillis();
63
64          // Parse command line arguments.
65          coinId = Hex.toByteArray (args[0]);
66          N = Integer.parseInt (args[1]);
67
68          // Set up mask for leading zeroes.
69          mask = ~((1L << (64 - N)) - 1L);
70
71          // Set up for computing hash digests.
72          coinIdPlusNonce = new byte [coinId.length + 8];
73          System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
74             coinId.length);
75          sha256 = new SHA256();
76          digest = new byte [sha256.digestSize()];
77
78          // Try all nonces until the digest has N leading zero bits.
79          for (long nonce = 0L; nonce <= 0x7FFFFFFFFFFFFFFFL;
80                ++ nonce)
81             {
82             // Test nonce.
83             Packing.unpackLongBigEndian
84                (nonce, coinIdPlusNonce, coinId.length);
85             sha256.hash (coinIdPlusNonce);
86             sha256.digest (digest);
87             sha256.hash (digest);
88             sha256.digest (digest);
89             if ((Packing.packLongBigEndian (digest, 0) & mask) ==
90                   0L)
91                {
92                // Stop timing and print result.
93                t2 = System.currentTimeMillis();
94                System.out.printf ("Coin ID = %s%n",
95                   Hex.toString (coinId));
96                System.out.printf ("Nonce   = %s%n",
97                   Hex.toString (nonce));
98                System.out.printf ("Digest  = %s%n",
99                   Hex.toString (digest));
100               System.out.printf ("%d msec%n", t2 – t1);
101               break;
102               }
103            }
104         }
105      }
106
107      // The task requires one core.
108      protected static in coresRequired()
109         {
110         return 1;
111         }
112      }
113   }
```

Listing 14.1. MineCoinClu.java (part 2)

Note that the whole job finishes as soon as the longest-running task finishes, as we'd expect when the tasks are performed in parallel in separate cores.

Got more Bitcoins? No problem!

```
$ java pj2 edu.rit.pj2.example.MineCoinClu \
  28 0123456789abcdef 3141592653589793 face2345abed6789 \
  0f1e2d3c4b5a6879
Job 72 launched Tue Jun 24 18:37:36 EDT 2014
Job 72 started Tue Jun 24 18:37:36 EDT 2014
Coin ID = 3141592653589793
Nonce   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
4301 msec
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc
4c1e3329ca4
25801 msec
Coin ID = 0f1e2d3c4b5a6879
Nonce   = 0000000001fe1c82
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e8
4f4c821cb92
67730 msec
Coin ID = face2345abed6789
Nonce   = 00000000195365d1
Digest  = 000000091061e29a6e915cd9c4ddef6962c9de0fc253c6cca82bc
8e3125a8085
860377 msec
Job 72 finished Tue Jun 24 18:51:57 EDT 2014 time 860718 msec
```

Under the Hood

When you run a Parallel Java 2 task on a multicore node, only one process is involved—the `pj2` process—and the task's parallel team threads all run within that process. But when you run a Parallel Java 2 job on a cluster, a whole bunch of processes running on multiple nodes have to get involved under the hood. These processes constitute the Parallel Java 2 *middleware* (Figure 14.2). The processes communicate with each other over the cluster's backend network using TCP sockets.

As already mentioned, you kick off the job by logging into the cluster's frontend node and running the `pj2` program, creating the *job process.* After instantiating the Job subclass and calling its `main()` method to define the job's rules, the job contacts the *Tracker,* a so-called "daemon" process that is always present on the frontend node. The job tells the Tracker that a new job has launched. Either immediately, or at a later time when resources are available, the Tracker tells the job to start. The job then requests the Tracker to run the job's task or tasks. Each request includes the Task subclass to be exe-

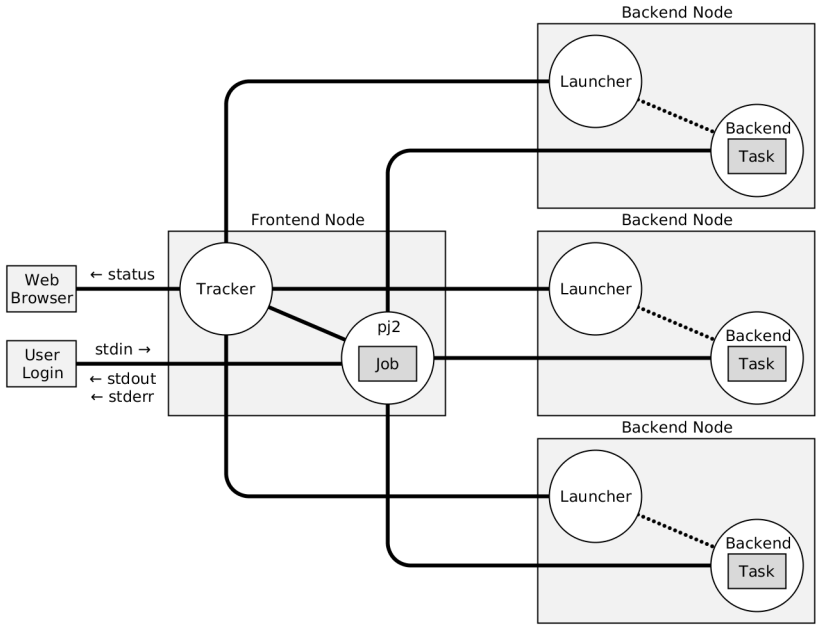


Figure 14.2. Parallel Java 2 cluster middleware

cuted, the resources the task requires (such as the number of cores needed), and other information. The task requests go into a queue in the Tracker.

The Tracker is keeping track of which cores on which cluster nodes are busy or idle. As idle cores become available, the Tracker assigns each task to one of the cluster's backend nodes. A *Launcher* daemon process is running on each backend node. For each task, the Tracker tells the Launcher to create a *backend process* on the Launcher's node. The backend process runs a special Backend program that is part of the middleware. The backend contacts the job and obtains the Task subclass to be executed, the task's command line arguments, and other information. The backend instantiates the Task subclass and calls the task's `main()` method, passing in the command line arguments. The `main()` method then carries out the task's computation. All this happens simultaneously for each of the job's tasks. Each task ends up executing in its own process on one of the cluster's backend nodes.

All the aforementioned processes remain in communication with each other as shown in Figure 14.2. Each backend periodically exchanges *heartbeat* messages with the job. If a heartbeat message fails to arrive when expected, the job (or the backend) detects that the backend (or the job) has failed, so the job (or the backend) terminates itself. Likewise, each Launcher

and each job exchanges heartbeat messages with the Tracker. If a Launcher fails, the Tracker no longer schedules tasks on that Launcher's node. If a job fails, the tracker removes the job and its tasks from the Tracker's queue. If the Tracker fails, the Launchers and the jobs terminate themselves.

Earlier we saw that a task can print stuff on `System.out`. (A task can also print on `System.err`.) These printouts appear on the console where the user typed the `pj2` command; that is, the console of the frontend process running on the frontend node. But the tasks are not running in the frontend process, or even on the frontend node! How can a task running in a backend process on a backend node cause printouts to appear on the frontend node's console? The Parallel Java 2 middleware takes care of this automatically. As we have already seen, all the characters a task prints are stored in an internal buffer. In a single-node environment, when the task terminates (or when it calls `flush()`), the buffer's contents are printed on the console. In a cluster environment, when the task terminates (or when it calls `flush()`), the middleware sends the buffer's contents from the backend process to the frontend process, and then the frontend process prints the characters on its console (the job's console). Furthermore, each task's print buffer is printed as a unit, without being intermingled with any other task's print buffer.

When a backend's task finishes, the backend informs the job, and the backend process terminates. When all of the job's tasks have finished, the job informs the Tracker, and the job process terminates. The Tracker then removes the job and its tasks from the Tracker's queue.

All of the above is going on for multiple users logged into the cluster and running multiple jobs simultaneously. The Tracker keeps track of all the tasks in all the jobs and ensures that each task will run only when the needed resources are available. In this way, the tasks have full use of the cores to which they are assigned, ensuring that every job gets the maximum possible performance out of the cluster.

The Tracker has a web interface. Using any web browser, you can go to the Tracker's URL and display the status of each node, core, job, and task in the cluster. The web interface provides status only; you can't change the order of jobs in the queue, terminate jobs, and so on.

Although I've described the Parallel Java 2 middleware at some length, you don't really need to be aware of the middleware when writing a cluster parallel program. Just define a Job subclass and Task subclasses as necessary. Then run the `pj2` program. The middleware does all its work behind the scenes.

One final detail: Depending on how the cluster is configured, you might need to use the `jar` option when you run the `pj2` program. This lets you provide a Java archive (JAR) containing the compiled class files for your program, which gets sent to each backend process. If the backend processes can't access the program's class files in the file system, and you omit the `jar`

option, the backend processes won't be able to instantiate the tasks, and the job will fail. See the `pj2` Javadoc for further information.

Points to Remember

- In a massively parallel program, each computation is sequential, and all the computations run in parallel independently with no interaction.
- Write a Parallel Java 2 cluster program as a subclass of class Job.
- Put the job's computational code in subclasses of class Task.
- Put code to define the job's rules in the job's `main()` method. The rules specify the tasks to be executed.
- Use the "`java pj2`" command to run your Parallel Java 2 cluster program. Use the `jar` option if necessary.

Chapter 15
Hybrid Parallel

The massively parallel bitcoin mining program in Chapter 14 doesn't necessarily take full advantage of the cluster's parallel processing capabilities. Suppose I run the program on the `tardis` cluster, which has 10 nodes with four cores per node, 40 cores total. Because the program mines each bitcoin sequentially on a single core, I have to mine 40 or more bitcoins to take full advantage of the cluster. If I mine fewer than 40 bitcoins, some of the cores will be idle. That's not good. I want to put those idle cores to use.

I can achieve better utilization of the cores if I run the *multithreaded* bitcoin mining program on each node, rather than the sequential program. That way, I would have four cores working on each bitcoin, rather than just one core. This is a *hybrid* parallel program (Figure 15.1). I run one process on each node, each process mining a different bitcoin. Inside each process I run multiple threads, one thread on each core, each thread testing a different series of nonces on the same bitcoin. The program has two levels of parallelism. It is massively parallel (with no interaction) between the nodes, and it is multithreaded parallel within each node. Consequently, I would expect to see a speedup relative to the original cluster program.

Listing 15.1 gives the hybrid parallel MineCoinClu2 program. The outer Job subclass is the same as in the previous MineCoinClu program. The job's `main()` method specifies a rule for each coin ID on the command line. Each rule runs an instance of the nested MineCoinTask defined later. The task's command line argument strings are *N*, the number of most significant zero bits in the digest, and the coin ID.

Next comes the nested MineCoinTask, a subclass of class Task, that contains the code for one of the bitcoin mining tasks. Here is where this program differs from the previous program. This program's task subclass is almost identical to the multithreaded parallel MineCoinSmp task from Chapter 3, rather than the single-threaded MineCoinSeq task. This program's task has the same parallel for loop as MineCoinSmp, with the same leapfrog schedule that divides the work of testing the possible nonces among the threads of the parallel thread team, each thread running on its own core of the node.

The MineCoinTask subclass does not override the `coresRequired()` method. So by default an instance of the task will use all the cores on the node where the task is running. The Tracker takes the tasks' required characteristics into account when scheduling tasks on the cluster. The Tracker will not execute a MineCoinTask instance until there is a node all of whose cores are idle.

I ran the hybrid parallel MineCoinClu2 program on the `tardis` cluster, giving it four coin IDs to mine. Here's what it printed:

```
$ java pj2 edu.rit.pj2.example.MineCoinClu2 \
  28 0123456789abcdef 3141592653589793 face2345abed6789 \
  0f1e2d3c4b5a6879
Job 74 launched Tue Jun 24 20:46:06 EDT 2014
```

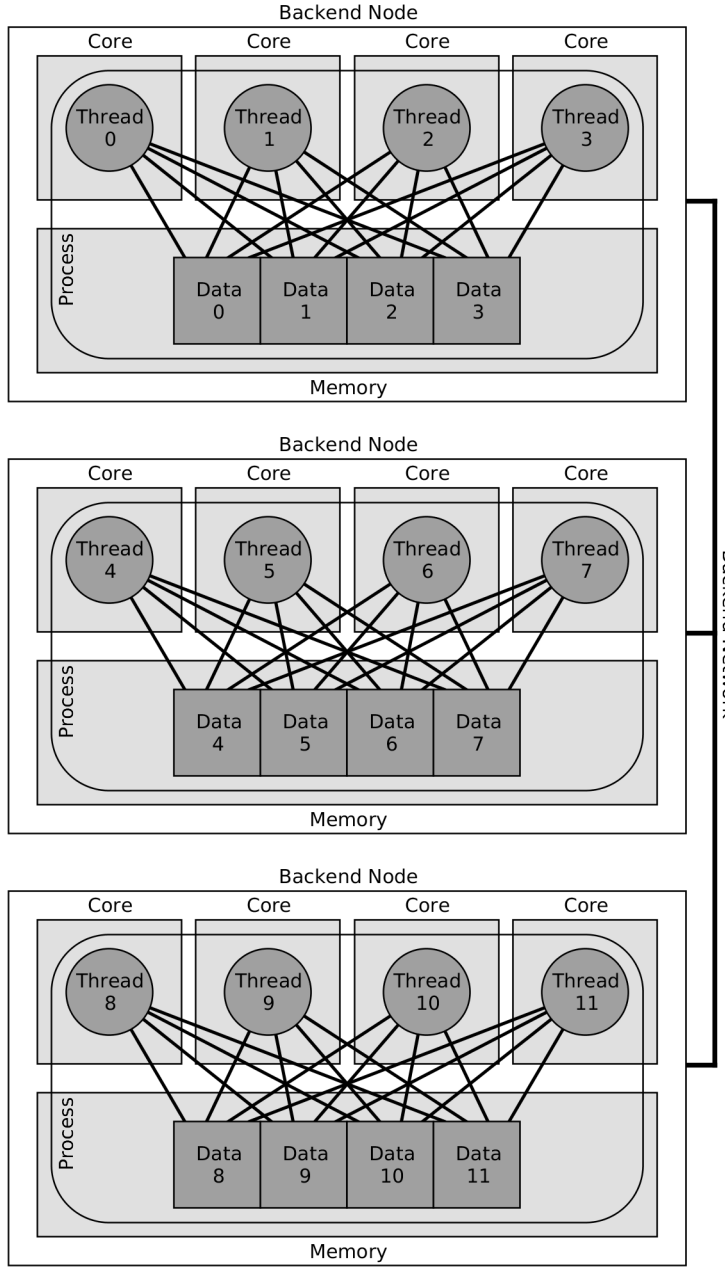


Figure 15.1. Hybrid parallel program running on a cluster

```
Job 74 started Tue Jun 24 20:46:07 EDT 2014
Coin ID = 3141592653589793
Nonce   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
1162 msec
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc
4c1e3329ca4
6466 msec
Coin ID = 0f1e2d3c4b5a6879
Nonce   = 0000000001fe1c82
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e8
4f4c821cb92
16873 msec
Coin ID = face2345abed6789
Nonce   = 00000000195365d1
Digest  = 000000091061e29a6e915cd9c4ddef6962c9de0fc253c6cca82bc
8e3125a8085
217812 msec
Job 74 finished Tue Jun 24 20:49:44 EDT 2014 time 218188 msec
```

For comparison, here's what the original MineCoinClu program printed, running on the `tardis` cluster with the same input:

```
$ java pj2 edu.rit.pj2.example.MineCoinClu \
  28 0123456789abcdef 3141592653589793 face2345abed6789 \
  0f1e2d3c4b5a6879
Job 72 launched Tue Jun 24 18:37:36 EDT 2014
Job 72 started Tue Jun 24 18:37:36 EDT 2014
Coin ID = 3141592653589793
Nonce   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
4301 msec
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc
4c1e3329ca4
25801 msec
Coin ID = 0f1e2d3c4b5a6879
Nonce   = 0000000001fe1c82
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e8
4f4c821cb92
67730 msec
Coin ID = face2345abed6789
Nonce   = 00000000195365d1
Digest  = 000000091061e29a6e915cd9c4ddef6962c9de0fc253c6cca82bc
8e3125a8085
860377 msec
Job 72 finished Tue Jun 24 18:51:57 EDT 2014 time 860718 msec
```

```
1  package edu.rit.pj2.example;
2  import edu.rit.crypto.SHA256;
3  import edu.rit.pj2.Job;
4  import edu.rit.pj2.LongLoop;
5  import edu.rit.pj2.Task;
6  import edu.rit.util.Hex;
7  import edu.rit.util.Packing;
8  public class MineCoinClu2
9     extends Job
10    {
11    // Job main program.
12    public void main
13       (String[] args)
14       {
15       // Parse command line arguments.
16       if (args.length < 2) usage();
17       int N = Integer.parseInt (args[0]);
18       if (1 > N || N > 63) usage();
19
20       // Set up one task for each coin ID.
21       for (int i = 1; i < args.length; ++ i)
22          rule() .task (MineCoinTask.class) .args (args[i], args[0]);
23       }
24
25    // Print a usage message and exit.
26    private static void usage()
27       {
28       System.err.println ("Usage: java pj2 " +
29          "edu.rit.pj2.example.MineCoinClu2 <N> <coinid> " +
30          "[<coinid> ...]");
31       System.err.println ("<N> = Number of leading zero bits " +
32          "(1 .. 63)");
33       System.err.println ("<coinid> = Coin ID (hexadecimal)");
34       throw new IllegalArgumentException();
35       }
36
37    // Class MineCoinClu2.MineCoinTask provides the Task that
38    // computes one coin ID's nonce in the MineCoinClu2 program.
39    private static class MineCoinTask
40       extends Task
41       {
42       // Command line arguments.
43       byte[] coinId;
44       int N;
45
46       // Mask for leading zeroes.
47       long mask;
48
49       // Timestamps.
50       long t1, t2;
51
52       // Task main program.
53       public void main
54          (String[] args)
55          throws Exception
56          {
57          // Start timing.
58          t1 = System.currentTimeMillis();
```

Listing 15.1. MineCoinClu2.java (part 1)

Comparing the two printouts, we see that the hybrid parallel program found exactly the same golden nonce for each coin ID as the original cluster parallel program, except it finished more quickly. To be precise, the speedups were 3.701, 3.990, 4.014, and 3.950, respectively for the four coin IDs. This shows that each task was indeed utilizing all four cores of the node where the task was running.

Under the Hood

As mentioned previously, the Parallel Java 2 cluster middleware includes a Tracker daemon running on the cluster's frontend node. The Tracker makes all the scheduling decisions for jobs and tasks running on the cluster. When a job executes one of its rules, the job sends a message to the Tracker, telling it to launch the rule's task. The message includes the required characteristics of the node on which the task is to run. These characteristics are specified by overriding protected static methods in the task subclass. You can specify any or all of the following:

- Override `coresRequired()` to specify the number of CPU cores the task requires, either a specific number of cores, or `ALL_CORES`. If not overridden, the default is to use all the cores on the node where the task runs.
- Override `gpusRequired()` to specify the number of GPU accelerators the task requires, either a specific number of GPUs, or `ALL_GPUS`. If not overridden, the default is to require no GPUs. (We will study GPU accelerated parallel programming later in the book.)
- Override `nodeNameRequired()` to specify the name of the node on which the task must run, either a specific node name, or `ANY_NODE_NAME`. If not overridden, the default is to let the task run on any node regardless of the node name.

You can also specify these characteristics in the rule that defines the task, by calling the appropriate methods on the TaskSpec. Refer to the Parallel Java 2 documention for further information.

The Tracker puts all launched tasks into a queue. There is one queue for each node in the cluster; tasks that require a specific node name go in the queue for that node. There is one additional queue for tasks that do not require a specific node name. The Tracker's scheduling policy is first to start tasks from the node-specific queues on those nodes, then to start tasks from the non-node-specific queue on any available nodes, until the queue is empty or until the first task in the queue requires more resources (CPU cores, GPU accelerators) than are available. Whenever a task finishes and its resources go idle, the Tracker starts as many pending tasks as possible. To guarantee fair access to the cluster's resources, the Tracker starts tasks from the queues in strict first-in-first-out (FIFO) order.

```
59
60          // Parse command line arguments.
61          coinId = Hex.toByteArray (args[0]);
62          N = Integer.parseInt (args[1]);
63
64          // Set up mask for leading zeroes.
65          mask = ~((1L << (64 - N)) - 1L);
66
67          // Try all nonces until the digest has N leading zero bits.
68          parallelFor (0L, 0x7FFFFFFFFFFFFFFFL)
69             .schedule (leapfrog) .exec (new LongLoop()
70             {
71             // For computing hash digests.
72             byte[] coinIdPlusNonce;
73             SHA256 sha256;
74             byte[] digest;
75
76             public void start() throws Exception
77                {
78                // Set up for computing hash digests.
79                coinIdPlusNonce = new byte [coinId.length + 8];
80                System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
81                   coinId.length);
82                sha256 = new SHA256();
83                digest = new byte [sha256.digestSize()];
84                }
85
86             public void run (long nonce) throws Exception
87                {
88                // Test nonce.
89                Packing.unpackLongBigEndian
90                   (nonce, coinIdPlusNonce, coinId.length);
91                sha256.hash (coinIdPlusNonce);
92                sha256.digest (digest);
93                sha256.hash (digest);
94                sha256.digest (digest);
95                if ((Packing.packLongBigEndian (digest, 0) & mask) ==
96                      0L)
97                   {
98                   // Stop timing and print result.
99                   t2 = System.currentTimeMillis();
100                  System.out.printf ("Coin ID = %s%n",
101                     Hex.toString (coinId));
102                  System.out.printf ("Nonce   = %s%n",
103                     Hex.toString (nonce));
104                  System.out.printf ("Digest  = %s%n",
105                     Hex.toString (digest));
106                  System.out.printf ("%d msec%n", t2 - t1);
107                  stop();
108                  }
109               }
110            });
111         }
112      }
113   }
```

Listing 15.1. MineCoinClu2.java (part 2)

I have found the Tracker's strict FIFO scheduling policy to be adequate for my teaching and research. I have not had the need, for example, to put priorities on tasks, so that higher-priority tasks can go ahead of lower-priority tasks in the queues. If your needs are different, you can write a modified version of the Tracker with a different scheduling policy; that's why I've released the Parallel Java 2 Library under the GNU GPL free software license.

Points to Remember

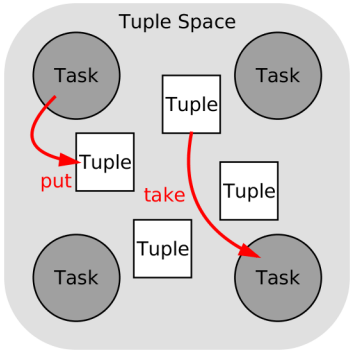
- A hybrid parallel cluster program exhibits multiple levels of parallelism. It runs in multiple processes, one process per node of the cluster; each process runs multiple threads, one thread per core of the node.
- A task in a job can use the multithreaded parallel programming constructs, such as parallel for loops, to run on multiple cores in a node.
- If necessary, specify a task's required characteristics—number of CPU cores, number of GPU accelerators, node name—by overriding the appropriate methods in the task subclass.

Chapter 16
Tuple Space

The massively parallel and hybrid parallel bitcoin mining programs in Chapters 14 and 15 were uncoupled parallel programs; there was no communication between the tasks. But in the next chapter we will start to encounter *coupled* parallel programs where the tasks *do* need to communicate with each other. So first, we have to look at how inter-task communication works in Parallel Java 2 programs.

A job maintains a conceptual repository of information called *tuple space*. David Gelernter originated the tuple space concept in a 1985 paper.* Tuple space holds *tuples*. Each tuple is an object whose fields contain appropriate information. The job's tasks can put tuples into tuple space and take tuples out of tuple space; any task can put or take any tuple. Tasks communicate with each other via tuple space. When multiple jobs run on a cluster, each job has its own separate tuple space; it is not possible to communicate between different jobs, only between tasks in the same job.

I liken tuple space to the ocean. Fish (tasks) inhabit the ocean (tuple space), and the water of the ocean surrounds and permeates all the fish. Whatever one fish puts into the water (tuples), any fish can take out of the water.



Recall that a job defines a number of *rules*. Each rule in turn defines one or more *tasks*. The rules in a job, the tasks in a rule, and the tuples taken and put by a task are all intimately interrelated.

Figure 16.1 is a timeline of an example job illustrating the three possible kinds of rules. Time increases from left to right as the job executes. This particular job consists of four rules.

The first and second rules are *start rules*. A job can have any number of start rules. The start rules are coded like this in the job's `main()` method:

```
rule() .task (TaskA.class);
rule() .task (TaskB.class);
```

The first rule says, "At the start of the job, run an instance of task class

* D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, January 1985.

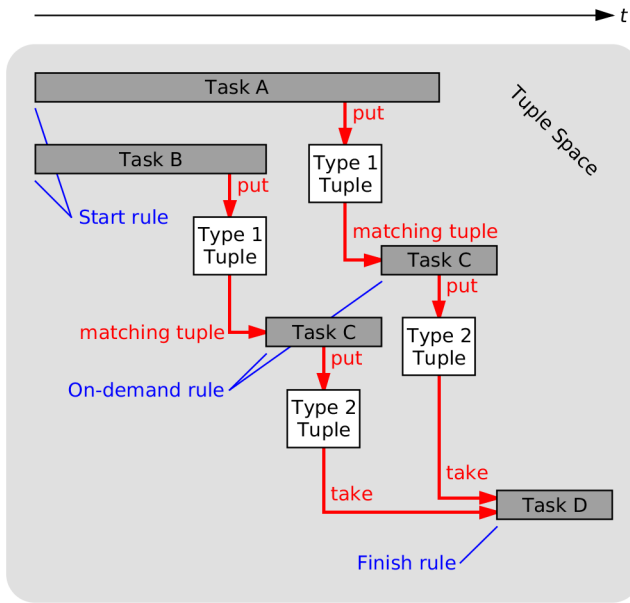


Figure 16.1. Example of rules, tasks, and tuples in a job

TaskA." The second rule says, "At the start of the job, run an instance of task class TaskB." Other attributes of the task, such as the task's command line arguments, can also be specified as part of the rule.

Each rule *fires* at some point during the job's execution. A start rule fires when the job commences executing. Thus, an instance of Task A and an instance of Task B start at time zero in Figure 16.1. (Keep in mind that the job's `main()` method only *defines* the rules. The job actually begins executing once the job's `main()` method returns.)

Task A and Task B each put a tuple into tuple space. A tuple is an object that is an instance of a subclass of class edu.rit.pj2.Tuple. In Figure 16.1, Task A and Task B each create an instance of class Type1Tuple. A tuple object carries content, which is stored in the tuple's fields as defined in the tuple subclass. Each task calls the `putTuple()` method to put its tuple into tuple space.

The third rule in this example job is an *on-demand rule*. A job can have any number of on-demand rules. The on-demand rule is coded like this in the job's `main()` method:

```
rule() .whenMatch (new Type1Tuple()) .task (TaskC.class);
```

This rule says, "Whenever a tuple appears in tuple space that matches the

given Type1Tuple, run an instance of task class TaskC." The `whenMatch()` method's argument is a *template* tuple. The rule fires when there is a *match* between the given template and a *target* tuple in tuple space. By default, a target matches a template if the target is an instance of the same class as the template, or if the target is an instance of a subclass of the template's class. (The tuple matching criterion can be changed; we will see examples of this later.)

An on-demand rule can fire not at all, once, or more than once during the course of a job. At a certain point in Figure 16.1, Task B puts a Type 1 tuple into tuple space. This tuple matches the third rule's template, so at this point an instance of Task C starts. Later, Task A puts another Type 1 tuple into tuple space. This tuple also matches the third rule's template, so at this point another instance of Task C starts. The two Task C instances run independently.

When an on-demand rule fires, the target tuple that triggered the rule is automatically taken out of tuple space and is provided to the task as the *matching tuple*. The task can retrieve its matching tuple by calling the `getMatchingTuple()` method. (A task triggered by a start rule, like Task A or Task B, has no matching tuple.) Information, recorded in the fields of the tuple, flows from one task to another task via tuple space; this is how different parts of the cluster parallel program communicate with each other.

An on-demand rule can specify more than one template, by including more than one `whenMatch()` clause in the rule definition. In this case the rule fires whenever *every* template has a matching target in tuple space. Each template must match a *different* target. If there are matches for some but not all of the templates, the rule does not fire. (The criterion for matching multiple templates can be changed.) When the rule fires, all the matching target tuples are automatically taken out of tuple space and are provided to the task as its multiple matching tuples.

In Figure 16.1, each instance of Task C puts a Type 2 tuple into tuple space. But because there are no rules that match a Type 2 tuple, no tasks are triggered when the Type 2 tuples appear. The tuples simply sit in tuple space.

The fourth and final rule in this example job is a *finish rule*. A job can have any number of finish rules. The finish rule was coded like this in the job's `main()` method:

```
rule() .atFinish() .task (TaskD.class);
```

This rule says, "When all other tasks have finished executing, run an instance of task class TaskD." In other words, a finish rule triggers a task to run at the end of the job. If a job has more than one finish rule, then more than one finish task is started at the end of the job. In Figure 16.1, Task D starts as soon as the last Task C finishes.

An on-demand task automatically obtains matching tuples when the task

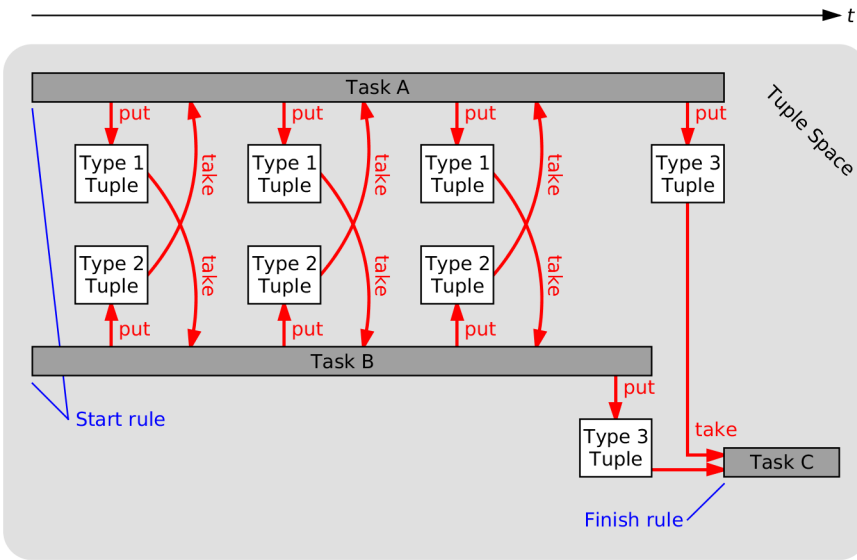


Figure 16.2. Another example of rules, tasks, and tuples in a job

triggers. But that's not the only way a task can obtain tuples. A task can also explicitly *take* a tuple out of tuple space by calling the `takeTuple()` method. The `takeTuple()` method's argument is a template tuple. The `takeTuple()` method waits until a target tuple exists in tuple space that matches the template tuple. Then the `takeTuple()` method removes that target tuple from tuple space and returns the target tuple. This is what Task D does; it repeatedly takes a Type 2 tuple.

Figure 16.2 shows another example of a job. This job has two start rules and one finish rule. Task A and Task B start when the job commences. Each task computes a series of intermediate results. Task B needs Task A's intermediate results, and Task A needs Task B's intermediate results. Task A executes code like this:

```
Type2Tuple in;
Type2Tuple template = new Type2Tuple();
for (...)
   {
   ... // Calculate intermediate result
   putTuple (new Type1Tuple (...));
   in = takeTuple (template);
   ...
   }
```

After calculating an intermediate result, Task A packages that up in a Type 1

tuple and puts the tuple into tuple space. Task A then takes a tuple that matches a certain template. The template is a Type 2 tuple. The `takeTuple()` method blocks until a Type 2 tuple is present in tuple space. When Task B puts such a tuple, the `takeTuple()` method unblocks, the matching tuple is removed from tuple space, and the `takeTuple()` method returns the tuple that was removed (`in`). Task B executes similar code, except the tuple types are reversed:

```
Type1Tuple in;
Type1Tuple template = new Type1Tuple();
for (...)
   {
   ... // Calculate intermediate result
   putTuple (new Type2Tuple (...));
   in = takeTuple (template);
   ...
   }
```

Eventually, Task A and Task B complete their calculations. Each task puts its final results into tuple space as a Type 3 tuple. The finish rule then fires, Task C runs, Task C takes the Type 3 tuples out of tuple space, and Task C reports the final results.

The following methods in class edu.rit.pj2.Task allow a task to interact with tuple space. For further information, refer to the Parallel Java 2 documentation.

- `getMatchingTuple()` lets an on-demand rule's task retrieve its matching tuples, which were automatically taken out of tuple space when the rule fired.
- `putTuple()` lets a task put a tuple—or, optionally, multiple copies of a tuple—into tuple space.
- `takeTuple()` lets a task take a tuple out of tuple space. A target tuple in tuple space that matches a given template tuple is removed and returned. If there are no matching tuples, `takeTuple()` blocks until a matching tuple shows up. If there is more than one matching tuple, `takeTuple()` removes and returns one of them; the matching tuple is chosen in an unspecified manner.
- `tryToTakeTuple()` does a *conditional* take. This is the same as a regular take, except if there are no matching tuples, `tryToTakeTuple()` does not block and immediately returns null.
- `readTuple()` lets a task make a copy of a tuple from tuple space. A target tuple in tuple space that matches a given template tuple is copied and returned, while the original target tuple remains in tuple space. If there are no matching tuples, `readTuple()` blocks until a matching tuple shows up. If there is more than one matching tuple, `readTuple()` copies and returns one of them; the matching tuple is chosen in an unspecified

manner.

- `tryToReadTuple()` does a *conditional* read. This is the same as a regular read, except if there are no matching tuples, `tryToReadTuple()` does not block and immediately returns null.
- `addTupleListener()` lets a task specify a tuple related action to be performed at a later time. The argument is a *tuple listener* object, an instance of a subclass of class edu.rit.pj2.TupleListener. When constructing a tuple listener object, you specify a template and an action, either "read" or "take." You also override the tuple listener's `run()` method. After that, when a tuple that matches the tuple listener's template appears in tuple space, the tuple listener is triggered. The matching tuple is read or taken from tuple space, and the tuple listener's `run()` method is called, passing in the matching tuple as the argument. The `run()` method does whatever is necessary to process the tuple. The `addTupleListener()` method returns immediately; this lets the task proceed to do other work, rather than blocking the task until a matching tuple appears in tuple space.

As mentioned previously, by default, a target tuple matches a template tuple if the target is an instance of the same class as the template, or if the target is an instance of a subclass of the template's class. The matching algorithm for a particular kind of tuple can be changed by overriding the `matchClass()` and `matchContent()` methods in the tuple subclass. The `matchClass()` method lets you specify which type or types of target tuple match a template tuple. The `matchContent()` method lets you compare the content (fields) of the template and the target to decide whether there is a match.

As you've probably gathered, tuple space is very flexible. Coupled with the rule-driven parallel task execution, the possibilities are endless for organizing and coordinating the pieces of a cluster parallel program. In the next few chapters we'll see several example programs that illustrate the various kinds of rules and the various patterns for inter-task communication using tuple space.

To use tuples in a cluster parallel program, you have to define one or more *tuple subclasses.* A tuple subclass must fulfill these requirements:

- The tuple subclass must extend the base class edu.rit.pj2.Tuple.
- The tuple subclass must define fields to carry the tuple's content. The fields can be anything at all—primitive types, objects, arrays, and so on.
- The tuple subclass must define a no-argument constructor.
- The tuple subclass must implement the `writeOut()` and `readIn()` methods, which are declared in interface edu.rit.io.Streamable, which is implemented by class edu.rit.pj2.Tuple. We'll see examples of these methods below.

A tuple subclass can have other constructors and methods in addition to the

ones listed above.

Once you've defined a tuple subclass, you can create tuples—instances of the tuple subclass—as you would any Java objects. However, the Parallel Java 2 middleware assumes that a tuple is immutable; that is, the tuple's state will not change once it has been initialized. If you need to change a tuple's contents, create a *deep copy* of the original tuple and change the copy.

To illustrate how to write a tuple subclass, let's use the bitcoin mining program from Chapter 14. Suppose that a bitcoin mining task, instead of printing its results, wants to package its results into a tuple. Listing 16.1 shows what the tuple subclass would look like.

The tuple subclass, MineCoinResult, extends the Tuple base class (line 6). I defined four fields to hold the tuple's content, namely the bitcoin mining task's results—the coin ID, nonce, digest, and running time in milliseconds (lines 9–12). I made the fields public so I could access them directly; making the fields private and adding public accessor methods is overkill for this simple tuple subclass. I defined the required no-argument constructor (lines 15–17) that leaves the fields at their default values. I also defined another constructor (lines 20–30) that initializes the fields to given values. A bitcoin mining task would use the latter constructor to package its results into a tuple. The constructor makes a deep copy of its arguments by *cloning* the byte arrays rather than *assigning* the byte array references. If the bitcoin mining task later changes the contents of the byte arrays passed in as the constructor arguments, the tuple's byte arrays do not change (the tuple must be immutable).

Class MineCoinResult defines the two required methods, `writeOut()` and `readIn()` (lines 33–52). These methods are inherited from the superclass Tuple, which implements interface edu.rit.io.Streamable, which declares those methods. A class that implements interface Streamable defines a *streamable object.* A streamable object can write its contents to an *out stream* (class edu.rit.io.OutStream). Class OutStream has methods for writing values of primitive types (`int`, `long`, and so on), strings, arrays, and objects. Class MineCoinResult's `writeOut()` method calls a few of these methods to write its fields. Each of these methods converts its argument to a sequence of bytes and writes the bytes to the given out stream. Likewise, a streamable object can read its contents from an *in stream* (class edu.rit.io.InStream). Class InStream has methods for reading values of all kinds. Class MineCoinResult's `readIn()` method calls a few of these methods to read its fields. Each of these methods reads a sequence of bytes from the given in stream, converts the bytes to a value of the proper kind, and returns the value. The fields must be read back in exactly the same order as they were written out.

Tuples have to be streamable objects because tuples are sent from one process (node) to another process when the parallel program runs on a cluster. But objects, such as tuples, cannot be sent directly from one process to

```
1  import edu.rit.io.InStream;
2  import edu.rit.io.OutStream;
3  import edu.rit.pj2.Tuple;
4  import java.io.IOException;
5  public class MineCoinResult
6     extends Tuple
7     {
8     // Content fields.
9     public byte[] coinID;
10    public long nonce;
11    public byte[] digest;
12    public long msec;
13
14    // Construct a new uninitialized mine coin result tuple.
15    public MineCoinResult()
16       {
17       }
18
19    // Construct a new mine coin result tuple with the given content.
20    public MineCoinResult
21       (byte[] coinID,
22        long nonce,
23        byte[] digest,
24        long msec)
25       {
26       this.coinID = (byte[]) coinID.clone();
27       this.nonce = nonce;
28       this.digest = (byte[]) digest.clone();
29       this.msec = msec;
30       }
31
32    // Write this tuple to the given out stream.
33    public void writeOut
34       (OutStream out)
35       throws IOException
36       {
37       out.writeByteArray (coinID);
38       out.writeLong (nonce);
39       out.writeByteArray (digest);
40       out.writeLong (msec);
41       }
42
43    // Read this tuple from the given in stream.
44    public void readIn
45       (InStream in)
46       throws IOException
47       {
48       coinID = in.readByteArray();
49       nonce = in.readLong();
50       digest = in.readByteArray();
51       msec = in.readLong();
52       }
53    }
```

Listing 16.1. Tuple subclass MineCoinResult

another. Byte streams, though, *can* be sent between processes. So a process needing to send an object to another process must convert the object to a sequence of bytes and send the byte stream. The process at the other end must then receive the byte stream and convert it back to an object. The Parallel Java 2 Library does most of the work under the hood. You, the programmer, just have to supply the code to write and read a streamable object's fields, namely the `writeOut()` and `readIn()` methods.

What I've just described sounds suspiciously like Java's built-in object serialization capability in package java.io. You're correct; the Parallel Java 2 Library has its own object serialization capability. Why not use java.io? Mainly because Java Object Serialization produces byte sequences that are far longer than they need to be. As we will discover as we study cluster parallel programming, minimizing the number of bytes sent from process to process is extremely important. I designed Parallel Java 2's streamable object capability to generate as few bytes as possible.

I'm not going to write a cluster parallel bitcoin mining program that uses tuples to report its results. (You can do that yourself if you like.)

Under the Hood

Tuple space is implemented as follows. The tuples are stored inside the Job object running in the job process on the cluster's frontend node. The job maintains a list of all the tuples that have been put into tuple space but have not yet been taken out. The job also maintains a list of pending take and read requests whose templates have not yet matched any tuples.

When a task calls `putTuple()`, the task process sends a message containing the tuple to the job process, which adds the tuple to its list. The task streams the tuple out as it sends the message, and the job streams the tuple in as it receives the message. This is why every tuple subclass needs to be streamable. When streaming in a tuple, the in stream needs to construct an instance of the tuple subclass in which to store the incoming content. This is why every tuple subclass needs to have a no-argument constructor.

When a task calls `takeTuple()`, the task sends a message containing the template to the job, which adds the template to the list of pending requests. The `takeTuple()` method then blocks waiting to receive a response. Whenever the job receives a take-request message or a put-tuple message, the job checks its list of pending requests against its list of tuples. If the job finds a match between a template and a tuple, the job removes the matching tuple from its list, and the job sends a response message with the tuple back to the task that originally made the request. The `takeTuple()` method in the task then unblocks and returns the tuple. The `tryToTakeTuple()` method works the same way, except if there is no tuple that matches the template, the job immediately sends back a response to that effect. The `readTuple()` and `try-`

`ToReadTuple()` methods work the same way, except the matching tuple is not removed from the job's list of tuples.

This design, with the job process acting as a central server of tuples, has a couple of consequences. First, the job process has to have enough memory to hold all the tuples that could ever be in tuple space at the same time. If the Java Virtual Machine's default heap size is not large enough, you have to specify a larger heap size when you run the "`java pj2`" command (refer to the `java` command documentation). Second, all inter-task communication via tuple space has to go through the central job process; the task processes do not communicate with each other directly. Thus, each put or take of a tuple incurs a certain amount of network communication overhead to transfer the tuple from the sending task to the job to the receiving task.

When I designed the cluster parallel programming features of the Parallel Java 2 Library, I decided to orient them primarily towards the loosely coupled and medium coupled region of the coupling spectrum (as described in Chapter 1). The tasks run for minutes or hours, not milliseconds; the tasks communicate with each other infrequently (relative to the speed of the CPU), not every microsecond; and the tasks communicate small to medium amounts of data, not enormous quantities of data. In such an environment, the time the job spends in network communication is small relative to the time spent in computation, so it's okay if all tuples go through the central job process. Because the tuples do not carry enormous quantities of data, it's okay to store them all in the job process's memory.

You can write tightly coupled cluster parallel programs with the Parallel Java 2 Library—programs that do frequent inter-task communication, or that communicate large quantities of data, or both. Just realize that these are outside the design center. As I said before, if you need a very high performance tightly coupled cluster parallel program, a parallel programming library intended for that environment, such as MPI, is probably a better choice.

I also had cloud computing in mind when I designed Parallel Java 2's cluster parallel programming features. When you run a Parallel Java 2 program on a "cloud cluster," the frontend process runs on your machine, but the backend processes run on virtual machines in the cloud. It's fairly easy to set up a connection from a virtual machine in the cloud to your machine (from a backend process to the frontend process). It's not so easy to set up a connection from one virtual machine to another in the cloud. I therefore decided to put tuple space in the frontend process and have the backend processes talk to the frontend process to transfer tuples, rather than have the backend processes talk directly to each other. (The Parallel Java 2 Library does not support clusters in the cloud yet.)

Points to Remember

- A Parallel Java 2 job consists of tasks specified by rules.
- When a rule fires, an instance of the rule's task is created and executed.
- Each task normally runs in its own separate process on one of the cluster's backend nodes.
- A start rule fires when the job commences execution.
- An on-demand rule fires whenever a target tuple appears in tuple space that matches a given template tuple. The target becomes the task's matching tuple.
- A finish rule fires at the end of the job, once all other tasks have finished.
- A task can put tuples into tuple space.
- A task can also read tuples in tuple space and take tuples out of tuple space, with or without blocking, by specifying a template to match.
- A task can use a tuple listener to read or take a tuple that matches a specified template.
- Normally, a target matches a template if the target is an instance of the same class as the template, or a subclass thereof.
- The tuple matching criterion can be altered by overriding methods in the tuple subclass.
- Every tuple subclass must have a no-argument constructor and must be streamable.
- Every tuple must be immutable—its state must not change once it has been initialized. Make a deep copy of a tuple and change the copy if necessary.

Chapter 17
Cluster Parallel Loops

The massively parallel bitcoin mining program in Chapter 15 *still* doesn't take full advantage of the cluster's parallel processing capabilities. Each bitcoin mining task uses all the cores on just one node. So on the 10-node `tardis` cluster, I have to mine 10 or more bitcoins to fully utilize all the cores. Furthermore, the most parallelism I can achieve for each bitcoin is the number of cores on a node, four cores in the case of `tardis`. What if I want to use *every core in the cluster to mine a single bitcoin* in parallel?

The multithreaded bitcoin mining program in Chapter 3 utilizes all the cores on one node by converting the plain sequential for loop over the nonces into a parallel for loop. Each parallel team thread, running on its own core in parallel with the other team threads, computes a subset of the nonces. I used a leapfrog schedule to partition the nonces among the threads. Running with, say, a team of four threads, thread 0 computes nonces 0, 4, 8, 12, . . .; thread 1 computes nonces 1, 5, 9, 13, . . .; and so on. When a thread finds the golden nonce, that thread tells the parallel for loop to stop, and then every thread exits the loop. All the machinery that creates the threads, that decides which threads would compute which nonces, that tells the threads to exit the loop, and so on is hidden inside the parallel for loop object. Because all the threads are in the same process, they all can access the loop control information, which is stored in hidden shared variables.

I can follow the same strategy to design a cluster parallel bitcoin mining program: I run a thread on each core of the cluster. Following the hybrid parallel programming pattern, the threads reside in tasks, one task on each node of the cluster. Then I have to partition the loop iterations among all the threads in all the tasks. Because the threads are not all in the same process, I can't use a hidden shared variable to do the partitioning.

This brings us to the *master-worker* pattern (Figure 17.1) for parallel loops in a cluster parallel program. There is a *master task* and a number of *worker tasks*, each worker having a unique *rank* (0, 1, 2, and so on). The master task is in charge of the loop schedule; it partitions the loop indexes into chunks in some manner, such as one chunk per worker task (a fixed schedule), multiple equal-sized chunks (a dynamic schedule), or any of the other schedules. Each worker sends a message to the master requesting a chunk of work for that worker's rank; the master sends a message back to the worker with the next available chunk for that rank. Each worker computes the designated chunk of work and puts the results in the appropriate place (the workers' results are not shown in Figure 17.1). Whenever a worker finishes a chunk, the worker requests and obtains the next available chunk from the master. This continues until the master informs the worker that there are no more chunks, whereupon the worker terminates.

In a Parallel Java 2 program, which uses tuple space for inter-task communication, the master-worker pattern is modified slightly (Figure 17.2). The job's main program partitions the loop iterations into chunks all at once and

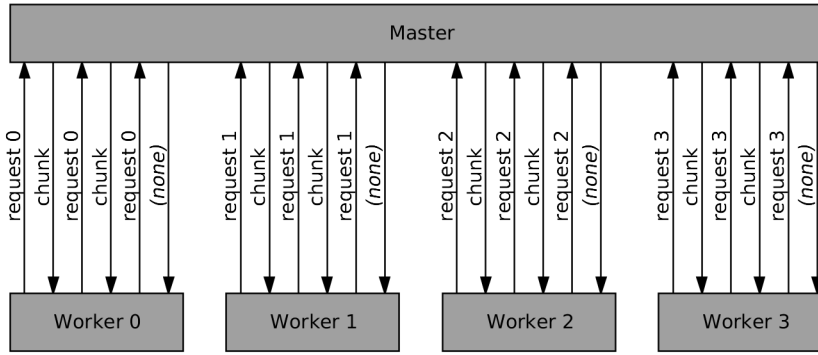


Figure 17.1. Master-worker pattern

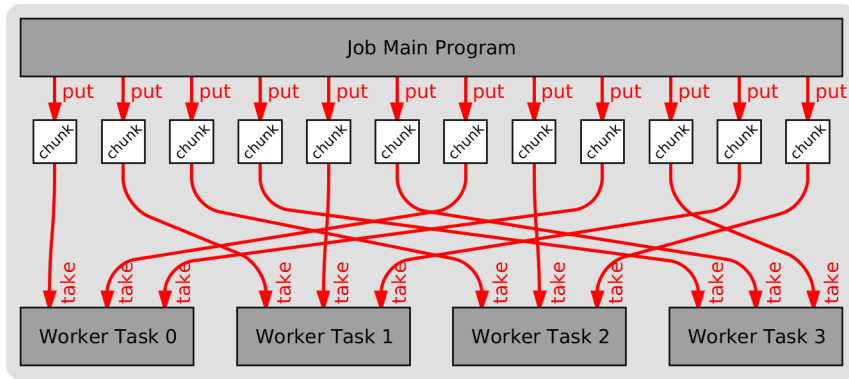


Figure 17.2. Master-worker pattern in tuple space

puts a tuple containing each chunk into tuple space. Then, once the worker tasks start, each worker repeatedly takes a chunk tuple out of tuple space, computes the designated chunk, and puts the result somewhere. The worker does a *conditional* take so that, when all the chunks have been used up, the worker will detect that there is no more work.

I also need to stop all the threads in all the tasks whenever one thread finds the golden nonce. Because the threads are not all in the same process, I can't use a hidden shared variable for this. Instead, I have to use tuple space. The thread that finds the golden nonce will inform the other threads by putting a special "stop tuple" into tuple space, where the other threads can see it.

Class Job and class Task provide methods to support the master-worker parallel loop pattern in a cluster parallel program, as well as the multi-

threaded parallel loop pattern in a multicore parallel program. Recall that in a multicore parallel program, you set up a parallel loop like this in the task class:

```
public class MyTask extends Task
   {
   public void main (String[] args)
      {
      parallelFor (0, 999999) .exec (new Loop()
         {
         public void run (int i)
            {
            Loop body code for iteration i
            }
         });
      }
   }
```

In a cluster parallel program, you set up a master-worker parallel loop like this; part is in the job class, the rest is in the task class:

```
public class MyJob extends Job
   {
   public void main (String[] args)
      {
      masterFor (0, 999999, MyTask.class);
      }
   }
public class MyTask extends Task
   {
   public void main (String[] args)
      {
      workerFor() .exec (new Loop()
         {
         public void run (int i)
            {
            Loop body code for iteration i
            }
         });
      }
   }
```

In the job main program, the `masterFor()` method specifies the inclusive lower and upper bounds for the loop as well as the worker task class. Under the hood, this sets up a start rule to run one or more instances of the worker task class. The number of worker tasks is specified by the job's `workers` property; you can specify this property on the `pj2` command line or by calling the job's `workers()` method; if not specified, the default is one worker task. The `masterFor()` method also partitions the index range among the worker tasks and writes the appropriate chunk tuples into tuple space. The index partitioning is specified by the job's `masterSchedule` and `masterChunk`

```
1  package edu.rit.pj2.example;
2  import edu.rit.crypto.SHA256;
3  import edu.rit.pj2.Job;
4  import edu.rit.pj2.LongLoop;
5  import edu.rit.pj2.Task;
6  import edu.rit.pj2.TupleListener;
7  import edu.rit.pj2.tuple.EmptyTuple;
8  import edu.rit.util.Hex;
9  import edu.rit.util.Packing;
10 public class MineCoinClu3
11    extends Job
12    {
13    // Job main program.
14    public void main
15       (String[] args)
16       {
17       // Parse command line arguments.
18       if (args.length != 2) usage();
19       String coinId = args[0];
20       int N = Integer.parseInt (args[1]);
21       if (1 > N || N > 63) usage();
22
23       // Set up master-worker cluster parallel for loop.
24       masterSchedule (leapfrog);
25       masterFor (0L, 0x7FFFFFFFFFFFFFFFL, WorkerTask.class)
26          .args (coinId, ""+N);
27       }
28
29    // Print a usage message and exit.
30    private static void usage()
31       {
32       System.err.println ("Usage: java pj2 [workers=<K>] " +
33          "edu.rit.pj2.example.MineCoinClu3 <coinId> <N>");
34       System.err.println ("<K> = Number of worker tasks " +
35          "(default 1)");
36       System.err.println ("<coinId> = Coin ID (hexadecimal)");
37       System.err.println ("<N> = Number of leading zero bits " +
38          "(1 .. 63)");
39       throw new IllegalArgumentException();
40       }
41
42    // Class MineCoinClu3.WorkerTask provides the worker Task in the
43    // MineCoinClu3 program. The worker tasks perform the golden
44    // nonce search.
45    private static class WorkerTask
46       extends Task
47       {
48       // Command line arguments.
49       byte[] coinId;
50       int N;
51
52       // Mask for leading zeroes.
53       long mask;
54
55       // For early loop exit.
56       volatile boolean stop;
57
```

Listing 17.1. MineCoinClu3.java (part 1)

properties; you can specify these on the `pj2` command line or by calling the job's `masterSchedule()` and `masterChunk()` methods; if not specified, the default is a fixed schedule.

In the worker task class main program, the `workerFor()` method creates a parallel loop object that executes the given loop body (a subclass of class Loop). Under the hood, the parallel loop object takes chunk tuples for the worker from tuple space and, for each index in each chunk, calls the given Loop object's `run()` method.

Like all parallel loops, the `workerFor()` parallel loop creates a team of threads, one thread per core by default, and executes the loop body in parallel within the worker task. A program with a master-worker parallel loop is thus automatically a hybrid parallel program; it consists of multiple worker tasks running on multiple nodes; within each node, it consists of multiple threads running on multiple cores.

Listing 17.1 gives the third version of the cluster parallel bitcoin mining program, class MineCoinClu3. The job main program first parses the command line arguments (lines 18–21): the coin ID and *N*, the number of most significant zero bits in the digest.

The job main program sets up the master portion of the master-worker cluster parallel for loop. Calling the `masterSchedule(leapfrog)` method on line 24 tells the master to partition the loop iterations for the workers using a leapfrog schedule, as I did in the multicore parallel program in Chapter 3. Calling the `masterFor()` method on line 25 does several things: it specifies the inclusive lower and upper bounds of the loop index range; it partitions the loop index range into chunks according to the specified leapfrog schedule; it puts one chunk tuple into tuple space for each chunk; and it adds a start rule to the job to fire up a certain number of worker tasks. Each task will be an instance of class WorkerTask (defined later). Each task's command line arguments are the same, namely the coin ID and *N* (line 26).

In the previous MineCoinClu and MineCoinClu2 programs, the job main program created *multiple* rules with a *single* task each. Here, the master in the job main program creates a *single* rule with *multiple* tasks. A single rule with multiple tasks defines a *task group*. Each task in a task group has a unique rank that serves to distinguish the tasks in the task group from one another. In a task group with K tasks, the task ranks are 0 through $K - 1$.

What's the difference between a task group and a bunch of single tasks? The Tracker treats the two cases differently when it schedules tasks to run on nodes. When there is a bunch of single tasks, the Tracker schedules the tasks independently. Depending on when computational resources become available, some of the tasks might start right away, and others might sit in the Tracker's queue for a while. That's fine if the tasks don't interact with each other. On the other hand, when there is a task group, the Tracker schedules all the tasks in the group as a unit. The Tracker will not start any of the tasks in

```
58          // Task main program.
59          public void main
60             (String[] args)
61             throws Exception
62             {
63             // Parse command line arguments.
64             coinId = Hex.toByteArray (args[0]);
65             N = Integer.parseInt (args[1]);
66
67             // Set up mask for leading zeroes.
68             mask = ~((1L << (64 - N)) - 1L);
69
70             // Early loop exit when any task finds the golden nonce.
71             addTupleListener (new TupleListener<EmptyTuple>
72                (new EmptyTuple())
73                {
74                public void run (EmptyTuple tuple)
75                   {
76                   stop = true;
77                   }
78                });
79
80             // Try all nonces until the digest has N leading zero bits.
81             workerFor() .schedule (leapfrog) .exec (new LongLoop()
82                {
83                byte[] coinIdPlusNonce;
84                SHA256 sha256;
85                byte[] digest;
86
87                public void start()
88                   {
89                   coinIdPlusNonce = new byte [coinId.length + 8];
90                   System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
91                      coinId.length);
92                   sha256 = new SHA256();
93                   digest = new byte [sha256.digestSize()];
94                   }
95
96                public void run (long nonce) throws Exception
97                   {
98                   if (stop) stop();
99                   Packing.unpackLongBigEndian (nonce, coinIdPlusNonce,
100                     coinId.length);
101                  sha256.hash (coinIdPlusNonce);
102                  sha256.digest (digest);
103                  sha256.hash (digest);
104                  sha256.digest (digest);
105                  if ((Packing.packLongBigEndian (digest, 0) & mask)
106                        == 0L)
107                     {
108                     putTuple (new EmptyTuple());
109                     System.out.printf ("Coin ID = %s%n",
110                        Hex.toString (coinId));
111                     System.out.printf ("Nonce   = %s%n",
112                        Hex.toString (nonce));
113                     System.out.printf ("Digest  = %s%n",
114                        Hex.toString (digest));
```

Listing 17.1. MineCoinClu3.java (part 2)

the group until there are enough resources to run *all* the tasks in the group. This is critical if the tasks have to interact with each other. In the MineCoinClu3 program, the tasks do interact with each other—namely, one task informs the other tasks when it finds the golden nonce, so the other tasks can stop—and so the tasks have to be specified as a task group.

Next comes the code for the worker task (line 45). Let's think about how all the tasks discover that one of the tasks found the golden nonce. This involves inter-task communication, which has to go through tuple space. As mentioned previously, when a task finds the golden nonce, the task can put a special "stop tuple" into tuple space, and the other tasks can detect this by reading the stop tuple. Each task *reads* the stop tuple, rather than *taking* the stop tuple, so that the stop tuple remains in tuple space for the other tasks to read.

However, while waiting for the stop tuple to show up, I want the task to be computing a series of nonces. This means the task cannot block trying to read the stop tuple. This is exactly the situation for which tuple listeners are designed. As its first act, the task can set up a tuple listener, specifying that it wants to read a stop tuple. The task can then proceed to search for the golden nonce. Later, if and when a stop tuple shows up, the tuple listener can cause the golden nonce search to stop.

What shall the stop tuple subclass look like? The stop tuple carries no information; its mere presence in tuple space is enough to signal that the golden nonce was found. So the stop tuple subclass will have no fields. The Parallel Java 2 Library provides just such a tuple subclass, namely class edu.rit.pj2.tuple.EmptyTuple. I will use that for my stop tuple.

Returning to the code, the task adds a tuple listener for the stop tuple (lines 71–78). The tuple listener constructor's argument specifies the template to match, namely an EmptyTuple (line 72). Because a tuple operation was not specified, the default operation is "read." The tuple listener's `run()` method (line 74), specified in an anonymous inner class, is called when an EmptyTuple is read from tuple space. The `run()` method sets the shared global `stop` flag to true. Further down in the code, the golden nonce search polls this flag and exits its loop when the flag becomes true, that is, when an EmptyTuple appears.

Note that the shared global `stop` flag is declared with the `volatile` keyword (line 56). Refer back to Chapter 13 to refresh your memory about why this flag needs to be declared `volatile`.

Next comes the worker portion of the master-worker cluster parallel for loop. Calling the `workerFor()` method on line 81 creates a *worker parallel for loop* object. This is similar to the parallel for loop objects we studied earlier, except the loop index range is not specified. Instead, the worker parallel for loop gets chunks of loop indexes by taking chunk tuples out of tuple space. The worker parallel for loop uses a leapfrog schedule (line 81) to fur-

```
115                     stop();
116                     }
117                 }
118             });
119         }
120     }
121 }
```

Listing 17.1. MineCoinClu3.java (part 3)

ther subdivide each chunk among the parallel team threads. Each team thread executes its own copy of the loop body, an instance of an anonymous inner subclass of class LongLoop (line 82). Each team thread calls its own loop body's `start()` method (lines 87–94) once before commencing the loop iterations, and it calls the `run()` method (lines 96–117) to perform each loop iteration, passing in the loop index (the nonce to be tested). The task stays in the worker parallel for loop until the nonce hits the largest possible value (which it almost surely never will) or until the shared global `stop` flag is set to true by the tuple listener (line 98).

If the task finds the golden nonce (lines 105–116), the task puts a stop tuple into tuple space to inform the other tasks (line 108). The stop tuple is an instance of class EmptyTuple. The task also does an early loop exit after printing the results (line 115). Whenever one task finds the golden nonce, all the tasks exit their loops, the tasks terminate, and the job terminates.

To sum up, the MineCoinClu3 program is a hybrid parallel program. The master partitions the loop index range among the worker tasks, using a leapfrog schedule. Each worker task's worker parallel for loop further subdivides each chunk of the loop index range among the threads in the task, again using a leapfrog schedule. All this happens automatically under the hood in the `masterFor()` and `workerFor()` methods.

I ran the MineCoinSeq program on one node of the `tardis` cluster, and I ran the MineCoinClu3 program with one to ten nodes (worker tasks), using strong scaling, for coin ID = `fedcba9876543210`, and for $N = 28$ and 29 most significant zero bits. Here are the commands I used:

```
$ java pj2 debug=makespan \
  edu.rit.pj2.example.MineCoinSeq fedcba9876543210 28
Coin ID = fedcba9876543210
Nonce   = 0000000006ee7a3e
Digest  = 000000084be04f3b20d2aa095debebf6a84241e4048b5cc90ee97
08b0fc74086
Job 3 makespan 233904 msec
$ java pj2 debug=makespan workers=2 \
  edu.rit.pj2.example.MineCoinClu3 fedcba9876543210 28
Coin ID = fedcba9876543210
Nonce   = 0000000006ee7a3e
```

```
Digest  = 000000084be04f3b20d2aa095debebf6a84241e4048b5cc90ee97
08b0fc74086
Job 10 makespan 29813 msec
```

For the cluster parallel program, the `workers` parameter specifies the number of worker tasks *K*. Omitting the `threads` parameter causes each worker task to use one thread for each core of the node on which it is running, so the cluster parallel program ran with 4, 8, 12, . . . 40 cores on `tardis`. Figure 17.3 plots the running times and efficiencies I observed. The running time model is

$$T = (5.67\times10^{-3} + 5.82\times10^{-4}\,N)K + (0.564 + 1.98\times10^{-6}\,N)\div K \qquad (17.1)$$

The program exhibits excellent strong scaling efficiencies all the way out to 40 cores.

Under the Hood

In a master-worker cluster parallel for loop, the master puts a number of chunk tuples (class edu.rit.pj2.Chunk or class edu.rit.pj2.LongChunk) into tuple space. Each chunk has the following fields:

- `rank` — Rank of the worker task that will execute the chunk, or `ANY` if any worker task can execute the chunk.
- `lb` — Loop index lower bound.
- `ub` — Loop index upper bound.
- `stride` — Loop index stride (the amount to increase the loop index on each iteration).

The Chunk and LongChunk classes' `matchContent()` methods are defined so that a template chunk will match a target chunk if the template's rank is `ANY`, if the target's rank is `ANY`, or if the template's rank equals the target's rank. The other fields in the chunk don't matter for purposes of matching.

The MineCoinClu3 program's loop index goes from 0 to 7FFFFFFF-FFFFFFFF hexadecimal, partitioned using a leapfrog schedule. Suppose there are two worker tasks; then the master puts these two chunk tuples into tuple space:

LongChunk
`rank` = 0
`lb` = 0
`ub` = 7FFFFFFFFFFFFFFF
`stride` = 2

LongChunk
`rank` = 1
`lb` = 1
`ub` = 7FFFFFFFFFFFFFFF
`stride` = 2

Worker task 0's worker parallel for loop takes a LongChunk tuple out of tuple space, using a template with `rank` set to 0. This template matches the first of the two chunk tuples. This chunk tells worker task 0 to do loop in-

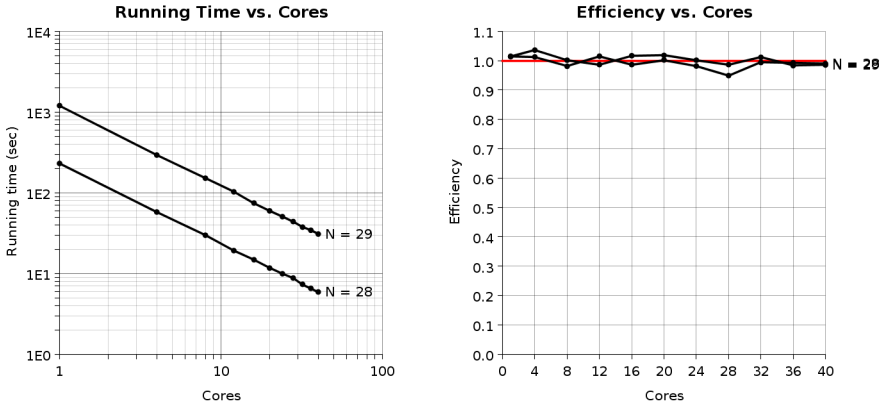


Figure 17.3. MineCoinClu3 strong scaling performance metrics

dexes 0, 2, 4, . . . 7FFFFFFFFFFFFFF. Now suppose the worker parallel for loop has four threads. This chunk is further subdivided among the threads using a leapfrog schedule. Thread 0 does loop indexes 0, 8, 16, . . . ; thread 1 does loop indexes 2, 10, 18, . . . ; thread 2 does loop indexes 4, 12, 20, . . . ; thread 3 does loop indexes 6, 14, 22, ...

Worker task 1's worker parallel for loop takes a LongChunk tuple out of tuple space, using a template with `rank` set to 1. This template matches the second of the two chunk tuples. This chunk tells worker task 1 to do loop indexes 1, 3, 5, . . . 7FFFFFFFFFFFFFF. Worker task 1's worker parallel for loop's thread 0 does loop indexes 1, 9, 17, . . . ; thread 1 does loop indexes 3, 11, 19, . . . ; thread 2 does loop indexes 5, 13, 21, . . . ; thread 3 does loop indexes 7, 15, 23, ... In this way, all the loop indexes are covered in a leapfrog fashion among all the threads in all the worker tasks.

Let's consider how a task, running in a backend process as part of a job, communicates with tuple space. When a task puts a tuple, the task sends a put message to the job's frontend process over the cluster's backend network. The message consists of the tuple's contents. The code for sending the message is executed by the task thread that calls the `putTuple()` method.

Receiving a tuple works as follows. When a task takes or reads a tuple, the task sends a take or read request message to the frontend process. The message contains the template for the take or read. The code for sending the message is executed, again, by the task thread that calls the `takeTuple()` or `readTuple()` method. After that, however, the task thread blocks on an internal data structure of pending tuple requests, waiting for the frontend to send a response.

The backend process has a separate internal thread that reads incoming messages from the frontend process. The incoming messages are read and processed one at a time. When the frontend process finds a tuple that matches a previously requested template, the frontend process sends a response message to the backend process; the response message contains the matching tuple. The backend process's internal thread receives this message, puts the matching tuple into the internal data structure of pending tuple requests, and unblocks the task thread. The task thread retrieves the matching tuple and returns it from the `takeTuple()` or `readTuple()` method call.

A tuple listener works differently. When a task thread in the backend process calls the `addTupleListener()` method, the task stores the tuple listener object in an internal list, and the task sends a take or read request message to the frontend process as before. The `addTupleListener()` method then returns, allowing the task thread to proceed. Later, when the frontend process sends a message containing a tuple that matches the tuple listener's template, the internal message receiving thread reads this incoming message and then calls the tuple listener's `run()` method, passing in the matching tuple.

Consequently, the tuple listener's `run()` method *must not do any lengthy processing.* Rather, the tuple listener should just record the matching tuple or set a flag, as the bitcoin mining worker task does, and return right away; then the task thread can process the tuple matched event. Alternatively, the tuple listener could spawn a separate thread to process the event. Because the internal message receiving thread will not receive and process the next message until the `run()` method returns, doing lengthy processing in the `run()` method will interfere with the cluster middleware's operation.

Points to Remember

- When multiple tasks need to interact with each other, specify them as a task group with a single rule.
- To get a work sharing parallel loop across a cluster parallel program, use the master-worker pattern.
- In the job's `main()` method, code the master portion by calling the `masterFor()` method.
- In the worker task's main() method, code the worker portion by calling the `workerFor()` method.
- To get a global "flag" shared among the tasks of a cluster parallel program, use tuple space. Set the flag by putting a tuple, get the flag by reading a tuple, possibly using a tuple listener to read the tuple.
- Do not do any lengthy processing in a tuple listener's `run()` method.

Chapter 18
Cluster Parallel Reduction

Let's look at how to do reduction in a cluster parallel program. Recall that in a single-node multicore parallel program that does reduction, each thread computes a partial result, then the partial results are combined into one final result. Now extend that to a multi-node multicore, or hybrid, parallel program, consisting of multiple worker tasks each with multiple threads (Figure 18.1). Just as there are two levels of parallelism—intra-task parallelism among the threads in each task, and inter-task parallelism among the tasks in the job—there are also two levels of reduction. Each thread in each task computes a partial result (R_{00}, R_{01}, and so on); the partial results in each task are combined into one semifinal result (R_0, R_1, R_2, R_3); the semifinal results are combined into one final result R.

In the single-node parallel program with reduction, the threads each put their results into shared memory, and the results are combined using a reduction tree (Figure 4.4) in which various threads read various other threads' results from shared memory. This still works for the first level of reduction in a hybrid parallel program. But this no longer works for the second level of reduction. Running on separate nodes, the tasks do not share memory, and inter-task communication is needed to transfer one task's semifinal result into another task in order to do the reduction. In a Parallel Java 2 program, inter-task communication goes through tuple space. Therefore, in Figure 18.1 each task records its semifinal result in a tuple and puts the tuple into tuple space.

Like the Bitcoin mining program in Chapter 17, the worker tasks in Figure 18.1 are specified by a start rule in the job's `main()` method, and they start executing when the job commences. (The master portion of the master-worker cluster parallel for loop sets up this start rule.) An additional reduction task is specified by a finish rule; this task runs once all the worker tasks have finished. The reduction task takes the tuples with the semifinal results produced by the worker tasks, combines the semifinal results, and calculates the final result.

Why make this a hybrid parallel program? Why not have just one level of parallelism and one level of reduction? Then each task would have a single thread, and each task would put its semifinal result directly into tuple space, without needing the intra-task reduction. The answer has to do with the fact that any tuple space operation, such as putting a tuple, involves sending a message from one process to another. Each message takes a nonzero amount of time to traverse the cluster network. Because network speeds are relatively much slower than CPU speeds, in a cluster parallel program it's important to minimize both the number of messages and the size of each message. I minimize the number of messages (`putTuple()` operations) by doing the first level of reduction inside each task, in parallel with the other tasks. Then I only have to put as many semifinal result tuples as there are *tasks*, not as many tuples as there are *threads*—in Figure 18.1, four messages instead of 16.

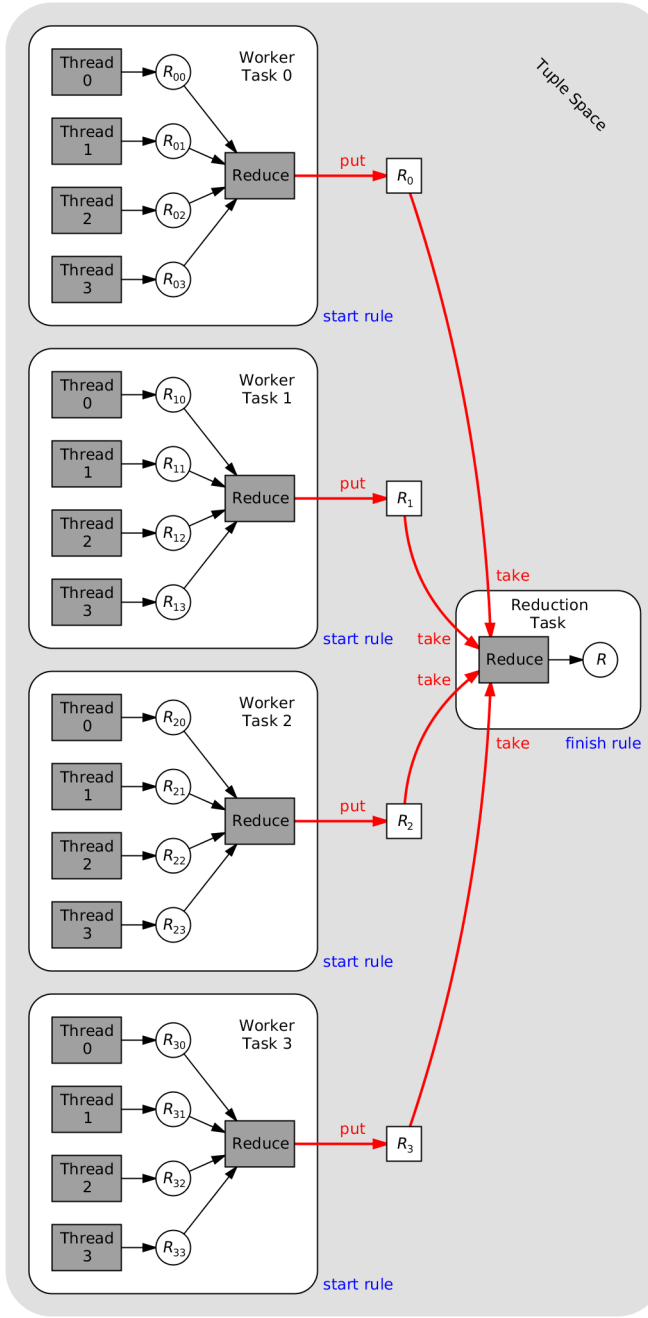


Figure 18.1. Cluster parallel reduction

Listing 18.1 gives the source code for PiClu, a cluster parallel version of the π estimating program from Chapter 4. It follows the pattern of Figure 18.1. The job's `main()` method specifies the master portion of a master-worker cluster parallel for loop (line 20). By default, the master partitions the loop iterations using a fixed schedule, and the worker tasks further subdivide each chunk of iterations using a fixed schedule. Because the loop body's running time is the same for each iteration, the load is inherently balanced, and a fixed schedule is appropriate. The `main()` method also specifies a finish rule that sets up the reduction task (lines 23–24). Calling the `runInJobProcess()` method specifies that the reduction task will run in the job process on the frontend node, rather than in a separate process on a backend node. The reduction task does not do any heavy computation, and there's no need to tie up a backend node to run the reduction task.

Class WorkerTask (lines 39–81) defines the worker tasks' computation. We've seen this code before; it is almost the same as the multicore PiSmp program in Chapter 4. However, this task specifies the worker portion of a master-worker cluster parallel for loop (line 61). Like the PiSmp program, class WorkerTask has a global reduction variable of type LongVbl.Sum, and the threads have thread-local reduction variables that are automatically reduced into the global variable.

The threads also have thread-local pseudorandom number generators (PRNGs). This time, however, the PRNG in each thread *in each task* must be initialized with a different seed, otherwise the tasks will generate the same sequence of random numbers. I do this on line 67 with the expression

```
seed + 1000*taskRank() + rank()
```

`taskRank()` returns the current task's rank within the task group, and `rank()` returns the current thread's rank within the task. If the program is run with four worker tasks each with four threads, task 0 initializes its threads' PRNGs with `seed` + 0, `seed` + 1, `seed` + 2, and `seed` + 3; task 1 initializes its threads' PRNGs with `seed` + 1000, `seed` + 1001, `seed` + 1002, and `seed` + 1003; task 2 initializes its threads' PRNGs with `seed` + 2000, `seed` + 2001, `seed` + 2002, and `seed` + 2003; and task 3 initializes its threads' PRNGs with `seed` + 3000, `seed` + 3001, `seed` + 3002, and `seed` + 3003. Every thread in every task gets a different seed.

Finally, instead of printing the answer, the worker task puts the global reduction variable containing the task's semifinal result into tuple space (line 79). Class LongVbl, as well as all the reduction variable classes in package edu.rit.pj2, are in fact tuple subclasses—they all extend class Tuple—so putting them into tuple space works. (Fancy that!)

Class ReduceTask (lines 85–102) defines the reduction task, which runs when all the worker tasks have finished. It has its own reduction variable, `count`; it calls the `tryToTakeTuple()` method repeatedly to obtain the

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Job;
3  import edu.rit.pj2.LongLoop;
4  import edu.rit.pj2.Task;
5  import edu.rit.pj2.vbl.LongVbl;
6  import edu.rit.util.Random;
7  public class PiClu
8     extends Job
9     {
10    // Job main program.
11    public void main
12       (String[] args)
13       {
14       // Parse command line arguments.
15       if (args.length != 2) usage();
16       long seed = Long.parseLong (args[0]);
17       long N = Long.parseLong (args[1]);
18
19       // Set up a task group of K worker tasks.
20       masterFor (0, N – 1, WorkerTask.class) .args (""+seed, ""+N);
21
22       // Set up reduction task.
23       Rule() .atFinish() .task (ReduceTask.class) .args (""+N)
24          .runInJobProcess();
25       }
26
27    // Print a usage message and exit.
28    private static void usage()
29       {
30       System.err.println ("Usage: java pj2 " +
31          "edu.rit.pj2.example.PiClu <seed> <N>");
32       System.err.println ("<seed> = Random seed");
33       System.err.println ("<N> = Number of random points");
34       throw new IllegalArgumentException();
35       }
36
37    // Class PiClu.WorkerTask performs part of the computation for
38    // the PiClu program.
39    private static class WorkerTask
40       extends Task
41       {
42       // Command line arguments.
43       long seed;
44       long N;
45
46       // Number of points within the unit circle.
47       LongVbl count;
48
49       // Main program.
50       public void main
51          (String[] args)
52          throws Exception
53          {
54          // Parse command line arguments.
55          seed = Long.parseLong (args[0]);
56          N = Long.parseLong (args[1]);
57
```

Listing 18.1. PiClu.java (part 1)

worker tasks' semifinal result tuples; and it accumulates those into the `count` variable using the `reduce()` method, which does a sum-reduce. When the `tryToTakeTuple()` method returns null, the reduction task knows that it has accumulated all the semifinal result tuples, so it prints the final answer.

Let's consider how the hybrid parallel PiClu program partitions the parallel loop iterations. Suppose I run the program with four worker tasks on a four-node cluster, each node having four cores, with N = 1,000,000 iterations. The job main program does not specify the master schedule, so the master partitions the loop index range 0 to 999,999 among the workers using the default fixed schedule, with an equal-sized chunk of indexes for each worker:

- Worker 0 — indexes 0 to 249,999
- Worker 1 — indexes 250,000 to 499,999
- Worker 2 — indexes 500,000 to 749,999
- Worker 3 — indexes 750,000 to 999,999

The worker task main program does not specify the parallel for loop schedule, so each worker partitions its own index chunk among the four parallel team threads using the default fixed schedule, with an equal-sized chunk of indexes for each team thread:

- Worker 0 thread 0 — indexes 0 to 62,499
- Worker 0 thread 1 — indexes 62,500 to 124,999
- Worker 0 thread 2 — indexes 125,000 to 187,499
- Worker 0 thread 3 — indexes 187,500 to 249,999
- Worker 1 thread 0 — indexes 250,000 to 312,499
- Worker 1 thread 1 — indexes 312,500 to 374,999
- Worker 1 thread 2 — indexes 375,000 to 437,499
- Worker 1 thread 3 — indexes 437,500 to 499,999
- Worker 2 thread 0 — indexes 500,000 to 562,499
- Worker 2 thread 1 — indexes 562,500 to 624,999
- Worker 2 thread 2 — indexes 625,000 to 687,499
- Worker 2 thread 3 — indexes 687,500 to 749,999
- Worker 3 thread 0 — indexes 750,000 to 812,499
- Worker 3 thread 1 — indexes 812,500 to 874,999
- Worker 3 thread 2 — indexes 875,000 to 937,499
- Worker 3 thread 3 — indexes 937,500 to 999,999

In this way, each thread in each worker executes the same number of loop iterations. Because the loop body's running time is the same in every iteration, the default fixed schedule results in a balanced load.

I ran the π estimating program on the `tardis` cluster to study the program's weak scaling performance. For the sequential version, I ran the PiSeq

```
58          // Generate n random points in the unit square, count how
59          // many are in the unit circle.
60          count = new LongVbl.Sum (0);
61          workerFor() .exec (new LongLoop()
62             {
63             Random prng;
64             LongVbl thrCount;
65             public void start()
66                {
67                prng = new Random (seed + 1000*taskRank() + rank());
68                thrCount = threadLocal (count);
69                }
70             public void run (long i)
71                {
72                double x = prng.nextDouble();
73                double y = prng.nextDouble();
74                if (x*x + y*y <= 1.0) ++ thrCount.item;
75                }
76             });
77
78          // Report results.
79          putTuple (count);
80          }
81       }
82
83    // Class PiClu.ReduceTask combines the worker tasks' results and
84    // prints the overall result for the PiClu program.
85    private static class ReduceTask
86       extends Task
87       {
88       // Reduce task main program.
89       public void main
90          (String[] args)
91          throws Exception
92          {
93          long N = Long.parseLong (args[0]);
94          LongVbl count = new LongVbl.Sum (0L);
95          LongVbl template = new LongVbl();
96          LongVbl taskCount;
97          while ((taskCount = tryToTakeTuple (template)) != null)
98             count.reduce (taskCount);
99          System.out.printf ("pi = 4*%d/%d = %.9f%n",
100            count.item, N, 4.0*count.item/N);
101         }
102      }
103   }
```

Listing 18.1. PiClu.java (part 2)

program from Chapter 4 on one node (one core). For the parallel version, I ran the PiClu program with one to ten worker tasks (4 to 40 cores). On one core, I ran it with 2, 4, 6, and 8 billion darts. On *K* cores, I ran it with *K* times as many darts as one core. Here are some of the commands I used:

```
$ java pj2 edu.rit.pj2.example.PiSeq 1234 2000000000
$ java pj2 workers=1 edu.rit.pj2.example.PiClu 1234 8000000000
$ java pj2 workers=2 edu.rit.pj2.example.PiClu 1234 16000000000
$ java pj2 workers=3 edu.rit.pj2.example.PiClu 1234 24000000000
$ java pj2 workers=4 edu.rit.pj2.example.PiClu 1234 32000000000
```

For each PiClu program run, I specified the number of worker tasks with the `workers=` option on the `pj2` command line.

Figure 18.2 plots the running times and efficiencies I observed. The running time model is $T = 8.86\times10^{-12}\ N + 2.02\times10^{-8}\ N\div K$, which yields an almost nonexistent sequential fraction of 0.000441.

Under the Hood

By calling the `runInJobProcess()` method, I specified the PiClu program's reduction task to run in the job's process on the frontend node. I did this to avoid tying up a backend node to run the reduction task. This is not the only reason one might want to run a task in the job's process. Here are some other reasons.

Suppose a user runs a cluster parallel program needs to read an input file. The job process runs in the user's account, and so has access to the user's files. But the backend processes typically do *not* run in the user's account; rather, they typically run in a separate Parallel Java account. This is for security; we don't want different users running jobs on the cluster at the same time to be able to access or interfere with each other's tasks. But this means that a task running in a backend process does not necessarily have access to the user's account. In particular, a backend process might not be able to read an input file in the user's account. One way to deal with this situation is for the user to make the input file readable by everyone. But again, for security, the user might not want to do this. An alternative is for the job to run the input file reading task *in the job process.* That way, the task's process *does* have access to the user's files. The input file reading task can then distribute the file's contents to other tasks via tuple space.

Similarly, if the program needs to write an output file rather than printing its results on the console, the file writing task can be specified to run in the job process. Other tasks send data to the file writing task via tuple space.

Another use case is if the program needs to display a graphical user interface (GUI) while running—to show the progress of the computation, for example. The task that drives the GUI must run in the job's process to be able to display windows on the user's screen. Other tasks can send progress up-

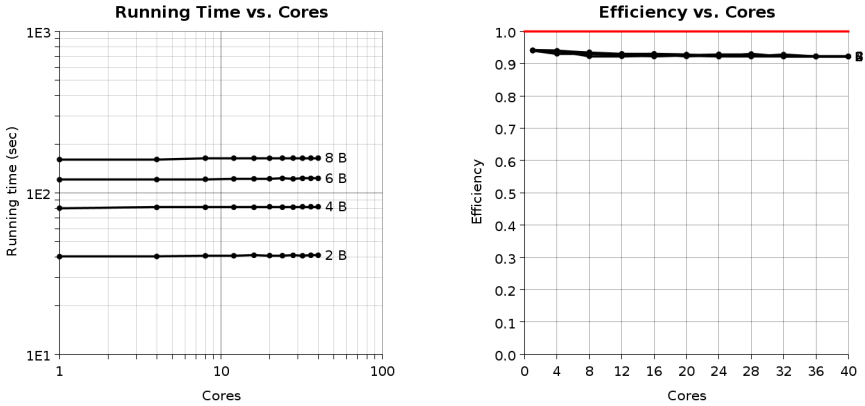


Figure 18.2. PiClu weak scaling performance metrics

dates to the GUI task via tuple space. Such a GUI task could even accept input from the user: to change parameters while the program is running, to stop the program gracefully (as opposed to killing it with an interrupt), and so on.

Points to Remember

- In a cluster parallel program with reduction, use the hybrid model with multiple multicore worker tasks.
- Do the first level of reduction in shared memory within each task.
- Do the second level of reduction in a separate finish task.
- Use tuple space to send semifinal results from the worker tasks to the finish task.
- To get good performance, make sure the time spent on computation vastly outweighs the time spent on message passing and reduction.

Chapter 19
Cluster Load Balancing

Consider how the last few cluster parallel programs partitioned their computations among the worker tasks and the threads within each task. The Bitcoin mining program in Chapter 17 partitioned the computations (nonces) among the tasks and threads using a leapfrog schedule. The π estimating program in Chapter 18 partitioned the computations (darts) among the tasks and threads using a fixed schedule. Fixed or leapfrog partitioning is appropriate when each loop iteration takes the same amount of time, so the load is inherently balanced.

Now let's consider a problem in which each loop iteration does *not* take the same amount of time, so the load is unbalanced. In a single-node multicore parallel program, we saw that we could balance the load, and reduce the parallel program's running time, by specifying a parallel for loop with a dynamic, proportional, or guided schedule. We can do the same in a cluster parallel program; the master portion of a master-worker cluster parallel for loop lets you specify a load balancing schedule when partitioning the loop iterations into chunks.

A hybrid parallel program, which has two levels of parallelism, has to do two levels of chunking: coarser-grained chunking at the task level, finer-grained chunking at the thread level within each task. An approach that usually works well is to partition the overall computation using a proportional schedule. This yields a number of chunks proportional to K, the number of worker tasks, with each chunk having the same number of loop indexes:

$$\text{Number of chunks} = K \times \mathit{chunkFactor}$$

There's a tradeoff, though. If the chunk factor is too large, there will be too many chunks, which will increase the message passing overhead—taking each chunk out of tuple space involves sending messages back and forth between the job process and the worker processes. But if the chunk factor is too small, there will be too few chunks to balance the load effectively. I've found that a chunk factor of 10 to 100 usually works well.

The master-worker pattern achieves coarse-grained load balancing at the task level. But any particular chunk of computation might itself exhibit an unbalanced load. So the hybrid parallel program also has to do fine-grained load balancing at the thread level within each task. This is achieved automatically by specifying the worker parallel for loop's schedule—for example, by specifying the `schedule` and `chunk` arguments on the `pj2` command line, or by hard-coding these into the worker parallel for loop.

Let's put these considerations into practice. Listing 19.1 gives the source code for TotientClu, a hybrid parallel version of the Euler totient program from Chapter 6. Recall that the Euler totient of n, $\Phi(n)$, is the number of numbers between 1 and $n - 1$ that are relatively prime to n. The program partitions the numbers between 1 and $n - 1$ among the worker tasks, using the master-worker pattern for load balancing. Each task further partitions each

```
1  package edu.rit.pj2.example;
2  import edu.rit.pj2.Job;
3  import edu.rit.pj2.LongLoop;
4  import edu.rit.pj2.Task;
5  import edu.rit.pj2.vbl.LongVbl;
6  import edu.rit.util.LongList;
7  public class TotientClu
8     extends Job
9     {
10    // Job main program.
11    public void main
12       (String[] args)
13       {
14       // Parse command line arguments.
15       if (args.length != 1) usage();
16       long n = Long.parseLong (args[0]);
17
18       // Set up a task group of K worker tasks.
19       masterFor (2, n - 1, WorkerTask.class) .args (""+n);
20
21       // Set up reduction task.
22       rule() .atFinish() .task (ReduceTask.class)
23          .runInJobProcess();
24       }
25
26    // Print a usage message and exit.
27    private static void usage()
28       {
29       System.err.println ("Usage: java pj2 [workers=<K>] " +
30          "edu.rit.pj2.example.TotientClu <n>");
31       System.err.println ("<K> = Number of worker tasks " +
32          "(default: 1)");
33       System.err.println ("<n> = Number whose totient to compute");
34       throw new IllegalArgumentException();
35       }
36
37    // Class TotientClu.WorkerTask provides a task that computes
38    // chunks of iterations in the totient computation.
39    private static class WorkerTask
40       extends Task
41       {
42       long n;
43       LongVbl phi;
44       LongList nFactors = new LongList();
45
46       // Worker task main program.
47       public void main
48          (String[] args)
49          throws Exception
50          {
51          n = Long.parseLong (args[0]);
52
53          phi = new LongVbl.Sum();
54          factorize (n, nFactors);
55
56          workerFor() .exec (new LongLoop()
57             {
58             LongList iFactors;
```

Listing 19.1. TotientClu.java (part 1)

chunk of work among its parallel team threads, using a parallel loop with a load balancing schedule. The threads' results are combined within each task using multicore parallel reduction, and the workers' results are combined using cluster parallel reduction.

The job's command line argument is *n*, the number whose totient is to be computed. Several `pj2` options must also be specified. The `workers` option gives the number of worker tasks. The `masterSchedule` and `masterChunk` options specify how to partition the computation at the master level. The `schedule` and `chunk` options specify how to partition the computation at the worker level. These options must be specified appropriately to achieve a balanced load.

I ran the totient program on the `tardis` cluster to study the program's strong scaling performance. I computed the totients of two numbers, n = 20,000,003 and n = 40,000,003 (both prime). For partitioning at the master level, I used a proportional schedule with a chunk factor of 100. For partitioning at the worker level, I used a guided schedule. To measure the sequential version, I ran the TotientSeq program from Chapter 6 on one node using commands like this:

```
$ java pj2 debug=makespan edu.rit.pj2.example.TotientSeq \
  20000003
```

To measure the parallel version on one core, I ran the TotientClu program with one worker task and one thread using commands like this:

```
$ java pj2 debug=makespan workers=1 \
  masterSchedule=proportional masterChunk=100 threads=1 \
  schedule=guided edu.rit.pj2.example.TotientClu 20000003
```

To measure the parallel version on multiple cores, I ran the TotientClu program with one to ten worker tasks and with all cores on each node (4 to 40 cores) using commands like this:

```
$ java pj2 debug=makespan workers=1 \
  masterSchedule=proportional masterChunk=100 \
  schedule=guided edu.rit.pj2.example.TotientClu 20000003
```

Figure 19.1 plots the running times and efficiencies I observed.

To derive a running time model, I need to know the problem size. There's no simple relationship between n, the number whose totient is being computed, and N, the amount of computation needed to calculate $\Phi(n)$. So I just used the program's running time in milliseconds on one core as the problem size. This yielded $N = 1.99\times10^5$ and 5.35×10^5 for n = 20,000,003 and 40,000,003, respectively. The running time model is $T = 0.205 + 7.33\times10^{-7} N + 0.001 N\div K$.

Figure 19.1 shows that the efficiencies degrade as the number of cores increases. This is due to the $(a + bN)$ term in the running time model. The sequential fraction is determined to be $F = 0.00176$ for n = 20,000,003 and F =

```
59             LongVbl thrPhi;
60             public void start()
61                {
62                iFactors = new LongList();
63                thrPhi = threadLocal (phi);
64                }
65             public void run (long i)
66                {
67                if (relativelyPrime (factorize (i, iFactors),
68                   nFactors))
69                      ++ thrPhi.item;
70                }
71             });
72
73          // Report result.
74          putTuple (phi);
75          }
76
77       // Store a list of the prime factors of x in ascending order
78       // in the given list.
79       private static LongList factorize
80          (long x,
81           LongList list)
82          {
83          list.clear();
84          long p = 2;
85          long psqr = p*p;
86          while (psqr <= x)
87             {
88             if (x % p == 0)
89                {
90                list.addLast (p);
91                x /= p;
92                }
93             else
94                {
95                p = p == 2 ? 3 : p + 2;
96                psqr = p*p;
97                }
98             }
99          if (x != 1)
100            list.addLast (x);
101         return list;
102         }
103
104      // Determine whether two numbers are relatively prime, given
105      // their lists of factors.
106      private static boolean relativelyPrime
107         (LongList xFactors,
108          LongList yFactors)
109         {
110         int xSize = xFactors.size();
111         int ySize = yFactors.size();
112         int ix = 0;
113         int iy = 0;
114         long x, y;
115         while (ix < xSize && iy < ySize)
116            {
```

Listing 19.1. TotientClu.java (part 2)

0.00112 for n = 40,000,003. The sequential fraction is smaller for the larger problem size. Consequently, the efficiencies for the larger problem size do not degrade as quickly as the efficiencies for the smaller problem size, as is apparent in the plot.

Why is the sequential fraction smaller for the larger problem size? The sequential portion of the cluster totient program consists mainly of sending messages to transfer chunk tuples from the master to the workers, sending messages to transfer the result tuples from the workers to the reduce task, performing the reduction, and printing the results. The parallelizable portion consists of examining all the numbers from 1 to $n - 1$ to calculate the totient. For a given number of workers, the sequential portion's running time is the same no matter what n is—because with a proportional schedule, the number of chunk tuples depends only on how many workers there are, and likewise for the number of result tuples. However, the parallelizable portion's running time increases as n increases. Therefore, the program's sequential fraction decreases as the problem size increases, and the program's efficiency goes up.

These observations illustrate a characteristic of cluster parallel programs that involve message passing: The performance improves as the ratio of computation to communication goes up. Communication includes messages to carry out the master-worker load balancing as well as messages to report the worker tasks' final results. Problems well-suited for a cluster parallel computer are those that require a lot of computation with little communication.

As a reminder, the approach the TotientClu program uses to calculate the Euler totient is rather inefficient. I'm using this as an example of why and how to do load balancing in a cluster parallel program, not as an example of how to calculate the totient efficiently. I chose to use an inefficient, computation-intensive, unbalanced-load algorithm to make my points about load balancing and computation-to-communication ratios.

Points to Remember

- In a cluster parallel program that needs load balancing, use the hybrid model with multiple multicore worker tasks.
- Do coarse-grained load balancing at the task level using the master-worker pattern.
- Do fine-grained load balancing at the thread level within each task by specifying the appropriate schedule for the parallel for loop.
- The job's `main()` method partitions the computation and puts chunk tuples containing chunks of work into tuple space.
- The worker tasks repeatedly take chunk tuples out of tuple space and perfom the indicated chunks of work.

```
117             x = xFactors.get (ix);
118             y = yFactors.get (iy);
119             if (x == y) return false;
120             else if (x < y) ++ ix;
121             else ++ iy;
122             }
123          return true;
124          }
125       }
126
127    // Class TotientClu.ReduceTask combines the worker tasks' results
128    // and prints the overall result.
129    private static class ReduceTask
130       extends Task
131       {
132       // Reduce task main program.
133       public void main
134          (String[] args)
135          throws Exception
136          {
137          LongVbl phi = new LongVbl.Sum();
138          LongVbl template = new LongVbl();
139          LongVbl taskPhi;
140          while ((taskPhi = tryToTakeTuple (template)) != null)
141             phi.reduce (taskPhi);
142          System.out.printf ("%d%n", phi.item + 1);
143          }
144       }
145 }
```

Listing 19.1. TotientClu.java (part 3)

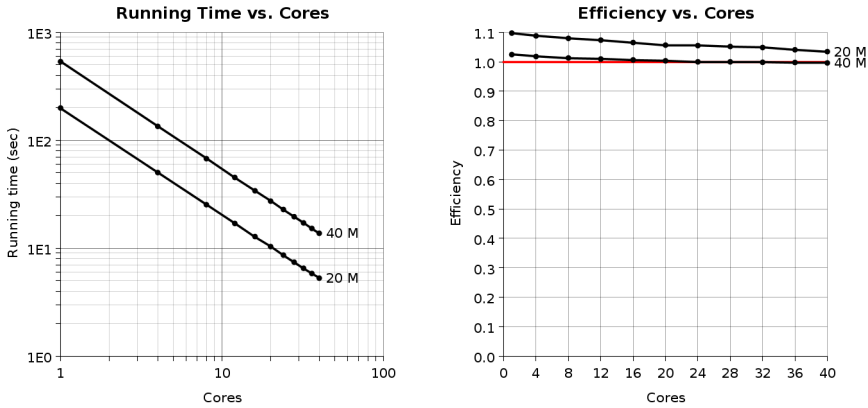
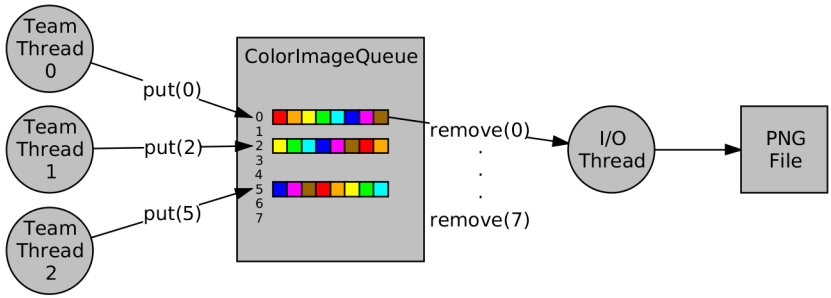


Figure 19.1. TotientClu strong scaling performance metrics

- To get good performance, make sure the time spent on computation vastly outweighs the time spent on message passing—taking chunk tuples, putting result tuples.

Chapter 20
File Output on a Cluster

Recall the single-node multicore parallel program from Chapter 7 that computes an image of the Mandelbrot Set. The program partitioned the image rows among the threads of a parallel thread team. Each thread computed the color of each pixel in a certain row, put the row of pixels into a ColorImageQueue, and went on to the next available row. Simultaneously, another thread took each row of pixels out of the queue and wrote them to a PNG file.



Now let's make this a cluster parallel program. The program will illustrate several features of the Parallel Java 2 Library, namely customized tuple subclasses and file output in a job. Studying the program's strong scaling performance will reveal interesting behavior that we didn't see with single-node multicore parallel programs.

Figure 20.1 shows the cluster parallel Mandelbrot Set program's design. Like the previous multicore version, the program partitions the image rows among multiple parallel team threads. Unlike the previous version, the parallel team threads are located in multiple separate worker tasks. Each worker task runs in a separate backend process on one of the backend nodes in the cluster. I'll use a master-worker parallel for loop to partition the image rows among the tasks and threads.

Also like the previous version, the program has an I/O thread responsible for writing the output PNG file, as well as a ColorImageQueue from which the I/O thread obtains rows of pixels. The I/O thread and the queue reside in an output task, separate from the worker tasks and shared by all of them. I'll run the output task in the job's process on the frontend node, rather than in a backend process on a backend node. That way, the I/O thread runs in the user's account and is able to write the PNG file in the user's directory. (If the output task ran in a backend process, the output task would typically run in a special Parallel Java account rather than the user's account, and the output task would typically not be able to write files in the user's directory.)

This is a *distributed memory* program. The worker task team threads compute pixel rows, which are located in the backend processes' memories. The output task's image queue is located in the frontend process's memory. Thus, it's not possible for a team thread to put a pixel row directly into the

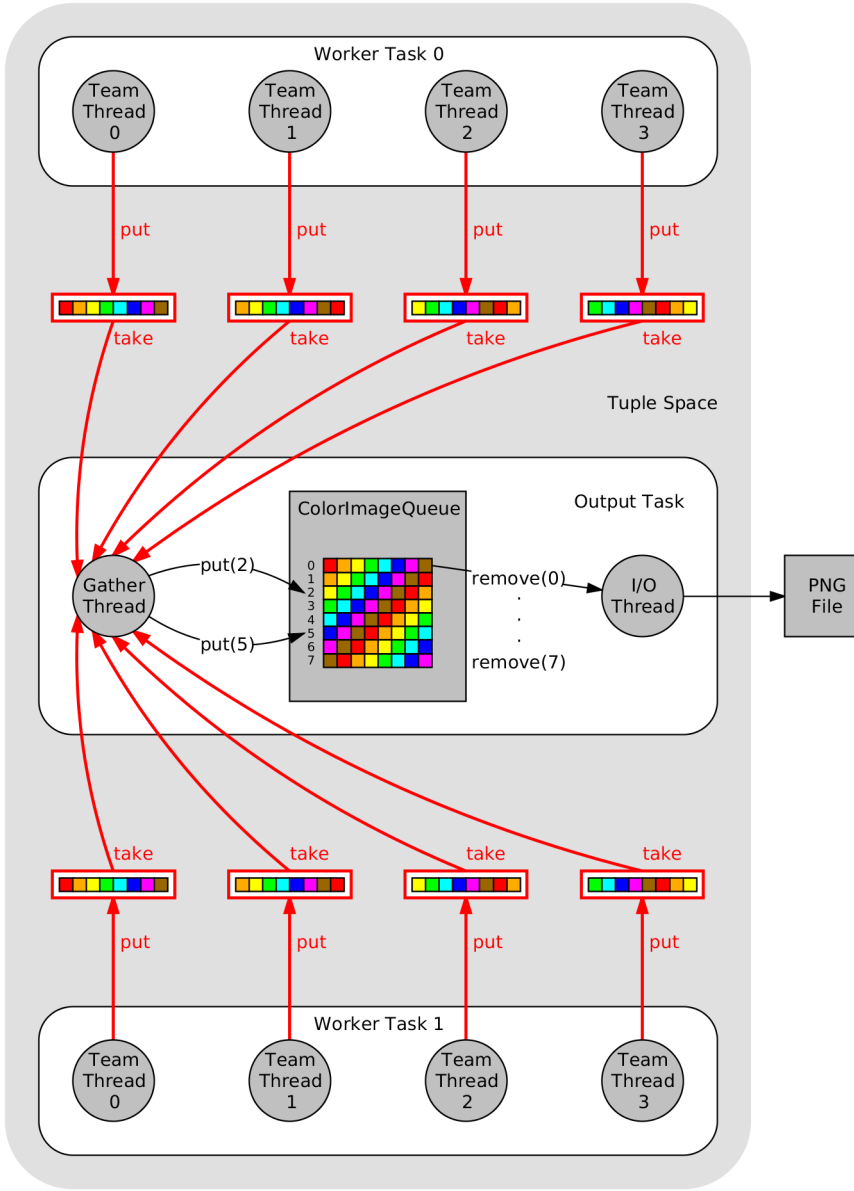


Figure 20.1. Cluster parallel Mandelbrot Set program

image queue, as the multicore parallel program could.

This is where tuple space comes to the rescue. When a team thread has computed a pixel row, the team thread packages the pixel row into an output tuple and puts the tuple into tuple space. The tuple also includes the row index. A second gather thread in the output task repeatedly takes an output tuple, extracts the row index and the pixel row, and puts the pixel row into the image queue at the proper index. The I/O thread then removes pixel rows from the image queue and writes them to the PNG file.

In this way, the computation's results are communicated from the worker tasks through tuple space to the output task. However, going through tuple space imposes a communication overhead in the cluster parallel program, which the multicore parallel program did not have. This communication overhead affects the cluster parallel program's scalability.

Listing 20.1 gives the source code for class edu.rit.pj2.example.MandelbrotClu. Like all cluster parallel programs, it begins with the job main program that defines the tasks. The `masterFor()` method (line 40) sets up a task group with *K* worker tasks, where *K* is specified by the `workers` option on the `pj2` command. The `masterFor()` method also sets up a master-worker parallel for loop that partitions the outer loop over the image rows, from 0 through `height` – 1, among the worker tasks. Because the running time is different in every loop iteration, the parallel for loop needs a load balancing schedule; I specified a proportional schedule with a chunk factor of 10 (lines 38–39). This partitions the outer loop iterations into 10 times as many chunks as there are worker tasks, and each task will repeatedly execute the next available chunk in a dynamic fashion.

The job main program also sets up the output task that will write the PNG file (lines 43–44), with the output task running in the job's process. Both the worker tasks and the output task are specified by start rules and will commence execution at the start of the job.

Next comes the OutputTuple subclass (line 71). It conveys a row of pixel colors, a ColorArray (line 74), along with the row index (line 73), from a worker task to the output task. The tuple subclass also provides the obligatory no-argument constructor (lines 76–78), `writeOut()` method (lines 88–92), and `readIn()` method (lines 94–98).

The WorkerTask class (line 102) is virtually identical to the single-node multicore MandelbrotSmp class from Chapter 7. There are only two differences. First, the worker task provides the worker portion of the master-worker parallel for loop (line 151). When the worker task obtains a chunk of row indexes from the master, the indexes are partitioned among the parallel team threads using a dynamic schedule for load balancing. Each loop iteration (line 160) computes the pixel colors for one row of the image, storing the colors in a per-thread color array (line 153). The second difference is that once all columns in the row have been computed, the worker task packages

```
1  package edu.rit.pj2.example;
2  import edu.rit.image.Color;
3  import edu.rit.image.ColorArray;
4  import edu.rit.image.ColorImageQueue;
5  import edu.rit.image.ColorPngWriter;
6  import edu.rit.io.InStream;
7  import edu.rit.io.OutStream;
8  import edu.rit.pj2.Job;
9  import edu.rit.pj2.Loop;
10 import edu.rit.pj2.Schedule;
11 import edu.rit.pj2.Section;
12 import edu.rit.pj2.Task;
13 import edu.rit.pj2.Tuple;
14 import java.io.BufferedOutputStream;
15 import java.io.File;
16 import java.io.FileOutputStream;
17 import java.io.IOException;
18 public class MandelbrotClu
19    extends Job
20    {
21    // Job main program.
22    public void main
23       (String[] args)
24       {
25       // Parse command line arguments.
26       if (args.length != 8) usage();
27       int width = Integer.parseInt (args[0]);
28       int height = Integer.parseInt (args[1]);
29       double xcenter = Double.parseDouble (args[2]);
30       double ycenter = Double.parseDouble (args[3]);
31       double resolution = Double.parseDouble (args[4]);
32       int maxiter = Integer.parseInt (args[5]);
33       double gamma = Double.parseDouble (args[6]);
34       File filename = new File (args[7]);
35
36       // Set up task group with K worker tasks. Partition rows among
37       // workers.
38       masterSchedule (proportional);
39       masterChunk (10);
40       masterFor (0, height - 1, WorkerTask.class) .args (args);
41
42       // Set up PNG file writing task.
43       rule() .task (OutputTask.class) .args (args)
44          .runInJobProcess();
45       }
46
47    // Print a usage message and exit.
48    private static void usage()
49       {
50       System.err.println ("Usage: java pj2 [workers=<K>] " +
51          "edu.rit.pj2.example.MandelbrotClu <width> <height> " +
52          "<xcenter> <ycenter> <resolution> <maxiter> <gamma> " +
53          "<filename>");
54       System.err.println ("<K> = Number of worker tasks (default " +
55          "1)");
56       System.err.println ("<width> = Image width (pixels)");
57       System.err.println ("<height> = Image height (pixels)");
58       System.err.println ("<xcenter> = X coordinate of center " +
```

Listing 20.1. MandelbrotClu.java (part 1)

the row index and the color array into an output tuple and puts the tuple into tuple space (line 189), whence the output task can take the tuple.

Last comes the OutputTask class (line 196), which runs in the job's process. After setting up the PNG file writer and the color image queue (lines 219–224), the task runs two parallel sections simultaneously in two threads (line 227).

The first section (lines 230–240) repeatedly takes an output tuple out of tuple space and puts the tuple's pixel data into the image queue at the row index indicated in the tuple. The `takeTuple()` method is given a blank output tuple as the template; this matches any output tuple containing any pixel row, no matter which worker task put the tuple. The tuple's row index ensures that the pixel data goes into the proper image row, regardless of the order in which the tuples arrive. The first section takes exactly as many tuples as there are image rows (the `height` argument).

The second section (lines 242–249) merely uses the PNG image writer to write the PNG file.

Each worker task terminates when there are no more chunks of pixel rows to calculate. The output task terminates when the first parallel section has taken and processed all the output tuples and the second parallel section has finished writing the PNG file. At that point the job itself terminates.

I ran the Mandelbrot Set program on the `tardis` cluster to study the program's strong scaling performance. I computed images of size 3200×3200, 4525×4525, 6400×6400, 9051×9051, and 12800×12800 pixels. For partitioning at the master level, the program is hard-coded to use a proportional schedule with a chunk factor of 10. For partitioning at the worker level, the program is hard-coded to use a dynamic schedule. To measure the sequential version, I ran the MandelbrotSeq program from Chapter 7 on one node using commands like this:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MandelbrotSeq \
  3200 3200 -0.75 0 1200 1000 0.4 ms3200.png
```

To measure the parallel version on one core, I ran the MandelbrotClu program with one worker task and one thread using commands like this:

```
$ java pj2 debug=makespan workers=1 cores=1 \
  edu.rit.pj2.example.MandelbrotClu 3200 3200 -0.75 0 1200 \
  1000 0.4 ms3200.png
```

To measure the parallel version on multiple cores, I ran the MandelbrotClu program with one to ten worker tasks and with all cores on each node (4 to 40 cores) using commands like this:

```
$ java pj2 debug=makespan workers=2 \
  edu.rit.pj2.example.MandelbrotClu 3200 3200 -0.75 0 1200 \
  1000 0.4 ms3200.png
```

```
59          "point");
60       System.err.println ("<ycenter> = Y coordinate of center " +
61          "point");
62       System.err.println ("<resolution> = Pixels per unit");
63       System.err.println ("<maxiter> = Maximum number of " +
64          "iterations");
65       System.err.println ("<gamma> = Used to calculate pixel hues");
66       System.err.println ("<filename> = PNG image file name");
67       throw new IllegalArgumentException();
68       }
69
70    // Tuple for sending results from worker tasks to output task.
71    private static class OutputTuple
72       extends Tuple
73       {
74       public int row;               // Row index
75       public ColorArray pixelData;  // Row's pixel data
76
77       public OutputTuple()
78          {
79          }
80
81       public OutputTuple
82          (int row,
83           ColorArray pixelData)
84          {
85          this.row = row;
86          this.pixelData = pixelData;
87          }
88
89       public void writeOut (OutStream out) throws IOException
90          {
91          out.writeUnsignedInt (row);
92          out.writeObject (pixelData);
93          }
94
95       public void readIn (InStream in) throws IOException
96          {
97          row = in.readUnsignedInt();
98          pixelData = (ColorArray) in.readObject();
99          }
100      }
101
102   // Worker task class.
103   private static class WorkerTask
104      extends Task
105      {
106      // Command line arguments.
107      int width;
108      int height;
109      double xcenter;
110      double ycenter;
111      double resolution;
112      int maxiter;
113      double gamma;
114
115      // Initial pixel offsets from center.
116      int xoffset;
```

Listing 20.1. MandelbrotClu.java (part 2)

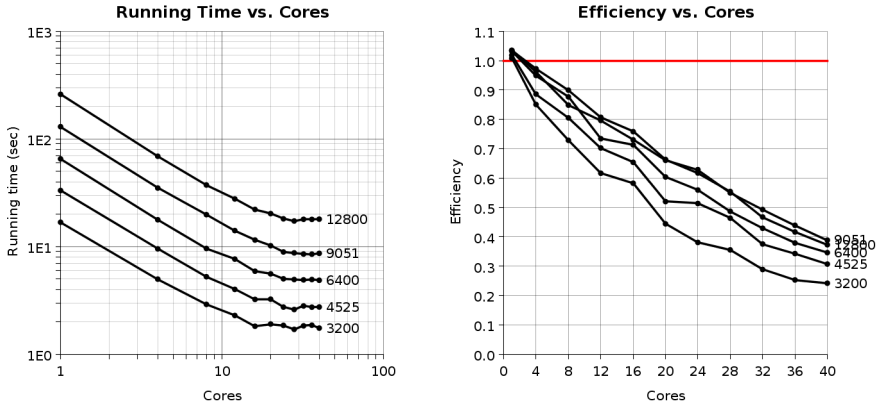


Figure 20.2. MandelbrotClu strong scaling performance metrics

Figure 20.2 plots the running times and efficiencies I observed. The running time plots' behavior is peculiar. The running times decrease as the number of cores increases, more or less as expected with strong scaling, but only up to a certain point. At around 16 or 20 cores, the running time plots flatten out, and there is no further reduction as more cores are added. Also, the efficiency plots show that there's a steady decrease in efficiency as more cores are added, much more of a drop than we've seen before. What's going on?

Here is the program's running time model fitted to the data:

$$T = (0.614 + 9.67\times10^{-11}\,N) + (4.74\times10^{-12}\,N)\cdot K + (0.0357 + 7.15\times10^{-9}\,N) \div K\,. \tag{20.1}$$

Plugging a certain problem size N into Equation 20.1 yields a running time formula with three terms: a constant term, a term directly proportional to K, and a term inversely proportional to K. For example, the 3200×3200-pixel image has a problem size (number of inner loop iterations) N of 2.23×10^9. For that problem size, the formula becomes

$$T = 0.830 + 0.0106\cdot K + 16.0 \div K\,. \tag{20.2}$$

Figure 20.3 plots these three terms separately in black, along with their sum T in red. Because the third term's coefficient is so much larger than the other terms' coefficients, the third term dominates for small K values, and T decreases as K increases. But as K gets larger, the third term gets smaller, while the second term gets larger. Eventually the third term becomes smaller than the second term. After that, the running time T increases along with the second term as K increases.

```
117         int yoffset;
118
119         // Table of hues.
120         Color[] huetable;
121
122         // Worker task main program.
123         public void main
124            (String[] args)
125            throws Exception
126            {
127            // Parse command line arguments.
128            width = Integer.parseInt (args[0]);
129            height = Integer.parseInt (args[1]);
130            xcenter = Double.parseDouble (args[2]);
131            ycenter = Double.parseDouble (args[3]);
132            resolution = Double.parseDouble (args[4]);
133            maxiter = Integer.parseInt (args[5]);
134            gamma = Double.parseDouble (args[6]);
135
136            // Initial pixel offsets from center.
137            xoffset = -(width - 1) / 2;
138            yoffset = (height - 1) / 2;
139
140            // Create table of hues for different iteration counts.
141            huetable = new Color [maxiter + 2];
142            for (int i = 1; i <= maxiter; ++ i)
143               huetable[i] = new Color().hsb
144                  (/*hue*/ (float) Math.pow ((double)(i – 1)/maxiter,
145                           gamma),
146                   /*sat*/ 1.0f,
147                   /*bri*/ 1.0f);
148            huetable[maxiter + 1] = new Color().hsb (1.0f, 1.0f, 0.0f);
149
150            // Compute all rows and columns.
151            workerFor() .schedule (dynamic) .exec (new Loop()
152               {
153               ColorArray pixelData;
154
155               public void start()
156                  {
157                  pixelData = new ColorArray (width);
158                  }
159
160               public void run (int r) throws Exception
161                  {
162                  double y = ycenter + (yoffset - r) / resolution;
163
164                  for (int c = 0; c < width; ++ c)
165                     {
166                     double x = xcenter + (xoffset + c) / resolution;
167
168                     // Iterate until convergence.
169                     int i = 0;
170                     double aold = 0.0;
171                     double bold = 0.0;
172                     double a = 0.0;
173                     double b = 0.0;
174                     double zmagsqr = 0.0;
```

Listing 20.1. MandelbrotClu.java (part 3)

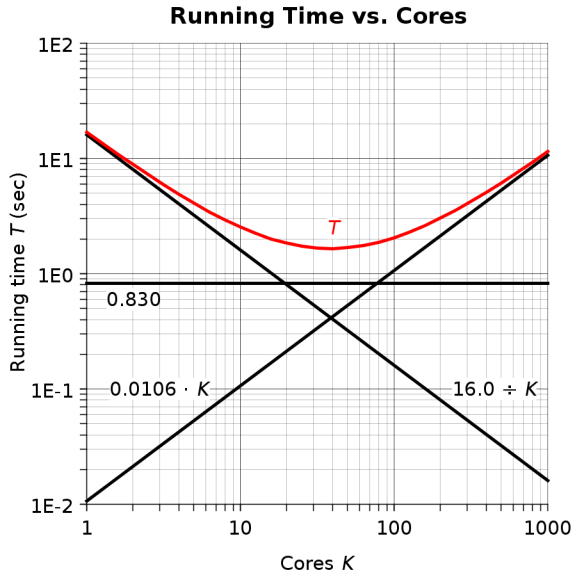


Figure 20.3. MandelbrotClu running time model, 3200×3200-pixel image

How many cores does it take for T to turn around and start increasing? Elementary calculus tells us that the minimum of a function occurs when the function's derivative goes to zero. The derivative of Equation 20.2 is

$$dT/dK = 0.0106 - 16.0 \div K^2 \,. \tag{20.3}$$

Setting $dT/dK = 0$ and solving yields $K = 38.9$. So when I ran the MandelbrotClu program on 40 cores, I went just barely past the point of minimum running time. If I had a bigger cluster, I would have seen the running times start to increase again as I ran on more than 40 cores.

There's an important lesson here. When doing strong scaling on a cluster parallel computer, you don't necessarily want to run the program on all the cores in the cluster. Rather, you want to run the program *on the number of cores that minimizes the running time.* This might be fewer than the total number of cores. Measuring the program's performance and deriving a running time model, as I did above, lets you determine the optimum number of cores to use.

In fact, you might be willing to run on even fewer cores than that. The plots show diminishing returns in the running time reductions as the number of cores approaches the optimum. For the images I computed, the running times on 20 cores were very nearly the same as the running times on 40 cores. So with the 40-core `tardis` cluster, I could compute two images on 20

```
                  while (i <= maxiter && zmagsqr <= 4.0)
                     {
                     ++ i;
                     a = aold*aold - bold*bold + x;
                     b = 2.0*aold*bold + y;
                     zmagsqr = a*a + b*b;
                     aold = a;
                     bold = b;
                     }

                  // Record number of iterations for pixel.
                  pixelData.color (c, huetable[i]);
                  }

               putTuple (new OutputTuple (r, pixelData));
               }
            });
         }
      }

   // Output PNG file writing task.
   private static class OutputTask
      extends Task
      {
      // Command line arguments.
      int width;
      int height;
      File filename;

      // For writing PNG image file.
      ColorPngWriter writer;
      ColorImageQueue imageQueue;

      // Task main program.
      public void main
         (String[] args)
         throws Exception
         {
         // Parse command line arguments.
         width = Integer.parseInt (args[0]);
         height = Integer.parseInt (args[1]);
         filename = new File (args[7]);

         // Set up for writing PNG image file.
         writer = new ColorPngWriter (height, width,
            new BufferedOutputStream
               (new FileOutputStream (filename)));
         filename.setReadable (true, false);
         filename.setWritable (true, false);
         imageQueue = writer.getImageQueue();

         // Overlapped pixel data gathering and file writing.
         parallelDo (new Section()
            {
            // Pixel data gathering section.
            public void run() throws Exception
               {
               OutputTuple template = new OutputTuple();
```

Listing 20.1. MandelbrotClu.java (part 4)

cores each in about the same time as I could compute one image on 40 cores.

We've seen that the term proportional to K in the running time model eventually causes the running time to increase. But where is that term coming from? *It's coming from the inter-task communication.* In particular, it's coming from the communication that results when the worker tasks (under the hood) take parallel loop chunk tuples that the master put into tuple space. Recall that the master uses a proportional schedule. Therefore, the more worker tasks (cores) there are, the more chunk tuples there are. Each chunk tuple requires a certain amount of time to traverse the cluster's backend network. Therefore, the more worker tasks there are, the more time the program spends on network communication transferring chunk tuples from the master to the workers. This accounts for the $0.0106 \cdot K$ term in the running time model.

What about the output tuples? Don't they require some communication time as well? Yes, they do. But for a given image, the number of output tuples is the same—one output tuple per image row—*no matter how many worker tasks (cores) there are.* Thus, the communication time for the output tuples is part of the first term in the running time model, the term that does not depend on K.

Points to Remember

- In a cluster parallel program that must write (or read) a file, consider doing the file I/O in a task that runs in the job's process.
- Use tuple space to convey the worker tasks' results to the output task. Define a tuple subclass whose fields hold the output results.
- When doing strong scaling on a cluster parallel program, as the number of cores increases, the running time initially decreases, but eventually starts to increase again.
- Use the program's running time model, fitted to the program's measured running time data, to determine the optimum number of cores on which to run the program—the number of cores that minimizes the running time.

```
233                OutputTuple tuple;
234                for (int i = 0; i < height; ++ i)
235                   {
236                   tuple = (OutputTuple) takeTuple (template);
237                   imageQueue.put (tuple.row, tuple.pixelData);
238                   }
239                }
240             },
241
242          new Section()
243             {
244             // File writing section.
245             public void run() throws Exception
246                {
247                writer.write();
248                }
249             });
250          }
251       }
252    }
```

Listing 20.1. MandelbrotClu.java (part 5)

Chapter 21
Interacting Tasks

Let's tackle the *N*-body zombie program from Chapter 8 and develop a cluster parallel version. This program will be more challenging than the other cluster parallel programs we've seen, for several reasons. First, in the previous programs the outer loop had no sequential dependencies, so I was able to parallelize it using the master-worker cluster parallel loop pattern, and I was able to implement that pattern simply by calling the `masterFor()` and `workerFor()` methods. In the zombie program, the outer loop does have sequential dependencies, so I have to parallelize the middle loop rather than the outer loop; but as we will see, this is not quite as simple. Second, in the previous programs there were no data structures that had to be shared among the tasks of the job. In the zombie program there are shared data structures, namely the arrays holding the zombies' (*x*, *y*) coordinates; but it is not possible to directly share data among tasks running in separate processes. Third, in most of the previous cluster parallel programs, the only inter-task communication occurred at the end of the computation, when each worker sent its semifinal result to the reduce task. In the cluster parallel Mandelbrot Set program, the workers sent pixel rows to the output task, but the workers did not interact with each other. As we will see, the zombie program requires frequent and copious communication among the workers.

Following the hybrid parallel programming pattern on a cluster parallel computer (Figure 1.11), the computation is partitioned across multiple nodes, with one task (process) on each node and multiple threads in each process. Following the distributed memory pattern, the data—the zombies' current and next positions—is likewise partitioned across the nodes. Partitioning the computation and the data, though, requires a bit of thought.

At each time step, the zombie program calculates the next position of each zombie. In the single-node multicore version, I partitioned the next-position computation among several threads (cores), each thread calculating just one "slice" of the next positions. I'll do the same in the cluster version; each worker task will calculate just one slice of the next positions. But this means that each worker task needs to allocate storage *only for its own slice* of the next positions, not for all of them. On the other hand, calculating any zombie's next position requires reading all the zombies' current positions. Therefore, each worker task needs to allocate storage for *all* of the current positions. The storage allocation looks like Figure 21.1.

In this example, there are 24 zombies and four worker tasks. Each worker allocates all 24 elements for the current position arrays `x` and `y`. (Each array element is labeled with its index.) Because calculating any zombie's next position takes the same amount of time, partitioning the next positions equally among the workers will result in a balanced load. After partitioning, each worker ends up with a six-element slice. Thus, each worker allocates only six elements for the next position arrays `xnext` and `ynext`. Note that in worker tasks 1 and higher, the `xnext` and `ynext` array indexes are not the

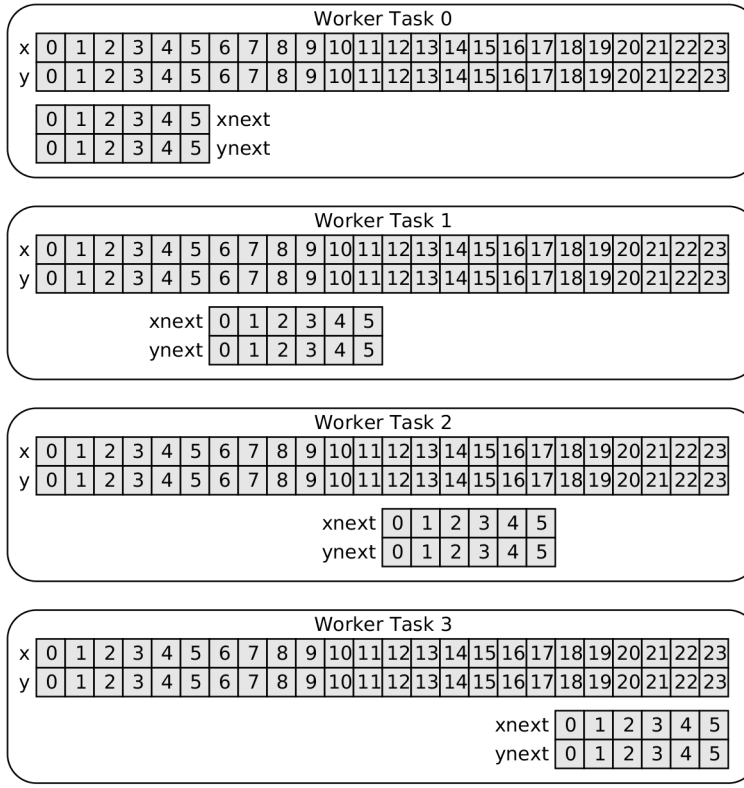


Figure 21.1. Zombie program storage allocation

same as the corresponding zombie's `x` and `y` array indexes; this subtle detail will turn up later when we write the code.

During each outer loop iteration (each time step), the middle loop iterates over just the worker's own slice of the zombies, calculates those zombies' next positions, and stores them in the `xnext` and `ynext` arrays. For each zombie in the worker's slice, the inner loop iterates over all the zombies' current positions in the `x` and `y` arrays. These calculations proceed independently in parallel in all the workers.

At the end of each outer loop iteration, the zombies' just-calculated next positions need to become the current positions for the next time step. Each worker can update its own slice by copying the `xnext` and `ynext` arrays back to the proper indexes in the `x` and `y` arrays. Each worker must obtain the remaining slices of `x` and `y` from the other workers. Because the workers are in different processes, this has to be done through tuple space (message passing). Each worker puts multiple copies of a tuple into tuple space, one copy

for each of the other workers, containing the worker's own next positions. Each of the other workers takes one of these copies and stores its contents in the proper indexes in the `x` and `y` arrays. Figure 21.2 shows the tuples put by worker task 0 and taken by worker tasks 1 through 3. In a similar manner, worker tasks 1 through 3 put tuples containing their slices, and the other tasks take those tuples.

Each worker also puts one additional copy of its next positions tuple into tuple space. This extra copy is taken by a snapshot task at each time step. Before the first time step, at designated time steps during the calculation (if so specified on the command line), and after the last time step, the snapshot task prints the zombies' positions.

Consequently, the zombie program's inter-task communication is frequent (it happens on every time step) and copious (each tuple contains numerous zombie (x, y) coordinates). In other words, the cluster parallel zombie program is tightly coupled. It will be interesting to see what effect the frequent and copious communication has on the program's running time.

Turning to the code (Listing 21.1), the ZombieClu job main program sets up a start rule with a task group consisting of K worker tasks, where the number of workers K is specified on the `pj2` command line (lines 33–35). Each task's command line arguments are the same as those of the job. The job main program also sets up a separate start rule with a snapshot task that will print snapshots of the zombie positions at the beginning, the end, and any intermediate time steps (lines 38–39). The snapshot task runs in the job process so it doesn't tie up a backend node.

To convey the results of each time step from each worker task to the other tasks, the program uses the ZombieTuple subclass (line 64). The tuple has fields for the rank of the worker task that sent the tuple, the time step, the lower bound index of the worker's slice, the zombies' (x, y) coordinates, and the "delta." In addition to calculating the zombies' next positions, the worker tasks also add up the total distance the zombies moved (the delta), in order to detect when the zombies reach the equilibrium state. However, with the data partitioned among the workers, each worker can calculate only its own part of the delta. To calculate the total delta, each worker must add in all the other worker's partial deltas. Each worker can then determine whether the total delta is below the threshold, and terminate itself if so. Thus, along with the zombies' next positions, each worker includes its partial delta in the zombie tuple sent to the other workers.

Because tuples are streamable objects, class ZombieTuple defines a no-argument constructor; the `writeOut()` method that writes all the fields; and the `readIn()` method that reads all the fields. Class ZombieTuple also overrides the `matchContent()` method. When deciding whether a certain target tuple matches a certain template, the job first checks whether the target is an instance of the template's class or a subclass thereof. If so, the job then calls

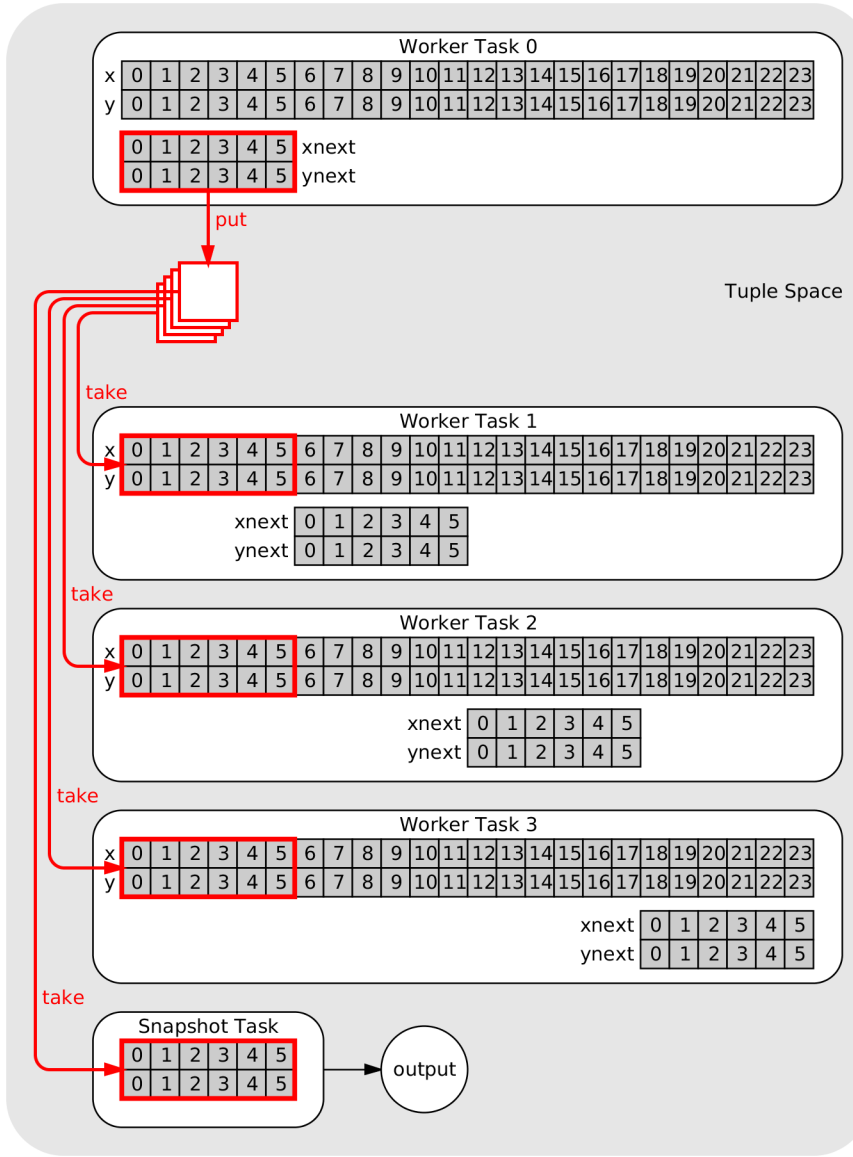


Figure 21.2. Zombie program inter-task communication

the `matchContent()` method on the template, passing the target as the argument. If the method returns true, the target matches. The default implementation of the `matchContent()` method always returns true, so any target will match the template no matter what is stored in the target's fields. This is not what I need for the zombie program. Instead, I overrode the `matchContent()` method so that a target will match a template only if the worker rank and time step fields are the same in the template and the target. This lets each worker match and take the correct zombie tuple from every other worker at each time step.

Next comes the worker task class (line 127). The first difference between this cluster version and the multicore version in Chapter 8 is the storage allocation for the zombies' positions. The worker allocates storage for all n current positions in the `x` and `y` arrays (lines 176–177). To determine this worker's slice of the next positions, the worker calls the `Chunk.partition()` method, passing in the lower and upper bounds of the entire index range (0 to $n - 1$), the number of equal-sized slices into which to partition that range (*i.e.*, the number of workers), and the index of the desired slice (*i.e.*, this worker's rank). From the Chunk object that is returned, the worker extracts the lower and upper bound indexes of the slice, `lb` and `ub`, and the length of the slice, `len`. The worker allocates storage for `len` next positions in the `xnext` and `ynext` arrays (lines 181–186).

Each worker task proceeds to set all the zombies' current positions to their random initial positions (lines 189–194). Because every worker seeds its own pseudorandom number generator with the same seed, every worker's PRNG generates exactly the same sequence of random numbers, and every worker ends up with exactly the same initial positions.

Each worker begins the triply-nested loop to calculate the zombies' positions as they move—the sequential outer loop over the time steps (line 200); the parallel middle loop over the zombies *in the worker's slice*, from `lb` to `ub` rather than 0 to $n - 1$ (line 205); and the sequential inner loop over all the zombies (line 221). Note the array index used to store the new (x, y) coordinates into the `xnext` and `ynext` arrays (lines 235–236). The zombie index `i` has to be offset by the lower bound of the worker's slice `lb` to get the proper index for the `xnext` and `ynext` arrays.

After the middle loop finishes, the just-calculated next positions must become the current positions. The multicore version did this by swapping the array references, but that won't work in the cluster version. Instead, here is where each worker puts copies of a zombie tuple into tuple space, containing the worker's slice of the next positions plus the worker's partial delta (lines 248–249). Each worker puts `size` copies of the tuple: `size` – 1 copies for the other workers, plus one copy for the snapshot task. Each worker then takes zombie tuples from the other workers out of tuple space. Note how, in the `takeTuple()` method call, the template's `step` and `rank` fields are set to the

```
1  package edu.rit.pj2.example;
2  import edu.rit.io.InStream;
3  import edu.rit.io.OutStream;
4  import edu.rit.pj2.Chunk;
5  import edu.rit.pj2.Job;
6  import edu.rit.pj2.Loop;
7  import edu.rit.pj2.Task;
8  import edu.rit.pj2.Tuple;
9  import edu.rit.pj2.vbl.DoubleVbl;
10 import edu.rit.util.Random;
11 import java.io.IOException;
12 import static java.lang.Math.*;
13 public class ZombieClu
14    extends Job
15    {
16    // Job main program.
17    public void main
18       (String[] args)
19       {
20       // Parse command line arguments.
21       if (args.length != 9) usage();
22       long seed = Long.parseLong (args[0]);
23       int N = Integer.parseInt (args[1]);
24       double W = Double.parseDouble (args[2]);
25       double G = Double.parseDouble (args[3]);
26       double L = Double.parseDouble (args[4]);
27       double dt = Double.parseDouble (args[5]);
28       double eps = Double.parseDouble (args[6]);
29       int steps = Integer.parseInt (args[7]);
30       int snap = Integer.parseInt (args[8]);
31
32       // Set up a task group of K worker tasks.
33       int K = workers();
34       if (K == DEFAULT_WORKERS) K = 1;
35       rule() .task (K, WorkerTask.class) .args (args);
36
37       // Set up snapshot task.
38       rule() .task (SnapshotTask.class) .args (args) .args (""+K)
39          .runInJobProcess();
40       }
41
42    // Print a usage message and exit.
43    private static void usage()
44       {
45       System.err.println ("Usage: java pj2 [workers=<K>] " +
46          "edu.rit.pj2.example.ZombieClu <seed> <N> <W> <G> <L> " +
47          "<dt> <eps> <steps> <snap>");
48       System.err.println ("<K> = Number of worker tasks " +
49          "(default: 1)");
50       System.err.println ("<seed> = Random seed");
51       System.err.println ("<N> = Number of bodies");
52       System.err.println ("<W> = Region size");
53       System.err.println ("<G> = Attraction factor");
54       System.err.println ("<L> = Attraction length scale");
55       System.err.println ("<dt> = Time step size");
56       System.err.println ("<eps> = Convergence threshold");
57       System.err.println ("<steps> = Number of time steps " +
58          "(0 = until convergence)");
```

Listing 21.1. ZombieClu.java (part 1)

proper time step and worker rank; this ensures that the template will match the proper zombie tuple. Each worker stores this tuple's contents into the proper indexes of the current position arrays, and sums up the partial deltas (lines 264–268). For the worker's own rank, the worker just copies the next position arrays into the proper indexes of the current position arrays, without going through tuple space (lines 258–260).

One subtle detail: It is imperative that every worker sum up the partial deltas *in the same order.* Why? Because floating point arithmetic is not exact. If different workers summed up the partial deltas in different orders, different workers might end up with slightly different total deltas; one worker's total delta might end up below the threshold while the other workers' total deltas did not; one worker might therefore terminate while the other workers did not; and the program would come to a standstill. Summing up the partial deltas in the same order in every worker prevents this from happening.

At this point every worker's current positions have been updated with their new values for the next time step. The worker checks for termination (lines 275–277) and, if it's not time to terminate yet, goes on to the next iteration of the outer loop. Once the outer time step loop finishes, each worker's `run()` method returns and the worker terminates.

Last comes the snapshot task (line 283). This task follows along with what the worker tasks are doing, time step by time step. The snapshot task begins by generating and printing all the zombies' initial positions, using a PRNG initialized with the same seed as the worker tasks. The snapshot task then executes a loop over all the time steps. At each time step, the snapshot task takes a series of snapshot tuples out of tuple space, one from each worker task. By setting the `step` and `rank` fields properly in each `takeTuple()` method call's template, the snapshot task ensures that it takes snapshot tuples in ascending order of time step, and within each time step in ascending order of worker task rank. At the specified snapshot intervals, the snapshot task prints the zombies' positions as recorded in the workers' zombie tuples. The snapshot task also adds up the workers' partial deltas, and exits its outer loop under the exact same conditions as the workers. At this point the snapshot task terminates, the worker tasks have also terminated, and so the job terminates.

Whew! What a program. This is the sort of thing you have to write if you need to partition data across tasks (nodes) of a cluster parallel program, and if the tasks have to communicate data back and forth among themselves.

To study the ZombieClu program's weak scaling performance, I ran the program on the `tardis` cluster using the same approach as in Chapter 10. I ran the program on five problem sizes on one core, n = 200, 300, 400, 600, and 800 zombies. I told the program to do s = 10,000 time steps (rather than stopping at equilibrium). I scaled the program up from one to ten workers (4 to 40 cores). I increased n in proportion to the square root of the number of

```
59          System.err.println ("<snap> = Snapshot interval (0 = none)");
60          throw new IllegalArgumentException();
61          }
62
63       // Tuple with results of one time step.
64       private static class ZombieTuple
65          extends Tuple
66          {
67          public int rank;     // Worker task rank
68          public int step;     // Time step number
69          public int lb;       // Lower bound zombie index
70          public double[] x;   // Zombie X coordinates
71          public double[] y;   // Zombie Y coordinates
72          public double delta; // Zombie position delta
73
74          public ZombieTuple()
75             {
76             }
77
78          public ZombieTuple
79             (int rank,
80              int step,
81              int lb,
82              double[] x,
83              double[] y,
84              double delta)
85             {
86             this.rank = rank;
87             this.step = step;
88             this.lb = lb;
89             this.x = x;
90             this.y = y;
91             this.delta = delta;
92             }
93
94          public boolean matchContent
95             (Tuple target)
96             {
97             ZombieTuple t = (ZombieTuple) target;
98             return this.rank == t.rank && this.step == t.step;
99             }
100
101         public void writeOut
102            (OutStream out)
103            throws IOException
104            {
105            out.writeInt (rank);
106            out.writeInt (step);
107            out.writeInt (lb);
108            out.writeDoubleArray (x);
109            out.writeDoubleArray (y);
110            out.writeDouble (delta);
111            }
112
113         public void readIn
114            (InStream in)
115            throws IOException
116            {
```

Listing 21.1. ZombieClu.java (part 2)

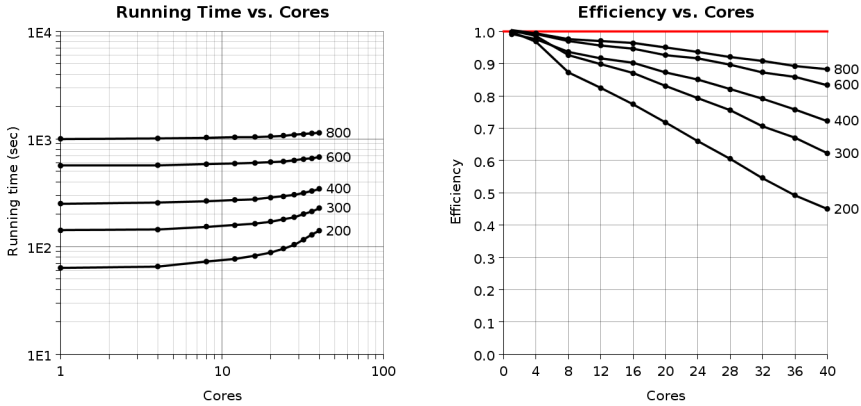


Figure 21.3. ZombieClu weak scaling performance metrics

cores. Figure 21.3 plots the running times and efficiencies I observed.

An overall trend is apparent in the running time and efficiency data. As the number of cores increases, at first the running times stay approximately the same, as I would expect under weak scaling. But as the number of cores continues to increase, *the running times go up.* Correspondingly, *the efficiencies go down*, ending up around 45 to 90 percent at 40 cores depending on the problem size.

Why do the running times go up? Here is the fitted running time model:

$$T = 4.72\times10^{-11}N + (1.60 + 6.59\times10^{-12}N) \cdot K + 1.57\times10^{-7}N \div K \qquad (21.1)$$

But with weak scaling, the problem size N increases as the number of cores K increases. For the zombie program running on one core, the problem size is $N_1 = sn^2$. On K cores, the problem size is $N = KN_1$. Substituting KN_1 for N in Equation 21.2 gives this alternative formula for the running time in terms of the problem size on one core and the number of cores:

$$T = 1.57\times10^{-7}N_1 + (1.60 + 4.72\times10^{-11}N_1) \cdot K + 6.59\times10^{-12}N_1 \cdot K^2 \qquad (21.2)$$

Rather than T being constant, as would be the case for ideal weak scaling, T is actually a *quadratic* function of the number of cores, although with rather small coefficients on the terms proportional to K and K^2. Still, as K increases, those terms also increase, causing the running time to go up as is apparent in the plots.

What in the program is causing the running time to increase with K? It is because, as the number of cores increases, *the amount of inter-task communication increases.* The number of zombie tuples sent and received through the cluster backend network is proportional to the number of worker tasks. Fur-

```
117            rank = in.readInt();
118            step = in.readInt();
119            lb = in.readInt();
120            x = in.readDoubleArray();
121            y = in.readDoubleArray();
122            delta = in.readDouble();
123            }
124         }
125
126      // Worker task class.
127      private static class WorkerTask
128         extends Task
129         {
130         // Command line arguments.
131         long seed;
132         int N;
133         double W;
134         double G;
135         double L;
136         double dt;
137         double eps;
138         int steps;
139
140         // Task group size and worker rank.
141         int size;
142         int rank;
143
144         // Current zombie positions.
145         double[] x;
146         double[] y;
147
148         // Next zombie positions.
149         int lb, ub, len;
150         double[] xnext;
151         double[] ynext;
152
153         // For detecting convergence.
154         DoubleVbl delta = new DoubleVbl.Sum();
155
156         // Task main program.
157         public void main
158            (String[] args)
159            throws Exception
160            {
161            // Parse command line arguments.
162            seed = Long.parseLong (args[0]);
163            N = Integer.parseInt (args[1]);
164            W = Double.parseDouble (args[2]);
165            G = Double.parseDouble (args[3]);
166            L = Double.parseDouble (args[4]);
167            dt = Double.parseDouble (args[5]);
168            eps = Double.parseDouble (args[6]);
169            steps = Integer.parseInt (args[7]);
170
171            // Get task group size and worker rank.
172            size = groupSize();
173            rank = taskRank();
174
```

Listing 21.1. ZombieClu.java (part 3)

thermore, the amount of data transmitted in all the zombie tuple increases too, because the number of zombies increases due to weak scaling as the number of cores increases. So as the number of cores increases, there are more messages to put and take tuples containing more data, and it requires more time to send all the messages. This is time that cannot be parallelized. Thus, while the running time for the parallelizable computation stays the same due to weak scaling, the running time for the communication continually increases as the number of cores increases.

However, notice that the running time for the larger problem sizes does not go up as much as for the smaller problem sizes as K increases. This is because the coefficient of the term proportional to K in Equation 21.2 increases very little as the problem size N_1 increases. The first term in Equation 21.2, on the other hand, goes up a lot as N_1 increases. (The third term is negligible in comparison to the other two terms for the number of cores we are considering.) For example, for $n = 200$ zombies, $N_1 = 10{,}000 \cdot 200^2 = 4.00 \times 10^8$, and T is about $62.8 + 1.62K$. But for $n = 800$ zombies, $N_1 = 10{,}000 \cdot 800^2 = 6.40 \times 10^9$, and T is about $1000 + 1.90K$. For the larger problem size, the second term is much smaller relative to the first term, so the running time increase with K is relatively not as large.

Consequently, the efficiencies for the larger problem sizes are better than for the smaller problem sizes. If I had calculated 1,000 or 2,000 zombies on one core and had done weak scaling onto more cores, I would have seen very good efficiencies, even with all the inter-task communication.

Putting it another way, the amount of computation in the zombie program is proportional to the square of the number of zombies: at each time step, the program has to do n^2 iterations of the innermost loop. On the other hand, the amount of communication in the zombie program is proportional just to the number of zombies: at each time step, the n zombies' positions are communicated among the tasks. Therefore, as n increases, the ratio of communication to computation goes down, the relative overhead due to communication goes down, and the efficiencies droop to a lesser extent. As I've said before, to get good efficiencies in a cluster parallel program with message passing, the amount of computation needs to be large relative to the amount of communication.

Back at the end of Chapter 16 I said that tightly coupled cluster parallel programs, like the ZombieClu program, are outside Parallel Java 2's design center. Now we can see why. The ZombieClu program *works* perfectly; it calculates the correct series of zombie positions. But the ZombieClu program does not *scale* well due to its large amount of communication. It's possible to get good scaling with this program, but to do so I have to calculate very large problem sizes. To get the 90 percent efficiency I observed on 40 cores, for example, I had to compute $800 \cdot 40^{1/2} = 5{,}060$ zombies. The program would scale better if implemented using a high-performance message passing li-

```
175            // Allocate storage for current positions for all zombies.
176            x = new double [N];
177            y = new double [N];
178
179            // Allocate storage for next positions for just this
180            // worker's slice of zombies.
181            Chunk slice = Chunk.partition (0, N - 1, size, rank);
182            lb = slice.lb();
183            ub = slice.ub();
184            len = (int) slice.length();
185            xnext = new double [len];
186            ynext = new double [len];
187
188            // Initialize zombies' (x,y) coordinates.
189            Random prng = new Random (seed);
190            for (int i = 0; i < N; ++ i)
191               {
192               x[i] = prng.nextDouble()*W;
193               y[i] = prng.nextDouble()*W;
194               }
195
196            // Do time steps.
197            int t = 0;
198            ZombieTuple template = new ZombieTuple();
199            ZombieTuple zt = null;
200            for (;;)
201               {
202               delta.item = 0.0;
203
204               // Do each zombie i for this worker's slice of zombies.
205               parallelFor (lb, ub) .exec (new Loop()
206                  {
207                  DoubleVbl thrDelta;
208
209                  public void start()
210                     {
211                     thrDelta = threadLocal (delta);
212                     }
213
214                  public void run (int i)
215                     {
216                     double vx = 0.0;
217                     double vy = 0.0;
218                     double dx, dy, d, v;
219
220                     // Accumulate velocity due to every other body j.
221                     for (int j = 0; j < N; ++ j)
222                        {
223                        if (j == i) continue;
224                        dx = x[j] - x[i];
225                        dy = y[j] - y[i];
226                        d = sqrt(dx*dx + dy*dy);
227                        v = G*exp(-d/L) - exp(-d);
228                        vx += v*dx/d;
229                        vy += v*dy/d;
230                        }
231
232                     // Move body i in the direction of its velocity.
```

Listing 21.1. ZombieClu.java (part 4)

brary like MPI. Even with MPI, though, the program would still eventually run into scaling issues.

Tightly coupled parallel programs, like the ZombieClu program in particular and *N*-body programs in general, are really best suited to run on a single multicore node, like the ZombieSmp program in Chapter 8. There, each thread can directly access all the current positions in shared memory, without needing time-consuming message passing. Multicore machines nowadays have much larger main memories and many more cores (including GPU accelerator cores) than they did decades ago when MPI was invented, so a single multicore node now can solve much larger problems (because of larger main memories) at a much larger computation rate (because of more cores) than back then. Still, a problem too large for a single multicore node has to be run on a cluster, with all the programming difficulty and message passing overhead that entails.

Under the Hood

The worker tasks in the ZombieClu program were specified as a task group in a single rule in the job main program. Consequently, the Tracker schedules all the worker tasks as a unit. The Tracker will not start any of the worker tasks until there are enough idle nodes to start all the worker tasks, one worker on each node. Because every worker task communicates with every other worker task, it is critical that they all be started as a unit, otherwise the program would come to a standstill.

As we saw in previous chapters, class edu.rit.pj2.Chunk provides the chunk tuple used in a master-worker cluster parallel for loop to specify a chunk of loop iterations, where the loop index is type `int`. Class edu.rit.pj2.LongChunk does the same for a loop index of type `long`. These classes also provide the static `partition()` method, which partitions a given index range in the same manner as a fixed schedule: one chunk for each of a given number of workers, each chunk (except possibly the last) having the same number of indexes. The ZombieClu program uses the chunk returned by the `partition()` method to partition the zombie next position arrays among the worker tasks (the length of the next position arrays is that of the chunk). The ZombieClu program also uses this chunk to partition the middle loop indexes among the worker tasks (the lower and upper loop index bounds are those of the chunk). Thus, the ZombieClu program partitions the middle parallel loop using, in effect, a fixed schedule at the job level. Each worker's parallel loop in turn partitions its index range among the threads of the task using the default fixed schedule. This fixed partitioning is appropriate because the computation is inherently balanced.

At each time step, each worker task puts *K* copies of a zombie tuple into tuple space, where *K* is the number of workers. This is done by specifying

```
233                     dx = vx*dt;
234                     dy = vy*dt;
235                     xnext[i-lb] = x[i] + dx;
236                     ynext[i-lb] = y[i] + dy;
237
238                     // Accumulate position delta.
239                     thrDelta.item += abs(dx) + abs(dy);
240                     }
241                  });
242
243               // Advance to next time step.
244               ++ t;
245
246               // Send new zombie positions to the other workers and to
247               // the snapshot task.
248               putTuple (size, new ZombieTuple
249                  (rank, t, lb, xnext, ynext, delta.item));
250
251               // Receive new zombie positions from the other workers.
252               double totalDelta = 0.0;
253               template.step = t;
254               for (int i = 0; i < size; ++ i)
255                  {
256                  if (i == rank)
257                     {
258                     System.arraycopy (xnext, 0, x, lb, len);
259                     System.arraycopy (ynext, 0, y, lb, len);
260                     totalDelta += delta.item;
261                     }
262                  else
263                     {
264                     template.rank = i;
265                     zt = takeTuple (template);
266                     System.arraycopy (zt.x, 0, x, zt.lb, zt.x.length);
267                     System.arraycopy (zt.y, 0, y, zt.lb, zt.y.length);
268                     totalDelta += zt.delta;
269                     }
270                  }
271
272               // Stop when position delta is less than convergence
273               // threshold or when the specified number of time steps
274               // have occurred.
275               if ((steps == 0 && totalDelta < eps) ||
276                  (steps != 0 && t == steps))
277                     break;
278               }
279            }
280         }
281
282      // Snapshot task class.
283      private static class SnapshotTask
284         extends Task
285         {
286         // Task main program.
287         public void main
288            (String[] args)
289            throws Exception
290            {
```

Listing 21.1. ZombieClu.java (part 5)

the number of copies as the first argument of the `putTuple()` method call (line 248). However, to reduce the number of bytes sent over the network, and hence reduce the amount of time spent, this method call sends only one message, not *K* messages. The message consists of the tuple's contents and the number of copies (*K*). The job process receives the put-tuple message. To reduce the number of bytes of storage allocated in the job process, the job stores *K references* to the one tuple in the job's internal tuple list. When a worker sends a take-tuple message to the job process, the job process sends the tuple back to the worker and removes one of the references. After all *K* references have gone away, the tuple goes away. In this manner, the tuple contents are transmitted over the network only once and occupy storage in the job process only once, no matter how many "copies" are put into tuple space.

Points to Remember

- Partition the program's data among the worker tasks where necessary. Use the `partition()` method of class Chunk or LongChunk to do the partitioning.
- When tasks need to communicate data among each other during a computation, the sending task wraps the data in a tuple and puts the tuple into tuple space; the receiving task takes the tuple out of tuple space and extracts the data.
- Such tuple classes must include fields that identify which part of the computation the data is coming from, in addition to fields holding the data itself.
- Such tuple classes must override the `matchContent()` method to check whether the identifying fields in the target equal the identifying fields in the template.
- To get good performance in a cluster parallel program with tasks that communicate during the computation, the time spent in computation must vastly exceed the time spent in communication.

```
291          // Parse command line arguments.
292          long seed = Long.parseLong (args[0]);
293          int N = Integer.parseInt (args[1]);
294          double W = Double.parseDouble (args[2]);
295          double eps = Double.parseDouble (args[6]);
296          int steps = Integer.parseInt (args[7]);
297          int snap = Integer.parseInt (args[8]);
298          int K = Integer.parseInt (args[9]);
299
300          // Print zombies' initial (x,y) coordinates.
301          Random prng = new Random (seed);
302          for (int i = 0; i < N; ++ i)
303             {
304             double x_i = prng.nextDouble()*W;
305             double y_i = prng.nextDouble()*W;
306             System.out.printf ("0\t%d\t%g\t%g%n", i, x_i, y_i);
307             System.out.flush();
308             }
309
310          // Do time steps.
311          int t = 0;
312          ZombieTuple template = new ZombieTuple();
313          ZombieTuple[] zt = new ZombieTuple [K];
314          for (;;)
315             {
316             // Advance to next time step.
317             ++ t;
318
319             // Receive and print new zombie positions from the
320             // workers.
321             double totalDelta = 0.0;
322             template.step = t;
323             for (int i = 0; i < K; ++ i)
324                {
325                template.rank = i;
326                zt[i] = takeTuple (template);
327                totalDelta += zt[i].delta;
328                }
329             if (snap > 0 && t % snap == 0)
330                snapshot (t, zt);
331
332             // Stop when position delta is less than convergence
333             // threshold or when the specified number of time steps
334             // have occurred.
335             if ((steps == 0 && totalDelta < eps) ||
336                (steps != 0 && t == steps))
337                   break;
338             }
339
340          // Print zombies' final positions.
341          if (snap == 0 || t % snap != 0)
342             snapshot (t, zt);
343          }
344       }
345
346       // Print a snapshot of the zombies' positions.
347       private static void snapshot
348          (int t,
```

Listing 21.1. ZombieClu.java (part 6)

```
349           ZombieTuple[] zt)
350          {
351          for (int i = 0; i < zt.length; ++ i)
352             {
353             ZombieTuple zt_i = zt[i];
354             for (int j = 0; j < zt_i.x.length; ++ j)
355                System.out.printf ("%d\t%d\t%g\t%g%n",
356                   t, zt_i.lb + j, zt_i.x[j], zt_i.y[j]);
357             System.out.flush();
358             }
359          }
360       }
361    }
```

Listing 21.1. ZombieClu.java (part 7)

Chapter 22
Cluster Heuristic Search

A massively parallel randomized approximation (MPRA) algorithm for solving a hard combinatorial problem—like the minimum vertex cover problem from Chapters 11 and 12—can potentially find an approximate solution closer to the exact solution by doing more iterations, by examining more candidate solutions. And because the candidates can all be examined independently, with no sequential dependencies, an MPRA program is ideally suited for a parallel computer. Indeed, in Chapter 12 we developed a single-node multicore MPRA program for the minimum vertex cover problem, and we scaled it up to the full number of cores on one node by increasing the number of iterations in proportion to the number of cores (weak scaling).

On a cluster parallel computer, we can scale an MPRA program up even further, utilizing all the cores on multiple nodes instead of just one node.

Recall the design of the multicore parallel MinVerCovSmp3 program from Chapter 12. It follows the typical massively parallel with reduction pattern. The design of the *cluster* parallel MinVerCovClu3 program follows the *cluster* parallel reduction pattern (Figure 22.1). The program consists of multiple worker tasks. Each worker task consists of multiple threads. The iterations, each iteration examining a randomly chosen candidate cover, are partitioned among the workers and threads. Each thread finds the best candidate, the one with the fewest vertices, among the candidates the thread examines. Within each worker, the per-thread results are reduced together, yielding the best candidate for that worker. Each worker puts a tuple containing the worker's semifinal result into tuple space. A separate reduction task takes these tuples, reduces the semifinal results together, and outputs the best candidate found by all the threads in all the workers.

The design of the cluster parallel minimum vertex cover MPRA program, MinVerCovClu3, is similar to that of the cluster parallel PiClu program from Chapter 18, with one difference. The PiClu program's inputs were merely a few parameters obtained from the command line. The minimum vertex cover program, however, gets its input graph by reading a file. In a cluster version of the program, the worker tasks typically run in a special Parallel Java account on the cluster's backend nodes and might not be able to read files in the user's account. Therefore, code in the job process, which does run in the user's account, must read the graph input file and distribute the graph to the workers via tuple space.

Turning to the code in Listing 22.1, the MinVerCovClu3 program begins with the job main program (line 20). It reads the graph input file and sets up the adjacency matrix, like all the prior versions (lines 31–45). But then it wraps the adjacency matrix inside an ObjectArrayTuple, which is a tuple subclass in the Parallel Java 2 Library that holds an array of objects, and puts the tuple into tuple space (line 48). This code runs in the job process in the user's account, before defining any tasks. The job main program goes on to define a group of worker tasks using the master-worker parallel loop pattern

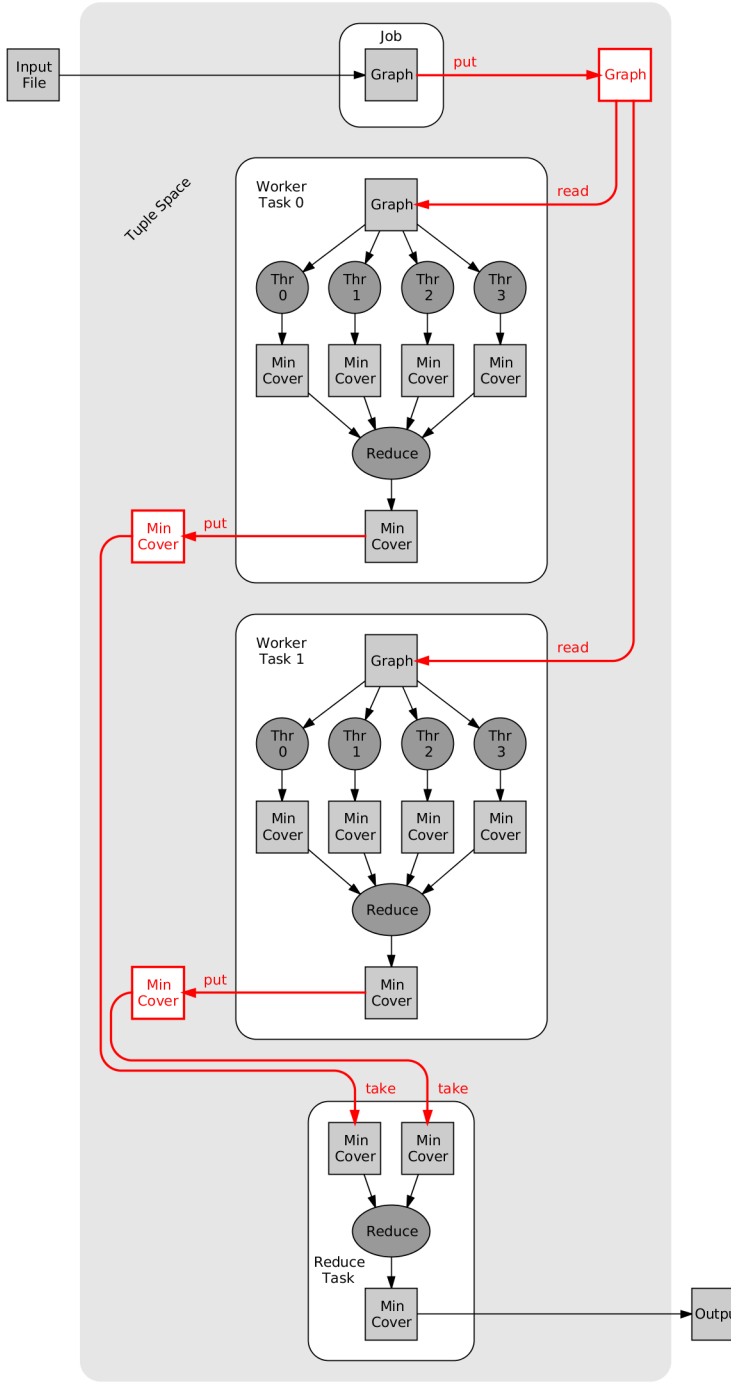


Figure 22.1. Cluster parallel minimum vertex cover program

(line 51) as well as a reduction task that will run in the job process when the worker tasks have finished (lines 54–55).

One caution, though: The program has to spend a certain amount of time reading in the file and communicating the graph through tuple space to all the workers. This time is part of the program's sequential portion. But because the graph file is small, and because the program does so much parallelizable computation, the sequential fraction due to reading and communicating the input file is small and does not degrade the program's scalability much. Later in the book we will study *big data* programs where the input files are very large. In those programs, reading the input files sequentially in the job main program would kill the performance. Rather, such programs will use the *map-reduce* paradigm to both read and process the input files in parallel.

The worker task class (line 79) is almost the same as the multicore parallel minimum vertex cover program from Chapter 12. Instead of reading the input file, the worker obtains the graph's adjacency matrix by reading the ObjectArrayTuple that the job main program put into tuple space (lines 105–106). The worker *reads* the tuple, rather than taking it, so the tuple will remain in tuple space for the other workers to read. The worker performs the worker portion of the master-worker parallel loop (line 110). The worker wraps its partially reduced result inside an ObjectTuple, which is a tuple subclass in the Parallel Java 2 Library that holds a single object, and puts the tuple into tuple space (line 135).

Finally, the reduction task (line 151) is similar to the one in the cluster parallel π estimating program. The reduction task reads the BitSet semifinal results put into tuple space by the worker tasks, reduces the semifinal results together into the final overall BitSet result, and prints the result.

To study the MinVerCovClu3 program's weak scaling performance, I ran the program on the `tardis` cluster on five different random graphs, with 100, 200, 300, 400, and 600 vertices, each graph with a density of 0.4. On one core I ran the program with 25 million iterations. I scaled the program up from one to ten workers (4 to 40 cores). I increased the number of iterations in proportion to the number of cores: 100 million iterations on 4 cores, 200 million iterations on 8 cores, and so on, up to one billion iterations on 40 cores. Figure 22.2 plots the running times and efficiencies I observed. The program experiences fairly good weak scaling efficiencies, about 80 percent or better, out to 40 cores. The running time plots show that the running time increases slightly as the program scales up to more cores (nodes). Why? Because when the program runs with more tasks on more nodes, it spends more time on inter-task communication; each task has to read one adjacency list tuple and put one vertex set result tuple.

An exhaustive search program would require 2^{100} to 2^{600} iterations to find the exact minimum vertex covers for these 100- to 600-vertex graphs—far too long to be practical. The heuristic search MinVerCovClu3 program

```
1  package edu.rit.pj2.example;
2  import edu.rit.io.InStream;
3  import edu.rit.io.OutStream;
4  import edu.rit.pj2.Job;
5  import edu.rit.pj2.LongLoop;
6  import edu.rit.pj2.Task;
7  import edu.rit.pj2.tuple.ObjectTuple;
8  import edu.rit.pj2.tuple.ObjectArrayTuple;
9  import edu.rit.pj2.vbl.BitSetVbl;
10 import edu.rit.util.BitSet;
11 import edu.rit.util.Random;
12 import edu.rit.util.RandomSubset;
13 import java.io.File;
14 import java.io.IOException;
15 import java.util.Scanner;
16 public class MinVerCovClu3
17    extends Job
18    {
19    // Job main program.
20    public void main
21       (String[] args)
22       throws Exception
23       {
24       // Parse command line arguments.
25       if (args.length != 3) usage();
26       File file = new File (args[0]);
27       long seed = Long.parseLong (args[1]);
28       long N = Long.parseLong (args[2]);
29
30       // Read input file, set up adjacency matrix.
31       Scanner s = new Scanner (file);
32       int V = s.nextInt();
33       int E = s.nextInt();
34       if (V < 1) usage ("V must be >= 1");
35       BitSet[] adjacent = new BitSet [V];
36       for (int i = 0; i < V; ++ i)
37          adjacent[i] = new BitSet (V);
38       for (int i = 0; i < E; ++ i)
39          {
40          int a = s.nextInt();
41          int b = s.nextInt();
42          adjacent[a].add (b);
43          adjacent[b].add (a);
44          }
45       s.close();
46
47       // Put adjacency matrix into tuple space for workers.
48       putTuple (new ObjectArrayTuple<BitSet> (adjacent));
49
50       // Set up a task group of K worker tasks.
51       masterFor (0, N - 1, WorkerTask.class) .args (""+seed, ""+V);
52
53       // Set up reduction task.
54       rule() .atFinish() .task (ReduceTask.class) .args (""+V)
55          .runInJobProcess();
56       }
```

Listing 22.1. MinVerCovClu3.java (part 1)

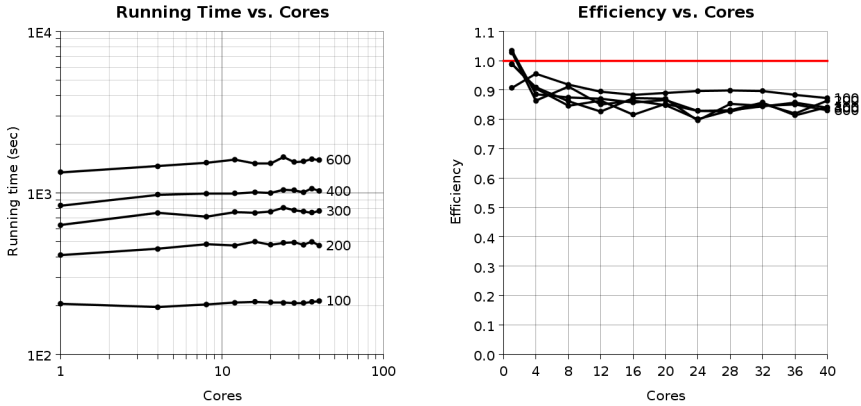


Figure 22.2. MinVerCovClu3 weak scaling performance metrics

found approximate solutions with about 3 to 27 minutes of computation, depending on the graph's size. How close were the approximate solutions to the exact solutions? There's no way to know without doing an exhaustive search. However, in general, the size of the minimum vertex cover the heuristic search finds does go down a bit as the number of iterations increases. Here are the sizes of the solutions the MinVerCovClu3 program found for each graph after doing 25 million iterations and after doing one billion iterations:

Graph size	*Cover size, 25 million iterations*	*Cover size, one billion iterations*
100	91	91
200	191	191
300	291	290
400	392	390
600	592	590

Points to Remember

- In a cluster parallel program that must read an input file, let the job main program read the file, wrap the file's contents in a tuple, and put the tuple into tuple space; and let the worker tasks read the tuple to obtain the input.
- However, reading an input file in this fashion makes sense only if the input file is small. For large input files, such as those for *big data* programs, use the map-reduce paradigm covered later.

```
57    // Print an error message and exit.
58    private static void usage
59       (String msg)
60       {
61       System.err.printf ("MinVerCovClu3: %s%n", msg);
62       usage();
63       }
64
65    // Print a usage message and exit.
66    private static void usage()
67       {
68       System.err.println ("Usage: java pj2 [workers=<K>] " +
69          "edu.rit.pj2.example.MinVerCovClu3 <file> <seed> <N>");
70       System.err.println ("<K> = Number of worker tasks "+
71          "(default 1)");
72       System.err.println ("<file> = Graph file");
73       System.err.println ("<seed> = Random seed");
74       System.err.println ("<N> = Number of trials");
75       throw new IllegalArgumentException();
76       }
77
78    // Worker task class.
79    private static class WorkerTask
80       extends Task
81       {
82       // Random seed.
83       long seed;
84
85       // Number of vertices.
86       int V;
87
88       // The graph's adjacency matrix. adjacent[i] is the set of
89       // vertices adjacent to vertex i.
90       BitSet[] adjacent;
91
92       // Minimum vertex cover.
93       BitSetVbl minCover;
94
95       // Worker task main program.
96       public void main
97          (String[] args)
98          throws Exception
99          {
100         // Parse command line arguments.
101         seed = Long.parseLong (args[0]);
102         V = Integer.parseInt (args[1]);
103
104         // Read adjacency matrix from tuple space.
105         adjacent = readTuple (new ObjectArrayTuple<BitSet>())
106            .item;
107
108         // Check randomly chosen candidate covers.
109         minCover = new BitSetVbl.MinSize (V) .add (0, V);
110         workerFor() .exec (new LongLoop()
111            {
112            BitSetVbl thrMinCover;
113            BitSet candidate;
114            Random prng;
```

Listing 22.1. MinVerCovClu3.java (part 2)

```
115             RandomSubset rsg;
116             public void start()
117                {
118                thrMinCover = threadLocal (minCover);
119                candidate = new BitSet (V);
120                prng = new Random (seed + 1000*taskRank() + rank());
121                rsg = new RandomSubset (prng, V, true);
122                }
123             public void run (long i)
124                {
125                candidate.clear();
126                rsg.restart();
127                while (! isCover (candidate))
128                   candidate.add (rsg.next());
129                if (candidate.size() < thrMinCover.size())
130                   thrMinCover.copy (candidate);
131                }
132             });
133
134          // Send best candidate cover to reduction task.
135          putTuple (new ObjectTuple<BitSetVbl> (minCover));
136          }
137
138       // Returns true if the given candidate vertex set is a cover.
139       private boolean isCover
140          (BitSet candidate)
141          {
142          boolean covered = true;
143          for (int i = 0; covered && i < V; ++ i)
144             if (! candidate.contains (i))
145                covered = adjacent[i].isSubsetOf (candidate);
146          return covered;
147          }
148       }
149
150    // Reduction task class.
151    private static class ReduceTask
152       extends Task
153       {
154       // Reduction task main program.
155       public void main
156          (String[] args)
157          throws Exception
158          {
159          // Parse command line arguments.
160          int V = Integer.parseInt (args[0]);
161
162          // Reduce all worker task results together.
163          BitSetVbl minCover = new BitSetVbl.MinSize (V) .add (0, V);
164          ObjectTuple<BitSetVbl> template =
165             new ObjectTuple<BitSetVbl>();
166          ObjectTuple<BitSetVbl> taskCover;
167          while ((taskCover = tryToTakeTuple (template)) != null)
168             minCover.reduce (taskCover.item);
```

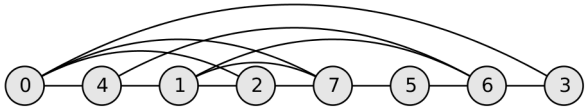
Listing 22.1. MinVerCovClu3.java (part 3)

```
169           // Print final result.
170           System.out.printf ("Cover =");
171           for (int i = 0; i < V; ++ i)
172              if (minCover.contains (i))
173                 System.out.printf (" %d", i);
174           System.out.println();
175           System.out.printf ("Size = %d%n", minCover.size());
176           }
177        }
178     }
```

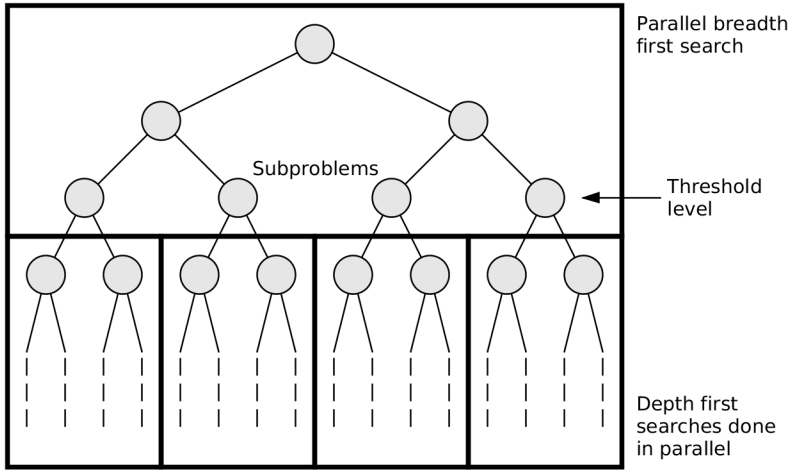
Listing 22.1. MinVerCovClu3.java (part 4)

Chapter 23
Cluster Work Queues

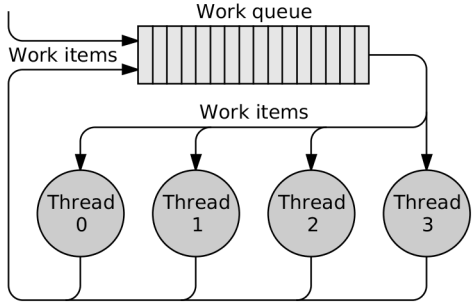
Recall the Hamiltonian cycle problem from Chapter 13. The problem is to decide whether a given graph has a Hamiltonian cycle. A Hamiltonian cycle is a path that visits every vertex in the graph exactly once and returns to the starting point, like the cycle 0, 4, 1, 2, 7, 5, 6, 3 in this graph:



In Chapter 13 I developed a multicore parallel program to solve the Hamiltonian cycle problem by doing an exhaustive search of all paths in the graph. The program was designed to do a parallel breadth first search down to a certain threshold level in the search tree of all paths, resulting in some number of subproblems (partial paths). The program then did multiple depth first searches in parallel, each depth first search working on one subproblem in a single thread.



I implemented this design using the parallel work queue pattern. There was a work queue of work items. Each parallel team thread repeatedly took a work item out of the queue and processed the item, possibly adding further items to the queue, until there were no more work items.



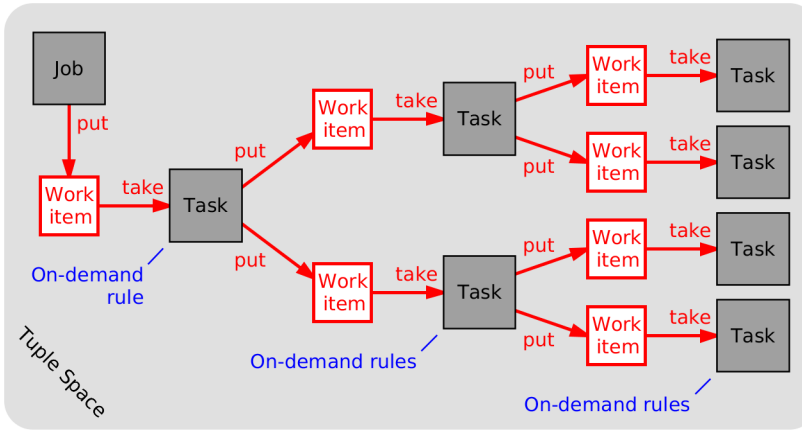
```
1  package edu.rit.pj2.example;
2  import edu.rit.io.InStream;
3  import edu.rit.io.OutStream;
4  import edu.rit.pj2.Job;
5  import edu.rit.pj2.Task;
6  import edu.rit.pj2.Tuple;
7  import edu.rit.pj2.TupleListener;
8  import edu.rit.pj2.tuple.EmptyTuple;
9  import edu.rit.pj2.tuple.ObjectArrayTuple;
10 import edu.rit.util.BitSet;
11 import java.io.File;
12 import java.io.IOException;
13 import java.util.Formatter;
14 import java.util.Scanner;
15 public class HamCycClu
16    extends Job
17    {
18    // Job main program.
19    public void main
20       (String[] args)
21       throws Exception
22       {
23       // Parse command line arguments.
24       if (args.length != 2) usage();
25       File file = new File (args[0]);
26       int threshold = Integer.parseInt (args[1]);
27
28       // Read input file, set up adjacency matrix.
29       Scanner s = new Scanner (file);
30       int V = s.nextInt();
31       int E = s.nextInt();
32       if (V < 1) usage ("V must be >= 1");
33       BitSet[] adjacent = new BitSet [V];
34       for (int i = 0; i < V; ++ i)
35          adjacent[i] = new BitSet (V);
36       for (int i = 0; i < E; ++ i)
37          {
38          int a = s.nextInt();
39          int b = s.nextInt();
40          adjacent[a].add (b);
41          adjacent[b].add (a);
42          }
43       s.close();
44       putTuple (new ObjectArrayTuple<BitSet> (adjacent));
45
46       // Set up first work item.
47       putTuple (new StateTuple (V));
48
49       // Perform a search task each time there's a new work item.
50       rule() .whenMatch (new StateTuple()) .task (SearchTask.class)
51          .args (""+threshold);
52
53       // Task to print negative result.
54       rule() .atFinish() .task (ResultTask.class)
55          .runInJobProcess();
56       }
57
```

Listing 23.1. HamCycClu.java (part 1)

Now I want to make this a cluster parallel program, so I can have even more cores searching for a Hamiltonian cycle in parallel. I need a version of the parallel work queue pattern that runs on a cluster of multiple nodes. The threads processing the work items will be in tasks running on the nodes. However, in a distributed memory setting, the tasks cannot have a common work queue in shared memory. Instead, the program must send work items from task to task using intertask communication, that is, via tuple space.

In a cluster setting, the parallel work queue pattern consists of two parts:

- Tuple space itself acts as the work queue. Each work item is packaged into a tuple and is put into tuple space.
- Whenever a work item tuple is put into tuple space, an *on-demand rule* in the job spawns a new task, takes the tuple out of tuple space, and passes the tuple to the new task.



In the Hamiltonian cycle cluster parallel program, the tuples (work items) are partial paths, and each task performs either a breadth first or a depth first search on the partial path, depending on the search level. The breadth first search could put additional work items into tuple space, thereby spawning further tasks. The threshold level limits the number of tasks; once the threshold level is reached, a task does a depth first search rather than spawning more tuples and tasks.

Turning to the code for the cluster parallel HamCycClu program (Listing 23.1), the job main program begins by reading the input graph file, packaging the graph's adjacency matrix into a tuple, and putting the tuple into tuple space (lines 29–44). The graph tuple is not a work item; rather, it is a "global variable" that every task will read to obtain the graph. I did the same thing in the cluster parallel minimum vertex cover program in Chapter 22.

The job puts the first work item into tuple space (line 47). This is an instance of class StateTuple, defined further on, containing a partial path with just vertex 0.

```
58    // Tuple class for a search state.
59    private static class StateTuple
60       extends Tuple
61       {
62       public int[] path;
63       public int level;
64
65       public StateTuple()
66          {
67          }
68
69       public StateTuple
70          (int V)
71          {
72          this.path = new int [V];
73          for (int i = 0; i < V; ++ i)
74             this.path[i] = i;
75          this.level = 0;
76          }
77
78       public StateTuple
79          (int[] path,
80           int level)
81          {
82          this.path = (int[]) path.clone();
83          this.level = level;
84          }
85
86       public void writeOut
87          (OutStream out)
88          throws IOException
89          {
90          out.writeIntArray (path);
91          out.writeInt (level);
92          }
93
94       public void readIn
95          (InStream in)
96          throws IOException
97          {
98          path = in.readIntArray();
99          level = in.readInt();
100         }
101      }
102
103   // Search task.
104   private static class SearchTask
105      extends Task
106      {
107      // Command line arguments.
108      int threshold;
109
110      // Number of vertices.
111      int V;
112
113      // The graph's adjacency matrix. adjacent[i] is the set of
114      // vertices adjacent to vertex i.
115      BitSet[] adjacent;
```

Listing 23.1. HamCycClu.java (part 2)

The job sets up two rules. The first is an on-demand rule (lines 50–51) that is fired whenever a StateTuple appears in tuple space. When fired, the rule spawns a new instance of class SearchTask, defined further on. The search task will run on some available node of the cluster; if all nodes are busy, the search task will sit in the Tracker's queue until a node goes idle. The search task will obtain the threshold level as a command line argument (line 51). The search task will obtain the partial path to process from the matching state tuple, which is automatically taken out of tuple space when the rule fires. The search task will obtain the graph's adjacency matrix by reading the graph tuple. The second rule is a finish rule (lines 54–55) that spawns a new instance of class ResultTask once all the search tasks have finished.

When the job's `main()` method returns, the job commences. Because the `main()` method already put a state tuple into tuple space, and because the `main()` method set up an on-demand rule that fires whenever a state tuple is present in tuple space, the on-demand rule fires immediately and spawns the first search task to process the initial state tuple, namely the one containing a partial path with just vertex 0. The first search task puts additional state tuples into tuple space, causing further search tasks to be spawned.

The StateTuple class (lines 59–101) encapsulates a work item, namely the search state (partial path) represented by the `path` and `level` fields. Elements `path[0]` through `path[level]` contain the vertices of the path, in order. Elements `path[level+1]` through `path[V-1]` contain the vertices not in the path, in no particular order. The StateTuple class includes the obligatory no-argument constructor, `writeOut()` method, and `readIn()` method. It also includes a constructor that the job uses to create the initial work item and a constructor that the search tasks use to create subsequent work items.

The SearchTask class (line 104) is similar to the multicore parallel HamCycSmp class in Chapter 13. It begins with several global variable declarations as well as the inner State class that does the actual breadth first and depth first searches. The State class is also similar to the one in Chapter 13, except that the breadth first search adds a new subproblem to the work queue by putting a new state tuple into tuple space (line 158).

The SearchTask main program (line 222) obtains the threshold level from the command line (line 227) and reads the graph's adjacency matrix from tuple space (lines 230–232). The task adds a tuple listener for a stop tuple (an instance of class EmptyTuple), whose presence in tuple space signals that a Hamiltonian cycle has been found and the task should stop processing (lines 235–242)—like the cluster parallel bitcoin mining program in Chapter 17. The task gets the matching StateTuple that caused the task to be spawned, uses the partial path from the matching tuple to initialize a state object, and does a breadth first or depth first search from that state (lines 246–248). If the search found a Hamiltonian cycle, the task prints it and puts a stop tuple into

```
116         // For early loop exit.
117         volatile boolean found;
118
119         // Class for the search state.
120         private class State
121            {
122            // Vertices in the path.
123            private int[] path;
124
125            // Search level = index of last vertex in the path.
126            private int level;
127
128            // Construct a new search state object.
129            public State
130               (int[] path,
131                int level)
132               {
133               this.path = path;
134               this.level = level;
135               }
136
137            // Search the graph from this state.
138            public State search()
139               throws Exception
140               {
141               if (level < threshold)
142                  return bfs();
143               else
144                  return dfs();
145               }
146
147            // Do a breadth first search of the graph from this state.
148            private State bfs()
149               throws Exception
150               {
151               // Try extending the path to each vertex adjacent to the
152               // current vertex.
153               for (int i = level + 1; i < V; ++ i)
154                  if (adjacent (i))
155                     {
156                     ++ level;
157                     swap (level, i);
158                     putTuple (new StateTuple (path, level));
159                     -- level;
160                     }
161               return null;
162               }
163
164            // Do a depth first search of the graph from this state.
165            private State dfs()
166               {
167               // Base case: Check if there is an edge from the last
168               // vertex to the first vertex.
169               if (level == V - 1)
170                  {
171                  if (adjacent (0))
172                     return this;
173                  }
```

Listing 23.1. HamCycClu.java (part 3)

tuple space to inform the other tasks (lines 251–255).

The final piece of the program is the result task (lines 266–278), which runs when all the search tasks have finished. If at this point a stop tuple exists in tuple space, then a solution was found, and the result task prints nothing. Otherwise, a solution was not found, and the result task prints a message to that effect. That's all the result task does.

As previously noted, whenever a new subproblem is generated and a state tuple is put into tuple space, a new search task is spawned. Spawning a task is a fairly heavyweight operation: communication has to happen between the job process and the Tracker and between the Tracker and a Launcher on some node; the Launcher has to fire up a JVM to run a backend process on that node; and the backend process has to communicate with the job process to determine which task to run—all this before the actual task main program starts executing. Consequently, a cluster parallel program should not run a large number of tasks, or else the task startup overhead will degrade the program's performance. The cluster parallel Hamiltonian cycle program avoids this by ceasing to spawn new tasks once the search reaches the threshold level. The threshold level should be specified so there are enough tasks to balance the load, but not so many tasks that the program's performance degrades.

I ran the HamCycClu program on the `tardis` cluster to find Hamiltonian cycles in several different 30-vertex graphs, which had from 81 to 91 edges. For each of the graphs, a sequential version of the program took over 10 minutes. Running on the 40 cores of the 10 nodes of the cluster, the HamCycClu program took just a few seconds for each graph. Here are the running times and speedups I measured:

		Running Time (msec)		
V	*E*	*Sequential*	*HamCycClu*	*Speedup*
30	81	691671	1029	672
30	82	604620	1041	581
30	83	619187	1424	435
30	88	641281	1452	442
30	91	692783	3480	199

The threshold level was set at 2 in the HamCycClu program runs. Setting the threshold level higher caused the running time to go up—the higher the threshold level, the more subproblems were generated by the breadth first search phase, the more tasks were spawned, and the longer the program took to run.

Like the multicore parallel HamCycSmp program in Chapter 13, the cluster parallel HamCycClu program experienced ridiculously large speedups, far beyond the number of cores in the cluster—and for the same reason. On these graphs, with multiple cores in the cluster nodes processing multiple

```
174            // Recursive case: Try extending the path to each vertex
175            // adjacent to the current vertex.
176            else
177               {
178               for (int i = level + 1; i < V && ! found; ++ i)
179                  if (adjacent (i))
180                     {
181                     ++ level;
182                     swap (level, i);
183                     if (dfs() != null)
184                        return this;
185                     -- level;
186                     }
187               }
188
189            return null;
190            }
191
192         // Determine if the given path element is adjacent to the
193         // current path element.
194         private boolean adjacent
195            (int a)
196            {
197            return adjacent[path[level]].contains (path[a]);
198            }
199
200         // Swap the given path elements.
201         private void swap
202            (int a,
203             int b)
204            {
205            int t = path[a];
206            path[a] = path[b];
207            path[b] = t;
208            }
209
210         // Returns a string version of this search state object.
211         public String toString()
212            {
213            StringBuilder b = new StringBuilder();
214            Formatter f = new Formatter (b);
215            for (int i = 0; i <= level; ++ i)
216               f.format (" %d", path[i]);
217            return b.toString();
218            }
219         }
220
221      // Search task main program.
222      public void main
223         (String[] args)
224         throws Exception
225         {
226         // Parse command line arguments.
227         threshold = Integer.parseInt (args[0]);
228
229         // Get adjacency matrix.
230         adjacent = readTuple (new ObjectArrayTuple<BitSet>())
231            .item;
```

Listing 23.1. HamCycClu.java (part 4)

subproblems in parallel, it so happened that at least one of the cores went to work on a subproblem where a Hamiltonian cycle was found right away. Consequently, the program stopped early, and the running time was small.

Under the Hood

This is the first cluster parallel program we've studied that used an on-demand rule in the job. The other kinds of rules only fire once: a start rule fires once when the job commences, a finish rule fires once at the end of the job after all the other tasks have finished. In contrast, an on-demand rule can fire not at all, once, or more than once, depending on whether matching tuples appear in tuple space as the job progresses.

As we have seen, the job process running in the cluster's frontend node is the central repository for the tuples in tuple space. The job process also has a list of all the job's rules. In the case of an on-demand rule, the job remembers the template associated with the rule. Whenever a task puts a tuple into tuple space, a message goes from the task to the job via the cluster's backend network. Processing the incoming message, the job tries to match the tuple with the on-demand rules' templates. If it finds a match, the job communicates with the Tracker to launch a new instance of the task specified in the on-demand rule. Once the new task has launched, the new task contacts the job, whereupon the job takes the matching tuple out of tuple space and sends it to the task.

Points to Remember

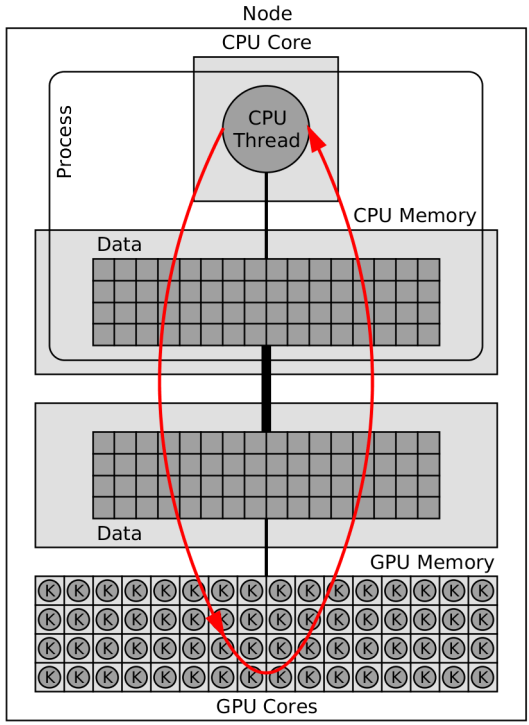
- Reiterating Chapter 13, some NP problems can be solved by traversing a *search tree* consisting of all possible solutions.
- A search tree can be traversed by a *breadth first search* or a *depth first search.*
- A parallel program traversing a search tree should do a parallel breadth first search down to a certain threshold level, then should do multiple depth first searches in parallel thereafter.
- Use the *parallel work queue* pattern to do a parallel loop when the number of iterations is not known before the loop starts.
- In a cluster parallel program, tuple space itself acts as the work queue, and the work items are encapsulated in tuples put into tuple space.
- To process the work items, include an on-demand rule in the job.
- The on-demand rule's template is specified to match a work item tuple.
- The on-demand rule is specified with a task to process the work item.
- Whenever a work item tuple appears in tuple space, the on-demand rule fires and spawns a new task; this causes the program to "loop" through the work items.

```
232            V = adjacent.length;
233
234            // Early loop exit when any task finds a Hamiltonian cycle.
235            addTupleListener (new TupleListener<EmptyTuple>
236               (new EmptyTuple())
237               {
238               public void run (EmptyTuple tuple)
239                  {
240                  found = true;
241                  }
242               });
243
244            // Search for a Hamiltonian cycle starting from the state
245            // that triggered this search task.
246            StateTuple tuple = (StateTuple) getMatchingTuple (0);
247            State hamCycle = new State (tuple.path, tuple.level)
248               .search();
249
250            // Report positive result.
251            if (hamCycle != null)
252               {
253               putTuple (new EmptyTuple());
254               System.out.printf ("Hamiltonian cycle =%s%n", hamCycle);
255               }
256            }
257
258         // The search task requires one core.
259         protected static int coresRequired()
260            {
261            return 1;
262            }
263         }
264
265      // Result task.
266      private static class ResultTask
267         extends Task
268         {
269         // Task main program.
270         public void main
271            (String[] args)
272            throws Exception
273            {
274            // Report negative result.
275            if (tryToReadTuple (new EmptyTuple()) == null)
276               System.out.printf ("No Hamiltonian cycle%n");
277            }
278         }
279
280      // Print an error message and exit.
281      private static void usage
282         (String msg)
283         {
284         System.err.printf ("HamCycClu: %s%n", msg);
285         usage();
286         }
287      }
```

Listing 23.1. HamCycClu.java (part 5)

- The task obtains the work item tuple by calling the `getMatchingTuple()` method.
- The task adds a work item to the work queue by putting a work item tuple into tuple space.
- The program incurs overhead each time a new task is spawned. Design the program so there will be enough tasks to balance the load, but not so many tasks that the program's performance degrades.

PART IV
GPU ACCELERATION



Parallel program running on a graphics processing unit accelerator

Chapter 24
GPU Massively Parallel

We now turn to graphics processing unit (GPU) accelerated parallel programming. In contrast to an unaccelerated node that might have a few or a few dozen CPU cores, a GPU accelerator might have hundreds or thousands of GPU cores. A GPU accelerated node can perform computations potentially many times faster than an unaccelerated multicore node.

To ease into GPU programming, I'll use a simple example: computing the *outer product* of two vectors. The inputs are two n-element vectors (arrays) **A** and **B**. The output is an $n \times n$-element matrix **C**, computed as follows:

$$C_{ij} = A_i \times B_j, \ 0 \le i \le n-1, \ 0 \le j \le n-1 \tag{24.1}$$

Here is an example of the outer product of two 10-element vectors:

	B	3	6	9	12	15	18	21	24	27	30
A											
2		6	12	18	24	30	36	42	48	54	60
4		12	24	36	48	60	72	84	96	108	120
6		18	36	54	72	90	108	126	144	162	180
8		24	48	72	96	120	144	168	192	216	240
10		30	60	90	120	150	180	210	240	270	300
12		36	72	108	144	180	216	252	288	324	360
14		42	84	126	168	210	252	294	336	378	420
16		48	96	144	192	240	288	336	384	432	480
18		54	108	162	216	270	324	378	432	486	540
20		60	120	180	240	300	360	420	480	540	600

C

If I had a computer with 100 cores, I could compute the above matrix's 100 elements all at once, each on its own separate core, in a massively parallel fashion, and get a theoretical speedup of 100 over a single core. Or consider computing the outer product of two 1000-element vectors. If I had a computer with one million cores, again I could compute all of the matrix's elements at the same time in parallel, one element on each core.

A real compute node would be unlikely to have one million cores. But Nvidia Corporation, maker of GPU-based graphics cards, has developed a *programming abstraction* that makes a GPU accelerator look as though it has a nearly unlimited number of *virtual* cores. Nvidia calls this programming abstraction the *Compute Unified Device Architecture (CUDA).* The Parallel Java 2 Library supports writing GPU accelerated parallel programs that run on CUDA-capable graphics cards.

In this and the next few chapters, I'll be using CUDA along with Parallel Java 2 to teach GPU parallel programming. CUDA's capabilities are vast, and I'm not going to try to cover them all. I will hit CUDA's basic features. You can pick up the rest from the CUDA documentation.

Figure 24.1 shows the hardware architecture of a typical CUDA GPU. This particular one is an Nvidia Tesla C2075, one of Nvidia's older-genera-

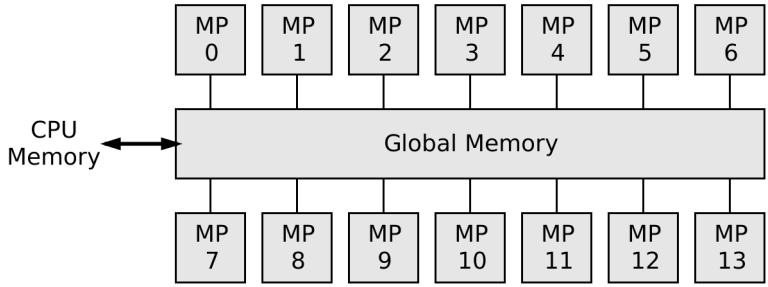


Figure 24.1. CUDA GPU hardware architecture

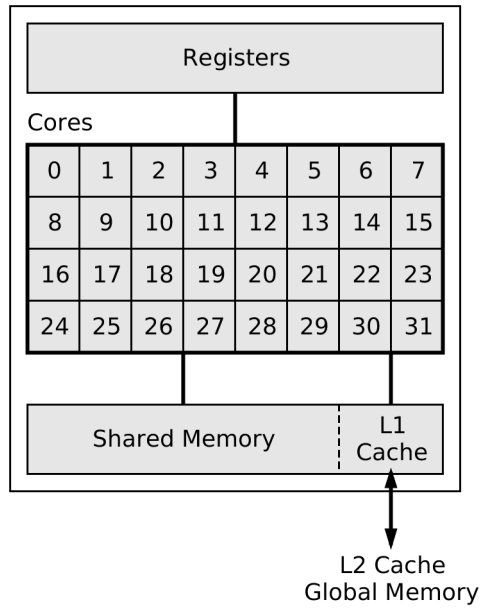


Figure 24.2. CUDA multiprocessor architecture

tion GPU cards designed for high performance parallel computing. The `kraken` machine at RIT is a multicore node with 80 CPU cores and four Tesla C2075 accelerators. I'll be using `kraken` to measure the performance of the GPU parallel programs in this book.

The Tesla C2075 has 14 *multiprocessors (MPs)*, each connected to a 6-gigabyte *global memory*. (Other GPUs have different numbers of multiprocessors and different amounts of global memory.) The global memory is connected via a high-speed bus to the CPU memory of the machine hosting the GPU; this lets the host CPU transfer data to and from the GPU global mem-

ory. Each multiprocessor can read and write data anywhere in the common global memory. The global memory is built from the same, comparatively slow, DRAM chips as are used in the CPU's main memory.

Figure 24.2 is an expanded view of one Tesla C2075 multiprocessor. It has 32 *cores;* a *register bank* with 32,768 high-speed 32-bit registers; a *shared memory;* and an *L1 cache.* The shared memory and the L1 cache together total 64 kilobytes. These can be configured either as 48 kilobytes of shared memory and 16 kilobytes of L1 cache, or as 16 kilobytes of shared memory and 48 kilobytes of L1 cache. (Other multiprocessors have different numbers of cores and registers, and different amounts of shared memory and L1 cache.) The shared memory and L1 cache use fast RAM circuitry, like the L1 cache in a CPU. The L1 cache connects to a somewhat slower L2 cache, which in turn connects to the very slow global memory. All the cores in the multiprocessor can access data located in their own multiprocessor's fast shared memory, but cannot access data located in other multiprocessors' shared memories. All the cores in all the multiprocessors can access data located in the slow global memory. Global memory data is cached in L2 and L1, thus reducing the time needed to access the data. However, the GPU's L2 and L1 caches are typically much smaller than a CPU's caches. GPU programs rely on additional techniques to reduce the effective access time for global memory data.

Summing up, the Tesla C2075 GPU has 14 multiprocessors each with 32 cores, for a total of 448 cores, as well as a 6-gigabyte memory. Other GPUs have different quantities of computational resources. The newer Nvidia Tesla K40 GPU accelerator, for example, has 2,880 cores and 12 gigabytes of memory.

To write GPU programs, however, you usually don't need to deal with these hardware details. Instead, you work with the CUDA programming model, which presents a high level abstraction of the GPU hardware. The abstraction consists of two key components: the computational grid, and the kernel function.

The *computational grid,* or just *grid,* is an abstraction of the GPU's multiprocessors and cores. Consider the outer product computation. The output is an $n \times n$ matrix. I want to compute the matrix elements, conceptually, all at once in parallel. To do so, I need an $n \times n$ matrix, or grid, of *threads*—one thread for each element. Each thread executes the same *kernel function;* the kernel function computes one output matrix element. The whole assemblage —a grid of threads, each thread executing the kernel function—is called the *computational kernel,* or just *kernel.*

Figure 24.3 depicts a two-dimensional grid of threads, suitable for computing the elements of a matrix. The grid is partitioned into *thread blocks,* or just *blocks.* The blocks are located within a two-dimensional coordinate system. In this example, the grid dimensions are (6, 4); that is, 6 blocks in the x

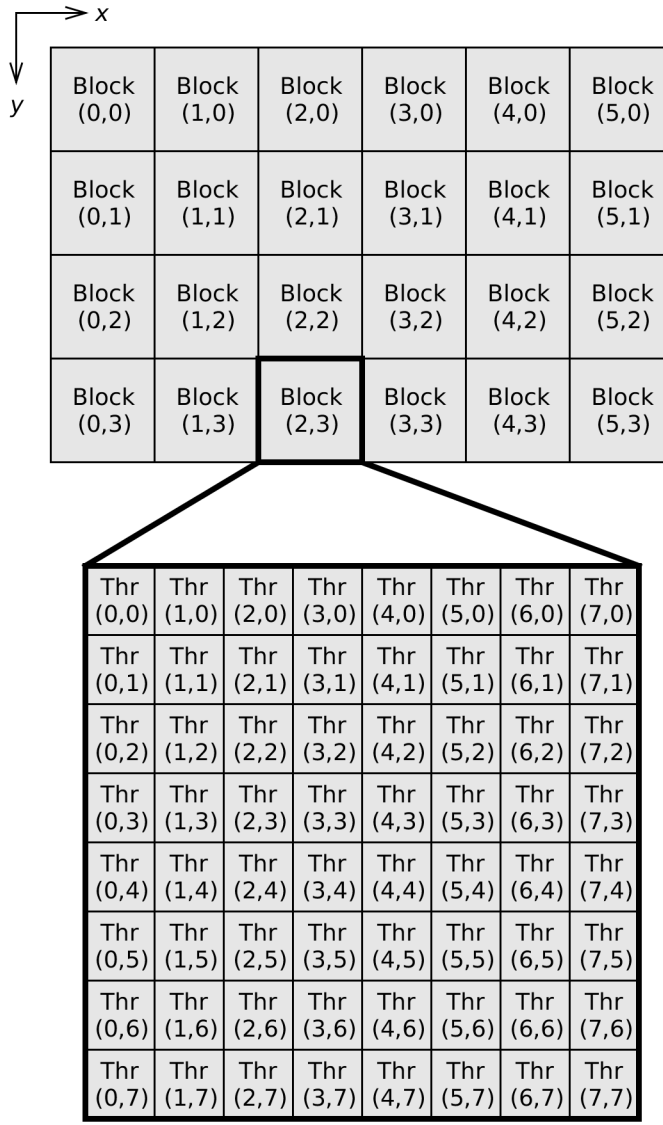


Figure 24.3. A two-dimensional grid of two-dimensional blocks

direction by 4 blocks in the *y* direction. Each block has an *x* coordinate in the range 0–5 and a *y* coordinate in the range 0–3.

Each block in turn consists of a number of threads. The threads are located within a two-dimensional coordinate system inside the block. In this example, the block dimensions are (8, 8); that is, 8 threads in the *x* direction by 8 threads in the *y* direction. Each thread has an *x* coordinate in the range 0–7 and a *y* coordinate in the range 0–7. Each thread in the entire grid is uniquely identified by its block coordinates in the grid and its thread coordinates in the block.

CUDA lets you configure a kernel with a one-dimensional, a two-dimensional, or a three-dimensional grid. Each dimension of the grid can be anything up to 65,535 blocks. CUDA also lets you configure a kernel with a one-dimensional, a two-dimensional, or a three-dimensional block. Each dimension of the block can be anything, as long as the total number of threads in the block is 1,024 or less (for a Tesla C2075). Thus, the maximum grid size is 65,535×65,535×65,535 blocks times 1,024 threads per block, for a total of 288 quadrillion threads.

For the outer product GPU program, I'll use a two-dimensional grid of two-dimensional blocks, because such a grid naturally matches the computation's structure. Other programs in later chapters will use one-dimensional grids and blocks. I myself have not had occasion to use a three-dimensional grid; but I can see how a scientist computing a 3-D problem, like molecular modeling or fluid dynamics, might want to.

The blocks in the grid are abstractions of the multiprocessors in the GPU, and the threads in the block are abstractions of the cores in the multiprocessor. As such, the threads in one block have access to one multiprocessor's fast shared memory. CUDA lets you declare program variables in shared memory; such variables are shared by all the threads in one block, but such variables are not accessible to threads in other blocks. CUDA also lets you declare program variables in global memory; such variables are shared by all the threads in all the blocks.

How is it possible to execute a grid with an enormous number of blocks, each block potentially containing over a thousand threads, on a GPU like the Tesla C2075 with only 14 multiprocessors of 32 cores each? The same way that dozens or hundreds of threads can execute on a CPU with only a few cores. The operating system—in particular, the GPU card's CUDA driver installed in the operating system—takes care of scheduling the grid's blocks to execute on the GPU's multiprocessors and scheduling each block's threads to execute on the multiprocessor's cores. You configure a kernel with the desired grid and block dimensions and tell the kernel to execute the desired kernel function; CUDA does the rest, ensuring that the kernel function is executed by each thread in the grid.

This brings us to the other part of the kernel, namely the kernel function.

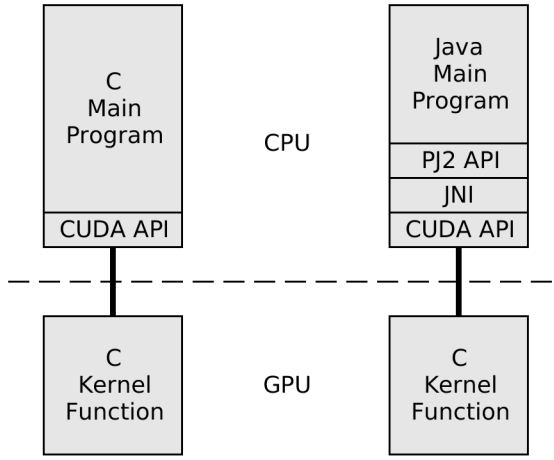


Figure 24.4. Main program and kernel function, in C and in Java

A CUDA program consists of two parts: a main program that runs on the host CPU, and a kernel function that runs in each thread on the GPU (Figure 24.4). If you're working purely with CUDA, you write both the main program and the kernel function in C, C++, or Fortran using the CUDA API, and you compile both pieces using the CUDA compiler, `nvcc`.

When doing GPU programming with the Parallel Java 2 Library, you write the main program in Java, and you write just the kernel function in C using CUDA. (I won't be using C++ or Fortran in this book.) You compile the kernel function using the CUDA compiler, `nvcc`, and you compile the rest using the standard Java compiler, `javac`. I'd much prefer to write the whole thing in Java; but at this time, I don't know of a way to write CUDA kernel functions in Java.

Parallel Java 2 provides Java classes for working with CUDA GPUs. Under the hood, these classes use the *Java Native Interface (JNI)* to invoke routines in the non-Java CUDA API, such as routines to configure a grid and execute a kernel function.

Combining a Java main program with a CUDA kernel in this way lets you take advantage of all of Parallel Java 2's features for single-core, multicore, and cluster parallel programming, and augment them with GPU acceleration. For example, you can easily write a parallel program that runs on multiple CPU cores, with each CPU core making use of its own GPU accelerator. This lets the program run on multiple GPUs at once, like the `kraken` machine's four Tesla C2075 GPUs. (We will see an example of just such a program later.)

Now we're ready to start designing the GPU parallel vector outer product program. A typical GPU program follows this sequence of steps, which I call the *computational arc:*

- Running on the CPU, the main program sets up variables in the CPU's memory and in the GPU's memory to hold the program's input and output data. Each variable on the CPU is said to *mirror* its counterpart on the GPU.
- The main program initializes the input variables in the CPU's memory with the input data.
- The main program copies the input variables from the CPU's memory to the GPU's memory.
- The main program configures a kernel with certain grid dimensions and block dimensions, and with a certain kernel function.
- The main program *launches* the kernel and waits for the kernel to finish.
- On the GPU, every thread in the grid executes the kernel function, and stores the computed output data in the output variables in the GPU's memory.
- Once the kernel finishes, back on the CPU, the main program copies the output variables from the GPU's memory to the CPU's memory.
- The main program prints the output data, stores the output data in a file, or does whatever else is necessary with the output data.

Figure 24.5 shows the vector outer product program's input and output variables, mirrored in the CPU and the GPU: the input vectors **A** and **B**, and the output matrix **C**. The arrows show the program's data flow as the computational arc progresses. Input data goes into **A** and **B** on the CPU; **A** and **B** are copied from the CPU to the GPU; the outer product is computed and stored into **C** on the GPU; **C** is copied from the GPU to the CPU; output data comes from **C** on the CPU.

As previously mentioned, the vector outer product program's kernel is configured with a two-dimensional grid of two-dimensional blocks. There is one thread in the grid for each element in the outer product matrix **C**. The kernel function computes one and only one matrix element, a different matrix element in each thread.

Listing 24.1 is the C source code for the kernel function, stored in the file OuterProductGpu.cu. (".cu" is the standard filename extension for a CUDA C source file.) The kernel function's name is `outerProduct` (line 7). Every kernel function must be declared this same way:

- `extern "C"` — Needed by Parallel Java 2 to access the kernel function.
- `__global__` — Needed by `nvcc` to compile the kernel function for the GPU. (That special CUDA keyword is "underscore underscore global underscore underscore".)

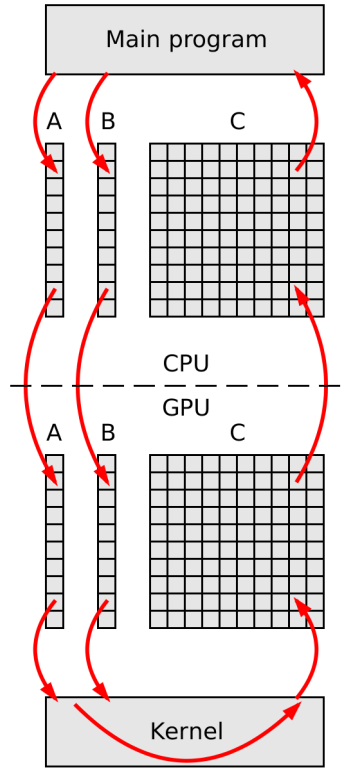


Figure 24.5. Vector outer product program variables

```
// Device kernel to compute the outer product matrix of two vectors.
// Called with a two-dimensional grid of two-dimensional blocks.
// a  First vector (input).
// b  Second vector (input).
// c  Outer product matrix (output).
// N  Vector length.
extern "C" __global__ void outerProduct
   (double *a,
    double *b,
    double **c,
    int N)
   {
   int row = blockIdx.y*blockDim.y + threadIdx.y;
   int col = blockIdx.x*blockDim.x + threadIdx.x;
   if (row < N && col < N)
      c[row][col] = a[row]*b[col];
   }
```

Listing 24.1. OuterProductGpu.cu

- `void` — A kernel function must not have a return value.

A kernel function can, and typically does, have arguments (lines 8–11). The vector outer product kernel function's arguments are pointers to the input `a` and `b` arrays, of type `double*` (this is C, remember); a pointer to the output `c` matrix, of type `double**`; and the length of the input vectors, `n`. The pointers and the length must be supplied as arguments because the arrays and the matrix will be dynamically allocated at run time based on user input.

The kernel function's sole purpose is to compute one output matrix element at a certain row and column (line 16). Each thread executing the kernel function must compute a different row and column. To do so, the thread must be able to determine its position within the grid.

A thread determines its position in the grid by querying four special pseudo-variables:

- `gridDim` gives the grid's dimensions: `gridDim.x` the *x* dimension, `gridDim.y` the *y* dimension, `gridDim.z` the *z* dimension.
- `blockDim` gives the dimensions of each block in the grid: `blockDim.x` the *x* dimension, `blockDim.y` the *y* dimension, `blockDim.z` the *z* dimension.
- `blockIdx` gives the indexes of the currently executing thread's block within the grid: `blockIdx.x` the *x* index, `blockIdx.y` the *y* index, `blockIdx.z` the *z* index.
- `threadIdx` gives the indexes of the currently executing thread within its block: `threadIdx.x` the *x* index, `threadIdx.y` the *y* index, `threadIdx.z` the *z* index.

For a one-dimensional grid or block, only the *x* dimensions and indexes are used. For a two-dimensional grid or block, only the *x* and *y* dimensions and indexes are used. For a three-dimensional grid or block, the *x*, *y*, and *z* dimensions and indexes are used.

The expression on line 13 computes the thread's *y* index (row) within the whole grid. For example, consider grid dimensions (6, 4), block dimensions (8, 8), and consider the thread at block index (2, 3), thread index (5, 4), as shown in Figure 24.3. The thread's row is

`blockIdx.y*blockDim.y + threadIdx.y` = 3×8 + 4 = 28.

Likewise, the expression on line 14 computes the thread's *x* index (column) within the whole grid. The thread's column is

`blockIdx.x*blockDim.x + threadIdx.x` = 2×8 + 5 = 21.

This particular thread computes the output matrix element at row 28, column 21, as you can verify by counting the rows and columns in Figure 24.3. (Remember that the row and column indexes start at 0.) Other threads compute

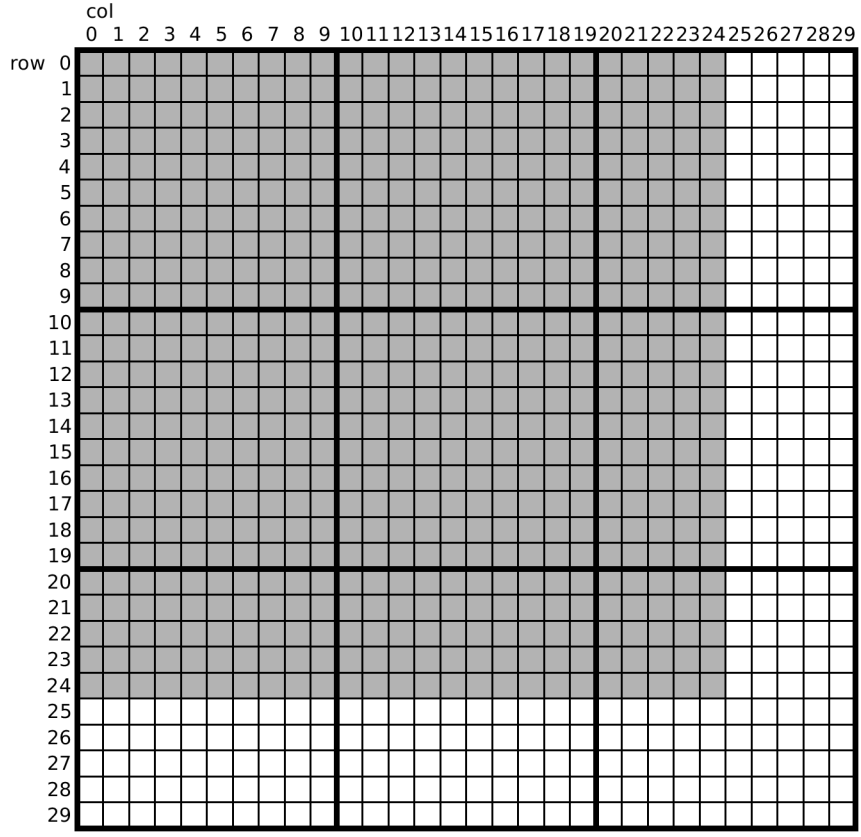


Figure 24.6. 3×3 grid of 10×10 blocks computing a 25×25 matrix

different rows and/or columns, as determined by their own block indexes and thread indexes.

Before proceeding to compute the matrix element, however, the kernel function has to check that the row index and the column index are in bounds (line 15). *This is critical*, because while all the blocks in the grid have the same dimensions, the input vector length *n* might not be a whole multiple of the blocks' *x* or *y* dimension. In that case, the grid will have to have extra blocks to cover the leftover matrix elements. For example, consider a 25×25-element matrix being computed by a grid of 10×10 blocks (Figure 24.6). The grid needs to have 3×3 blocks to cover all the matrix elements; but some of the threads do not have a corresponding matrix element. The test on line 15 ensures that such threads do not try to read or write nonexistent elements.

The kernel function is compiled with this command:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \
  --ptxas-options="-v" -o OuterProductGpu.cubin \
  OuterProductGpu.cu
```

This command tells the CUDA compiler to compile the source file OuterProductGpu.cu and to generate a *CUDA module file* named OuterProductGpu.cubin. (For a more detailed explanation of the compilation process, see the "Under the Hood" section below.) The module file ends up in the same directory as the .class files produced by the Java compiler.

Having written the kernel function in C, we turn to the other part of the GPU parallel vector outer product program, namely the main program written in Java, class OuterProductGpu (Listing 24.2). The main program class is typically a Task, like other Parallel Java 2 programs (line 11).

Parallel Java 2 makes the kernel function appear to be a regular Java method that you can call. This *kernel method* is declared in a *kernel interface* (lines 14–22). The name of the kernel interface can be anything; I called it OuterProductKernel. The kernel interface must extend interface edu.rit.gpu.Kernel. The name of the kernel method must be identical to the kernel function, `outerProduct`. The kernel method must be declared to return `void` and must be declared with the same arguments in the same order as the kernel function. The kernel interface must declare one and only one method.

The Java data types of the kernel method's arguments depend on the C data types of the kernel function's arguments. A complete list of the corresponding Java and C argument data types appears in the Javadoc documentation for interface edu.rit.gpu.Kernel. Here, the input arrays `a` and `b` of C type `double*` are declared as Java type edu.rit.gpu.GpuDoubleArray; the output matrix `c` of C type `double**` is declared as Java type edu.rit.gpu.GpuDoubleMatrix; and the argument `N` of C type `int` is declared as Java type `int`.

It is extremely important to declare the Java kernel method properly. If this does not correspond to the C kernel function exactly, the program will fail in strange and unpredictable ways when run. Because the Java compiler knows nothing about CUDA kernel functions, there is unfortunately no way for the Java compiler to catch errors in the kernel method declaration.

To access the GPU, the main program must first obtain a *GPU object* by calling the static `Gpu.gpu()` method (line 40). The `ensureComputeCapability()` method call on line 41 has to do with the way the kernel function was compiled; see the "Under the Hood" section below for further information.

The main program creates an array to hold the input **A** vector by calling the GPU object's `getDoubleArray()` method, passing in the array length (line 45). This method creates the array in the CPU's memory, stores a reference to the array in a GpuDoubleArray object, and returns the GpuDoubleArray. This method also creates the array in the GPU's global memory. Likewise, the main program creates an array to hold the input **B** vector (line 46). To create the output **C** matrix, the main program calls the GPU object's `get-`

```
1  package edu.rit.gpu.example;
2  import edu.rit.gpu.CacheConfig;
3  import edu.rit.gpu.Kernel;
4  import edu.rit.gpu.Gpu;
5  import edu.rit.gpu.GpuDoubleArray;
6  import edu.rit.gpu.GpuDoubleMatrix;
7  import edu.rit.gpu.Module;
8  import edu.rit.pj2.Task;
9  import edu.rit.util.Random;
10 public class OuterProductGpu
11    extends Task
12    {
13    // Kernel function interface.
14    private static interface OuterProductKernel
15       extends Kernel
16       {
17       public void outerProduct
18          (GpuDoubleArray a,
19           GpuDoubleArray b,
20           GpuDoubleMatrix c,
21           int N);
22       }
23
24    // GPU kernel block dimensions = NT x NT threads.
25    private static final int NT = 32;
26
27    // Task main program.
28    public void main
29       (String[] args)
30       throws Exception
31       {
32       long t1 = System.currentTimeMillis();
33
34       // Validate command line arguments.
35       if (args.length != 2) usage();
36       long seed = Long.parseLong (args[0]);
37       int N = Integer.parseInt (args[1]);
38
39       // Initialize GPU.
40       Gpu gpu = Gpu.gpu();
41       gpu.ensureComputeCapability (2, 0);
42
43       // Set up input vectors and output matrix.
44       Random prng = new Random (seed);
45       GpuDoubleArray a = gpu.getDoubleArray (N);
46       GpuDoubleArray b = gpu.getDoubleArray (N);
47       GpuDoubleMatrix c = gpu.getDoubleMatrix (N, N);
48       for (int i = 0; i < N; ++ i)
49          {
50          a.item[i] = prng.nextDouble()*20.0 - 10.0;
51          b.item[i] = prng.nextDouble()*20.0 - 10.0;
52          }
53       a.hostToDev();
54       b.hostToDev();
55
56       // Compute outer product.
57       Module module = gpu.getModule
58          ("edu/rit/gpu/example/OuterProductGpu.cubin");
```

Listing 24.2. OuterProductGpu.java (part 1)

`DoubleMatrix()` method (line 47), passing in the number of rows and columns. The main program must create the arrays and matrix this way, rather than constructing the arrays and matrix directly, so that the arrays' and matrix's contents can be transferred between the CPU and the GPU. Also note that the arrays' and matrix's data types are those of the previously declared kernel method's arguments.

For this example, the main program initializes the **A** and **B** vectors with random elements in the range –10.0 to +10.0 (lines 48–52), storing the values into the actual arrays, which are the `item` fields in the `a` and `b` objects. The main program then transfers the input vectors from the CPU to the GPU by calling the `hostToDev()` method on the `a` and `b` objects (lines 53–54).

With the input data set up and transferred to the GPU's memory, the main program proceeds to the kernel. The main program first obtains a *module object* by calling the GPU object's `getModule()` method (lines 57–58). The argument is the name of the CUDA module file that contains the compiled kernel function. The module file name is stated *relative to the Java class path.* Because class OuterProductGpu is in package edu.rit.gpu.example, the compiled Java class file OuterProductGpu.class is in directory edu/rit/gpu/example under the top level directory of the Java class path. The compiled CUDA module file is in this same directory, and so the compiled CUDA module file OuterProductGpu.cubin must be specified as "edu/rit/gpu/example/OuterProductGpu.cubin". If you are not using Java packages, all the compiled .class and .cubin files would typically be in the top level directory of the Java class path, and you would not need to include the directory name.

To configure and invoke the kernel, the main program obtains a *kernel object* by calling the module object's `getKernel()` method (lines 59–60), specifying the class of the kernel interface declared earlier. The `getKernel()` method returns a reference to a *proxy object* that implements the kernel interface. Thus, you can call the kernel method on the kernel object. You can also call other methods on the kernel object, which are declared in the Kernel superinterface.

The main program specifies the block and grid dimensions for the kernel by calling the kernel object's `setBlockDim()` and `setGridDim()` methods (lines 61–62). Each block is two-dimensional and consists of `NT`×`NT` threads, where `NT` was declared earlier to be 32 (line 25). Thus, each block will have the maximum possible number of threads, namely 1,024. The grid is also two-dimensional, with each dimension consisting of ceil(`N`/`NT`) blocks. (The ceiling function is computed with the expression `(N + NT – 1)/NT`; if `N` is not a multiple of `NT`, this rounds up to the next higher integer.) Thus, the entire grid will have enough threads to compute every matrix element, possibly with extra threads if `N` is not a multiple of `NT` as shown in Figure 24.6.

Because the kernel does not use the multiprocessor's shared memory, I want to configure the multiprocessor's fast RAM to consist of more L1 cache

```
59        OuterProductKernel kernel =
60           module.getKernel (OuterProductKernel.class);
61        kernel.setBlockDim (NT, NT);
62        kernel.setGridDim ((N + NT - 1)/NT, (N + NT - 1)/NT);
63        kernel.setCacheConfig (CacheConfig.CU_FUNC_CACHE_PREFER_L1);
64        long t2 = System.currentTimeMillis();
65        kernel.outerProduct (a, b, c, N);
66        long t3 = System.currentTimeMillis();
67
68        // Print results.
69        c.devToHost();
70
71        System.out.printf ("a[%d] = %.5f%n", 0,   a.item[0  ]);
72        System.out.printf ("a[%d] = %.5f%n", 1,   a.item[1  ]);
73        System.out.printf ("a[%d] = %.5f%n", N-2, a.item[N-2]);
74        System.out.printf ("a[%d] = %.5f%n", N-1, a.item[N-1]);
75
76        System.out.printf ("b[%d] = %.5f%n", 0,   b.item[0  ]);
77        System.out.printf ("b[%d] = %.5f%n", 1,   b.item[1  ]);
78        System.out.printf ("b[%d] = %.5f%n", N-2, b.item[N-2]);
79        System.out.printf ("b[%d] = %.5f%n", N-1, b.item[N-1]);
80
81        System.out.printf ("c[%d][%d] = %.5f%n",
82           0,   0,   c.item[0  ][0  ]);
83        System.out.printf ("c[%d][%d] = %.5f%n",
84           0,   1,   c.item[0  ][1  ]);
85        System.out.printf ("c[%d][%d] = %.5f%n",
86           0,   N-2, c.item[0  ][N-2]);
87        System.out.printf ("c[%d][%d] = %.5f%n",
88           0,   N-1, c.item[0  ][N-1]);
89        System.out.printf ("c[%d][%d] = %.5f%n",
90           1,   0,   c.item[1  ][0  ]);
91        System.out.printf ("c[%d][%d] = %.5f%n",
92           1,   1,   c.item[1  ][1  ]);
93        System.out.printf ("c[%d][%d] = %.5f%n",
94           1,   N-2, c.item[1  ][N-2]);
95        System.out.printf ("c[%d][%d] = %.5f%n",
96           1,   N-1, c.item[1  ][N-1]);
97        System.out.printf ("c[%d][%d] = %.5f%n",
98           N-2, 0,   c.item[N-2][0  ]);
99        System.out.printf ("c[%d][%d] = %.5f%n",
100          N-2, 1,   c.item[N-2][1  ]);
101       System.out.printf ("c[%d][%d] = %.5f%n",
102          N-2, N-2, c.item[N-2][N-2]);
103       System.out.printf ("c[%d][%d] = %.5f%n",
104          N-2, N-1, c.item[N-2][N-1]);
105       System.out.printf ("c[%d][%d] = %.5f%n",
106          N-1, 0,   c.item[N-1][0  ]);
107       System.out.printf ("c[%d][%d] = %.5f%n",
108          N-1, 1,   c.item[N-1][1  ]);
109       System.out.printf ("c[%d][%d] = %.5f%n",
110          N-1, N-2, c.item[N-1][N-2]);
111       System.out.printf ("c[%d][%d] = %.5f%n",
112          N-1, N-1, c.item[N-1][N-1]);
113
114       // Print running times.
115       long t4 = System.currentTimeMillis();
116       System.out.printf ("%d msec pre%n", t2 - t1);
```

Listing 24.2. OuterProductGpu.java (part 2)

and less shared memory. Increasing the amount of L1 cache will reduce the time required to access the arrays and matrix in the GPU's global memory. The main program configures the shared memory and L1 cache for the kernel by calling the kernel object's `setCacheConfig()` method (line 63), specifying that L1 is preferred over shared memory.

Finally, after all this setup, the main program launches the kernel by calling the kernel method on the kernel object (line 65), passing in the requisite arguments. At this point the main program blocks waiting for the kernel to finish execution. When the kernel method returns, the computation on the GPU is complete. The main program transfers the output matrix from the GPU to the CPU by calling the `devToHost()` method on the `c` object (line 69). The main program prints selected elements of the input **A** and **B** vectors and the output **C** matrix, obtaining the elements from the `item` fields of the `a`, `b`, and `c` objects. After printing the running times for various sections of the program, the task terminates.

The OuterProductGpu task class specifies two additional pieces of information: that it requires one CPU core (lines 133–136) and one GPU accelerator (lines 139–142). When the task is executed on a node or a cluster where the Parallel Java 2 Tracker is running, the Tracker ensures that the task runs on a node that has one idle CPU core and one idle GPU accelerator. The Tracker tells the task which GPU to use, and the `Gpu.gpu()` method on line 40 returns a GPU object referring to this GPU.

To compare the speed of the GPU with that of the CPU, I wrote a single-threaded CPU-only version of the vector outer product program in Java, class edu.rit.gpu.example.OuterProductSeq. I ran the CPU-only program on one core of `kraken` and the GPU program on one GPU accelerator of `kraken`. The `kraken` machine has four Intel Xeon E7-8850 processors, each with ten dual-hyperthreaded CPU cores, running at a 2.0 GHz clock rate; `kraken` also has four 448-core Nvidia Tesla C2075 GPU accelerators, running at a 1.15 GHz clock rate. Here are the programs' running times in milliseconds for vectors of various lengths *n*. Both the total running times and the running times for just the outer product calculation are listed.

	Total Time			*Calculation Time*		
n	*CPU*	*GPU*	*Ratio*	*CPU*	*GPU*	*Ratio*
1000	29	291	0.10	12	1	12.00
2000	58	355	0.16	23	1	23.00
4000	152	610	0.25	45	3	15.00
8000	668	1649	0.41	163	8	20.38
16000	3345	8711	0.38	624	33	18.91

The data shows that the GPU program's total running time is longer than the CPU-only program's. This is because the GPU program has to transfer two large vectors and a very large matrix between the CPU memory and the GPU

```
117         System.out.printf ("%d msec calc%n", t3 - t2);
118         System.out.printf ("%d msec post%n", t4 - t3);
119         System.out.printf ("%d msec total%n", t4 - t1);
120         }
121
122      // Print a usage message and exit.
123      private static void usage()
124         {
125         System.err.println ("Usage: java pj2 " +
126            "edu.rit.gpu.example.OuterProductGpu <seed> <N>");
127         System.err.println ("<seed> = Random seed");
128         System.err.println ("<N> = Vector length");
129         throw new IllegalArgumentException();
130         }
131
132      // Specify that this task requires one core.
133      protected static int coresRequired()
134         {
135         return 1;
136         }
137
138      // Specify that this task requires one GPU accelerator.
139      protected static int gpusRequired()
140         {
141         return 1;
142         }
143   }
```

Listing 24.2. OuterProductGpu.java (part 3)

memory—something the CPU-only program doesn't have to do. On the other hand, the parallelizable calculation runs 12 to 20 times faster on the GPU than on the CPU.

This example program, calculating a vector outer product, is not one that utilizes a GPU accelerator to best advantage. The calculation is trivial and doesn't take that long even on a single CPU core. I used this example primarily to introduce GPU and CUDA programming concepts. In the next few chapters we'll do some more serious computations on the GPU.

Under the Hood

Let's dissect the command that compiles the kernel function's kernel source file:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \
  --ptxas-options="-v" -o OuterProductGpu.cubin \
  OuterProductGpu.cu
```

The CUDA compiler can produce two kinds of compiled output. One alternative is a "PTX" file. This contains instructions for a device-independent virtual machine called PTX. PTX is analogous to the device-independent

Java Virtual Machine (JVM), and a PTX file is analogous to a Java class file. The Java compiler translates a Java source file into an intermediate Java class file. When the Java program is executed, the just-in-time compiler in the JVM translates the class file into machine code for the particular CPU where the JVM is running. In the same way, the CUDA compiler can translate a CUDA source file into an intermediate PTX file. When the GPU program is executed, the CUDA driver finishes the translation process and converts the PTX file into machine code for the particular GPU running the program. While this run-time translation imposes some overhead when the GPU program starts up, a PTX file can be executed on any kind of CUDA-capable GPU.

The other alternative is for the CUDA compiler to produce a "CUDA binary" file. The CUDA compiler translates the CUDA source file all the way down to machine instructions for a particular kind of GPU. This eliminates the overhead at program startup; but the price is that the CUDA binary file can be executed only on the particular kind of GPU for which it was compiled.

If portability across different kinds of GPUs is important, generate PTX files. If you're going to be using a specific GPU all the time, generate CUDA binary files and eliminate the startup overhead. That's what I usually do. The "`-cubin`" option tells the CUDA compiler to produce a CUDA binary file. (See the CUDA documentation for information about other options.)

Various CUDA-capable GPUs have different *compute capabilities*. A compute capability is denoted by a major version number and a minor version number, such as "compute capability 1.0." The compute capability specifies features that are supported, such as the number of registers in a multiprocessor, the maximum number of threads in a block, the maximum size of the multiprocessor's shared memory, the existence of certain specialized machine instructions, and so on. (See the CUDA documentation for a complete list of the compute capabilities and their supported features.) Each higher compute capability supports all the features of the lower compute capabilities and extends existing features or adds new features. The CUDA compiler needs to know the compute capability for which the source file is being compiled, so the compiler can flag as errors any features not supported by that compute capability. The "`-arch compute_20`" option specifies compute capability 2.0, which is the compute capability my Tesla C2075 GPU cards support.

Before launching a GPU kernel that was compiled assuming a certain compute capability, you need to verify that the GPU you're using in fact supports that compute capability. If it doesn't, the kernel function might fail if it tries to utilize a feature the GPU doesn't support. This is the purpose of the `ensureComputeCapability()` method call on line 41. The arguments are the major and minor version numbers, 2 and 0 in this case (compute capability

2.0). If the GPU supports the given compute capability or any higher compute capability, the method just returns. Otherwise, the method throws an exception that will abort the program.

When generating a CUDA binary file with the "`-cubin`" option, the CUDA compiler needs to know the GPU machine language into which to translate the source file. The "`-code sm_20`" option specifies the machine language associated with compute capability 2.0. (Again, see the CUDA documentation for a list of the GPU machine languages.)

The "`--ptxas-options="-v"`" option tells the CUDA compiler to turn on verbose output from the PTX assembler that is generating the machine code in the CUDA binary file. This verbose output includes some important information. Here is what the `nvcc` command prints:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \
  --ptxas-options="-v" -o OuterProductGpu.cubin \
  OuterProductGpu.cu
ptxas info: 0 bytes gmem
ptxas info: Compiling entry function 'outerProduct' for 'sm_20'
ptxas info: Function properties for outerProduct
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info: Used 10 registers, 48 bytes cmem[0]
```

A kernel function typically declares local variables. The vector outer product kernel function declared the local variables `row` and `col`. The compiler places local variables in registers in the multiprocessor's register bank. The compiler also uses registers to hold intermediate values of expressions. The last line of output above tells us that the compiled kernel function uses 10 registers. So each thread in a block, executing the kernel function, needs its own separate group of 10 registers. I configured the kernel with 1,024 threads per block, so each block requires 10,240 registers. A compute capability 2.0 GPU has 32,768 registers in each multiprocessor. So all is well.

If you configure a kernel with too many threads, such that a block requires more registers than a multiprocessor has, the kernel will fail to launch, and the Java main program will throw an exception. It's important to be aware of your kernel function's register usage. You have to make sure that the number of registers per thread times the number of threads per block does not exceed the number of registers per multiprocessor. You might have to reduce the number of threads per block to get them to fit.

Finally, the "`-o OuterProductGpu.cubin`" option gives the name of the output CUDA binary file the compiler is to generate, and "`OuterProductGpu.cu`" gives the name of the CUDA source file to compile.

Points to Remember

- A CUDA-capable GPU consists of a number of multiprocessors plus a global memory.

- Each multiprocessor has a number of cores, a bank of high-speed registers, a fast shared memory, and a fast L1 cache memory. The multiprocessors can be configured with more shared memory and less L1 cache, or vice versa.
- A CUDA kernel consists of a grid plus a kernel function.
- A grid consists of a number of blocks in a one-, two-, or three-dimensional arrangement.
- A block consists of a number of threads in a one-, two-, or three-dimensional arrangement.
- Each thread in the grid executes the same kernel function.
- Each thread in the grid has a unique identity, determined by the grid dimensions (`gridDim`), block dimensions (`blockDim`), block index (`blockIdx`), and thread index (`threadIdx`).
- A GPU parallel program consists of a task main program written in Java using the Parallel Java 2 Library, plus a kernel function written in C (or C++ or Fortran) written using CUDA.
- The GPU computational arc: Input data flows from the CPU memory to the GPU memory, calculations take place on the GPU, output data flows from the GPU memory to the CPU memory.
- In the Java main program, declare the kernel method in a kernel interface that extends interface edu.rit.gpu.Kernel.
- Use Java classes in package edu.rit.gpu, such as GpuDoubleArray and GpuDoubleMatrix, to set up variables mirrored in the CPU memory and the GPU memory. Call methods on these objects to transfer data from the CPU to the GPU or vice versa.
- To execute a kernel, use Java classes in package edu.rit.gpu to get a module object; get a kernel object that implements the kernel interface; configure the grid in the kernel object; and call the kernel method on the kernel object.
- Define the static `coresRequired()` and `gpusRequired()` methods in the task class.
- Be aware of the GPU's compute capability, and compile the kernel source file appropriately.
- Be aware of the kernel function's register usage, the number of registers the threads in a block require, and the maximum number of registers available in a multiprocessor.

Chapter 25
GPU Parallel Reduction

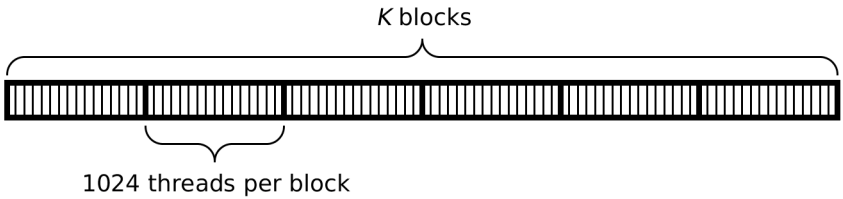
Let's try a more substantial GPU parallel program: our old friend, the π estimating program. Recall that the program throws N darts that land at random locations in the unit square, counts how many darts C fall within the inscribed circle quadrant, and estimates $\pi \approx 4C/N$. In a parallel version, the loop iterations (darts) are partitioned among K threads; each thread throws N/K darts and increments its own semifinal count; at the end, the semifinal counts are sum-reduced together, yielding the final total count from which to compute the result.

I will use this same approach for the GPU parallel π estimating program. But I have to be careful about how I partition the computation and about how I perform the reduction.

In the GPU parallel outer product program in Chaper 21, I set up the computational grid to match the structure of the output matrix. There was one thread for each matrix element. The threads were arranged into two-dimensional blocks, and the blocks were arranged into a two-dimensional grid, mirroring the two-dimensional array (matrix) being computed. Each thread computed one and only one matrix element.

In contrast, the GPU parallel π estimating program has no such structure. The computation's output is a single value, C, the total number of darts that landed inside the circle quadrant. So other considerations will dictate how I set up the computational grid.

I want to use the full parallelism available on the GPU. So the grid will be one-dimensional, and the number of blocks in the grid will be the same as the number of multiprocessors in the GPU. (As we will see, the Parallel Java 2 Library lets you query how many multiprocessors the GPU has.) Thus, when I run the kernel, each block in the grid will be assigned to its own multiprocessor, and all the blocks will run fully in parallel. Each block in turn will be one-dimensional, and the number of threads in the block will be the maximum possible. The maximum threads per block is a characteristic of the CUDA compute capability; for my compute-capability-2.0 GPU, that's 1024 threads. So if the GPU has K multiprocessors, the grid will have $1024 \cdot K$ threads; and each thread will do $N/(1024 \cdot K)$ loop iterations.

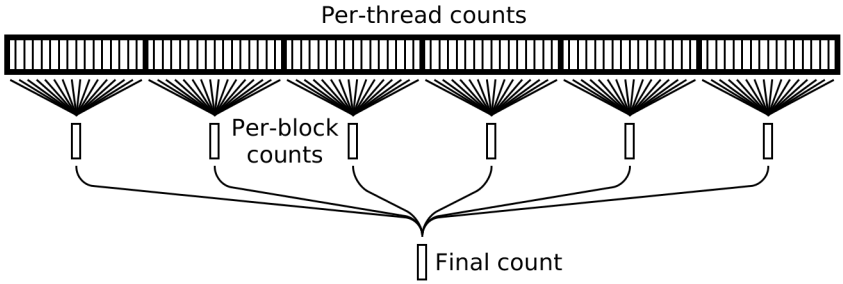


The kernel's output is the final total count C. This will be stored in the GPU's global memory, so the CPU can retrieve it. Following the parallel reduction pattern, each thread in the kernel will maintain its own per-thread counter. Reducing the per-thread counters together will take place in two

```
1  package edu.rit.gpu.example;
2  import edu.rit.gpu.Kernel;
3  import edu.rit.gpu.Gpu;
4  import edu.rit.gpu.GpuLongVbl;
5  import edu.rit.gpu.Module;
6  import edu.rit.pj2.Task;
7  public class PiGpu
8     extends Task
9     {
10    // Kernel function interface.
11    private static interface PiKernel
12       extends Kernel
13       {
14       public void computeRandomPoints
15          (long seed,
16           long N);
17       }
18
19    // Task main program.
20    public void main
21       (String[] args)
22       throws Exception
23       {
24       // Validate command line arguments.
25       if (args.length != 2) usage();
26       long seed = Long.parseLong (args[0]);
27       long N = Long.parseLong (args[1]);
28
29       // Initialize GPU.
30       Gpu gpu = Gpu.gpu();
31       gpu.ensureComputeCapability (2, 0);
32
33       // Set up GPU counter variable.
34       Module module = gpu.getModule
35          ("edu/rit/gpu/example/PiGpu.cubin");
36       GpuLongVbl count = module.getLongVbl ("devCount");
37
38       // Generate n random points in the unit square, count how many
39       // are in the unit circle.
40       count.item = 0;
41       count.hostToDev();
42       PiKernel kernel = module.getKernel (PiKernel.class);
43       kernel.setBlockDim (1024);
44       kernel.setGridDim (gpu.getMultiprocessorCount());
45       kernel.computeRandomPoints (seed, N);
46
47       // Print results.
48       count.devToHost();
49       System.out.printf ("pi = 4*%d/%d = %.9f%n", count.item, N,
50          4.0*count.item/N);
51       }
52
53    // Print a usage message and exit.
54    private static void usage()
55       {
56       System.err.println ("Usage: java pj2 " +
57          "edu.rit.gpu.example.PiGpu <seed> <N>");
58       System.err.println ("<seed> = Random seed");
```

Listing 25.1. PiGpu.java (part 1)

stages. Within each block, the threads in the block will use the multiprocessor's fast shared memory to add the per-thread counters together, yielding one semifinal count in shared memory for each block. The per-block counts will in turn be added together, yielding the final count in global memory.



Listing 25.1 is the Java main program, class PiGpu. The main program doesn't do much; it sets up the GPU kernel, runs the kernel, retrieves the kernel's result, and prints the answer. All the real work happens in parallel in the kernel.

Class PiGpu begins by declaring the GPU kernel interface (lines 11–17). The kernel function is named `computeRandomPoints()`, and it takes two arguments, the `seed` for the pseudorandom number generator (PRNG) and the number of darts to throw, `N`, both of Java type `long`.

The main program needs to access the variable in GPU global memory that will end up holding the kernel's result. To gain access to this variable, the main program first obtains a GPU object (lines 30–31) and obtains the compiled GPU module (lines 34–35). The main program then calls the `getLongVbl()` method on the module object, specifying the name of a certain variable in the GPU module, `devCount`. (We'll see the GPU variable's declaration when we look at the code for the kernel.) The `getLongVbl()` method returns a GpuLongVbl object, which is stored in the main program's `count` variable. The `count` object has a field of type `long`; this field (on the CPU) mirrors the `devCount` variable (on the GPU).

The main program is now ready to run the computation. First, it initializes the count to 0 and uploads the CPU variable to the GPU variable (lines 40–41); this initializes the count on the GPU. One-time initializations like this have to be done by the CPU main program, not by the GPU kernel function. If this were done in the kernel function, every thread in the grid would initialize the count, which would be incorrect. Next, the main program gets the kernel object; configures the grid to have 1024 threads per block and K blocks, where K is the number of multiprocessors on the GPU as returned by the `getMultiprocessorCount()` method; and calls the kernel method (lines 42–45).

When the kernel method returns, the computation has finished, and the

```
59          System.err.println ("<N> = Number of random points");
60          throw new IllegalArgumentException();
61          }
62
63       // Specify that this task requires one core.
64       protected static int coresRequired()
65          {
66          return 1;
67          }
68
69       // Specify that this task requires one GPU accelerator.
70       protected static int gpusRequired()
71          {
72          return 1;
73          }
74    }
```

Listing 25.1. PiGpu.java (part 2)

count on the GPU has been set to *C*, the total number of darts that landed inside the circle quadrant. The main program downloads the `count` object from the GPU to the CPU (line 48). The count object's `item` field now contains *C*, and the program uses that to print the results (lines 49–50).

To finish off the Java code, the PiGpu task class overrides the `coresRequired()` method (lines 64–67) to state that the task requires just one CPU core—because there is no multithreaded parallelism on the CPU side. The PiGpu task class also overrides the `gpusRequired()` method (lines 70-73) to state that the task requires one GPU accelerator; if this is omitted, the Tracker would assume the task requires no GPUs, the Tracker might assign the task to a node that has no GPU, and the program would fail.

Turning to the GPU side of the program, Listing 25.2 has the GPU kernel function as well as several variable declarations, written in C with CUDA. Line 1 includes the `Random.cu` source file, which defines a typedef `prng_t` for a PRNG as well as several functions for seeding the generator and extracting random numbers. (I'm not going to describe what's inside `Random.cu`.) Line 4 states that the grid is assumed to have `NT` = 1024 threads per block. (It's up to the main program to configure the grid properly.)

Line 7 declares the `devCount` variable, which will hold the final count *C*. The CUDA keyword `__device__` ("underscore underscore device underscore underscore") says that the variable is to be located in the GPU's global memory. The `devCount` variable is located in the global memory for two reasons: so that all the threads in all the blocks can access the count for purposes of reduction, and so that the count can be mirrored in the CPU main program.

Line 10 declares an array variable named `shrCount`. In each block, this array will hold the per-thread counts for purposes of reduction. The CUDA

keyword `__shared__` ("underscore underscore shared underscore underscore") says that the array is to be located in the multiprocessor's fast shared memory. Because each multiprocessor has its own separate shared memory, each block in the grid gets its own separate `shrCount` array. The array needs only as many elements as there are threads in a *block*, namely `NT` elements—*not* as many elements as there are threads in the whole grid. The `shrCount` array is located in the shared memory so that all the threads in one block can access the array during the reduction. There is no need (and in fact no way) for threads in one block to access the `shrCount` array in a different block.

Lines 17–19 declare the kernel function, `computeRandomPoints()`. The arguments are the PRNG's `seed` and the number of loop iterations `N`, both of type `unsigned long long int` in C. (In the Java main program, the corresponding kernel function arguments are of type `long`; see PiGpu.java lines 14–16.)

The kernel function begins by determining several quantities (lines 26–28). `thr` is the index of the currently executing thread within the block, obtained from the `threadIdx.x` pseudo-variable. `size` is the number of threads in the entire grid, which is the number of blocks in the grid (the `gridDim.x` pseudo-variable) times the number of threads per block. The *N* loop iterations need to be partitioned among this many threads. `rank` is the unique rank of the currently executing thread within the entire grid, computed from the block's *x* coordinate in the grid and the thread's *x* coordinate in the block.

When each thread executes the kernel function, the thread gets its own per-thread counter in the `count` local variable and its own per-thread PRNG of type `prng_t` in the `prng` local variable. These variables are declared on lines 22–23 and initialized on lines 31–32. The PRNG is initialized differently in each thread (`seed` plus `rank`); this ensures that each thread will generate different random dart locations. The per-thread counter is initialized to 0. We've now seen all the counter variables that will be involved in the reduction: the per-thread counts (line 22), the per-block counts in shared memory (line 10), and the final count in global memory (line 7).

The kernel is now ready to partition the loop iterations among the threads in the grid, and have each thread execute a subset of the loop iterations. The for loop on line 35 partitions the iterations using what amounts to a leapfrog schedule. Each thread's initial loop index is the same as the thread's rank. Going to the next iteration, each thread increases its loop index by the total number of threads in the grid. Suppose the grid consists of 14 blocks of 1024 threads each, for a total of 14336 threads, and suppose *N* is 100000. Then thread rank 0 performs loop indexes 0, 14336, 28672, 43008, 57344, 71680, and 86016; thread rank 1 performs loop indexes 1, 14337, 28673, 43009, 57345, 71681, and 86017; and so on. Each thread ends up performing an equal (or nearly equal) number of loop iterations; and because the running time of each loop iteration is the same, the load is balanced. In the loop body

```
#include "Random.cu"

// Number of threads per block.
#define NT 1024

// Overall counter variable in global memory.
__device__ unsigned long long int devCount;

// Per-thread counter variables in shared memory.
__shared__ unsigned long long int shrCount [NT];

// Device kernel to compute random points.
// Called with a one-dimensional grid of one-dimensional blocks, NB
// blocks, NT threads per block. NT must be a power of 2.
// seed  Pseudorandom number generator seed.
// N     Number of points.
extern "C" __global__ void computeRandomPoints
   (unsigned long long int seed,
    unsigned long long int N)
   {
   int thr, size, rank;
   unsigned long long int count;
   prng_t prng;

   // Determine number of threads and this thread's rank.
   thr = threadIdx.x;
   size = gridDim.x*NT;
   rank = blockIdx.x*NT + thr;

   // Initialize per-thread prng and count.
   prngSetSeed (&prng, seed + rank);
   count = 0;

   // Compute random points.
   for (unsigned long long int i = rank; i < N; i += size)
      {
      double x = prngNextDouble (&prng);
      double y = prngNextDouble (&prng);
      if (x*x + y*y <= 1.0) ++ count;
      }

   // Shared memory parallel reduction within thread block.
   shrCount[thr] = count;
   __syncthreads();
   for (int i = NT/2; i > 0; i >>= 1)
      {
      if (thr < i)
         shrCount[thr] += shrCount[thr+i];
      __syncthreads();
      }

   // Atomic reduction into overall counter.
   if (thr == 0)
      atomicAdd (&devCount, shrCount[0]);
   }
```

Listing 25.2. PiGpu.cu

(lines 37–39), a random (x, y) dart location is extracted from the per-thread PRNG, and the per-thread counter is incremented if the dart falls inside the circle quadrant. Because each thread is updating its own per-thread PRNG and counter, no thread synchronization is needed at this point.

After finishing the loop iterations, the kernel is ready to perform the reduction. The first stage of reduction is to add up all the per-thread counts within the block. This is done in parallel using a reduction tree, like the one we encountered in the multicore parallel π estimating program in Chapter 4 (Figure 25.1). There, the code to do the reduction was hidden inside the reduction variable classes in the Parallel Java 2 Library, and the parallel for loop did the reduction automatically. CUDA has nothing analogous to a reduction variable class, so here I have to code the reduction tree myself (lines 43–50). The reduction makes use of the multiprocessor's fast shared memory, which all the threads in the block can access.

Each thread first stores its per-thread counter variable (`count`) into one of the elements of the shared memory array (`shrCount[thr]`); the array index is the same as the thread's index within the block. Before proceeding, each thread has to wait until all the threads in the block have stored their per-thread counters. In other words, at this point the threads need to wait at a barrier. The `__syncthreads()` function ("underscore underscore syncthreads"), a special CUDA function, performs the barrier synchronization. The barrier is implemented in hardware and is very fast.

Now the shared memory parallel reduction can commence. The reduction proceeds in a number of rounds; each iteration of the loop on lines 45–50 is one round. The loop index `i` is the *stride* for the current round; the stride starts at half the number of threads in the block, namely 512; the stride is halved at each loop iteration; so the strides for the rounds are 512, 256, 128, . . . , 4, 2, 1; when the stride hits 0, the loop stops. The stride is halved by right-shifting it one bit position (`i >>= 1`), which is equivalent to dividing it by 2. During each round, each thread whose index is less than the stride adds its own array element (`shrCount[thr]`) together with the array element whose index is the stride higher (`shrCount[thr+i]`), and stores the result back into its own array element. These additions are all done in parallel. Again, after the additions there is a barrier, to ensure that all threads have completed their additions before proceeding to the next round of reduction.

Thus, in the first round with a stride of 512, thread 0 adds array element 512 into array element 0, thread 1 adds element 513 into element 1, . . . , thread 511 adds element 1023 into element 511; threads 512–1023 do nothing. In the second round with a stride of 256, thread 0 adds array element 256 into array element 0, thread 1 adds element 257 into element 1, . . . , thread 255 adds element 511 into element 255; threads 256–1023 do nothing. And so on. This implements exactly the reduction tree shown in Figure 25.1 (extended upwards to accommodate 1024 elements). Because each thread is up-

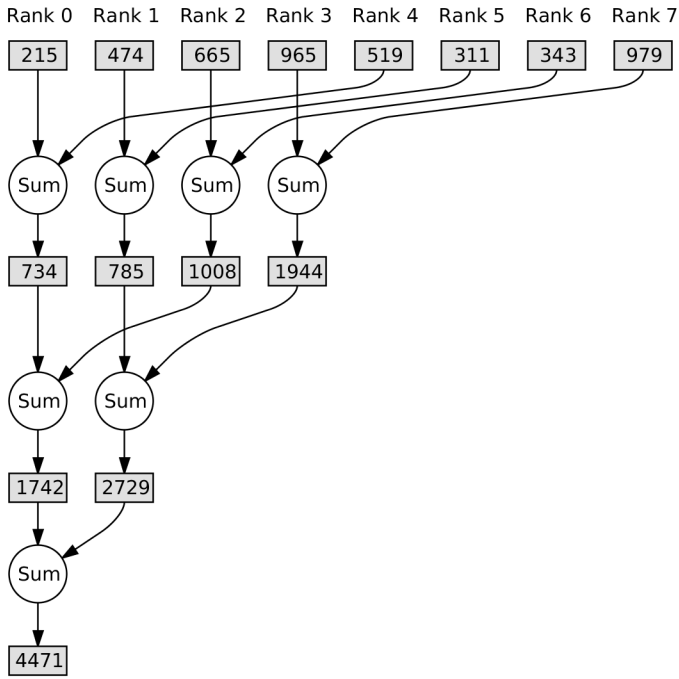


Figure 25.1. Sum-reduce parallel reduction tree

dating a different array element, the threads do not need to synchronize with each other *during* the additions; the threads only need to synchronize at the barrier at the *end* of each round; this minimizes the synchronization overhead. When the reduction loop exits, the per-block count—the sum of all the per-thread counts—ends up stored in `shrCount[0]`.

This shared memory parallel reduction code is specifically designed for the case where the number of threads in the block is a power of 2. If, in some other program that does reduction, the number of threads is not a power of 2, the code would have to be altered a little. The initial stride would be the smallest power of 2 greater than or equal to half the number of threads in the block. The if statement in the loop body would check that the thread index is less than the stride *and* that the thread index plus the stride is less than the number of threads in the block.

Now the second stage of reduction happens. Each thread must add its own per-block count into the global count variable, `devCount`. This is done by a single thread in each block, namely thread rank 0 (line 53). Multiple threads, one in each block, are attempting to update the global variable concurrently; therefore, these threads do need to synchronize with each other. I get the necessary synchronization by using an *atomic operation* to do the up-

date (line 54). The `atomicAdd()` function, a special CUDA function, adds its second argument (the per-block count, `shrCount[0]`) to the variable referred to by the first argument (the global count, `devCount`) and stores the result back into that variable. Furthermore, the `atomicAdd()` function ensures that only one thread at a time performs the operation. "Atomic" means "not divisible" or "not interruptible"—a thread doing an atomic add on a variable will not be interrupted by another thread trying to do an atomic add on that variable until the first add is complete. Various atomic operations are available in different CUDA compute capabilities; `atomicAdd()` of unsigned long long integer variables is supported in compute capability 2.0 GPUs like mine. (See the CUDA documentation for a list of the supported atomic operations.) This is another reason to verify the GPU's compute capability before proceeding with the program.

When all the threads in all the blocks in the grid have finished executing the kernel function, the global `devCount` variable contains the sum of all the per-block counts, which is the sum of all the per-thread counts, which is the total number of darts that landed inside the circle quadrant, *C*. At this point, back in the CPU main program, the kernel method returns, and the CPU main program downloads *C* from the GPU and prints the result. Done!

I compared the total running times of the single-threaded CPU-only version of the π estimating program (PiSeq) with the GPU parallel version (PiGpu) on the `kraken` machine for various problem sizes, using commands like this:

```
$ java pj2 edu.rit.pj2.example.PiSeq 142857 1000000
$ java pj2 edu.rit.gpu.example.PiGpu 142857 1000000
```

Here are the running times I observed, in milliseconds. I also listed the estimate for π that the PiGpu program computed.

N	*CPU*	*GPU*	*Ratio*	π *Estimate*
1×10^{6}	50	96	0.5	3.142316000
1×10^{7}	181	87	2.1	3.141395200
1×10^{8}	1233	116	10.6	3.141559320
1×10^{9}	11415	277	41.2	3.141609640
1×10^{10}	113127	1985	57.0	3.141612167
1×10^{11}	1129808	19088	59.2	3.141598614
1×10^{12}	11300994	190098	59.4	3.141594103

Once the problem size becomes large enough that the fixed overhead no longer dominates the running time, the GPU program is 60 times faster than the CPU program on `kraken`. It took the CPU over three hours to compute one trillion darts. It took the GPU only three minutes. That's a computation rate of over five billion darts per second. No human darts champion in any pub on the globe ever threw darts that fast.

Under the Hood

The GPU parallel π estimating program's `main()` method, which runs on the CPU, declares a variable named `count` (PiGpu.java line 36). The kernel declares a variable named `devCount` (PiGpu.cu line 7) located in the GPU's global memory. The `count` variable on the CPU "mirrors" the `devCount` variable on the GPU. Let's look more closely at how mirrored variables work.

The `count` object was created by calling the `getLongVbl("devCount")` method on the module object obtained from the GPU object. As we have seen, `count` is an instance of class GpuLongVbl, with a public field named `item` of type `long`. The `getLongVbl()` method calls a CUDA function to get the address in the GPU's global memory of the GPU variable named `devCount`. The address of `devCount` is stored in a private field of the `count` object. When the main program calls `count.hostToDev()` (line 41), the `hostToDev()` method in turn calls a CUDA function to copy eight bytes—the size of a `long` variable—from the address of the count object's `item` field on the CPU to the address of `devCount` on the GPU. When the main program calls `count.devToHost()` (line 48), the `devToHost()` method in turn calls a CUDA function to copy eight bytes from the address of `devCount` on the GPU to the address of the count object's `item` field on the CPU. In this way, I can keep the variable's value in sync between the CPU and the GPU.

The GpuDoubleArray and GpuDoubleMatrix classes—which the outer product program in Chapter 24 used—work the same way, as do all of the GPU variable classes in package edu.rit.gpu. Each class stores the variable's GPU global memory address in a private field. The `hostToDev()` and `devToHost()` methods call CUDA functions to copy the variable's value from one side to the other. The Java methods use the Java Native Interface (JNI) to call the non-Java CUDA functions.

However, copying the mirrored variable does not happen automatically. You have to call the `hostToDev()` or `devToHost()` method explicitly. Why? Because it's not necessarily true that every change to the variable's value on one side needs to be immediately reflected on the other side. (We'll see an example of this in Chapter 27.)

Here's the command that compiles the PiGpu.cu source file, producing the PiGpu.cubin CUDA binary file:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \
  --ptxas-options="-v" -o PiGpu.cubin PiGpu.cu
ptxas info: 8 bytes gmem, 4 bytes cmem[14]
ptxas info: Compiling entry function 'computeRandomPoints' for
'sm_20'
ptxas info: Function properties for computeRandomPoints
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info: Used 20 registers, 8192 bytes smem, 48 bytes
cmem[0], 24 bytes cmem[16]
```

The kernel function uses 20 registers, and there are 1,024 threads per block, so each block requires 20,480 registers; that fits within the 32,768 registers available in each multiprocessor. The `shrCount` array occupies 8,192 bytes of shared memory (1,024 array elements of 8 bytes each); that fits within the 48 kilobytes of shared memory available in each multiprocessor. So this kernel will have no trouble running on my compute-capability-2.0 GPU.

Points to Remember

- Configure the kernel grid to match the problem structure.
- If the problem has no inherent two-dimensional or three-dimensional structure, go with a one-dimensional grid of one-dimensional blocks.
- Use the multiprocessor's shared memory to do parallel reduction within a block, following the reduction tree pattern.
- Using the `__syncthreads()` function, synchronize the threads at a barrier immediately before beginning the reduction tree and at the end of each round of the reduction tree.
- Use the GPU's global memory to store the final result of the reduction.
- Only one thread in the block should do the reduction into the global memory variable.
- Use CUDA's atomic operations to synchronize the threads in different blocks that are updating the global memory variable.

Chapter 26
Multi-GPU Programming

Suppose I have a node with more than one GPU accelerator. The `kraken` machine, for example, has four Nvidia Tesla C2075 GPU cards, as well as 80 CPU cores (40 dual-hyperthreaded cores). Each Tesla card has 448 GPU cores. How can I utilize all 1,792 GPU cores on this node?

One way is to use the massively parallel approach introduced in Chapter 14 in the context of a cluster parallel computer. I can run four separate GPU accelerated parallel programs on `kraken` at once, each program running on a separate GPU.

But what I really might like to do is run *one* parallel program and scale it up to use all the GPUs on the node. To illustrate this approach, let's change the GPU parallel π estimating program so it can run on multiple GPUs.

The simplest way to do this is to combine the multicore paradigm from Chapter 4 with the GPU accelerated paradigm from Chapter 25 (Figure 26.1). The program consists of multiple threads. There is one thread *for each GPU* on the node (not one thread for each CPU core). The N loop iterations (dart throws) are partitioned among the threads, in the manner with which we are familiar. Each thread runs a computational kernel on its own GPU—in fact, the exact same kernel as in Chapter 25. The kernel computes the number of darts within the circle quadrant for the thread's portion of the loop iterations. These kernel results become the semifinal counts for each thread—C_0, C_1, and so on. The threads' semifinal counts are sum-reduced, again in the manner with which we are familiar, to produce the final count C.

The second version of the GPU parallel π estimating program, class PiGpu2 (Listing 26.1), begins the same as the first version, with the command line arguments `seed` and `N` (lines 13–14) and the kernel function interface (lines 18–24). It's the same kernel function interface as the previous version. In addition, there is a reduction variable of type LongVbl named `count` (line 15); this will hold the final count after the per-thread semifinal counts are reduced together.

The task main program (line 26) begins by obtaining the command line arguments and initializing the `count` global variable to do a sum reduction (line 36). Next the program sets up a parallel thread team. The `parallelDo()` method on line 39 creates the thread team, where the first argument is the number of threads in the team, namely the number of GPU accelerators, and the second argument is the parallel section object containing the code each team thread will execute. The number of GPUs is determined by calling the `Gpu.allowedDeviceCount()` method, which returns the number of GPUs this process is allowed to use. As we will see later, the default is to use all the GPUs on the node; but this can be overridden with an option on the `pj2` command line.

Each parallel team thread now calls the `run()` method on its own copy of the parallel section object defined in the anonymous inner class starting on line 40. The thread first creates its own per-thread reduction variable,

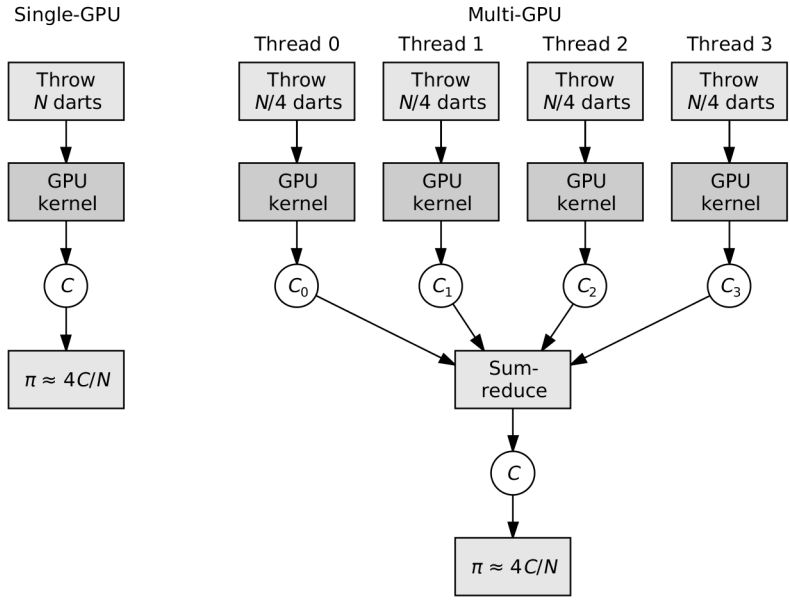


Figure 26.1. Estimating π with a single GPU and with multiple GPUs

```
1  package edu.rit.gpu.example;
2  import edu.rit.gpu.Kernel;
3  import edu.rit.gpu.Gpu;
4  import edu.rit.gpu.GpuLongVbl;
5  import edu.rit.gpu.Module;
6  import edu.rit.pj2.LongChunk;
7  import edu.rit.pj2.Section;
8  import edu.rit.pj2.Task;
9  import edu.rit.pj2.vbl.LongVbl;
10 public class PiGpu2
11    extends Task
12    {
13    private static long seed;
14    private static long N;
15    private static LongVbl count;
16
17    // Kernel function interface.
18    private static interface PiKernel
19       extends Kernel
20       {
21       public void computeRandomPoints
22          (long seed,
23           long N);
24       }
```

Listing 26.1. PiGpu2.java (part 1)

`thrCount`, linked to the global reduction variable `count` (line 44). The thread next obtains a GPU object (line 47). Under the hood, the `Gpu.gpu()` method returns a different GPU object, representing a different GPU accelerator, to each calling thread. Thus, each thread ends up working with its own separate GPU. The rest of the `run()` method is almost identical to the single-GPU π estimating program in Chapter 25, except for two things.

The first difference is that before launching the GPU kernels, the N loop iterations must be partitioned among the parallel team threads. This is done by calling the `LongChunk.partition()` method on line 57. The method partitions the total index range (0 through $N-1$) into as many equal-sized chunks as there are threads in the team (`threads()`) and returns the chunk associated with the current thread's rank in the team (`rank()`). The length of this chunk (`length()`) is the number of iterations (`thrN`) the current thread, that is, the current thread's GPU kernel, will perform.

The second difference is in the arguments passed to the GPU kernel method on lines 67–68. Each thread's kernel must generate a different sequence of random values; so the seed passed to the kernel is the seed from the command line plus one million times the current thread's rank. Thus, the seed for thread rank 0 is just `seed`; the seed for thread rank 1 is `seed` + 1,000,000; the seed for thread rank 2 is `seed` + 2,000,000; and so on. Inside the kernel, each GPU thread in turn adds its own rank to this seed, and the result is used to initialize each GPU thread's pseudorandom number generator (PRNG). Thus, each GPU thread's PRNG is initialized with a different seed and generates a different sequence of random values. (This assumes that the kernel will have fewer than one million GPU threads, which seems reasonable.) Also, the number of iterations the kernel will perform is specified as `thrN`, the per-thread number—not `N`, the total number.

After each thread's kernel method returns, the thread downloads the kernel's count and stores it in the thread's own per-thread `thrCount` variable (lines 72–73). After all the threads have finished, the per-thread counts are automatically sum-reduced into the global `count` variable, which is used to print the answer (lines 78–79).

Although the PiGpu2 program runs with a team of multiple threads, most of the time the threads are blocked waiting for the GPU kernel method to return. There is no need to tie up a whole core for each thread. Accordingly, the `coresRequired()` method is overridden to specify that the program needs only one core (lines 83–86). When the Tracker schedules the program to run, the Tracker needs to find a node with only one idle core; all the threads will share this core. On the other hand, the PiGpu2 program wants to use all the GPU accelerators on the node, and the `gpusRequired()` method is overridden to specify this (lines 90–93). When the Tracker schedules the program to run, the Tracker needs to find a node all of whose GPUs are idle. You can specify that the program use a particular number of GPUs by including the

```
25    // Task main program.
26    public void main
27       (String[] args)
28       throws Exception
29       {
30       // Validate command line arguments.
31       if (args.length != 2) usage();
32       seed = Long.parseLong (args[0]);
33       N = Long.parseLong (args[1]);
34
35       // Set up global counter variable.
36       count = new LongVbl.Sum (0);
37
38       // Run one CPU thread for each GPU on the node.
39       parallelDo (Gpu.allowedDeviceCount(), new Section()
40          {
41          public void run() throws Exception
42             {
43             // Set up per-thread counter variable.
44             LongVbl thrCount = threadLocal (count);
45
46             // Initialize per-thread GPU.
47             Gpu gpu = Gpu.gpu();
48             gpu.ensureComputeCapability (2, 0);
49
50             // Set up GPU counter variable.
51             Module module = gpu.getModule
52                ("edu/rit/gpu/example/PiGpu.cubin");
53             GpuLongVbl devCount = module.getLongVbl ("devCount");
54
55             // Determine how many of the N points this thread will
56             // compute.
57             long thrN = LongChunk.partition (0, N - 1, threads(),
58                rank()) .length() .longval();
59
60             // Generate thrN random points in the unit square,
61             // count how many are in the unit circle.
62             devCount.item = 0;
63             devCount.hostToDev();
64             PiKernel kernel = module.getKernel (PiKernel.class);
65             kernel.setBlockDim (1024);
66             kernel.setGridDim (gpu.getMultiprocessorCount());
67             kernel.computeRandomPoints (seed + 1000000L*rank(),
68                thrN);
69
70             // Get per-thread count, automatically reduced into
71             // global count.
72             devCount.devToHost();
73             thrCount.item = devCount.item;
74             }
75          });
76
77       // Print results.
78       System.out.printf ("pi = 4*%d/%d = %.9f%n", count.item, N,
79          4.0*count.item/N);
80       }
```

Listing 26.1. PiGpu2.java (part 2)

"gpus=" option on the pj2 command line.

To study the PiGpu2 program's weak scaling performance, I ran the program on one to four GPUs on the kraken machine. The scale factor K was the number of GPU accelerators (not the number of CPU cores). I ran the program with problem size N = 100 billion, 200 billion, 500 billion, one trillion, and two trillion darts with one GPU. As I increased K, I also increased N in the same proportion. Here are examples of the commands I used:

```
$ java pj2 debug=makespan gpus=1 edu.rit.gpu.example.PiGpu2 \
  142857 100000000000
$ java pj2 debug=makespan gpus=2 edu.rit.gpu.example.PiGpu2 \
  142857 200000000000
```

Figure 26.2 plots the running times and efficiencies I observed. The fitted running time model is

$$T = 1.28 + 9.39\times10^{-15}\,N + 0.180\,K + 1.90\times10^{-10}\,N\div K \qquad (26.1)$$

The program's sequential portion takes 1.28 seconds (plus a negligible term proportional to N); this yields a sequential fraction ranging from 0.06 for the smallest problem size down to 0.003 for the largest problem size. Once again, we see that as the amount of computation increases, the overhead due to the fixed sequential portion diminishes, resulting in higher efficiencies. Each parallel team thread takes 0.180 seconds to do its one-time initialization, mostly setting up the thread's GPU. Each dart throw takes 1.90×10^{-10} seconds, for a computation rate of 5.26 billion darts per second.

Below are the estimates for π calculated by the program for various problem sizes N. Note how the the estimate improves—Δ, the relative difference between the program's estimate and the actual value of π, trends downward—as the number of darts increases. Programs like this must do enormous numbers of iterations to get accurate answers, which makes such programs attractive candidates for parallelization.

N	π Estimate	Δ
4×10^{11}	3.141591085	4.99×10^{-7}
8×10^{11}	3.141589897	8.77×10^{-7}
2×10^{12}	3.141591885	2.45×10^{-7}
4×10^{12}	3.141593160	1.61×10^{-7}
8×10^{12}	3.141593379	2.31×10^{-7}

Points to Remember

- To run a single parallel program on multiple GPUs, dedicate a separate thread to each GPU.
- Call Gpu.allowedDeviceCount() to determine the number of GPU accelerators the program is allowed to use.

```
81    // Specify that this task requires one core. (Parallel team
82    // threads will share the core.)
83    protected static int coresRequired()
84       {
85       return 1;
86       }
87
88    // Specify that this task requires all GPU accelerators on the
89    // node.
90    protected static int gpusRequired()
91       {
92       return ALL_GPUS;
93       }
94 }
```

Listing 26.1. PiGpu2.java (part 3)

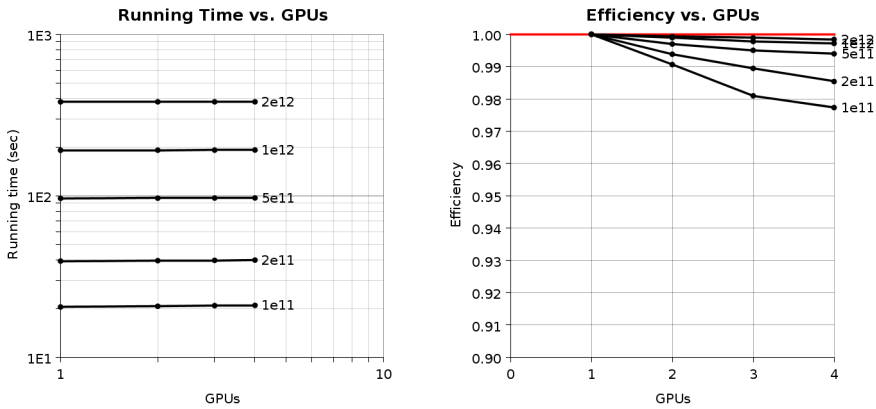


Figure 26.2. PiGpu2 weak scaling performance metrics

- Use the `parallelDo` statement to set up a parallel thread team with one thread for each GPU.
- Each thread gets its own Gpu object, runs its own computational kernel, and retrieves its own kernel's result.
- Reduce the per-thread results together using reduction variables with the multithreaded reduction pattern.
- Override the `coresRequired()` method to specify that the task requires one core. All the parallel team threads will share this core.
- Override the `gpusRequired()` method to specify that the task requires all the GPUs on the node.

Chapter 27
GPU Sequential Dependencies

Recall the *N*-body zombie program from Chapter 8. The sequential version was based on a triply-nested loop, as shown in the pseudocode below. (x_i, y_i) is the current position of zombie *i;* (vx_i, vy_i) is the net velocity of zombie *i;* (vx_{ij}, vy_{ij}) is the velocity of zombie *i* relative to zombie *j;* and ($nextx_i$, $nexty_i$) is the next position of zombie *i.* The first loop goes over the time steps; the second loop goes over the zombies and calculates each zombie's next position; the third loop also goes over the zombies and accumulates the velocity relative to every other zombie. Δ is the total absolute distance all the zombies moved during the time step; when this falls below a threshold ε, the zombies have reached equilibrium.

```
Initialize zombie positions
Repeat: (time step loop)
  For i = 0 to N – 1:
    (vxi, vyi) ← (0, 0)
    For j = 0 to N – 1, j ≠ i:
      Compute (vxij, vyij) using equations (7.1), (7.2), and (7.3)
      (vxi, vyi) ← (vxi + vxij, vyi + vyij)
    (nextxi, nextyi) ← (xi + vxi · dt, yi + vyi · dt)
  Replace current zombie positions with next zombie positions
  If Δ < ε:
    Exit time step loop
```

To make the multicore parallel version of the zombie program, I noted that the time step loop has sequential dependencies; I cannot calculate the next time step until I've finished calculating the previous time step; so the outer loop had to remain a regular, non-parallel loop. But the second loop does not have sequential dependencies; the next position of one zombie does not depend on the next position of any other zombie; so the second loop could become a parallel loop, calculating all the zombies' next positions in parallel:

```
Initialize zombie positions
Repeat: (time step loop)
  Parallel for i = 0 to N – 1:
    (vxi, vyi) ← (0, 0)
    For j = 0 to N – 1, j ≠ i:
      Compute (vxij, vyij) using equations (7.1), (7.2), and (7.3)
      (vxi, vyi) ← (vxi + vxij, vyi + vyij)
    (nextxi, nextyi) ← (xi + vxi · dt, yi + vyi · dt)
  Replace current zombie positions with next zombie positions
  If Δ < ε:
    Exit time step loop
```

Now let's make a GPU parallel version of the zombie program. The time step loop still has to remain a regular, non-parallel loop. But the second loop can be done in parallel, so I will do that part on the GPU. The GPU computational kernel calculates the zombies' next positions and Δ, given the zombies' current positions, for one time step. The time step loop runs on the CPU and executes the kernel repeatedly, updates the positions, and checks for convergence:

```
Initialize zombie positions
Repeat: (time step loop)
  Execute GPU computational kernel
  Replace current zombie positions with next zombie positions
  If Δ < ε:
    Exit time step loop
```

The GPU kernel has to do this, expressed as sequential pseudocode:

```
For i = 0 to N – 1:
  (vxi, vyi) ← (0, 0)
  For j = 0 to N – 1, j ≠ i:
    Compute (vxij, vyij) using equations (7.1), (7.2), and (7.3)
    (vxi, vyi) ← (vxi + vxij, vyi + vyij)
  (nextxi, nextyi) ← (xi + vxi · dt, yi + vyi · dt)
```

How shall I parallelize this pseudocode to run on the GPU? Recall that the GPU kernel has two levels of parallelism: the kernel can run multiple blocks in parallel on the GPU's multiprocessors, and each block can run multiple threads in parallel on the multiprocessor's cores. So a natural way to partition the computation is to parallelize the outer loop iterations across the blocks, and to parallelize the inner loop iterations across the threads in a block. The kernel will have a one-dimensional grid of one-dimensional blocks. Each block will compute the next position of one zombie. The threads in the block will compute the zombie's velocity relative to every other zombie. These relative velocities must be added together to get the zombie's net velocity; this calls for a parallel sum-reduce within the block, like the one in the GPU parallel π estimating program in Chapter 26. The kernel must also calculate Δ, the total absolute distance all the zombies moved. Each block can calculate a partial Δ for its own zombie. The blocks' Δ values must then all be added together; this is done with atomic operations on a variable in global memory, again like the GPU parallel π estimating program.

Turning to the GPU kernel code (Listing 27.1), line 2 declares that there will be 256 threads per block. The kernel's inner loop iterations will be partitioned among this many threads. Why 256 threads? Why not have 1024 threads, the maximum allowed in a compute-capability-2.0 GPU? I discuss this question in the "Under the Hood" section below.

Line 5 declares the `devDelta` variable in the GPU's global memory. The kernel will store the total Δ in this variable. Because it is located in global memory, all threads in all blocks can access it.

Lines 8–9 declare arrays of X and Y velocity values, one element for each thread in the block, located in the block's fast shared memory. These arrays will be used in the parallel sum-reduce within the block to compute the block's zombie's net velocity. (There is one element for each *thread*, not one element for each zombie.)

Lines 12–27 define a subroutine to do atomic addition on a variable of type `double`. The subroutine adds the given `value` to the given variable `v` and stores the result back in `v`. Furthermore, the addition is done atomically, so multiple threads adding values to the variable will not interfere with each other. I had to write my own subroutine because a compute-capability-2.0 GPU does not have an atomic add operation for type `double`. See the "Under the Hood" section below for an explanation of how the `atomicAdd` subroutine works.

The `timeStep` kernel function begins on line 34. The first four arguments are pointers to arrays of type `double` for the zombies' current X and Y positions `xpos` and `ypos` and the next X and Y positions `xnext` and `ynext`. These have to be passed as arguments, and cannot be declared as variables like `devDelta`, because the number of zombies is not known at compile time; rather, the user specifies the number of zombies when running the program. Indeed, the fifth argument is the number of zombies `N`, which is the length of the arrays. The sixth and seventh arguments `G` and `L` are part of the formula for the zombie's relative velocity. The eighth argument `dt` is the time step size.

When the kernel function is called, the `xpos` and `ypos` arrays contain the kernel's inputs, namely all the zombies' current positions for the current time step. The `G`, `L`, and `dt` arguments are also inputs. The `xnext` and `ynext` arrays will hold the kernel's outputs, namely all the zombie's next positions after the current time step. The `devDelta` variable is also an output.

The kernel function code proper begins by initializing several variables on lines 44–49. The variable `i` is the index of this block's zombie; because each block is calculating the next position of one particular zombie, `i` is just the index of the block within the grid, from 0 to `N` – 1. The variable `j` is the rank of the thread within the block, from 0 to `NT` – 1 (0 to 255).

When the kernel function is executing, the outer loop iterations over the zombies have already been partitioned—each outer loop iteration is being calculated by one of the blocks in the grid, in parallel with all the other blocks. Now, in the kernel function, the inner loop iterations have to be partitioned among the threads in the block. The for loop starting on line 54 partitions the inner loop iterations using, in effect, a leapfrog schedule, similar to the one in the GPU parallel π estimating program. The loop index `k` starts at

```
1  // Number of threads per block.
2  #define NT 256
3
4  // Variables in global memory.
5  __device__ double devDelta;
6
7  // Per-thread variables in shared memory.
8  __shared__ double shrVx [NT];
9  __shared__ double shrVy [NT];
10
11 // Atomically set double variable v to the sum of itself and value.
12 __device__ void atomicAdd
13    (double *v,
14     double value)
15    {
16    double oldval, newval;
17    do
18       {
19       oldval = *v;
20       newval = oldval + value;
21       }
22    while (atomicCAS
23       ((unsigned long long int *)v,
24        __double_as_longlong (oldval),
25        __double_as_longlong (newval))
26          != __double_as_longlong (oldval));
27    }
28
29 // Device kernel to update zombie positions after one time step.
30 // Called with a one-dimensional grid of one-dimensional blocks,
31 // N blocks, NT threads per block. N = number of zombies. Each block
32 // updates one zombie. Each thread within a block computes the
33 // velocity with respect to one other zombie.
34 extern "C" __global__ void timeStep
35    (double *xpos,
36     double *ypos,
37     double *xnext,
38     double *ynext,
39     int N,
40     double G,
41     double L,
42     double dt)
43    {
44    int i = blockIdx.x;      // Index of this block's zombie
45    int j = threadIdx.x;     // Index of this thread within block
46    double xpos_i = xpos[i]; // This zombie's current X position
47    double ypos_i = ypos[i]; // This zombie's current X position
48    double vx = 0.0;         // This zombie's X velocity
49    double vy = 0.0;         // This zombie's Y velocity
50    int k;
51    double dx, dy, d, v;
52
53    // Compute and accumulate velocity w.r.t. every other zombie.
54    for (k = j; k < N; k += NT)
55       {
56       if (k == i) continue;
57       dx = xpos[k] - xpos_i;
58       dy = ypos[k] - ypos_i;
```

Listing 27.1. ZombieGpu.cu (part 1)

the thread's own rank and increases by the number of threads in the block (256). For example, suppose there are 1000 zombies. In thread rank 0, the loop index `k` will be 0, 256, 512, 768; in thread rank 1, the loop index `k` will be 1, 257, 513, 769; and so on. In this manner, all inner loop indexes from 0 to 999 will be calculated by some thread in the block. Each thread will perform the same, or nearly the same, number of inner loop iterations; and because each inner loop iteration takes the same amount of time, the load is balanced. The inner loop body calculates the relative velocity between zombie `i` (the block's zombie) and zombie `k` (one of the other zombies) and accumulates the relative velocity into the thread's local variables `vx` and `vy`. Because each thread is updating its own local variables, the threads do not need to synchronize with each other at this point.

Once the loop finishes, each thread in the block has calculated a partial net velocity. These partial net velocities must be added together to get the zombie's total net velocity. This is accomplished using a shared memory parallel reduction tree (lines 66–77). The reduction tree code is the same as in the GPU parallel π estimating program, except that we are reducing two quantities at the same time, namely `vx` and `vy`. When the reduction finishes, `shrVx[0]` and `shrVy[0]` end up holding the X and Y components of the zombie's net velocity.

The kernel function can now calculate the zombie's next position, as well as the zombie's partial Δ. These calculations must be done by a single thread; so thread rank 0 executes lines 82–93, and the other threads do nothing. Lines 83–84 retrieve the zombie's net velocity from shared memory; lines 87–88 calculate the zombie's net change in position; lines 89–90 calculate the zombie's next position and store it in the `xnext` and `ynext` arrays in GPU global memory, at the zombie's own index. Because the thread 0s in all the blocks are updating different elements in `xnext` and `ynext`, the threads do not need to synchronize with each other at this point. Lastly, line 93 adds the zombie's absolute change in position to the `devDelta` variable in GPU global memory, using the `atomicAdd` subroutine defined earlier to synchronize the threads.

When the GPU kernel finishes, its outputs have been stored in GPU global memory. The `xnext` and `ynext` arrays contain all the zombies' next positions after the current time step. The `devDelta` variable contains the total absolute change in position of all the zombies for the current time step.

Having written the C kernel, I'm now able to write the Java main program, class ZombieGpu (Listing 27.2). The program declares variables to hold the zombies' current X and Y positions (lines 25–26) and their next X and Y positions (lines 29–30). These variables are of type GpuDoubleArray, which provide arrays of type `double` allocated in the GPU and mirrored in the CPU. The GPU kernel takes the `xpos` and `ypos` arrays as inputs, and calculates the `xnext` and `ynext` arrays as outputs. The main program also de-

```
      d = sqrt(dx*dx + dy*dy);
      v = G*exp(-d/L) - exp(-d);
      vx += v*dx/d;
      vy += v*dy/d;
      }

   // Compute net velocity via shared memory parallel reduction.
   shrVx[j] = vx;
   shrVy[j] = vy;
   __syncthreads();
   for (k = NT/2; k > 0; k >>= 1)
      {
      if (j < k)
         {
         shrVx[j] += shrVx[j+k];
         shrVy[j] += shrVy[j+k];
         }
      __syncthreads();
      }

   // Single threaded section.
   if (j == 0)
      {
      // Get net velocity.
      vx = shrVx[0];
      vy = shrVy[0];

      // Move zombie in the direction of its velocity.
      dx = vx*dt;
      dy = vy*dt;
      xnext[i] = xpos_i + dx;
      ynext[i] = ypos_i + dy;

      // Accumulate position delta.
      atomicAdd (&devDelta, abs(dx) + abs(dy));
      }
   }
```

Listing 27.1. ZombieGpu.cu (part 2)

```
package edu.rit.gpu.example;
import edu.rit.gpu.CacheConfig;
import edu.rit.gpu.Gpu;
import edu.rit.gpu.GpuDoubleArray;
import edu.rit.gpu.GpuDoubleVbl;
import edu.rit.gpu.Kernel;
import edu.rit.gpu.Module;
import edu.rit.pj2.Task;
import edu.rit.util.Random;
public class ZombieGpu
   extends Task
   {
```

Listing 27.2. ZombieGpu.java (part 1)

clares a variable to hold Δ (line 33), which is another output calculated by the kernel. This variable is of type GpuDoubleVbl, which provides a single value of type `double` located in the GPU and mirrored in the CPU.

Next comes the kernel interface (lines 36–48). The interface declares the kernel method `timeStep()` and its arguments, which are the same as the kernel function's arguments in the GPU code. However, the `xpos`, `ypos`, `xnext`, and `ynext` arguments are declared to be Java type GpuDoubleArray rather than C type `double*`—that is, the same type as the corresponding variables on lines 25–30. (This is Java, not C; you can't do pointers in Java.)

The main program proper begins on line 51. After parsing the command line arguments, initializing the GPU object, verifying the compute capability, and loading the compiled kernel module, the program creates the four objects `xpos`, `ypos`, `xnext`, and `ynext` (lines 74–77). Each object contains, in its `item` field, an array of *N* elements of type double. Each array is also allocated storage for *N* elements of type `double` in the GPU's global memory. The program also creates the `delta` object (line 78). This object's `item` field, a value of type `double`, mirrors the `devDelta` variable in the GPU's global memory, which was declared in the kernel code.

Next the main program sets up the computational kernel (lines 81–84). The kernel is configured with a one-dimensional grid of one-dimensional blocks, with *N* blocks (one for each zombie) and 256 threads per block (as assumed by the kernel code). The kernel is also configured with more L1 cache and less shared memory. On my Nvidia Tesla C2075 GPUs, this configures each multiprocessor with 48 kilobytes of L1 cache and 16 kilobytes of shared memory. The shared memory holds the `shrVx` and `shrVy` arrays; each array has 256 elements of type `double`; a `double` value occupies 8 bytes; so the two arrays occupy 4 kilobytes; this fits within the 16 kilobyte limit. I want the L1 cache to be as large as possible because the threads will be reading the zombies' current positions from the `xpos` and `ypos` arrays in global memory, and I want these cached in L1 to speed up the memory accesses. The 48-kilobyte L1 cache can hold up to 3,072 zombies' X and Y coordinates without experiencing cache misses.

The program initializes the zombies' (x, y) coordinates to random values on the CPU (lines 88–93) and uploads these to the GPU (lines 94–95). The program also prints a snapshot of the zombies' initial positions (line 99).

The program now commences the outermost loop over the time steps (line 102). At each time step, the program reinitializes Δ to 0 and uploads it to the GPU (lines 105–106). The program then calls the `timeStep()` kernel method (line 107), which executes the computational kernel. When the kernel method returns, the `xnext` and `ynext` arrays on the GPU contain the zombies' calculated next positions, and the `devDelta` variable on the GPU contains Δ.

Note that the GPU kernel is being executed repeatedly, once for every iteration of the outer time step loop. The variables located in GPU memory—

```
13    // Command line arguments.
14    long seed;
15    int N;
16    double W;
17    double G;
18    double L;
19    double dt;
20    double eps;
21    int steps;
22    int snap;
23
24    // Current body positions.
25    GpuDoubleArray xpos;
26    GpuDoubleArray ypos;
27
28    // Next body positions.
29    GpuDoubleArray xnext;
30    GpuDoubleArray ynext;
31
32    // For detecting convergence.
33    GpuDoubleVbl delta;
34
35    // Kernel function interface.
36    private static interface ZombieKernel
37       extends Kernel
38       {
39       public void timeStep
40          (GpuDoubleArray xpos,
41           GpuDoubleArray ypos,
42           GpuDoubleArray xnext,
43           GpuDoubleArray ynext,
44           int N,
45           double G,
46           double L,
47           double dt);
48       }
49
50    // Task main program.
51    public void main
52       (String[] args)
53       throws Exception
54       {
55       // Parse command line arguments.
56       if (args.length != 9) usage();
57       seed = Long.parseLong (args[0]);
58       N = Integer.parseInt (args[1]);
59       W = Double.parseDouble (args[2]);
60       G = Double.parseDouble (args[3]);
61       L = Double.parseDouble (args[4]);
62       dt = Double.parseDouble (args[5]);
63       eps = Double.parseDouble (args[6]);
64       steps = Integer.parseInt (args[7]);
65       snap = Integer.parseInt (args[8]);
66
67       // Initialize GPU.
68       Gpu gpu = Gpu.gpu();
69       gpu.ensureComputeCapability (2, 0);
70
```

Listing 27.2. ZombieGpu.java (part 2)

`xpos`, `ypos`, `xnext`, and `ynext`—retain their values in between kernel executions, which is essential for this program to compute the correct results.

To go on to the next time step, the zombies' next positions need to become the zombies' new current positions. There's no need to copy the elements from one array to the other, and there's not even any need to download the array elements from the GPU to the CPU. All that's necessary is to swap the array object references on the CPU (lines 113–115). The program downloads Δ from the GPU, checks for convergence, and exits the time step loop if so (lines 120–123). If the program has not reached convergence, the program prints out a snapshot of the zombies' positions every `snap` time steps, where `snap` was specified on the command line (lines 126–127); and the program repeats the time step loop. One final snapshot takes place when the program is finished (line 131).

The `snapshot()` method (lines 135–143) downloads the current zombie positions into the `xpos` and `ypos` objects on the CPU, and prints those. Taking a snapshot is the only time the zombies' positions need to be transferred from the GPU to the CPU. This happens once at the beginning of the program, once at the end, and every `snap` time steps in between.

To compare the CPU's performance to the GPU's performance on the zombie simulation, I ran the ZombieSeq program and the ZombieGpu program on the `kraken` machine, using commands like this:

```
$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSeq \
  142857 100 5.00 0.5 10 0.00001 0.001 0 0
$ java pj2 debug=makespan edu.rit.gpu.example.ZombieGpu \
  142857 100 5.00 0.5 10 0.00001 0.001 0 0
```

I ran the programs for various numbers of zombies N and various initial areas W (the second and third command line arguments). Here are the running times T in milliseconds I observed, as well as the number of time steps needed to reach convergence:

N	W	Steps	CPU T	GPU T	Ratio
100	5.00	15869	12034	1704	7.1
200	7.07	13043	39568	2187	18.1
500	11.18	10186	192903	3615	53.4
1000	15.81	9308	706896	7947	90.0
2000	22.36	9595	2880873	27286	105.6

For a large enough N-body simulation—a problem size for which I might really want to take advantage of the GPU's computational power—the GPU is over 100 times faster than the CPU. Different computations, running on different CPU and GPU hardware, would experience different time ratios. Still, in my experience GPU programs can be one to two orders of magnitude faster than equivalent CPU programs.

```
71          // Set up GPU variables.
72          Module module = gpu.getModule
73             ("edu/rit/gpu/example/ZombieGpu.cubin");
74          xpos = gpu.getDoubleArray (N);
75          ypos = gpu.getDoubleArray (N);
76          xnext = gpu.getDoubleArray (N);
77          ynext = gpu.getDoubleArray (N);
78          delta = module.getDoubleVbl ("devDelta");
79
80          // Set up GPU kernel.
81          ZombieKernel kernel = module.getKernel (ZombieKernel.class);
82          kernel.setBlockDim (256);
83          kernel.setGridDim (N);
84          kernel.setCacheConfig (CacheConfig.CU_FUNC_CACHE_PREFER_L1);
85
86          // Set zombies' initial (x,y) coordinates at random in a WxW
87          // square region.
88          Random prng = new Random (seed);
89          for (int i = 0; i < N; ++ i)
90             {
91             xpos.item[i] = prng.nextDouble()*W;
92             ypos.item[i] = prng.nextDouble()*W;
93             }
94          xpos.hostToDev();
95          ypos.hostToDev();
96
97          // Snapshot all bodies' initial positions.
98          int t = 0;
99          snapshot (t);
100
101         // Do repeated time steps.
102         for (;;)
103            {
104            // Do one time step.
105            delta.item = 0.0;
106            delta.hostToDev();
107            kernel.timeStep (xpos, ypos, xnext, ynext, N, G, L, dt);
108
109            // Advance to next time step.
110            ++ t;
111
112            // Update positions.
113            GpuDoubleArray tmp;
114            tmp = xpos; xpos = xnext; xnext = tmp;
115            tmp = ypos; ypos = ynext; ynext = tmp;
116
117            // Stop when position delta is less than convergence
118            // threshold or when the specified number of time steps
119            // have occurred.
120            delta.devToHost();
121            if ((steps == 0 && delta.item < eps) ||
122                  (steps != 0 && t == steps))
123               break;
124
125            // Snapshot all bodies' positions every <snap> time steps.
126            if (snap > 0 && (t % snap) == 0)
127               snapshot (t);
128            }
```

Listing 27.2. ZombieGpu.java (part 3)

Under the Hood

Here's how I decided to use `NT` = 256 threads per block in the GPU kernel. After writing the complete program, I ran the program on a typical problem with `NT` defined to be different powers of 2, namely 32, 64, 128, 256, 512, and 1024. I picked powers of 2 to simplify the parallel reduction code in the kernel function. I measured the program's running time for each value of `NT`, and picked the `NT` value that yielded the smallest running time. Fewer than 256 or more than 256 threads per block turned out to yield larger running times. There's no simple explanation for this. In my experience, rather than trying to predict the optimum value of `NT` ahead of time, it's easier to measure the program's actual performance and adjust `NT` accordingly.

The ZombieGpu kernel included the `atomicAdd()` function for a variable of type `double` because CUDA compute capability 2.0 does not have that function built in, so I had to write it myself. I used a function that is built in, called `atomicCAS()`, which stands for "atomic compare and swap."

The atomic CAS operation takes three arguments: a variable, an old value, and a new value. The atomic CAS operation also uses the variable's current value. The operation does the following: If the variable's current value equals the old value, then set the variable's value to the new value and return the variable's previous value; otherwise leave the variable's value unchanged and return the variable's current value. Furthermore, the operation is done atomically; when one thread is in the middle of an atomic CAS on a variable, it will not be interrupted by any other thread trying to do an atomic CAS on that variable. Stated another way, atomic CAS *compares* the variable to the old value, and if they are the same, *swaps* the variable with the new value. Atomic CAS is typically implemented in hardware.

Here is pseudocode for atomically updating a variable using the atomic CAS operation:

```
1  Do:
2      old value = current value of the variable
3      new value = value to be stored in the variable
4  While atomicCAS (variable, old value, new value) ≠ old value
```

Step 2 queries the variable's current value. Step 3 computes the variable's updated value; usually the updated value depends on the current value. Step 4 *tries* to atomically set the variable to the new value. The attempt will not succeed, however, if at Step 4 the variable's value is not the same as the old value retrieved at Step 2. Why might the value not be the same? Because another thread might have changed the variable's value between the time this thread did Step 2 and the time this thread did Step 4. In that case, the value returned by the atomic CAS will not be the same as the old value the thread is expecting; so the thread stays in the do-while loop, re-queries the vari-

```
129
130         // Snapshot all bodies' final positions.
131         snapshot (t);
132         }
133
134      // Snapshot all bodies' positions.
135      private void snapshot
136         (int t)
137         {
138         xpos.devToHost();
139         ypos.devToHost();
140         for (int i = 0; i < N; ++ i)
141            System.out.printf ("%d\t%d\t%g\t%g%n",
142               t, i, xpos.item[i], ypos.item[i]);
143         }
144
145      // Print a usage message and exit.
146      private static void usage()
147         {
148         System.err.println ("Usage: java pj2 " +
149            "edu.rit.pj2.example.ZombieGpu <seed> <N> <W> <G> <L> " +
150            "<dt> <eps> <steps> <snap>");
151         System.err.println ("<seed> = Random seed");
152         System.err.println ("<N> = Number of bodies");
153         System.err.println ("<W> = Region size");
154         System.err.println ("<G> = Attraction factor");
155         System.err.println ("<L> = Attraction length scale");
156         System.err.println ("<dt> = Time step size");
157         System.err.println ("<eps> = Convergence threshold");
158         System.err.println ("<steps> = Number of time steps (0 = " +
159            "until convergence)");
160         System.err.println ("<snap> = Snapshot interval (0 = none)");
161         throw new IllegalArgumentException();
162         }
163
164      // Specify that this task requires one core.
165      protected static int coresRequired()
166         {
167         return 1;
168         }
169
170      // Specify that this task requires one GPU accelerator.
171      protected static int gpusRequired()
172         {
173         return 1;
174         }
175   }
```

Listing 27.2. ZombieGpu.java (part 4)

able's value, recomputes the updated value, and retries the atomic CAS. The thread might stay in this loop for several iterations. When the update succeeds—that is, when the atomic CAS returns the old value the thread is expecting—the thread gets out of the loop and goes on.

Using atomic CAS, I can atomically update a variable any way I please. But there's still a problem: CUDA compute capability 2.0 has atomic CAS operations for integer variables—types `int`, `unsigned int`, `long long int`, and `unsigned long long int`. CUDA compute capability 2.0 does not have atomic CAS operations for floating point variables—types `float` and `double`.

To deal with this problem, I'll trick the atomic CAS operation into thinking that a value of type `double` is really a value of type `unsigned long long int`. I can get away with this because both types occupy the same amount of memory, eight bytes. On line 23, I cast the variable pointer from `double*` to `unsigned long long int*`; this doesn't change the pointer, but it makes it look like the correct type for the atomic CAS. On lines 24–26, I run the double values through the special `__double_as_long_long()` CUDA function. This special CUDA function *does not do a type conversion*—it does not convert a floating point value to an integer value, which would alter the value's bit pattern. Rather, this special CUDA function *leaves the value's bit pattern intact* and merely reinterprets that bit pattern as being a long integer, which is the correct type for the atomic CAS. The atomic CAS then *manipulates those bit patterns:* it compares the variable's current value's bit pattern to the old value's bit pattern, and if they are the same, swaps the variable with the new value's bit pattern and returns the variable's previous value's bit pattern. The result is the same as if the atomic CAS operation had operated on the double values directly.

The ZombieGpu main program mirrors the `xpos`, `ypos`, `xnext`, `ynext`, and `delta` variables between the CPU and the GPU. In Chapter 25 I pointed out that mirrored variables are not copied automatically between the CPU and the GPU; rather, you have to call the `hostToDev()` and `devToHost()` methods explicitly. Now we can see why. In the zombie program, there's no need to transfer the `xpos` and `ypos` arrays from the GPU to the CPU at every time step. The arrays need to be transferred only when the program is printing a snapshot of the zombies' positions. Refraining from transferring the arrays at other times reduces the program's sequential overhead. The `xnext` and `ynext` arrays never need to be transferred at all; they are used solely in the kernel. On the other hand, the `delta` variable *does* need to be transferred at each time step, so the CPU can check the convergence criterion. But because `delta` is just a single `double` value, the overhead is minimal.

Points to Remember

- For a program with sequential dependencies, consider doing the sequential loop in the CPU main program and the parallelizable loop or loops in the GPU kernel.
- The GPU has two levels of parallelism: the blocks within the grid, and the threads within the block. Keep this flexibility in mind when parallelizing a program for the GPU.
- If a section of code in the GPU kernel function must be performed by only one thread in the block, enclose the code section in an if statement that checks whether the thread index is 0.
- When mirroring variables on the CPU and the GPU, transfer the variables from one side to the other only when necessary.
- Determine the kernel's number of threads per block `NT` empirically: measure the program's running time for different `NT` values, and pick the one that yields the smallest running time.
- Use the `atomicCAS()` function to atomically update a variable, if CUDA does not support the desired updating operation.
- Use the `__double_as_long_long()` function, and similar functions (see the CUDA documentation), to treat a bit pattern of one type as a bit pattern of another type for purposes of atomic CAS.

Chapter 28
Objects on the GPU

The GPU parallel programs we've studied so far all used primitive types to represent the data being calculated. The outer product program used arrays and matrices of type `double`. The π estimating programs used counters of type `long`. The zombie program used arrays of type `double` for the zombie positions. In the Java main programs, I used instances of classes such as GpuLongVbl, GpuDoubleArray, and GpuDoubleMatrix to mirror the GPU variables on the CPU. Each such instance had an `item` field of the appropriate Java type, such as `long`, `double[]`, or `double[][]`. I called the `hostToDev()` and `devToHost()` methods on those instances to transfer the data between the CPU and the GPU.

For some programs, however, the appropriate type to represent the data is an object, not a primitive type. Consider the zombie program. The program needs to store the zombies' positions. Viewed abstractly, a zombie's position is a two-dimensional vector consisting of the X coordinate and the Y coordinate. In an object-oriented language such as Java, the proper way to represent a vector is a class with two fields:

```
public class Vector
   {
   public double x;
   public double y;
   }
```

I can also define methods to do operations on vectors, such as add two vectors, subtract two vectors, determine the magnitude of a vector, and so on. I can then use the vector class in other data structures, such as an array of vectors holding positions for multiple zombies. Manipulating the positions by calling vector methods is easier to read, easier to maintain, and less defect prone than the non-object-oriented approach I took in the previous chapter, where the zombies' positions were stored in two separate arrays of X coordinates and Y coordinates.

A non-object-oriented language like C does not have objects. C does, however, have structures. The proper way to represent a vector in C is a structure with two fields:

```
typedef struct
   {
   double x;
   double y;
   }
   vector_t;
```

Apart from immaterial differences in syntax, the Java class and the C structure are really the same data type. When I write a GPU parallel program involving vectors, I want to write the Java main program code using the Vector class, and I want to write the C kernel code using the `vector_t` structure. Then I want to transfer instances of the Vector class from the CPU to the

```
1  // Number of threads per block.
2  #define NT 256
3
4  // Structure for a 2-D vector.
5  typedef struct
6     {
7     double x;
8     double y;
9     }
10    vector_t;
11
12 // Vector addition; a = b + c. Returns a.
13 __device__ vector_t *vectorAdd
14    (vector_t *a,
15     vector_t *b,
16     vector_t *c)
17    {
18    a->x = b->x + c->x;
19    a->y = b->y + c->y;
20    return a;
21    }
22
23 // Vector subtraction; a = b - c. Returns a.
24 __device__ vector_t *vectorSubtract
25    (vector_t *a,
26     vector_t *b,
27     vector_t *c)
28    {
29    a->x = b->x - c->x;
30    a->y = b->y - c->y;
31    return a;
32    }
33
34 // Scalar product; a = a*b. Returns a.
35 __device__ vector_t *scalarProduct
36    (vector_t *a,
37     double b)
38    {
39    a->x *= b;
40    a->y *= b;
41    return a;
42    }
43
44 // Returns the magnitude of a.
45 __device__ double vectorMagnitude
46    (vector_t *a)
47    {
48    return sqrt (a->x*a->x + a->y*a->y);
49    }
50
51 // Variables in global memory.
52 __device__ double devDelta;
53
54 // Per-thread variables in shared memory.
55 __shared__ vector_t shrVel [NT];
56
```

Listing 28.1. ZombieGpu2.cu (part 1)

GPU and have them end up as `vector_t` structures on the GPU; and vice versa when transferring in the other direction.

The Parallel Java 2 Library has exactly this capability. To illustrate it, I'm going to develop a second version of the GPU parallel zombie program, this time using vector objects and structures.

Listing 28.1 is the C code for the GPU kernel. It begins by declaring a `vector_t` structure for a two-dimensional vector (lines 5–10). Because C is not an object oriented programming language, I can't define *methods* on this structure. However, I can define *functions* that act like methods. To define a "method," I simply define a function whose first argument, `a`, is a pointer to the structure I want the function to manipulate. I can then access the structure's fields using syntax like `a->x` and `a->y`. Lines 12–49 define several such vector methods to do vector addition, vector subtraction, scalar multiplication, and vector magnitude. (If I were writing the kernel in C++ instead of C, I could define a class with real methods; but I'm going to stick with C.)

Each of the vector functions begins with the special CUDA keyword `__device__` ("underscore underscore device underscore underscore"). This signals the CUDA compiler that the function is not a kernel function, but just a subroutine to be compiled into GPU code. (A kernel function begins with `__global__` rather than `__device__`.)

The rest of the kernel is functionally the same as the ZombieGpu kernel in Chapter 27, and I'm not going to describe the new kernel in detail. I will point out where the new kernel uses the vector structure instead of separate X and Y coordinate arrays. Line 55 declares an array of velocity vectors in the block's shared memory; this array is used in the sum-reduction that computes the zombie's net velocity. The `timeStep()` kernel function's first two arguments are pointers to arrays of vectors holding the current zombie positions and the next zombie positions (lines 81–82). In the kernel function code, the velocity computations are carried out using the vector methods defined earlier (lines 100–103). Compare this code to the equivalent code in Chapter 27, and see if you agree that with this code it's easier to understand what's going on. The sum-reduction that computes the zombie's net velocity likewise uses one of the vector methods defined earlier (lines 107–114), as does the single-threaded code section that computes the zombie's next position after the time step (lines 117–127).

The Java main program, class ZombieGpu2 (Listing 28.2), begins by defining a Java class Vector that corresponds to the kernel's `vector_t` structure (line 16). Class Vector is a subclass of class edu.rit.gpu.Struct (line 17). The Struct superclass declares methods that the Parallel Java 2 Library uses to transfer instances of class Vector on the CPU to instances of structure `vector_t` on the GPU and vice versa. Class Vector declares the same fields as structure `vector_t`, namely `x` and `y` of type `double` (lines 19–20), as well as a constructor to initialize them (lines 23–29). Next come three methods that

```
57 // Atomically set double variable v to the sum of itself and value.
58 __device__ void atomicAdd
59    (double *v,
60     double value)
61    {
62    double oldval, newval;
63    do
64       {
65       oldval = *v;
66       newval = oldval + value;
67       }
68    while (atomicCAS
69       ((unsigned long long int *)v,
70        __double_as_longlong (oldval),
71        __double_as_longlong (newval))
72          != __double_as_longlong (oldval));
73    }
74
75 // Device kernel to update zombie positions after one time step.
76 // Called with a one-dimensional grid of one-dimensional blocks, N
77 // blocks, NT threads per block. N = number of zombies. Each block
78 // updates one zombie. Each thread within a block computes the
79 // velocity with respect to one other zombie.
80 extern "C" __global__ void timeStep
81    (vector_t *pos,
82     vector_t *next,
83     int N,
84     double G,
85     double L,
86     double dt)
87    {
88    int i = blockIdx.x;        // Index of this block's zombie
89    int j = threadIdx.x;       // Index of this thread within block
90    vector_t pos_i = pos[i];   // This zombie's current position
91    vector_t vel = {0.0, 0.0}; // This zombie's velocity
92    int k;
93    vector_t posdiff;
94    double d, v;
95
96    // Compute and accumulate velocity w.r.t. every other zombie.
97    for (k = j; k < N; k += NT)
98       {
99       if (k == i) continue;
100      vectorSubtract (&posdiff, &pos[k], &pos_i);
101      d = vectorMagnitude (&posdiff);
102      v = G*exp(-d/L) - exp(-d);
103      vectorAdd (&vel, &vel, scalarProduct (&posdiff, v/d));
104      }
105
106   // Compute net velocity via shared memory parallel reduction.
107   shrVel[j] = vel;
108   __syncthreads();
109   for (k = NT/2; k > 0; k >>= 1)
110      {
111      if (j < k)
112         vectorAdd (&shrVel[j], &shrVel[j], &shrVel[j+k]);
113      __syncthreads();
114      }
```

Listing 28.1. ZombieGpu2.cu (part 2)

are declared in the Struct superclass. The static `sizeof()` method (lines 32–35) returns the amount of storage occupied by one instance of the `vertex_t` structure, namely 16 bytes—two fields of type `double`, each occupying 8 bytes. This method is static because the Library must be able to determine the structure's size before any instances of the class have been created. The `toStruct()` method (lines 39–44) is called by the Library when an instance of class Vector is being transferred from the CPU to the GPU. The `toStruct()` method must write, into the given byte buffer, the bytes of the `vector_t` structure exactly as they would appear in the GPU's memory. In general, this is done by calling methods on the byte buffer, such as `putInt()`, `putDouble()`, and so on, for each of the fields in the class; see the "Under the Hood" section below for further discussion. Here, the `toStruct()` method puts the `x` and `y` fields into the byte buffer. The `fromStruct()` method (lines 48–53) is called by the Library when an instance of class Vector is being transferred from the GPU to the CPU. The `fromStruct()` method must read, from the given byte buffer, the bytes of the `vector_t` structure exactly as they would appear in the GPU's memory. In general, this is done by calling methods on the byte buffer, such as `getInt()`, `getDouble()`, and so on, for each of the fields in the class; see the "Under the Hood" section below for further discussion. Here, the `fromStruct()` method gets the `x` and `y` fields from the byte buffer.

The kernel interface is declared in lines 77–87. As always, the Java argument types correspond to the C argument types: Java type `int` for C type `int`; Java type `double` for C type `double`. The current position array and next position array arguments are of C type `vector_t*`; in Java these become type GpuStructArray. Class GpuStructArray provides an array of structures on the GPU mirrored as an array of objects on the CPU. Class GpuStructArray is a generic class, here specified with the generic type parameter <Vector>, indicating that it is an array of Vector objects.

The task main program proper begins on line 90. It is functionally the same as the task main program in Chapter 27, except it uses the Vector class. Lines 113–114 create arrays of `vector_t` structures on the GPU to hold the zombies' current and next positions. The arrays are mirrored on the CPU in the `item` fields of the `pos` and `next` variables. These variables are of type GpuStructArray<Vector>, the same type as the kernel function arguments. Lines 125–131 initialize the `pos` array elements to random (*x*, *y*) coordinates and initialize the `next` array elements to (0, 0). The initial `pos` array is then transferred from the CPU to the GPU (line 132). Similarly, the `snapshot()` method transfers the `pos` array from the GPU to the CPU (line 174) and prints the position vectors.

To compare the CPU's performance to the GPU's performance on the zombie simulation using structures, I ran the ZombieSeq program and the

```
115
116    // Single threaded section.
117    if (j == 0)
118       {
119       // Get net velocity.
120       vel = shrVel[0];
121
122       // Move zombie in the direction of its velocity.
123       vectorAdd (&next[i], &pos_i, scalarProduct (&vel, dt));
124
125       // Accumulate position delta.
126       atomicAdd (&devDelta, abs(vel.x) + abs(vel.y));
127       }
128    }
```

Listing 28.1. ZombieGpu2.cu (part 3)

```
 1 package edu.rit.gpu.example;
 2 import edu.rit.gpu.CacheConfig;
 3 import edu.rit.gpu.Gpu;
 4 import edu.rit.gpu.GpuDoubleVbl;
 5 import edu.rit.gpu.GpuStructArray;
 6 import edu.rit.gpu.Kernel;
 7 import edu.rit.gpu.Module;
 8 import edu.rit.gpu.Struct;
 9 import edu.rit.pj2.Task;
10 import edu.rit.util.Random;
11 import java.nio.ByteBuffer;
12 public class ZombieGpu2
13    extends Task
14    {
15   // Structure for a 2-D vector.
16    private static class Vector
17       extends Struct
18       {
19       public double x;
20       public double y;
21
22       // Construct a new vector.
23       public Vector
24          (double x,
25           double y)
26          {
27          this.x = x;
28          this.y = y;
29          }
30
31       // Returns the size in bytes of the C struct.
32       public static long sizeof()
33          {
34          return 16;
35          }
36
```

Listing 28.2. ZombieGpu2.java (part 1)

ZombieGpu2 program on the `kraken` machine, using commands like this:

```
$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSeq \
  142857 100 5.00 0.5 10 0.00001 0.001 0 0
$ java pj2 debug=makespan edu.rit.gpu.example.ZombieGpu2 \
  142857 100 5.00 0.5 10 0.00001 0.001 0 0
```

I ran the programs for various numbers of zombies *N* and various initial areas *W* (the second and third command line arguments). Here are the running times *T* in milliseconds I observed, as well as the number of time steps needed to reach convergence:

N	*W*	Steps	CPU *T*	GPU *T*	Ratio
100	5.00	15869	12034	1743	6.9
200	7.07	13043	39568	2211	17.9
500	11.18	10186	192903	3485	55.4
1000	15.81	9308	706896	7228	97.8
2000	22.36	9595	2880873	23602	122.1

The GPU program is over 120 times faster than the CPU program for a large enough problem size.

In fact, the GPU program using vector structures (ZombieGpu2) is even faster than the GPU program using separate X and Y coordinate arrays (ZombieGpu). Here are the running times *T* in milliseconds for the ZombieGpu and ZombieGpu2 programs:

N	*W*	Steps	ZombieGpu *T*	ZombieGpu2 *T*
100	5.00	15869	1704	1743
200	7.07	13043	2187	2211
500	11.18	10186	3615	3485
1000	15.81	9308	7947	7228
2000	22.36	9595	27286	23602

With 2000 zombies, the ZombieGpu2 program is about 14 percent faster than the ZombieGpu program. Why? Most likely because the ZombieGpu2 program is more “cache friendly.” The X and Y coordinates of each vector structure are stored in adjacent memory locations. When the kernel code pulls a vector’s X coordinate from GPU global memory, a number of adjacent memory locations also get pulled into the L2 and L1 caches—possibly including the vector’s Y coordinate. Then when the kernel code accesses the vector’s Y coordinate, much of the time the Y coordinate is already in the cache and can be retrieved quickly without needing to access global memory again. In contrast, the ZombieGpu program is not as cache friendly: a vector’s X and Y coordinates are not in adjacent memory locations, rather they are in separate X and Y coordinate arrays. When the kernel code pulls in a vector’s X coordinate, the Y coordinate is usually *not* also pulled into the cache; so when the

```
37       // Write this Java object to the given byte buffer as a C
38       // struct.
39       public void toStruct
40          (ByteBuffer buf)
41          {
42          buf.putDouble (x);
43          buf.putDouble (y);
44          }
45
46       // Read this Java object from the given byte buffer as a C
47       // struct.
48       public void fromStruct
49          (ByteBuffer buf)
50          {
51          x = buf.getDouble();
52          y = buf.getDouble();
53          }
54       }
55
56    // Command line arguments.
57    long seed;
58    int N;
59    double W;
60    double G;
61    double L;
62    double dt;
63    double eps;
64    int steps;
65    int snap;
66
67    // Current body positions.
68    GpuStructArray<Vector> pos;
69
70    // Next body positions.
71    GpuStructArray<Vector> next;
72
73    // For detecting convergence.
74    GpuDoubleVbl delta;
75
76    // Kernel function interface.
77    private static interface ZombieKernel
78       extends Kernel
79       {
80       public void timeStep
81          (GpuStructArray<Vector> pos,
82           GpuStructArray<Vector> next,
83           int N,
84           double G,
85           double L,
86           double dt);
87       }
88
89    // Task main program.
90    public void main
91       (String[] args)
92       throws Exception
93       {
94       // Parse command line arguments.
```

Listing 28.2. ZombieGpu2.java (part 2)

kernel accesses the Y coordinate, it has to go back to global memory to get it, which is slower than getting the Y coordinate from the cache.

Under the Hood

Writing the `toStruct()` and `fromStruct()` methods in a subclass of class edu.rit.gpu.Struct requires knowing how the fields of a C structure are laid out in the GPU's memory. If the kernel function and the main program are both written in C, the compiler takes care of the structure layout automatically. But when the main program is written in Java, the Java compiler knows nothing about the C structure's layout, and the onus is on the programmer.

When working with the Parallel Java 2 Library's C structure capability, my first recommendation is to use only simple structures whose fields are all primitive types, like the `vector_t` structure in the ZombieGpu2 program. More complicated structures, especially structures with fields that are pointers to variable-sized dynamically-allocated data, become too difficult to deal with.

Confining our attention to structures with primitive fields, the first thing to understand is the correspondence between C types and Java types, and the size (number of bytes) occupied by each:

C type	*Java type*	*Size*
`double`	`double`	8
`long long int`	`long`	8
`unsigned long long int`	`long`	8
`float`	`float`	4
`int`	`int`	4
`unsigned int`	`int`	4
`short int`	`short`	2
`unsigned short int`	`short`	2
`char`	`byte`	1
`unsigned char`	`byte`	1

My second recommendation is that when defining the C structure in the kernel code, put all the 8-byte fields (if any) first, put all the 4-byte fields (if any) second, put all the 2-byte fields (if any) third, and put all the 1-byte fields (if any) last. This ensures (in conjunction with the padding rule described below) that each field is properly aligned in memory: 8-byte fields aligned to addresses that are multiples of 8 bytes, 4-byte fields aligned to addresses that are multiples of 4 bytes, and so on.

Once the fields of the C structure are defined in the kernel code, you can define the corresponding Java class in the main program code. The Java class's fields are named the same as the C structure's fields, *appear in the*

```
95        if (args.length != 9) usage();
96        seed = Long.parseLong (args[0]);
97        N = Integer.parseInt (args[1]);
98        W = Double.parseDouble (args[2]);
99        G = Double.parseDouble (args[3]);
100       L = Double.parseDouble (args[4]);
101       dt = Double.parseDouble (args[5]);
102       eps = Double.parseDouble (args[6]);
103       steps = Integer.parseInt (args[7]);
104       snap = Integer.parseInt (args[8]);
105
106       // Initialize GPU.
107       Gpu gpu = Gpu.gpu();
108       gpu.ensureComputeCapability (2, 0);
109
110       // Set up GPU variables.
111       Module module = gpu.getModule
112          ("edu/rit/gpu/example/ZombieGpu2.cubin");
113       pos = gpu.getStructArray (Vector.class, N);
114       next = gpu.getStructArray (Vector.class, N);
115       delta = module.getDoubleVbl ("devDelta");
116
117       // Set up GPU kernel.
118       ZombieKernel kernel = module.getKernel (ZombieKernel.class);
119       kernel.setBlockDim (256);
120       kernel.setGridDim (N);
121       kernel.setCacheConfig (CacheConfig.CU_FUNC_CACHE_PREFER_L1);
122
123       // Set zombies' initial (x,y) coordinates at random in a WxW
124       // square region. Also allocate zombies' next positions.
125       Random prng = new Random (seed);
126       for (int i = 0; i < N; ++ i)
127          {
128          pos.item[i] = new Vector
129             (prng.nextDouble()*W, prng.nextDouble()*W);
130          next.item[i] = new Vector (0, 0);
131          }
132       pos.hostToDev();
133
134       // Snapshot all bodies' initial positions.
135       int t = 0;
136       snapshot (t);
137
138       // Do repeated time steps.
139       for (;;)
140          {
141          // Do one time step.
142          delta.item = 0.0;
143          delta.hostToDev();
144          kernel.timeStep (pos, next, N, G, L, dt);
145
146          // Advance to next time step.
147          ++ t;
148
149          // Update positions.
150          GpuStructArray<Vector> tmp;
151          tmp = pos; pos = next; next = tmp;
152
```

Listing 28.2. ZombieGpu2.java (part 3)

same order, and use the Java types corresponding to the C types as shown above.

The Java class's `sizeof()` method returns the size of the C structure in bytes. This is the sum of the sizes of the fields, *plus possibly some extra padding.* Padding must be added if necessary to make the size of the structure be a multiple of the size of the largest field in the structure—the first field, according to my recommendation above. This ensures that if an array of structures is created, each structure in the array is properly aligned.

The ZombieGpu2 program defined this C structure and Java class:

```
typedef struct
   {
   double x;
   double y;
   }
   vector_t;
```

```
private static class Vector
   {
   double x;
   double y;
   . . .
   }
```

The sum of the field sizes is 16. This is a multiple of the largest field size, 8. So no extra padding is needed, and the `sizeof()` method returns 16.

Here's another example:

```
typedef struct
   {
   double a;
   double b;
   int c;
   }
   example_t;
```

```
private static class Example
   {
   double a;
   double b;
   int c;
   . . .
   }
```

The sum of the field sizes is 20. This is not a multiple of the largest field size, 8. So extra padding is needed, and the `sizeof()` method would return 24 rather than 20.

With the C structure's fields laid out as recommended above, the Java class's `toStruct()` method must put the fields into the byte buffer passed as an argument, *in the order the fields are declared*, using the ByteBuffer method corresponding to each field's type: `putDouble()`, `putLong()`, `putFloat()`, `putInt()`, `putShort()`, or `putByte()`. Likewise, the Java class's `fromStruct()` method must get the fields from the byte buffer passed as an argument, *in the order the fields are declared*, using the ByteBuffer method corresponding to each field's type: `getDouble()`, `getLong()`, `getFloat()`, `getInt()`, `getShort()`, or `getByte()`.

In addition to the fields and methods described above, the Java class can define other constructors and methods as needed by the Java main program.

In the Java code, a Java variable that mirrors a C structure is created by calling the `getStructVbl()` method on a Gpu object or by calling the `getStructVbl()` method on a Module object. These methods return an instance of class GpuStructVbl<T>, where T is the Java class corresponding to the C

```
153         // Stop when position delta is less than convergence
154         // threshold or when the specified number of time steps
155         // have occurred.
156         delta.devToHost();
157         if ((steps == 0 && delta.item < eps) ||
158            (steps != 0 && t == steps))
159               break;
160
161         // Snapshot all bodies' positions every <snap> time steps.
162         if (snap > 0 && (t % snap) == 0)
163            snapshot (t);
164         }
165
166      // Snapshot all bodies' final positions.
167      snapshot (t);
168      }
169
170   // Snapshot all bodies' positions.
171   private void snapshot
172      (int t)
173      {
174      pos.devToHost();
175      for (int i = 0; i < N; ++ i)
176         System.out.printf ("%d\t%d\t%g\t%g%n",
177            t, i, pos.item[i].x, pos.item[i].y);
178      }
179
180   // Print a usage message and exit.
181   private static void usage()
182      {
183      System.err.println ("Usage: java pj2 " +
184         "edu.rit.pj2.example.ZombieGpu2 <seed> <N> <W> <G> <L> " +
185         "<dt> <eps> <steps> <snap>");
186      System.err.println ("<seed> = Random seed");
187      System.err.println ("<N> = Number of bodies");
188      System.err.println ("<W> = Region size");
189      System.err.println ("<G> = Attraction factor");
190      System.err.println ("<L> = Attraction length scale");
191      System.err.println ("<dt> = Time step size");
192      System.err.println ("<eps> = Convergence threshold");
193      System.err.println ("<steps> = Number of time steps (0 = " +
194         "until convergence)");
195      System.err.println ("<snap> = Snapshot interval (0 = none)");
196      throw new IllegalArgumentException();
197      }
198
199   // Specify that this task requires one core.
200   protected static int coresRequired()
201      {
202      return 1;
203      }
204
205   // Specify that this task requires one GPU accelerator.
206   protected static int gpusRequired()
207      {
208      return 1;
209      }
210   }
```

Listing 28.2. ZombieGpu2.java (part 4)

structure. The address of the C structure in GPU memory is stored inside the GpuStructVbl object. The GpuStructVbl object has a public `item` field of type T; *this field is initially null* and must be assigned an instance of class T. Thereafter, calling the GpuStructVbl object's `hostToDev()` method uploads the `item` field to the GPU. The `hostToDev()` method creates a byte buffer, calls the `toStruct()` method on the `item` field, which puts the object's contents into the byte buffer, and transfers the byte buffer's contents to GPU memory at the address of the GPU variable; the number of bytes transferred is determined by calling the `item` field's `sizeof()` method. Likewise, calling the GpuStructVbl's `devToHost()` method downloads the `item` field from the GPU. The `devToHost()` method creates a byte buffer, transfers the contents of GPU memory from the address of the GPU variable into the byte buffer, and calls the `fromStruct()` method on the `item` field, which gets the object's contents from the byte buffer.

A Java variable that mirrors an array of C structures is created by calling the `getStructArray()` method on a Gpu object or by calling the `getStructArray()` method on a Module object. These methods return an instance of class GpuStructArray<T>, where T is the Java class corresponding to the C structure. The address of the C structure array in GPU memory is stored inside the GpuStructArray object. The GpuStructArray object has a public `item` field of type T[]; *the elements of this array are initially null* and must be assigned instances of class T. Thereafter, calling the GpuStructArray object's `hostToDev()` method uploads the `item` array to the GPU, and calling the `devToHost()` method downloads the `item` array from the GPU. The `toStruct()` and `fromStruct()` methods are called on each `item` array element to effect the transfer.

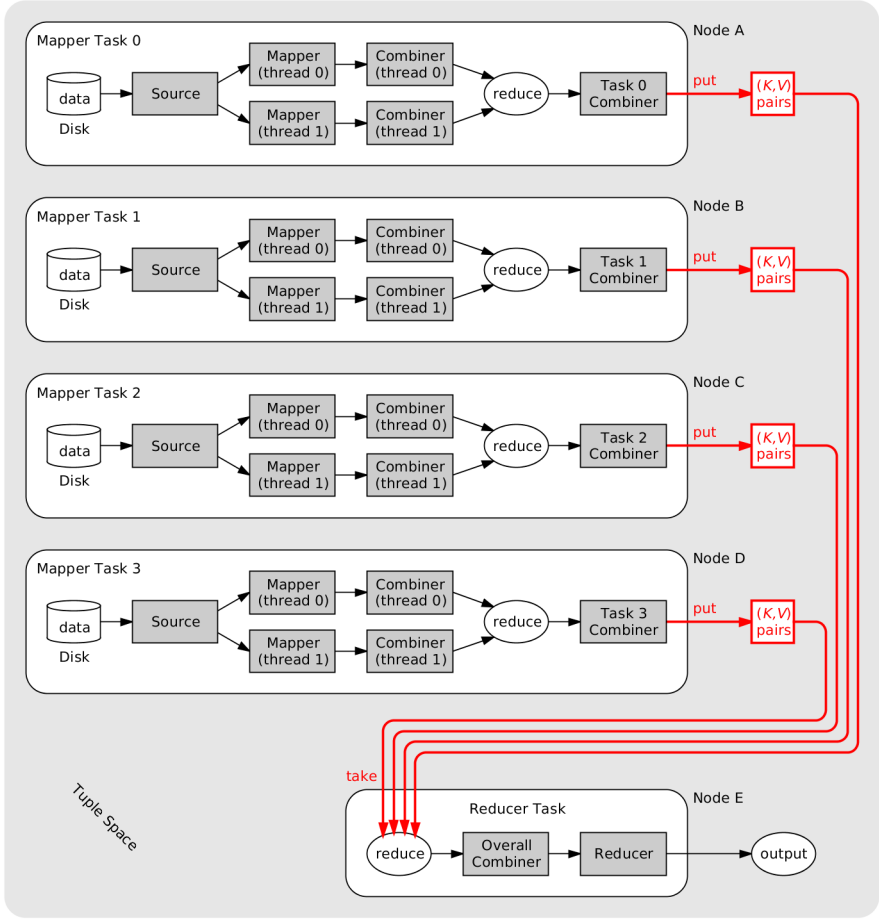
Points to Remember

- When designing the kernel function for a GPU parallel program, consider whether data might better be represented with a C structure.
- Limit the C structure to consist of fields of primitive types.
- Lay out the structure with the largest fields first and the smallest fields last.
- In the Java main program, define a class that extends class edu.rit.gpu.Struct, with the same fields as the C structure. Also define the class's `sizeof()`, `toStruct()`, and `fromStruct()` methods.
- To mirror a C structure variable, create an instance of generic class edu.rit.gpu.GpuStructVbl. Assign an instance of the Java class to the GpuStructVbl's `item` field.
- To mirror an array of C structures, create an instance of generic class edu.rit.gpu.GpuStructArray. Assign an instance of the Java class to each

element of the GpuStructArray's `item` field.

- Use the usual `hostToDev()` and `devToHost()` methods to transfer structures from the CPU to the GPU and vice versa.
- Representing data with a structure might make the program more cache friendly, thereby improving its performance.

PART V
BIG DATA



Parallel Java Map-Reduce job running on a cluster parallel computer

Chapter 29
Basic Map-Reduce

We turn now to the last part of the book, to consider how to write parallel *big data* applications. This kind of application spends only a little bit of CPU time on each data item, but works with an enormous number of data items. Such applications might better be called *little CPU big data* applications. Processing the data items in parallel can speed up the application.

You can write a big data application using the multicore and cluster parallel programming constructs we've studied up to this point. However, many big data applications fit a certain pattern: *map-reduce*. A parallel programming library that supports the higher-level map-reduce pattern can let you code big data applications with less effort than a library that supports only lower-level multicore and cluster parallel programming constructs. One such library is Apache's Hadoop (hadoop.apache.org).

Before I begin, though, I want to dispel two myths or misconceptions about big data parallel programming.

The first myth is "Big data = map-reduce." Some folks feel that every parallel big data application has to be programmed using the map-reduce paradigm. Some folks even feel that *every* parallel application, no matter how much or how little data the application deals with, has to be programmed using the map-reduce paradigm. While the map-reduce paradigm is convenient to use due to libraries like Hadoop, a programmer who uses map-reduce for every program is like a carpenter who has only a hammer in her toolbox. To paraphrase an old saying, "If your only tool is Hadoop, every program looks like a map-reduce job"—even if the map-reduce paradigm is ill-suited to the application. Some of the parallel programs we've studied so far fit the map-reduce pattern, but many did not. The wise parallel programmer has many parallel programming tools in her toolbox, and she uses the appropriate tools for each job rather than shoehorning everything into map-reduce.

The second myth is "Map-reduce = Hadoop." A popular map-reduce library written in Java, Hadoop is designed to support parallel processing of *really big* data sets on *really big* clusters. As such, it includes features needed in that kind of environment, such as fault tolerant distributed processing and a replicated file system. In my experience, however, some data oriented applications deal only with *big* data, not *really big* data—quantities measured in gigabytes rather than terabytes, say. For such applications, I've found Hadoop to be overkill. Simpler map-reduce libraries that omit Hadoop's advanced capabilities can process the data in less time than Hadoop. I'm going to be using *Parallel Java Map Reduce (PJMR)*, a simple map-reduce library built on top of the Parallel Java 2 Library's cluster programming capabilities, to teach you map-reduce parallel programming in Java.

I'll introduce map-reduce parallel programming with a *small* data example. (Later I'll scale up to bigger data sizes.) Suppose I have several novels in the form of plain text files, like *A Tale of Two Cities* by Charles Dickens:

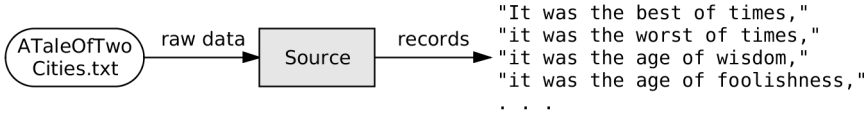
> It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way . . .

I want to analyze these files and produce a *concordance*, which is a list of the unique words in the files. I also want to count how many times each unique word occurs in the files. The concordance should look like this:

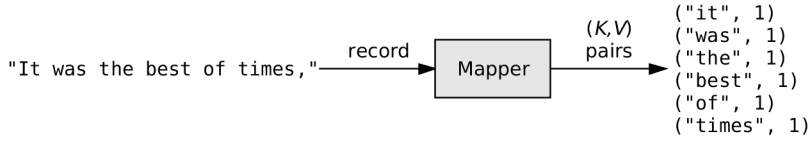
```
14 hats
1 un-regular
2 stimulant
41 progress
8 blue-eyed
1 quay
5 dictating
2 snuffers
10 disparage
. . .
```

That is, the word "hats" occurred 14 times in the novels, the word "un-regular" occurred once, the word "stimulant" occurred twice, and so on.

Consider a series of processing steps that begins with the file containing *A Tale of Two Cities* and ends with the concordance for that one file. In the first step, a *source* object generates *records* containing the raw input data. The records can be anything, depending on the application: lines of text from a file, rows of data from a database, objects in JSON (JavaScript Object Notation) format from a repository, and so on. In our case, each record will be one line from the text file containing the novel.



In the second step, a *mapper* object extracts from each record zero or more *(key, value)* or *(K,V) pairs*. The key can be any data type, and so can the value. Each pair contains information pertinent to the application. The concordance consists of words and their associated counts. So in our case, a pair will consist of a word (the key) with a count of 1 (the value).



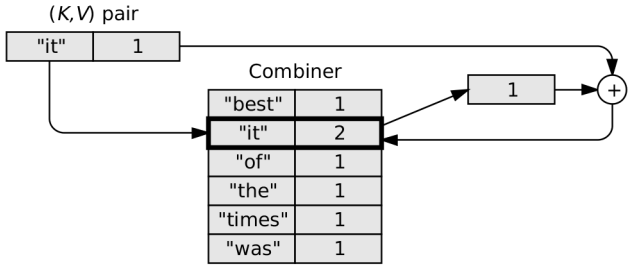
This object is called a "mapper" because it transforms, or *maps*, a record to a pair or pairs.

The third processing step is to absorb each (*K*,*V*) pair into a *combiner* object. The combiner is a special kind of hash table data structure. Like a hash table, the combiner is a collection that associates keys with values. However, in a combiner, the values are stored in *reduction variables.* The reduction operation can be anything. In our example, after absorbing the (*K*,*V*) pairs from the first record in ATaleOfTwoCities.txt, the combiner looks like this:

Combiner

"best"	1
"it"	1
"of"	1
"the"	1
"times"	1
"was"	1

The next (*K*,*V*) pair added to the combiner is `("it",1)`, namely the first word in the next record. The combiner looks up the pair's key *K* = `"it"` in the hash table, yielding a reduction variable with the value 1. The pair's value *V* = 1 is then reduced into the reduction variable. What is the reduction operation? That depends on the application. In our case, we want to add up the counts for each word, so the reduction operation is summation. The pair's value and the reduction variable's value are added together, and the result, 2, is stored back in the reduction variable in the hash table:

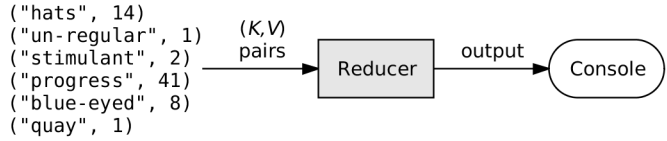


As the (*K*,*V*) pairs are absorbed into the combiner, the words' associated counts are incremented—because the reduction operation of the reduction variables in the combiner is addition, and the value in each pair is *V* = 1. If a word that does not yet exist in the combiner is absorbed, a new entry, consisting of the word *K* plus a new reduction variable with an initial value of 0, is inserted into the hash table, and then the value *V* = 1 is added to the reduction variable. After all the (*K*,*V*) pairs from all the records from the input file have been processed, the combiner ends up holding the information needed to generate the concordance, namely a list of the unique words and the count for each word.

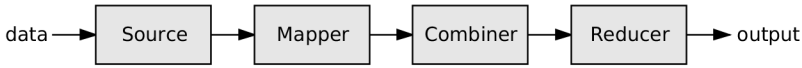
The object is called a combiner because it *combines* all the (*K*,*V*) pairs

together. Specifically, for each unique key *K* in the series of pairs, the combiner combines, or reduces, together all the values *V* associated with that key in the series of pairs, using some reduction operation.

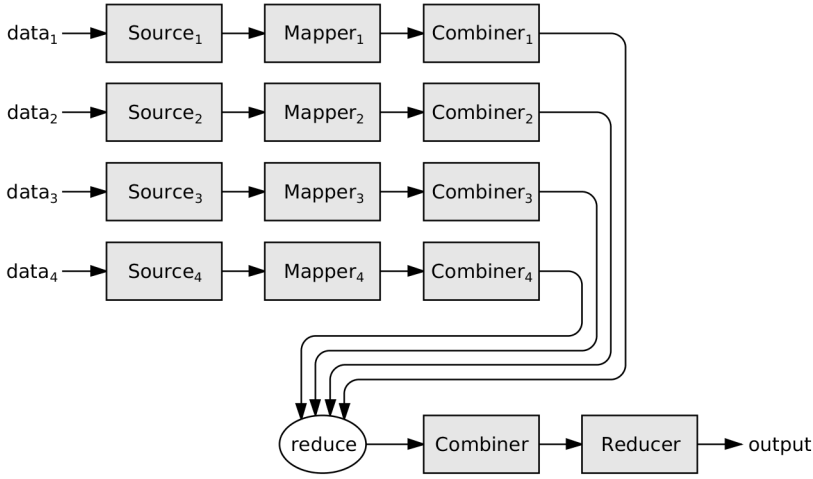
In the fourth and final processing step, the (*K*,*V*) pairs in the combiner are fed to a *reducer* object. The reducer uses these pairs to produce the application's final output. The reducer might or might not do further processing on the pairs. In our case, the reducer merely prints each pair's count and word.



The map-reduce workflow consists of the four processing steps performed by the four objects—source, mapper, combiner, reducer. The "map" part of "map-reduce" is done by the mapper object. The "reduce" part of "map-reduce" is done in two stages: the first stage is when (*K*,*V*) pairs are absorbed into the combiner, the second stage is done by the reducer object.



So far, the map-reduce program has analyzed just one input file. In general, though, I want to analyze several input files containing several novels, and produce a combined concordance for all the novels. Here's where some parallel computing comes in. I set up separate source, mapper, and combiner objects for each input file, and I run these *in parallel* to process all the input files simultaneously. At the end, I reduce the per-file combiners together into one overall combiner, then I feed the overall combiner's contents to the reducer to produce the output.



In the above parallel map-reduce workflow, the per-file combiners are reduced together into the overall combiner as follows. The overall combiner starts out as an empty hash table. Then each (*K,V*) pair from each per-file combiner is absorbed into the overall combiner in the manner described previously, using the reduction operation of the reduction variables in the combiner. In the concordance example, the overall combiner ends up containing the unique words in all the input files; and the count for each word ends up being the sum of the counts for that word in all the input files.

When designing a map-reduce application, the first critical decision is what the key and the value in the (*K,V*) pairs will be. This is because the central data structure in the map-reduce workflow is the combiner, that is, a hash table that associates keys *K* with values *V.* The second critical decision is what reduction variable will be used for the values. This is because every new (*K,V*) pair's value is reduced into the value associated with the pair's key in the combiner, using the reduction variable's reduction operation. You must make these two critical decisions so the final overall combiner contains the data needed to produce the application's output.

Although different map-reduce programs operate on different data and do different mapping and reducing computations, the overall map-reduce workflow is the same. Therefore, a map-reduce library, like Hadoop or PJMR, can provide most of the code for a map-reduce program. You, the programmer, only have to write the bits of code that are specific to your application. With PJMR, these are the pieces you have to write:

- You must write code for a *source object* that obtains your raw data from wherever it is stored. PJMR includes classes for source objects that read plain text files. Many big data sets are in the form of text files, so you can often use PJMR's classes without needing to write your own source objects.
- You must write code for a *mapper object.* This object has a method whose argument is one record from a source object. The method analyzes the record and adds zero or more (*K,V*) pairs to a combiner object, which is also passed in as an argument.
- You must write code for the *reduction variable* in the combiner object. (Refer back to Chapter 5, which describes how to write a reduction variable class.) However, if a suitable reduction variable class already exists in the Parallel Java 2 Library, you can use that.
- You must write code for a *reducer object.* This object has a method whose argument is one (*K,V*) pair from the overall combiner. The method uses that information to produce the program's output.
- You must write a *main program* to configure all the above objects and start the map-reduce job.

Now we're ready to write the actual PJMR program to produce a concordance of several text files: class edu.rit.pjmr.example.Concordance01 (Listing 29.1).

The class for the whole program must extend class edu.rit.pjmr.PjmrJob (line 11). This is a generic class. I will skip over the generic type parameters for now and return to them later. The PjmrJob subclass has a `main()` method, just like all Parallel Java 2 programs (lines 14–27). The `main()` method's purpose is to configure the other objects in the PJMR job. I'm going to defer a discussion of the `main()` method until I've covered the other classes in the program.

The source object's class in a PJMR job implements interface edu.rit-.pjmr.Source. A source object extracts records from some data source and presents each record to the PJMR job as a (key, value) pair. The key consists of information identifying the record, and the value is the actual contents of the record. Interface Source is a generic interface with two generic type parameters designating the data types of the key and the value.

For the source objects in the concordance program, I used the predefined edu.rit.pjmr.TextFileSource class. This class obtains input data from a text file, which is exactly what I need. This class implements generic interface Source. The record's key data type is class edu.rit.pjmr.TextId, which consists of the input file name and the line number; these uniquely identify the record. The record's value data type is class String, which contains the designated line from the input file.

For the mapper objects in the concordance program, I defined class MyMapper (line 38). I made it a nested class inside the main Concordance01 class for convenience. (Alternatively, I could have put class MyMapper in its own separate source file.) The mapper class must extend the PJMR base class edu.rit.pjmr.Mapper (line 39). This base class is a generic class. The first two generic type parameters are the key data type and the value data type of the records generated by the source object, namely TextId and String. The last two generic type parameters are the key data type and the value data type of the (*K*,*V*) pairs generated by the mapper object. Here, the keys *K* are words, which are instances of type String. The values *V* are word counts. The value data type must be a reduction variable class with the proper reduction operation for the application. Here, I want the values *V* to be long integers, and I want the reduction operation to be summation. So I can use the predefined edu.rit.pj2.vbl.LongVbl.Sum class. (I specified the fourth generic type parameter on line 39 as that class's superclass, LongVbl.)

Next comes the code for the `map()` method (line 43). The `map()` method's arguments are the key and the value of one record from the source object (types TextId and String, respectively) and a reference to the combiner object of type edu.rit.pjmr.Combiner. This is a generic class, whose generic type parameters designate the data types of the (*K*,*V*) pairs produced by the

mapper (types String and LongVbl, respectively).

The `map()` method's purpose is to analyze the contents of the record, namely the `inValue` argument; produce zero or more (*K*,*V*) pairs; and add these to the combiner. If necessary, the `map()` method can also use the record's key, namely the `inKey` argument, to produce the (*K*,*V*) pairs. The concordance program doesn't need to use the record's key (the file name and line number), so the `map()` method just ignores the `inKey` argument. The `map()` method uses a Scanner to extract whitespace-delimited words from the line of input. Non-letter characters at the beginning and the end of the word are discarded, and the word is converted to lowercase. The resulting word is added to the combiner by calling the `add()` method on the combiner (line 62). The `add()` method's arguments are the key and the value in one (*K*,*V*) pair. The value object must be an instance of the desired reduction variable class, which performs the desired reduction operation. Here, the value object is an instance of class LongVbl.Sum initialized with a count of 1 (line 41). Thus, the values (word counts) in the combiner will be long integers, and during reduction the values will be added together.

For the reducer object, I defined class MyReducer (line 68). I made it a nested class like the mapper class, but I could have put the reducer class in its own separate source file. The reducer class must extend the PJMR base class edu.rit.pjmr.Reducer (line 69). This base class is a generic class. The two generic type parameters are the key data type and the value data type of the (*K*,*V*) pairs generated by the mapper object, namely String and LongVbl.

I defined the `reduce()` method in the reducer class (line 71). The `reduce()` method's arguments are the key and the value of one (*K*,*V*) pair from the overall combiner. The `reduce()` method's purpose is to generate the requisite output from this (*K*,*V*) pair. Here, all the `reduce()` method has to do is print the word count *V* followed by the word *K*.

Having defined all the classes I need, I can return to the concordance program's main class, class Concordance01 (line 10). As previously mentioned, a PJMR program's main class must extend class PjmrJob (line 11). This is a generic class. The four generic type parameters are the same as those of the mapper class: the source object's record's key and value types (TextId and String) and the mapper object's (*K*,*V*) pair's key and value types (String and LongVbl).

Next comes the concordance program's `main()` method (line 14). The command line arguments are the names of the text files to be analyzed. For each file, I configure a *mapper task* for the map-reduce job (lines 19–22). A mapper task encapsulates the source, mapper, and combiner objects associated with one text file. Each mapper task is configured with a source object that is an instance of class TextFileSource reading from one of the input files on the command line. Each mapper task is also configured with a mapper object that is an instance of class MyMapper. Each mapper task is by default

```
1  package edu.rit.pjmr.example;
2  import edu.rit.pj2.vbl.LongVbl;
3  import edu.rit.pjmr.Combiner;
4  import edu.rit.pjmr.Mapper;
5  import edu.rit.pjmr.PjmrJob;
6  import edu.rit.pjmr.Reducer;
7  import edu.rit.pjmr.TextFileSource;
8  import edu.rit.pjmr.TextId;
9  import java.util.Scanner;
10 public class Concordance01
11    extends PjmrJob<TextId,String,String,LongVbl>
12    {
13    // PJMR job main program.
14    public void main
15       (String[] args)
16       {
17       if (args.length < 1) usage();
18
19       for (int i = 0; i < args.length; ++ i)
20          mapperTask()
21             .source (new TextFileSource (args[i]))
22             .mapper (MyMapper.class);
23
24       reducerTask() .reducer (MyReducer.class);
25
26       startJob();
27       }
28
29    // Print a usage message and exit.
30    private static void usage()
31       {
32       System.err.println ("Usage: java pj2 " +
33          "edu.rit.pjmr.example.Concordance01 <file> [<file> ...]");
34       throw new IllegalArgumentException();
35       }
36
37    // Mapper class.
38    private static class MyMapper
39       extends Mapper<TextId,String,String,LongVbl>
40       {
41       private static final LongVbl ONE = new LongVbl.Sum (1L);
42
43       public void map
44          (TextId inKey,   // Line number
45           String inValue, // Line from file
46           Combiner<String,LongVbl> combiner)
47          {
48          Scanner scanner = new Scanner (inValue);
49          while (scanner.hasNext())
50             {
51             // For each word, remove leading and trailing
52             // non-letters and convert to lowercase.
53             String s = scanner.next();
54             int i = 0;
55             while (i < s.length() &&
56                       ! Character.isLetter (s.charAt (i)))
57                ++ i;
```

Listing 29.1. Concordance01.java (part 1)

configured with a combiner object that is an instance of class Combiner (I don't have to specify this). Then I configure a *reducer task* for the map-reduce job (line 24). A reducer task encapsulates the overall combiner object and the reducer object for the job. The reducer task is configured with a reducer object that is an instance of class MyReducer. The reducer task is by default configured with an overall combiner object that is an instance of class Combiner (again, I don't have to specify this).

It's important to realize that the PjmrJob's `main()` method does not *execute* the map-reduce job, it only *configures* the job. To actually run the job, the `main()` method must call the `startJob()` method as its last act (line 26). If you don't call `startJob()`, the program will exit without doing anything.

That's it! A complete PJMR map-reduce program in 79 lines of code. Almost all the work of the map-reduce job is performed by PJMR classes. After calling the `main()` method to configure the job, PJMR calls your code only to perform each mapping step (the mapper object's `map()` method) and to perform each reducing step (the reducer object's `reduce()` method).

I ran the Concordance01 program on a dual-core desktop PC to analyze four of Charles Dickens's novels: *A Tale of Two Cities*, *Bleak House*, *David Copperfield*, and *Oliver Twist.* I used this command:

```
$ java pj2 edu.rit.pjmr.example.Concordance01 \
  AtaleOfTwoCities.txt BleakHouse.txt DavidCopperfield.txt \
  OliverTwist.txt
```

These novels' text files are small (between 0.7 and 2.0 megabytes each), and the program computes the concordance quickly (in about 5.1 seconds), even on a mere PC. In the next few chapters I'll use PJMR to analyze some much bigger data sets on a real parallel computer.

By the way, the ten most frequent words in the four novels, in descending order, are "the", "and", "to", "I", "of", "a", "in", "that", "it", and "he". The pronoun "he" occurs 12,299 times, while the pronoun "she" occurs only 5,515 times, less than half as often—indicative of gender attitudes in Dickens's time.

Under the Hood

Because the combiner is such an important part of PJMR's map-reduce workflow, it's helpful to peek under the hood of class Combiner to see what it does (Listing 29.2).

Class Combiner is a generic class (line 8) whose generic type parameters are the key and value data types of the mapper's (*K*,*V*) pairs. Class Combiner is a subclass of the general purpose Map class from the Parallel Java 2 Library (line 9), which is implemented as a hash table. As such, you can do anything to a combiner that you could do to a map, and the combiner has a few additional operations. Class Combiner is also a reduction variable; it im-

```
58             int j = s.length() - 1;
59             while (j >= 0 && ! Character.isLetter (s.charAt (j)))
60                -- j;
61             if (i <= j)
62                combiner.add (s.substring(i,j+1).toLowerCase(), ONE);
63             }
64          }
65       }
66
67    // Reducer class.
68    private static class MyReducer
69       extends Reducer<String,LongVbl>
70       {
71       public void reduce
72          (String key,    // Word
73           LongVbl value) // Number of occurrences
74          {
75          System.out.printf ("%s %s%n", value, key);
76          System.out.flush();
77          }
78       }
79    }
```

Listing 29.1. Concordance01.java (part 2)

```
1  package edu.rit.pjmr;
2  import edu.rit.pj2.Vbl;
3  import edu.rit.pj2.TerminateException;
4  import edu.rit.util.Action;
5  import edu.rit.util.Instance;
6  import edu.rit.util.Map;
7  import edu.rit.util.Pair;
8  public class Combiner<K,V extends Vbl>
9     extends Map<K,V>
10    implements Vbl
11    {
12    // Construct a new combiner.
13    public Combiner()
14       {
15       super();
16       }
17
18    // Construct a new combiner that is a copy of the given combiner.
19    public Combiner
20       (Combiner<K,V> combiner)
21       {
22       super (combiner);
23       }
24
25    // Add the given (key, value) pair into this combiner.
26    public void add
27       (K key,
28        V value)
29       {
30       if (! contains (key))
31          put (key, initialValue (key, value));
```

Listing 29.2. Combiner.java (part 1)

plements the Vbl interface (line 10). As such, one combiner object can be reduced into another combiner object. PJMR uses this ability to reduce each per-source combiner into the overall combiner. The combiner has two constructors (lines 13 and 19) that do the same thing as the superclass map's constructors.

The combiner's `add()` method (line 26) is passed the key and the value of a (*K*,*V*) pair to be added to the combiner. The `add()` method first checks whether the key already exists; if not, a new element is inserted into the map (lines 30–31). The new element consists of the given key and an initial value to be associated with the key. The initial value is obtained by calling the protected `initialValue()` method (line 73), passing in the given key and the given value.

As defined in the Combiner class, the `initialValue()` method calls the `Instance.newDefaultInstance()` method in the Parallel Java 2 Library. The `newDefaultInstance()` method in turn uses Java Reflection to create a new instance of the class of the value object being added; the new instance is created using that class's default (no-argument) constructor. The new instance is returned (except if the value was null, in which case null is returned). This is usually what you want—assuming the default constructor creates a new instance with an appropriate initial value. If not, you can define a subclass of class Combiner and override the protected `initialValue()` method to do what you want; and you can configure your PJMR job to use the Combiner subclass by calling the `combiner()` method in the PjmrJob main program.

Back in the combiner's `add()` method, after putting a new map element into the map if necessary, the `add()` method gets the map element associated with the given key, then reduces the given value into that map element's value (lines 32–33)—either the already-existing element's value, or the newly-inserted element's initial value.

Next come the three methods implemented from interface Vbl: `clone()`, `set()`, and `reduce()`. The `clone()` and `set()` methods delegate to the appropriate methods in the Map superclass. The `reduce()` method (line 50) iterates over the mappings in the combiner (reduction variable) being reduced into this combiner (line 53). For each such mapping, the `reduce()` method calls the `add()` method to add the mapping's key and value to this combiner (line 57). The `add()` method in turn reduces the value in the mapping into the value in this combiner.

Lastly, class Combiner overrides the protected `copyValue()` method from the Map superclass (line 63). The Map superclass calls this method whenever it needs to copy a value in the map. Class Combiner's `copyValue()` method returns a clone of the given value, thus performing a *deep copy* of the value. (Class Map's `copyValue()` method returns a reference to the given value, thus performing a *shallow copy;* this is not the behavior I

```
32          if (value != null)
33             get (key) .reduce (value);
34          }
35
36       // Create a clone of this shared variable.
37       public Object clone()
38          {
39          return super.clone();
40          }
41
42       // Set this shared variable to the given shared variable.
43       public void set
44          (Vbl vbl)
45          {
46          copy ((Combiner<K,V>)vbl);
47          }
48
49       // Reduce the given shared variable into this shared variable.
50       public void reduce
51          (Vbl vbl)
52          {
53          ((Combiner<K,V>)vbl).forEachItemDo (new Action<Pair<K,V>>()
54             {
55             public void run (Pair<K,V> pair)
56                {
57                add (pair.key(), pair.value());
58                }
59             });
60          }
61
62       // Copy the given value.
63       protected V copyValue
64          (V value)
65          {
66          return (V) value.clone();
67          }
68
69       // The add() method calls the initialValue() method when a
70       // certain key is being added to this combiner for the first
71       // time. The initialValue() method returns a new object which is
72       // the initial value to be associated with the new key.
73       protected V initialValue
74          (K key,
75           V value)
76          {
77          try
78             {
79             return value == null ? null :
80                (V) Instance.newDefaultInstance (value.getClass(),
81                   true);
82             }
83          catch (Throwable exc)
84             {
85             throw new IllegalArgumentException
86                ("Cannot create initial value", exc);
87             }
88          }
89       }
```

Listing 29.2. Combiner.java (part 2)

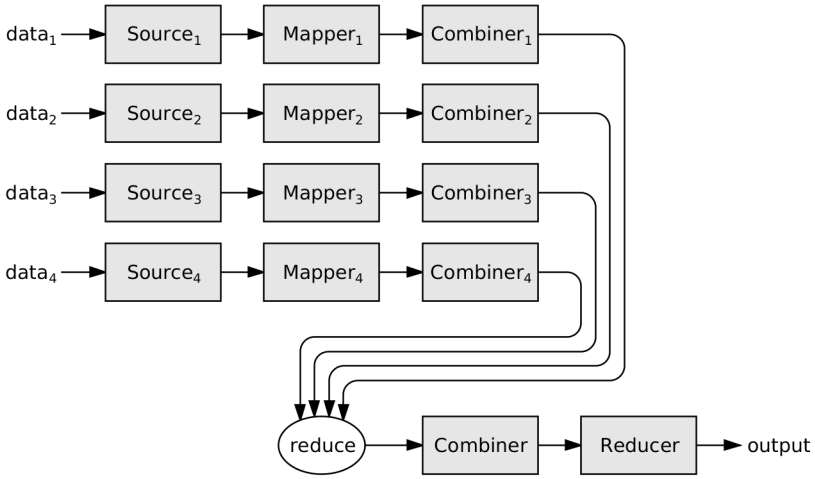
want for the combiner.)

Points to Remember

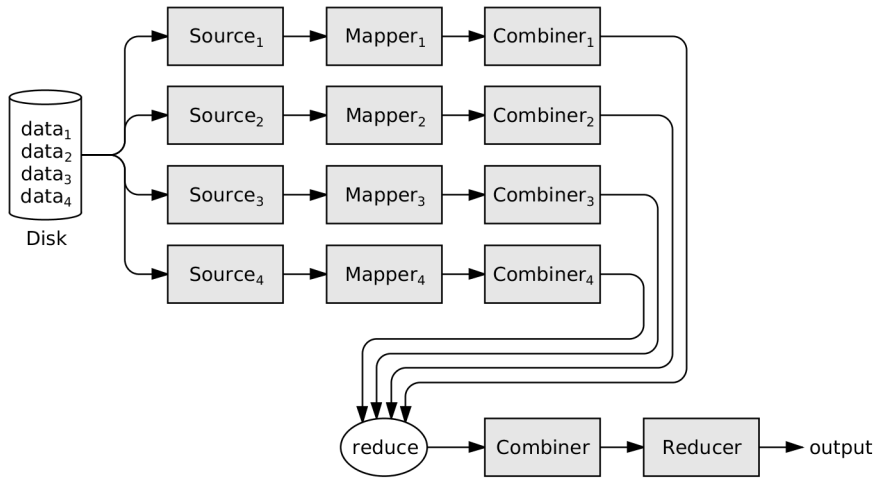
- Many parallel big data applications follow the *map-reduce* pattern.
- Using a map-reduce library, like Apache's Hadoop and Parallel Java Map Reduce (PJMR), can reduce the effort needed to develop a parallel big data application.
- However, big data does not necessarily equal map-reduce, and map-reduce does not necessarily equal Hadoop.
- A PJMR map-reduce workflow consists of four objects: source, mapper, combiner, reducer.
- The source object obtains raw input data from somewhere and produces records.
- The mapper object takes the records and extracts (*key*, *value*) or (*K*,*V*) pairs, where the key can be any data type and the value must be a reduction variable data type.
- The combiner object takes the (*K*,*V*) pairs and stores them in a special hash table data structure in which the values are reduced together using the reduction variable's reduction operation.
- The source, mapper, and combiner objects are replicated for each input data source and are run in parallel.
- After all the input data has been processed, the per-source combiners are reduced together into one overall combiner.
- The reducer object takes the (*K*,*V*) pairs from the overall combiner and produces the application's output.
- To design a PJMR map-reduce application, decide on the key and value data types for the (*K*,*V*) pairs, and decide on the reduction operation for the values.
- To program a PJMR application, define classes for the source objects (if necessary), the values' reduction variables (if necessary), the mapper objects, and the reducer objects. Then define a subclass of class PjmrJob with the job main program.
- The job main program must configure the sources, mappers, and reducers, and must call the `startJob()` method as its last act.
- Run the PJMR job like any Parallel Java 2 job, using the `pj2` launcher program.

Chapter 30
Cluster Map-Reduce

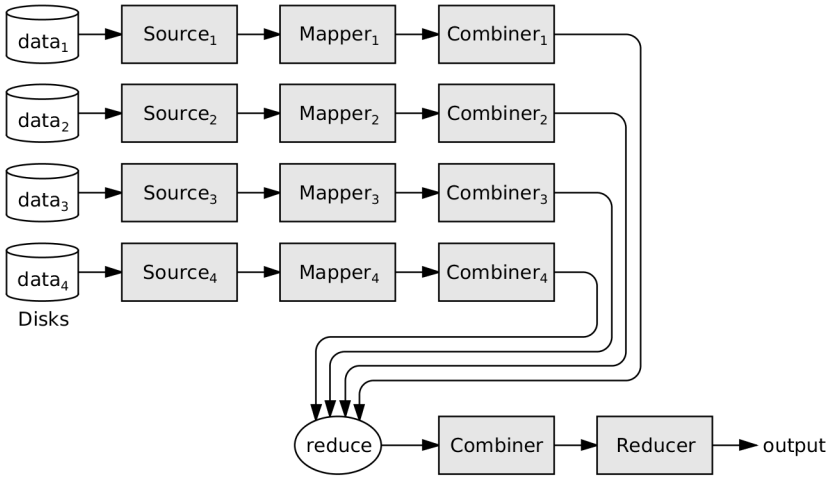
The parallel map-reduce workflow I introduced in the previous chapter featured multiple raw data sources, each with its own source, mapper, and combiner objects, followed by an overall combiner object and a reducer object. To speed up the application, the portions of the workflow associated with each data source were executed in parallel by separate threads.



I ran the program on a PC to analyze four text files containing novels by Charles Dickens. But this meant that the four raw data sources *all resided on the same disk drive.* Consequently, *the disk drive interface became a bottleneck* that to some extent nullified the program's parallelism. Even though the program ran in multiple threads on multiple cores, all the threads had to take turns reading their raw data sources sequentially through the one disk drive interface.



To get *full* parallelism during the mapping stage, each raw data source must reside on a *separate disk drive* with its own disk drive interface. Then the raw data sources no longer all have to go through the same bottleneck. Each parallel thread can read its own raw data source simultaneously with all the other threads.



It's possible to attach multiple disk drives to a node, just as a node can have multiple cores. But there is a limit to the number of disk drives a single node can handle, just as there is a limit to the number of cores on one node. And there is a limit to the amount of data that can be stored on one disk drive, just as there is a limit to the amount of main memory on one node.

Scaling up to big data analysis jobs requires shifting from a single-node computer to a *cluster* parallel computer. A cluster is not limited by the amount of disk storage a single node can handle. If you need more storage, add more nodes (with their disk drives) to the cluster. Of course, adding more nodes also means adding more cores to the cluster, thus also increasing the parallelism available for computing with the data on the new disk drives.

Figure 30.1 shows a PJMR map-reduce job running in the environment for which PJMR was designed, namely a cluster. I'll point out three features of the PJMR architecture.

First, the map-reduce job consists of multiple *mapper tasks* containing source, mapper, and combiner objects, as well as a *reducer task* containing combiner and reducer objects. Each mapper task runs on its own separate node in the cluster, in parallel with the other mapper tasks. Each mapper task's raw data source resides on the node's disk drive. This way, each source object can read its own disk drive without interference from other source objects, and the source object can download raw data at up to the full bandwidth of the disk interface. Consequently, the effective download bandwidth for the whole job is one disk's bandwidth multiplied by the number of nodes.

This eliminates the bottleneck that would occur if all the raw data resided on the same disk drive.

Second, sometimes the map-reduce job's running time can be decreased by configuring each mapper task with *multiple mapper objects*, running in parallel in multiple threads (cores) on the node, all getting records from the one source object. Why might this improve the performance? Because typically, the source object reads raw data from the disk drive in large chunks, then breaks the chunks into separate records. Multiple mapper objects running in multiple threads can then process these multiple records simultaneously. Also, if one thread has to stall waiting for the source to read the next chunk off the disk drive, another thread can still be occupied processing a previous record.

Third, PJMR follows the *cluster parallel reduction pattern.* Each mapper object has its own per-thread combiner. These are reduced together within each mapper task, yielding a per-task combiner. The (*K*,*V*) pairs in the per-task combiners are sent via tuple space to the reducer task, where they are absorbed into the overall combiner. This means that the key data type *K* and the value data type *V* must be streamable or serializable, so that they can be sent across the cluster's backend network encapsulated inside tuples.

Keep in mind that these architectural features are hidden inside PJMR. Even when running a PJMR map-reduce job on a cluster parallel computer, the only code you have to write is still just the mapper object, reducer object, and job main program, plus possibly the source object and the reduction variable. PJMR takes care of the rest.

To illustrate a cluster parallel PJMR map-reduce job, I'll analyze some web server log data. Courtesy of our system administrator, I obtained ten days' worth of logs from the RIT Computer Science web server. Each daily log is a text file. Each line of text is a log entry containing the IP address that made a web request, the date and time of the request, the URL that was requested, and other information. Here is a small portion of one log file:

```
157.55.39.63 - - [10/Jul/2015:00:00:03 -0400] "GET /~vcss243/La
66.249.73.178 - - [10/Jul/2015:00:00:07 -0400] "GET /images/Bay
207.46.13.80 - - [10/Jul/2015:00:00:12 -0400] "GET /~vcss233/pu
180.76.15.163 - - [10/Jul/2015:00:00:12 -0400] "GET /~mxt4877/C
180.76.15.158 - - [10/Jul/2015:00:00:17 -0400] "GET /~ats/ds-20
129.21.36.110 - - [10/Jul/2015:00:00:19 -0400] "GET /~displays/
157.55.39.78 - - [10/Jul/2015:00:00:21 -0400] "GET /usr/local/p
157.55.39.39 - - [10/Jul/2015:00:00:31 -0400] "GET /usr/local/p
80.240.138.58 - - [10/Jul/2015:00:00:32 -0400] "GET /~ark/pj2/d
180.76.15.148 - - [10/Jul/2015:00:00:35 -0400] "GET /~ats/java-
```

I partitioned the log files into ten chunks of roughly equal size and stored one chunk on the disk drive of each of the `tardis` cluster's ten backend nodes, which are named `dr00` through `dr09`. The log files occupy about 400 megabytes total, or 40 megabytes on each node—a medium size data set.

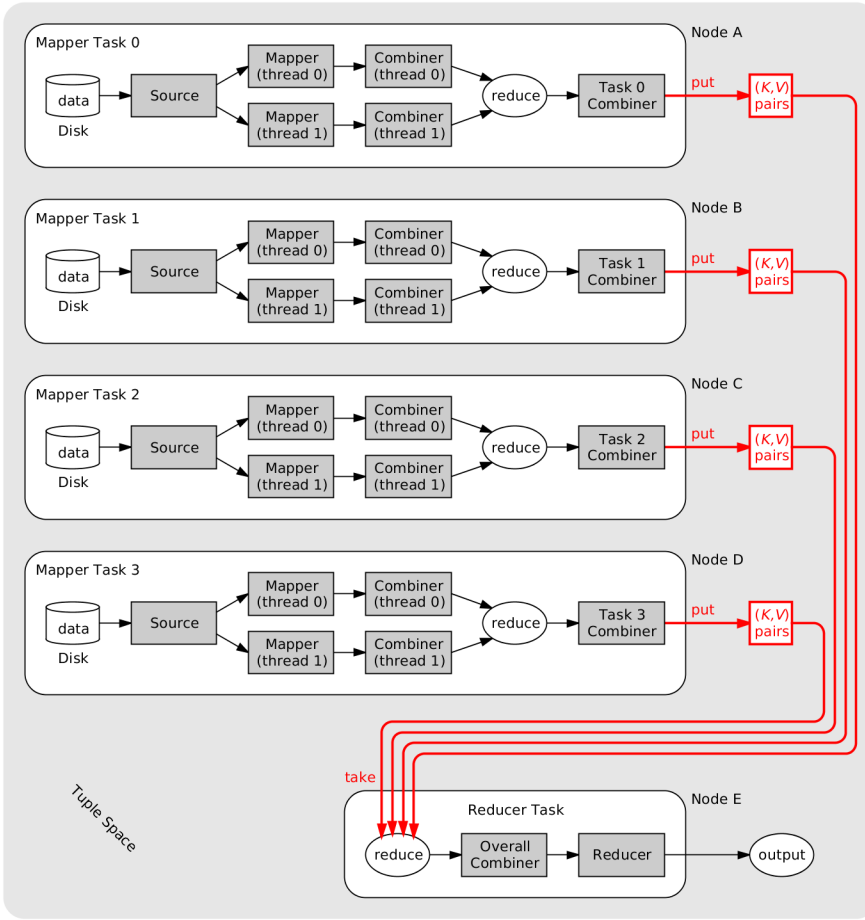


Figure 30.1. PJMR map-reduce job architecture

A PJMR job usually assumes the data to be analyzed is already located on the cluster's disk drives. You have to do the data partitioning yourself. (The Parallel Java 2 Library includes some programs to help with data partitioning, in package edu.rit.pjmr.util; refer to the Javadoc for further information.) However, you only need to partition the data once. Thereafter, you can run as many map-reduce jobs on the data as you want, doing various kinds of analysis.

The first web log analysis program is edu.rit.pjmr.example.WebLog01 (Listing 30.1). The program is run with this command:

```
$ java pj2 [threads=NT] edu.rit.pjmr.example.WebLog01 nodes \
  file [pattern]
```

The program's command line arguments are as follows:

- `NT` is the number of mapper threads in each mapper task (default: 1).
- `nodes` is a comma-separated list of one or more backend node names in the cluster. The program will run a mapper task on each node.
- `file` is the name of the web log file to be analyzed. The program assumes that each node has a file with this name, containing a portion of the web log data.
- `pattern` is an optional argument. If present, the pattern must be a regular expression as specified in Java platform class java.util.regex.Pattern, and the program will analyze only the web log entries that match the pattern. For example, the pattern could be a date, to limit the analysis to that date. If the *pattern* argument is omitted, the program will analyze all the web log entries.

The main program class (line 12) extends class PjmrJob (line 13). Because I will be using a TextFileSource as I did in the previous chapter, the source's key and value data types are TextId and String. The program will be mapping IP addresses to number of occurrences, similar to the concordance program in the previous chapter. Thus, the mapper's (*K*,*V*) data types are class edu.rit.pjmr.example.IPAddress for the key *K* and class LongVbl for the value *V*. (Class IPAddress is a separate class that stores an IP address in the form of an integer and that has a method to decide whether one IP address is smaller than another.)

After parsing the command line arguments, the program prints its *provenance* (lines 34–41). The provenance includes the command line used to run the program and the date and time when the program was run. This information is useful when examining the program's output at a later time. This information is also useful if the same analysis needs to be performed again.

For each backend node name specified (line 44), the program configures a mapper task to run on that node (line 45). The mapper task's source object is a TextFileSource reading the given web log file (line 46). The mapper task is configured with *NT* mapper objects, each an instance of class MyMapper; the *pattern* argument is passed to each mapper object (line 47).

The program also configures a reducer task (line 50), which can run on any available node. The reducer task includes a *customizer object* that is an instance of class MyCustomizer (line 51) and a reducer object that is an instance of class MyReducer (line 52). The customizer object lets you include extra functionality in the reducer task, as we will see later. (You can configure a mapper task with a customizer object too; I don't need that in this program.)

Lastly, the program starts the map-reduce job (line 54).

The mapper class MyMapper (line 68) introduces another feature of PJMR. The Mapper base class actually defines three methods: `start()`,

```
1  package edu.rit.pjmr.example;
2  import edu.rit.pj2.vbl.LongVbl;
3  import edu.rit.pjmr.Combiner;
4  import edu.rit.pjmr.Customizer;
5  import edu.rit.pjmr.Mapper;
6  import edu.rit.pjmr.PjmrJob;
7  import edu.rit.pjmr.Reducer;
8  import edu.rit.pjmr.TextFileSource;
9  import edu.rit.pjmr.TextId;
10 import java.util.Date;
11 import java.util.regex.Pattern;
12 public class WebLog01
13    extends PjmrJob<TextId,String,IPAddress,LongVbl>
14    {
15    // PJMR job main program.
16    public void main
17       (String[] args)
18       {
19       // Parse command line arguments.
20       if (args.length < 2 || args.length > 3) usage();
21       String[] nodes = args[0].split (",");
22       String file = args[1];
23       String pattern = null;
24       if (args.length >= 3)
25          {
26          pattern = args[2];
27          Pattern.compile (pattern); // Verify that pattern compiles
28          }
29
30       // Determine number of mapper threads.
31       int NT = Math.max (threads(), 1);
32
33       // Print provenance.
34       System.out.printf
35          ("$ java pj2 threads=%d edu.rit.pjmr.example.WebLog01",
36           NT);
37       for (String arg : args)
38          System.out.printf (" %s", arg);
39       System.out.println();
40       System.out.printf ("%s%n", new Date());
41       System.out.flush();
42
43       // Configure mapper tasks.
44       for (String node : nodes)
45          mapperTask (node)
46             .source (new TextFileSource (file))
47             .mapper (NT, MyMapper.class, pattern);
48
49       // Configure reducer task.
50       reducerTask()
51          .customizer (MyCustomizer.class)
52          .reducer (MyReducer.class);
53
54       startJob();
55       }
56
```

Listing 30.1. WebLog01.java (part 1)

`map()`, and `finish()`—not unlike the `start()`, `run()`, and `finish()` methods of a parallel loop. The mapper task operates as follows. Each thread in the mapper task gets its own separate mapper object. The thread calls the mapper object's `start()` method, once only, at the start of the map-reduce job. The `start()` method's arguments are an array of argument strings passed to the mapper task from the job main program, and a reference to the per-thread combiner. The `start()` method can do initializations, if necessary, before commencing the processing of records. If the `start()` method is omitted, the default is to do nothing. Next, the thread repeatedly gets a record from the mapper task's source object and passes the record's key and value, along with the per-thread combiner reference, to the mapper object's `map()` method. As we have already seen, the `map()` method extracts relevant information from the record and adds zero or more (*K*,*V*) pairs to the combiner. After the source object signals that there are no more records, the thread calls the mapper object's `finish()` method, once only, which can do finalizations if necessary. If the `finish()` method is omitted, the default is to do nothing.

In the WebLog01 program, the MyMapper class has a `start()` method (lines 74–80). The `args` argument is an array of strings specified back in the job main program, when the mapper object was configured (line 47):

```
.mapper (NT, MyMapper.class, pattern);
```

The argument strings, if any, appear after the mapper class name. There can be any number of argument strings, including none. Here, there is one argument string, namely the pattern (regular expression) that a web log entry must match in order to be included in the analysis, or null if the user did not specify a pattern. If a pattern was specified, the `start()` method compiles the pattern for later use and stores the resulting Pattern object in the `pattern` field (line 72); otherwise, the `pattern` field remains null.

Note how information specified by the user—in this case, the pattern—flows from the `pj2` command line to the job main program and from there to all the mapper objects, via the argument strings. This is how user input can affect the analysis the map-reduce job performs.

The MyMapper class's `map()` method (line 82) first checks whether the `inValue` argument, which is the contents of one record (line) from the web log file, matches the pattern the user specified, if any (line 87). If the line matches, or if no pattern was specified, the line is to be included in the analysis. The `map()` method then extracts the initial portion of the line up to the first whitespace character; this is the IP address string. The `map()` method constructs an IPAddress object from that string and adds the IPAddress object as the key, along with a LongVbl.Sum object initialized to 1 as the value (line 71), to the combiner. Thus, the number of occurrences associated with the IP address in the combiner is increased by 1. (This is similar to what the concordance program in the previous chapter did.) The WebLog01 program

```
58    // Print a usage message and exit.
59    private static void usage()
60       {
61       System.err.println ("Usage: java pj2 [threads=<NT>] " +
62          "edu.rit.pjmr.example.WebLog01 <nodes> <file> " +
63          "[<pattern>]");
64       throw new IllegalArgumentException();
65       }
66
67    // Mapper class.
68    private static class MyMapper
69       extends Mapper<TextId,String,IPAddress,LongVbl>
70       {
71       private static final LongVbl ONE = new LongVbl.Sum (1L);
72       private Pattern pattern;
73
74       public void start
75          (String[] args,
76           Combiner<IPAddress,LongVbl> combiner)
77          {
78          if (args[0] != null)
79             pattern = Pattern.compile (args[0]);
80          }
81
82       public void map
83          (TextId inKey,   // File name and line number
84           String inValue, // Line from file
85           Combiner<IPAddress,LongVbl> combiner)
86          {
87          if (pattern == null || pattern.matcher (inValue) .find())
88             {
89             int i = 0;
90             while (i < inValue.length() &&
91                ! Character.isWhitespace (inValue.charAt (i)))
92                   ++ i;
93             combiner.add (new IPAddress (inValue.substring (0, i)),
94                ONE);
95             }
96          }
97       }
98
99    // Reducer task customizer class.
100   private static class MyCustomizer
101      extends Customizer<IPAddress,LongVbl>
102      {
103      public boolean comesBefore
104         (IPAddress key_1, LongVbl value_1,
105          IPAddress key_2, LongVbl value_2)
106         {
107         if (value_1.item > value_2.item)
108            return true;
109         else if (value_1.item < value_2.item)
110            return false;
111         else
112            return key_1.compareTo (key_2) < 0;
113         }
114      }
115
```

Listing 30.1. WebLog01.java (part 2)

doesn't care about the rest of the web log file line, so the `map()` method just ignores the rest.

The MyCustomizer class (line 100) introduces yet another feature of PJMR. You can configure a mapper task or a reducer task, or both, with a *customizer object*. Here, I configured a customizer object only in the reducer task (line 51). The Customizer base class defines three methods: `start()`, `finish()`, and `comesBefore()`. Here, I only need the `comesBefore()` method. (See the Javadoc for class Customizer for descriptions of the other methods.) If a reducer task includes a customizer object, and if the customizer object's `comesBefore()` method is defined, the reducer task will *sort* the (*K*,*V*) pairs in the overall combiner before sending the pairs to the reducer object. The `comesBefore()` method's arguments are the key and value from one (*K*,*V*) pair and the key and value from another (*K*,*V*) pair. The `comesBefore()` method must return true if the first pair comes before the second pair in the desired sorted order. If the reducer task does not include a customizer object, or if the reducer task includes a customizer object but the `comesBefore()` method is not defined, the reducer task will not sort the (*K*,*V*) pairs. (This is what happened in the concordance program in the previous chapter.)

In the WebLog01 program, a (*K*,*V*) pair consists of an IP address (key) and a number of occurrences (value). The reducer task's customizer's `comesBefore()` method (line 103) compares the pairs such that pairs with larger numbers of occurrences come first. For equal numbers of occurrences, pairs with smaller IP addresses come first. This puts IP addresses that made the most requests at the top of the program's output.

Finally, the MyReducer class (line 117) generates the WebLog01 program's output. The `reduce()` method's arguments are the key and the value of one (*K*,*V*) pair from the overall combiner. Remember, these are fed to the `reduce()` method in sorted order. The `reduce()` method simply prints the number of occurrences (the value) followed by the IP address (the key).

I ran the WebLog01 program over the ten days' worth of web logs on the `tardis` cluster. Because I had partitioned the web log files among all ten backend nodes `dr00` through `dr09`, I specified all the node names in the command, causing the program to run a mapper task on every node. I also specified `threads=2`, causing the program to run two mapper objects in separate threads in each mapper task. This reduced the program's running time by about half, to about 11.0 seconds. Here are the first few lines the program printed (beginning with the provenance):

```
$ java pj2 threads=2 edu.rit.pjmr.example.WebLog01
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/weblog/weblog.txt
Mon Aug 03 08:57:39 EDT 2015
907360  108.45.93.78
38119   129.21.36.111
28307   129.21.36.110
```

```
116    // Reducer class.
117    private static class MyReducer
118       extends Reducer<IPAddress,LongVbl>
119       {
120       public void reduce
121          (IPAddress key, // IP address
122           LongVbl value) // Number of requests
123          {
124          System.out.printf ("%s\t%s%n", value, key);
125          System.out.flush();
126          }
127       }
128    }
```

Listing 30.1. WebLog01.java (part 3)

```
1      // Mapper class.
2      private static class MyMapper
3         extends Mapper<TextId,String,String,LongVbl>
4         {
5         private static final LongVbl ONE = new LongVbl.Sum (1L);
6         private static final Pattern getPattern =
7            Pattern.compile ("GET /([^ \\t\\n\\x0B\\f\\r/]+)");
8         private Pattern ipaddrPattern;
9         private Pattern pattern;
10
11        public void start
12           (String[] args,
13            Combiner<String,LongVbl> combiner)
14           {
15           ipaddrPattern = Pattern.compile ("^" + args[0] + "\\s");
16           if (args[1] != null)
17              pattern = Pattern.compile (args[1]);
18           }
19
20        public void map
21           (TextId inKey,   // File name and line number
22            String inValue, // Line from file
23            Combiner<String,LongVbl> combiner)
24           {
25           if ((pattern == null || pattern.matcher (inValue) .find())
26                 && ipaddrPattern.matcher (inValue) .find())
27              {
28              Matcher m = getPattern.matcher (inValue);
29              if (m.find())
30                 combiner.add (m.group (1), ONE);
31              }
32           }
33        }
```

Listing 30.2. WebLog05.java, class MyMapper

```
22890   68.180.229.52
19579   144.76.63.35
13929   32.208.223.22
9715    157.55.39.63
7474    66.249.67.46
7414    66.249.67.39
```

Wait a minute. During those ten days, one IP address—108.45.93.78—made *over nine hundred thousand* web requests. A DNS loookup shows that this IP address is associated with a host name that I believe represents a Verizon home customer. What URLs was that person, or web crawler as the case may be, requesting?

To find out, I wrote another PJMR map-reduce program, class edu.rit.pjmr.example.WebLog05. Most of the WebLog05 program is quite similar to the WebLog01 program, so I have listed only the significantly different piece, the mapper class (Listing 30.2). On the WebLog05 command line, the user specifies an IP address. The web log lines that begin with that IP address (and that match an optional regular expression pattern if any) are analyzed. For each such line, the mapper extracts the URL that follows the string `"GET /"`. If the URL refers to a document in a subdirectory, only the top-level directory (up to the next `"/"` character) is extracted. The program counts and prints the number of occurrences of each unique URL.

I ran the WebLog05 program over the web logs on the `tardis` cluster, specifying IP address 108.45.93.78. The program run took 4.7 seconds—less time than the previous run, because the program didn't have to analyze every web log entry. Here are the first few lines the program printed:

```
$ java pj2 threads=2 edu.rit.pjmr.example.WebLog05
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/weblog/weblog.txt 108.45.93.78
Mon Aug 03 09:37:30 EDT 2015
404807  ~ark
270939  usr
107520  ~ats
56753   %7Eats
10502   ~hpb
8094    ~wrc
6069    ~rpj
2019    ~afb
1568    ~ncs
```

This person appears to be requesting a dump of the entire RIT CS web site, for purposes unknown. The above URLs refer to the personal web sites of RIT CS faculty. The top web site belongs to user `~ark`, who happens to be me. (I have a pretty big web site. It includes web pages for all the courses I've taught over several decades, as well as Javadoc web pages for several large software libraries I've written, including the Parallel Java 2 Library.)

To narrow down when this person made all these web requests, I ran the

WebLog05 program again, this time specifying a specific date as the matching pattern. Here is what one of the program runs (running time: 4.1 seconds) printed, showing what this person requested on July 12, 2015:

```
$ java pj2 threads=2 edu.rit.pjmr.example.WebLog05
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/weblog/weblog.txt 108.45.93.78 12/Jul/2015
Mon Aug 03 09:40:27 EDT 2015
132682  usr
26454   ~ark
9133    ~hpb
7083    ~ats
5574    ~wrc
5087    ~rpj
2019    ~afb
1291    ~ncs
1278    ~anhinga
```

And here is what this person requested on July 13, 2015:

```
$ java pj2 threads=2 edu.rit.pjmr.example.WebLog05
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/weblog/weblog.txt 108.45.93.78 13/Jul/2015
Mon Aug 03 09:40:50 EDT 2015
378351  ~ark
138257  usr
100413  ~ats
56741   %7Eats
2520    ~wrc
1369    ~hpb
982     ~rpj
698     ~cs4
648     ~rwd
```

The chief takeaway message here is that map-reduce is a versatile pattern for analyzing large, unstructured data sets. You don't have to set up a database, with schemas, tables, and SQL; map-reduce works just fine to "query" plain text files. By writing various map-reduce programs, you can do as many different kinds of analysis as you want; and the map-reduce programs are neither lengthy nor complicated. You can run the analysis programs as many times as you want over the same data set, specifying different parameters each time. You can speed up the analysis programs by partitioning the data among the disk drives of a cluster parallel computer's backend nodes and running the map-reduce jobs in multiple parallel tasks on the cluster.

Under the Hood

PJMR is designed so that the first stage of reduction happens in the combiner objects inside the mapper tasks. The resulting (K,V) pairs are sent from each mapper task to the reducer task via tuple space. The second stage of reduction happens in the combiner object inside the reducer task.

This design has several consequences for PJMR map-reduce programs. As I mentioned earlier, the key data type *K* and the value data type *V* must be streamable or serializable, so they can be packaged into tuples and sent across the network.

Each per-thread combiner object in each mapper task, each per-task combiner in each mapper task, and the overall combiner in the reducer task are implemented as hash table data structures stored in main memory. Consequently, the (*K*,*V*) pairs stored in each task's combiners *must fit in the main memory* of the node where each task is running. The whole point of the map-reduce pattern is to *reduce* the raw data down to just the information essential to the analysis. PJMR assumes that this essential information is *much smaller* than the raw data and is therefore able to fit into the nodes' main memories.

Specifically, the combiner objects are stored in the JVM's heap. Depending on the size of your data, you might have to increase the JVM's maximum heap size above the default. To do this, include the `jvmflags` option on the `pj2` command line. For example, the command

```
$ java pj2 jvmflags="-Xmx2000m" ...
```

causes each backend process's JVM to run with the flag `"-Xmx2000m"`, which sets the JVM's maximum heap size to 2,000 megabytes. However, it's a bad idea to set the maximum heap size larger than the node's physical main memory size. If your map-reduce job needs more memory to store intermediate results than is available on your nodes, then you should be using a library like Hadoop, not PJMR. (Hadoop stores intermediate results on disk, not in main memory.)

Besides being stored in the nodes' main memories, the mapper tasks' combiners' (*K*,*V*) pairs have to be sent via tuple space to the reducer task. The amount of network traffic is proportional to the numbers and sizes of the keys and values in all the combiners after the mapping stage has finished. Again, PJMR assumes that these (*K*,*V*) pairs occupy much less space than the raw data, so that the map-reduce program spends most of its time running the mappers and comparatively little time sending and receiving tuples. If this is not the case, use Hadoop.

Points to Remember

- Large scale data analysis tasks need to run on a cluster.
- Partition the data set to be analyzed across the cluster backend nodes' disk drives. Don't store the whole data set on one disk drive.
- The program's running time can potentially be reduced by configuring each mapper task with more than one mapper object. Each mapper object runs in a separate thread (core) on a backend node.

- Define the mapper class's `start()`, `map()`, and `finish()` methods as necessary.
- Pass user-specified arguments from the job main program to the mapper objects as necessary. The arguments appear as an array of strings passed to the mapper object's `start()` method.
- Define the reducer class's `start()`, `reduce()`, and `finish()` methods as necessary.
- Pass user-specified arguments from the job main program to the reducer object as necessary. The arguments appear as an array of strings passed to the reducer object's `start()` method.
- Configure a mapper task or a reducer task with a customizer object as necessary.
- To sort the final (*K*,*V*) pairs into a desired order, configure the reducer task with a customizer object whose `comesBefore()` method defines the desired ordering.
- By writing various map-reduce programs, you can do as many different kinds of analysis of a data set as you want.
- You can run the analysis programs as many times as you want over the same data set, specifying different parameters each time.

Chapter 31
Big Data Analysis

My last parallel programming example will be another big data analysis program using the map-reduce paradigm. The program itself is not all that different from the previous concordance program and web log analysis programs. What differs is the scale of the data.

The U.S. government's National Oceanic and Atmospheric Administration (NOAA) runs the National Climatic Data Center (NCDC). NCDC maintains the Global Historical Climatology Network (GHCN), which collects and archives climate related data from weather stations all over the globe. NCDC publishes a data set of historical data from the GHCN going all the way back to before 1900, available on the web.* The GHCN data set consists of over 98,000 text files totaling 25.9 gigabytes. Google would consider this a puny data set, but to me that's big data, or at least middling big data.

Here is a portion of one of the GHCN data set files:

```
AEM00041194198301TMAX  276  S  302  S  252  S  235  S-9999
AEM00041194198301TMIN  140  S  134  S  143  S  158  S  145  S
AEM00041194198301TAVG  195H S  198H S  191H S  194H S  185H S
AEM00041194198302TMAX  196  S  210  S  201  S  195  S  212  S
AEM00041194198302TMIN-9999     111  S   99  S-9999     117  S
AEM00041194198302TAVG  164H S  154H S  157H S  151H S  171H S
AEM00041194198303TMAX  192  S  199  S  208  S  222  S  297  S
AEM00041194198303TMIN-9999     128  S  123  S  120  S-9999
AEM00041194198303TAVG  162H S  168H S  162H S  168H S  232H S
```

Each line contains climate data recorded at a particular weather station for each day of a particular month and year. (The lines are too long to display in their entirety.) The first eleven characters are the weather station identifier, `AEM00041194` in this file, which happens to be the Dubai International Airport. The next four characters are the year, `1983`. The next two characters are the month: January (`01`), February (`02`), and so on. The next four characters indicate the kind of climate data on the line: `TMAX` says the line contains maximum temperatures, `TMIN` is minimum temperatures, `TAVG` is average temperatures. (Some files include other kinds of climate data too.) The rest of the line consists of 31 eight-character fields, one for each day of the month. The first five characters of the field are the temperature reading in units of 0.1 Celsius. A value of `-9999` indicates a missing or nonexistent reading. The seventh character of the field is a space if the reading is valid, or is some non-space character if the reading is not valid. (The sixth and eighth characters of the field do not concern us.) Thus, at weather station `AEM00041194`, the maximum temperature recorded on January 1, 1983 was 27.6 C; on January 2, 30.2 C; on February 1, 19.6 C; and so on.

I want to write a program that calculates the average maximum temperature for each year in the data set. The program's output should look like this:

* http://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-ghcn

```
1  package edu.rit.pjmr.example;
2  import edu.rit.numeric.ListXYSeries;
3  import edu.rit.numeric.plot.Plot;
4  import edu.rit.pj2.vbl.DoubleVbl;
5  import edu.rit.pjmr.Combiner;
6  import edu.rit.pjmr.Customizer;
7  import edu.rit.pjmr.Mapper;
8  import edu.rit.pjmr.PjmrJob;
9  import edu.rit.pjmr.Reducer;
10 import edu.rit.pjmr.TextDirectorySource;
11 import edu.rit.pjmr.TextId;
12 import java.io.File;
13 import java.io.IOException;
14 import java.util.Date;
15 public class MaxTemp01
16    extends PjmrJob<TextId,String,Integer,DoubleVbl>
17    {
18    //PJMR job main program.
19    public void main
20       (String[] args)
21       {
22       // Parse command line arguments.
23       if (args.length != 5) usage();
24       String[] nodes = args[0].split (",");
25       String directory = args[1];
26       int yearlb = Integer.parseInt (args[2]);
27       int yearub = Integer.parseInt (args[3]);
28       String plotfile = args[4];
29
30       // Determine number of mapper threads.
31       int NT = Math.max (threads(), 1);
32
33       // Print provenance.
34       System.out.printf
35          ("$ java pj2 threads=%d edu.rit.pjmr.example.MaxTemp01",
36           NT);
37       for (String arg : args)
38          System.out.printf (" %s", arg);
39       System.out.println();
40       System.out.printf ("%s%n", new Date());
41       System.out.flush();
42
43       // Configure mapper tasks.
44       for (String node : nodes)
45          mapperTask (node)
46             .source (new TextDirectorySource (directory))
47             .mapper (NT, MyMapper.class, ""+yearlb, ""+yearub);
48
49       // Configure reducer task.
50       reducerTask()
51          .runInJobProcess()
52          .customizer (MyCustomizer.class)
53          .reducer (MyReducer.class, plotfile);
54
55       startJob();
56       }
57
```

Listing 31.1. MaxTemp01.java (part 1)

```
1900    17.7
1901    17.4
1902    17.5
1903    17.2
1904    17.2
1905    17.4
1906    17.7
1907    17.4
. . .
```

To produce the first line of output, the program considers all year-1900 `TMAX` records in all files of the data set; extracts all the temperature readings from these records, omitting missing and invalid readings; averages these readings; and prints the year (1900) followed by the average (17.7 C). The program does the same for 1901, 1902, and so on. In fact, I want to specify the range of years the program should analyze.

American mathematician Richard W. Hamming (1915–1998) famously said, "The purpose of computing is insight, not numbers."* A list of numbers, like the program output above, doesn't give much insight. To get better insight into climate trends, I also want the analysis program to generate a *plot* of average maximum temperature versus year.

The climate data analysis program is edu.rit.pjmr.example.MaxTemp01 (Listing 31.1). The program is run with this command:

```
$ java pj2 [threads=NT] edu.rit.pjmr.example.MaxTemp01 nodes \
  directory yearlb yearub plotfile
```

The program's command line arguments are as follows:

- *`NT`* is the number of mapper threads in each mapper task (default: 1).
- *`nodes`* is a comma-separated list of one or more backend node names in the cluster. The program will run a mapper task on each node.
- *`directory`* is the name of the directory containing the climate data files to be analyzed. The program assumes that each node has a directory with this name, containing a portion of the climate data files.
- *`yearlb`* is the first year to be analyzed.
- *`yearub`* is the last year to be analyzed.
- *`plotfile`* is the name of the file in which to store the plot of average maximum temperature versus year.

The main program class (line 15) extends class PjmrJob (line 16). As in the previous chapters, the source's key and value data types are TextId and String. The program will be mapping years (integers) to temperatures in units of 0.1 Celsius (floating point numbers). So for the mapper's (*K*,*V*) data types, I will use class Integer (in the Java platform) for the key *K* and class Dou-

* R. Hamming. *Numerical Methods for Scientists and Engineers.* McGraw-Hill, 1962, page vii.

```
58    // Print a usage message and exit.
59    private static void usage()
60       {
61       System.err.println ("Usage: java pj2 [threads=<NT>] " +
62          "edu.rit.pjmr.example.MaxTemp01 <nodes> <directory> " +
63          "<yearlb> <yearub> <plotfile>");
64       throw new IllegalArgumentException();
65       }
66
67    // Mapper class.
68    private static class MyMapper
69       extends Mapper<TextId,String,Integer,DoubleVbl>
70       {
71       private int yearlb;
72       private int yearub;
73
74       // Record year range.
75       public void start
76          (String[] args,
77           Combiner<Integer,DoubleVbl> combiner)
78          {
79          yearlb = Integer.parseInt (args[0]);
80          yearub = Integer.parseInt (args[1]);
81          }
82
83       // Process one data record.
84       public void map
85          (TextId id,
86           String data,
87           Combiner<Integer,DoubleVbl> combiner)
88          {
89          // If record is not 269 characters, ignore it.
90          if (data.length() < 269) return;
91
92          // Look only at TMAX records.
93          if (! data.substring (17, 21) .equals ("TMAX")) return;
94
95          // Get year.
96          int year = 0;
97          try
98             {
99             year = parseInt (data, 11, 15);
100            }
101         catch (NumberFormatException exc)
102            {
103            return;
104            }
105
106         // Look only at years in the specified range.
107         if (yearlb > year || year > yearub) return;
108
109         // Look at each day's maximum temperature.
110         for (int i = 21; i < 269; i += 8)
111            {
112            int tmax = 0;
113            try
114               {
115               tmax = parseInt (data, i, i + 5);
```

Listing 31.1. MaxTemp01.java (part 2)

bleVbl (in the Parallel Java 2 Library) for the value *V*. Class Integer wraps a value of type `int`. Class DoubleVbl provides a reduction variable that wraps a value of type `double`.

After parsing the command line arguments and printing the provenance, the program configures one mapper task to run on each specified backend node of the cluster (lines 44–47). The mapper task's source is an instance of class edu.rit.pjmr.TextDirectorySource (line 46). Whereas class TextFileSource (in the previous chapters) obtains records from just one file, class TextDirectorySource obtains records from all the files in the given directory; each record is one line from one of the files; the order in which the files are read is not specified. That's appropriate for the GHCN data set, which consists of a bunch of files rather than one file. The mapper task is configured with *NT* mapper objects, each an instance of class MyMapper (line 47). The lower and upper bounds of the range of years to be analyzed are passed as arguments to each mapper object.

The program configures one reducer task with a customizer object and a reducer object (lines 50–53). The plot file name is passed as an argument to the reducer object. The reducer task is run in the job process on the cluster's frontend node (line 51) so the reducer task can write the plot file in the user's account.

The mapper class's `start()` method (line 75) stores the lower and upper bound years for later use. The `map()` method (line 84) processes one record (line) from a climate data set file. Only `TMAX` records whose year lies in the specified range are analyzed. For each temperature reading that is not missing and that is valid, the `map()` method adds a (*K*,*V*) pair to the combiner. The key *K* is the year. The value *V* is the temperature reading, as an instance of class DoubleVbl.Mean. This class's reduction operation computes the average (mean) of all the values that are fed into a reduction variable.

The customer class's `comesBefore()` method (line 135) causes the reducer task to sort the (*K*,*V*) pairs in the overall combiner into ascending order of year. This is the order I want for the program's printout and plot.

The MaxTemp01 program's reducer class (line 148) does a bit more than the reducer classes in the previous chapters. Besides printing the results, the reducer task creates a plot of the results. To do so, the reducer class uses two classes from the Parallel Java 2 Library: class edu.rit.numeric.ListXYSeries and class edu.rit.numeric.plot.Plot.

Class ListXYSeries encapsulates a series of two-dimensional points (*x*,*y*). The *x* and *y* coordinates are type `double`. The class has methods to add a point to the series, retrieve a point from the series, and so on. The points stored in the series specify the points to appear on the plot.

Class Plot encapsulates a plot. The class has methods to set a title for the plot; set the length, title, lower and upper bounds, and so on of the X axis and the Y axis; as well as other methods too numerous to mention here. (Refer to

```
116                }
117             catch (NumberFormatException exc)
118                {
119                continue;
120                }
121             char qflag = data.charAt (i + 6);
122             if (tmax != -9999 && qflag == ' ')
123                combiner.add (year, new DoubleVbl.Mean (tmax));
124             }
125          }
126
127       // Parse an integer, ignoring leading and trailing whitespace.
128       private static int parseInt (String s, int from, int to)
129          {
130          return Integer.parseInt (s.substring (from, to) .trim());
131          }
132       }
133
134    // Reducer task customizer class.
135    private static class MyCustomizer
136       extends Customizer<Integer,DoubleVbl>
137       {
138       // Sort into ascending order of year (key).
139       public boolean comesBefore
140          (Integer key_1, DoubleVbl value_1,
141           Integer key_2, DoubleVbl value_2)
142          {
143          return key_1 < key_2;
144          }
145       }
146
147    // Reducer class.
148    private static class MyReducer
149       extends Reducer<Integer,DoubleVbl>
150       {
151       File plotfile;
152       ListXYSeries data;
153
154       // Initialize data series.
155       public void start (String[] args)
156          {
157          plotfile = new File (args[0]);
158          data = new ListXYSeries();
159          }
160
161       // Print the year (key) and the average maximum temperature
162       // (value).
163       public void reduce
164          (Integer key,
165           DoubleVbl value)
166          {
167          int year = key.intValue();
168          double mean = value.doubleValue()/10.0;
169          data.add (year, mean);
170          System.out.printf ("%d\t%.1f%n", year, mean);
171          System.out.flush();
172          }
```

Listing 31.1. MaxTemp01.java (part 3)

the Javadoc for class Plot.) The class also has a method to add a data series to the plot; the data is stored in a ListXYSeries object. You can specify the shape and color of the dots that are plotted; you can specify the width and color of the lines that connect the dots; you can specify to omit the dots or the lines. The default settings usually suffice.

Once you have set up a Plot object, you can display it in a window on the screen. You can also store the plot in a file. Later, you can view the plot with this command, which runs the View program in the Parallel Java 2 Library:

```
$ java View plotfile
```

The View program reads the plot from the given file and displays it in a window. The window has menus that let you alter the plot's appearance, store the plot as a PNG image file, store the plot as a PostScript file, and so on. (I've used class Plot to create all the plots in this book.)

Returning to the code, the reducer task calls the reducer's `start()` method (line 155) after all the mappers' (*K*,*V*) pairs have been absorbed into the overall combiner. The `start()` method stores the plot file name passed in as an argument and creates a ListXYSeries object to hold the data for the average maximum temperature versus year plot.

The reducer task calls the reducer's `reduce()` method (line 163) for each (*K*,*V*) pair from the overall combiner, *K* being the year and *V* being the average maximum temperature for that year in units of 0.1 Celsius. The reducer task presents the (*K*,*V*) pairs to the `reduce()` method in sorted order, that is, in ascending order of year. To convert the temperature value from units of 0.1 Celsius to units of Celsius, the `reduce()` method divides the temperature value by 10. The `reduce()` method prints the year and the temperature. The `reduce()` method also adds an (*x*,*y*) point to the plot series, with *x* being the year and *y* being the temperature.

After that, the reducer task calls the reducer's `finish()` method (line 174). The `finish()` method creates a Plot object; sets the plot's X and Y axis titles; turns off plotting dots for the data points, so that only the connecting lines will be plotted; and adds the data accumulated in the ListXYSeries object to the plot. The `finish()` method then stores the plot in the given file.

I downloaded the GHCN data set from the NCDC web site and stored one-tenth of the climate data files on the disk drives of each of the ten backend nodes of the `tardis` cluster. Then I ran the following command to find the average maximum temperature for each year from 1900 through 2015. I stored the plot in a "drawing file" named `MaxTemp01.dwg`. I configured the mapper tasks with two mapper objects running in two threads (cores) on each node. The program's running time was about 250 seconds.

```
$ java pj2 threads=2 edu.rit.pjmr.example.MaxTemp01 \
  dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09 \
  /var/tmp/ark/NOAA/ghcnd_all 1900 2015 MaxTemp01.dwg
```

```
173         // Generate plot.
174         public void finish()
175            {
176            Plot plot = new Plot()
177               .xAxisTitle ("Year")
178               .yAxisTitle ("Average Maximum Temperature (C)")
179               .seriesDots (null)
180               .xySeries (data);
181            try
182               {
183               Plot.write (plot, plotfile);
184               }
185            catch (IOException exc)
186               {
187               exc.printStackTrace (System.err);
188               }
189            }
190         }
191   }
```

Listing 31.1. MaxTemp01.java (part 4)

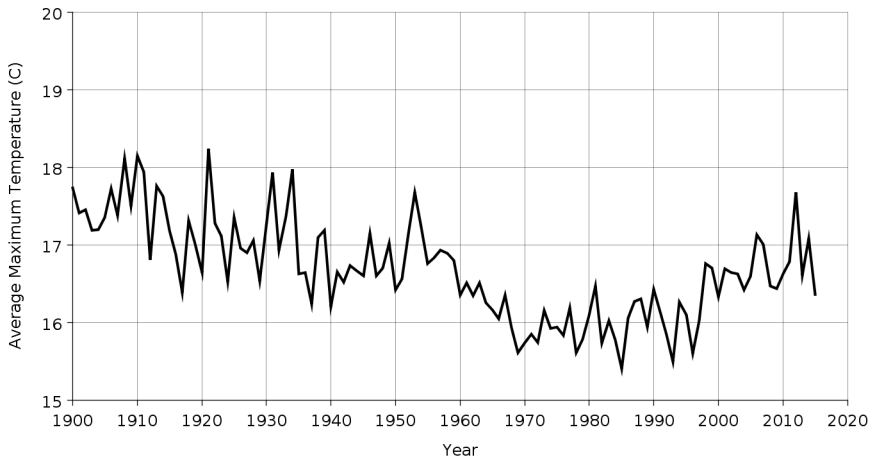


Figure 31.1. Climate data analysis results

Figure 31.1 shows the plot the program produced. Despite year-to-year fluctuations, some overall trends are apparent. The worldwide average maximum temperature trended downwards until about 1970, flattened out until about 1990, then started trending upwards at a faster rate. Recalling Hamming's maxim, this is an insight that would be difficult to see in a list of numbers. I personally believe the GHCN climate data shows evidence of global warming; you may disagree. Still, this example shows how a parallel map-reduce program can extract meaningful insights from a large data set.

Points to Remember

- "The purpose of computing is insight, not numbers."
  —Richard W. Hamming
- Consider whether presenting analysis results in the form of plots might yield better insights. However, do present analysis results as tables of numbers also.
- Use the Parallel Java 2 Library classes edu.rit.numeric.plot.Plot, edu.rit-.numeric.ListXYSeries, and other classes in package edu.rit.numeric to create plots right in your program.
- Store a plot object in a file for later viewing. Use the View program to view the plot.
- A task that needs to read or write files in the user's account should be specified to run in the job process on the cluster's frontend node.