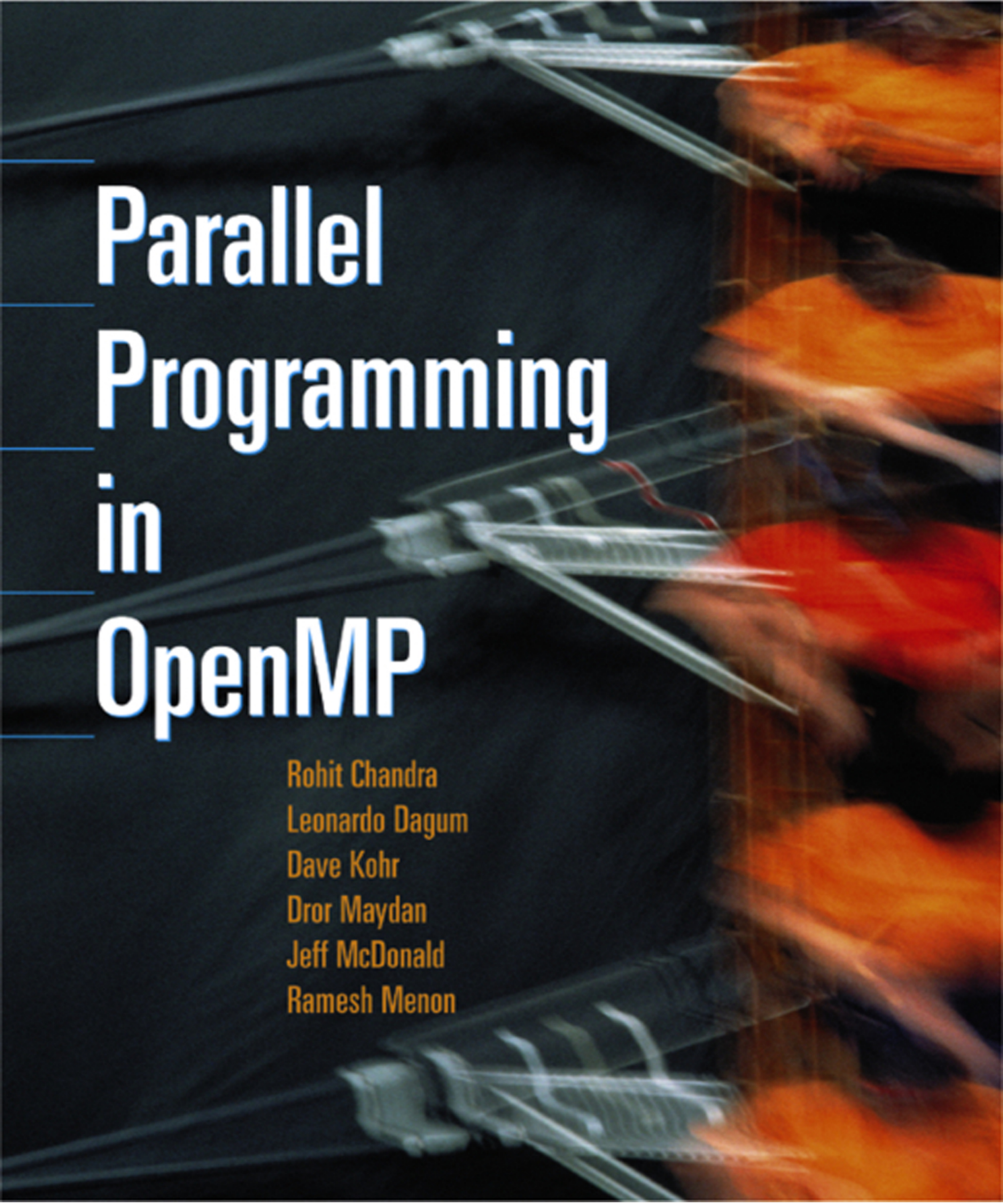
Parallel Programming in OpenMP

Rohit Chandra
Leonardo Dagum
Dave Kohr
Dror Maydan
Jeff McDonald
Ramesh Menon



Parallel Programming in OpenMP

About the Authors

Rohit Chandra is a chief scientist at NARUS, Inc., a provider of internet business infrastructure solutions. He previously was a principal engineer in the Compiler Group at Silicon Graphics, where he helped design and implement OpenMP.

Leonardo Dagum works for Silicon Graphics in the Linux Server Platform Group, where he is responsible for the I/O infrastructure in SGI's scalable Linux server systems. He helped define the OpenMP Fortran API. His research interests include parallel algorithms and performance modeling for parallel systems.

Dave Kohr is a member of the technical staff at NARUS, Inc. He previously was a member of the technical staff in the Compiler Group at Silicon Graphics, where he helped define and implement the OpenMP.

Dror Maydan is director of software at Tensilica, Inc., a provider of application-specific processor technology. He previously was an engineering department manager in the Compiler Group of Silicon Graphics, where he helped design and implement OpenMP.

Jeff McDonald owns SolidFX, a private software development company. As the engineering department manager at Silicon Graphics, he proposed the OpenMP API effort and helped develop it into the industry standard it is today.

Ramesh Menon is a staff engineer at NARUS, Inc. Prior to NARUS, Ramesh was a staff engineer at SGI, representing SGI in the OpenMP forum. He was the founding chairman of the OpenMP Architecture Review Board (ARB) and supervised the writing of the first OpenMP specifications.

Parallel Programming in OpenMP

Rohit Chandra
Leonardo Dagum
Dave Kohr
Dror Maydan
Jeff McDonald
Ramesh Menon

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ACADEMIC PRESS
A Harcourt Science and Technology Company
SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

Senior Editor	Denise E. M. Penrose
Senior Production Editor	Edward Wade
Editorial Coordinator	Emilia Thiuri
Cover Design	Ross Carron Design
Cover Image	© Stone/Gary Benson
Text Design	Rebecca Evans & Associates
Technical Illustration	Dartmouth Publishing, Inc.
Composition	Nancy Logan
Copyeditor	Ken DellaPenta
Proofreader	Jennifer McClain
Indexer	Ty Koontz
Printer	Courier Corporation

ACADEMIC PRESS
A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
http://www.academicpress.com

Academic Press
Harcourt Place, 32 Jamestown Road, London, NW1 7BY, United Kingdom
http://www.academicpress.com

Morgan Kaufmann Publishers
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
http://www.mkp.com

05 04 03 02 01 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data is available for this book.

ISBN 1-55860-671-8

This book is printed on acid-free paper.

We would like to dedicate this book to our families:

Rohit—To my wife and son, Minnie and Anand, and my parents

Leo—To my lovely wife and daughters, Joanna, Julia, and Anna

Dave—To my dearest wife, Jingjun

Dror—To my dearest wife and daughter, Mary and Daniella, and my parents, Dalia and Dan

Jeff—To my best friend and wife, Dona, and my parents

Ramesh—To Beena, Abishek, and Sithara, and my parents

This Page Intentionally Left Blank

Foreword

by John L. Hennessy
President, Stanford University

For a number of years, I have believed that advances in software, rather than hardware, held the key to making parallel computing more commonplace. In particular, the lack of a broadly supported standard for programming shared-memory multiprocessors has been a chasm both for users and for software vendors interested in porting their software to these multiprocessors. OpenMP represents the first vendor-independent, commercial "bridge" across this chasm.

Such a bridge is critical to achieve portability across different shared-memory multiprocessors. In the parallel programming world, the challenge is to obtain both this functional portability as well as performance portability. By performance portability, I mean the ability to have reasonable expectations about how parallel applications will perform on different multiprocessor architectures. OpenMP makes important strides in enhancing performance portability among shared-memory architectures.

Parallel computing is attractive because it offers users the potential of higher performance. The central problem in parallel computing for nearly 20 years has been to improve the "gain to pain ratio." Improving this ratio, with either hardware or software, means making the gains in performance come at less pain to the programmer! Shared-memory multiprocessing was developed with this goal in mind. It provides a familiar programming model, allows parallel applications to be developed incrementally, and supports fine-grain communication in a very cost effective manner. All of these factors make it easier to achieve high performance on parallel machines. More recently, the development of cache-coherent distributed shared memory has provided a method for scaling shared-memory architectures to larger numbers of processors. In many ways, this development removed the hardware barrier to scalable, shared-memory multiprocessing.

OpenMP represents the important step of providing a software standard for these shared-memory multiprocessors. Our goal now must be to learn how to program these machines effectively (i.e., with a high value for gain/pain). This book will help users accomplish this important goal. By focusing its attention on how to use OpenMP, rather than on defining the standard, the authors have made a significant contribution to the important task of mastering the programming of multiprocessors.

Contents

Preface

OpenMP is a parallel programming model for shared memory and distributed shared memory multiprocessors. Pioneered by SGI and developed in collaboration with other parallel computer vendors, OpenMP is fast becoming the de facto standard for parallelizing applications. There is an independent OpenMP organization today with most of the major computer manufacturers on its board, including Compaq, Hewlett-Packard, Intel, IBM, Kuck & Associates (KAI), SGI, Sun, and the U.S. Department of Energy ASCI Program. The OpenMP effort has also been endorsed by over 15 software vendors and application developers, reflecting the broad industry support for the OpenMP standard.

Unfortunately, the main information available about OpenMP is the OpenMP specification (available from the OpenMP Web site at *www.openmp.org*). Although this is appropriate as a formal and complete specification, it is not a very accessible format for programmers wishing to use OpenMP for developing parallel applications. This book tries to fulfill the needs of these programmers.

This introductory-level book is primarily designed for application developers interested in enhancing the performance of their applications by utilizing multiple processors. The book emphasizes practical concepts and tries to address the concerns of real application developers. Little background is assumed of the reader other than single-processor programming experience and the ability to follow simple program examples in the

Fortran programming language. While the example programs are usually in Fortran, all the basic OpenMP constructs are presented in Fortran, C, and C++.

The book tries to balance the needs of both beginning and advanced parallel programmers. The introductory material is a must for programmers new to parallel programming, but may easily be skipped by those familiar with the basic concepts of parallelism. The latter are more likely to be interested in applying known techniques using individual OpenMP mechanisms, or in addressing performance issues in their parallel program.

The authors are all SGI engineers who were involved in the design and implementation of OpenMP and include compiler writers, application developers, and performance engineers. We hope that our diverse backgrounds are positively reflected in the breadth and depth of the material in the text.

Organization of the Book

This book is organized into six chapters.

Chapter 1, "Introduction," presents the motivation for parallel programming by giving examples of performance gains achieved by some real-world application programs. It describes the different kinds of parallel computers and the one targeted by OpenMP. It gives a high-level glimpse of OpenMP and includes some historical background.

Chapter 2, "Getting Started with OpenMP," gives a bird's-eye view of OpenMP and describes what happens when an OpenMP parallel program is executed. This chapter is a must-read for programmers new to parallel programming, while advanced readers need only skim the chapter to get an overview of the various components of OpenMP.

Chapter 3, "Exploiting Loop-Level Parallelism," focuses on using OpenMP to direct the execution of loops across multiple processors. Loop-level parallelism is among the most common forms of parallelism in applications and is also the simplest to exploit. The constructs described in this chapter are therefore the most popular of the OpenMP constructs.

Chapter 4, "Beyond Loop-Level Parallelism: Parallel Regions," focuses on exploiting parallelism beyond individual loops, such as parallelism across multiple loops and parallelization of nonloop constructs. The techniques discussed in this chapter are useful when trying to parallelize an increasingly large portion of an application and are crucial for scalable performance on large numbers of processors.

Chapter 5, "Synchronization," describes the synchronization mechanisms in OpenMP. It describes the situations in a shared memory parallel program when explicit synchronization is necessary. It presents the various OpenMP synchronization constructs and also describes how programmers may build their own custom synchronization in a shared memory parallel program.

Chapter 6, "Performance," discusses the performance issues that arise in shared memory parallel programs. The only reason to write an OpenMP program is scalable performance, and this chapter is a must-read to realize these performance goals.

Appendix A, A Quick Reference to OpenMP, which details various OpenMP directives, runtime library routines, lock routines, and so on, can be found immediately following Chapter 6.

A presentation note regarding the material in the text. The code fragments in the examples are presented using a different font, shown below:

```
This is a code sample in an example.
```

```
This is an OpenMP construct in an example.
```

Within the examples all OpenMP constructs are highlighted in boldface monofont. Code segments such as variable names, or OpenMP constructs used within the text are simply highlighted using the regular text font, but in italics as in this *sample.*

Acknowledgments

This book is based entirely on the OpenMP effort, which would not have been possible without the members of the OpenMP Architectural Review Board who participated in the design of the OpenMP specification, including the following organizations: Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, International Business Machines (IBM), Kuck & Associates (KAI), Silicon Graphics, Inc. (SGI), Sun Microsystems, Inc., and the U.S. Department of Energy ASCI Program.

This book benefited tremendously from the detailed perusal of several reviewers, including George Adams of Purdue University, Tim Mattson of Intel Corporation, David Kuck of Kuck & Associates, Ed Rothberg of Ilog, Inc. (CPLEX Division), and Mary E. Zosel of the U.S. Department of Energy ASCI Program.

Writing a book is a very time-consuming operation. This book in particular is the joint effort of six authors, which presents a substantial additional challenge. The publishers, especially our editor Denise Penrose, were very patient and encouraging throughout the long process. This book would just not have been possible without her unflagging enthusiasm.

Our employer, SGI, was most cooperative in helping us make time to work on the book. In particular, we would like to thank Ken Jacobsen, Ron Price, Willy Shih, and Ross Towle for their unfailing support for this effort, and Wesley Jones and Christian Tanasescu for his help with several applications discussed in the book.

Finally, the invisible contributors to this effort are our individual families, who let us steal time in various ways so that we could moonlight on this project.

CHAPTER 1

Introduction

ENHANCED COMPUTER APPLICATION PERFORMANCE is the only practical purpose of parallel processing. Many computer applications continue to exceed the capabilities delivered by the fastest single processors, so it is compelling to harness the aggregate capabilities of multiple processors to provide additional computational power. Even applications with adequate single-processor performance on high-end systems often enjoy a significant cost advantage when implemented in parallel on systems utilizing multiple, lower-cost, commodity microprocessors. Raw performance and price performance: these are the direct rewards of parallel processing.

The cost that a software developer incurs to attain meaningful parallel performance comes in the form of additional design and programming complexity inherent in producing correct and efficient computer code for multiple processors. If computer application performance or price performance is important to you, then keep reading. It is the goal of both OpenMP and this book to minimize the complexity introduced when adding parallelism to application code.

In this chapter, we introduce the benefits of parallelism through examples and then describe the approach taken in OpenMP to support the development of parallel applications. The shared memory multiprocessor target architecture is described at a high level, followed by a brief explanation of why OpenMP was developed and how it came to be. Finally, a road map of the remainder of the book is presented to help navigate through the rest of the text. This will help readers with different levels of experience in parallel programming come up to speed on OpenMP as quickly as possible.

This book assumes that the reader is familiar with general algorithm development and programming methods. No specific experience in parallel programming is assumed, so experienced parallel programmers will want to use the road map provided in this chapter to skip over some of the more introductory-level material. Knowledge of the Fortran language is somewhat helpful, as examples are primarily presented in this language. However, most programmers will probably easily understand them. In addition, there are several examples presented in C and C++ as well.

1.1 Performance with OpenMP

Applications that rely on the power of more than a single processor are numerous. Often they provide results that are time-critical in one way or another. Consider the example of weather forecasting. What use would a highly accurate weather model for tomorrow's forecast be if the required computation takes until the following day to complete? The computational complexity of a weather problem is directly related to the accuracy and detail of the requested forecast simulation. Figure 1.1 illustrates the performance of the MM5 (mesoscale model) weather code when implemented using OpenMP [GDS 95] on an SGI Origin 2000 multiprocessor.[1] The graph shows how much faster the problem can be solved when using multiple processors: the forecast can be generated 70 times faster by using 128 processors compared to using only a single processor. The factor by which the time to solution can be improved compared to using only a single processor is called *speedup*.

These performance levels cannot be supported by a single-processor system. Even the fastest single processor available today, the Fujitsu VPP5000, which can perform at a peak of 1512 Mflop/sec, would deliver MM5 performance equivalent to only about 10 of the Origin 2000 processors demonstrated in the results shown in Figure 1.1.[2] Because of these

[1] MM5 was developed by and is copyrighted by the Pennsylvania State University (Penn State) and the University Corporation for Atmospheric Research (UCAR). MM5 results on SGI Origin 2000 courtesy of Wesley Jones, SGI.

[2] *http://box.mmm.ucar.edu/mm5/mpp/helpdesk/20000106.html.*

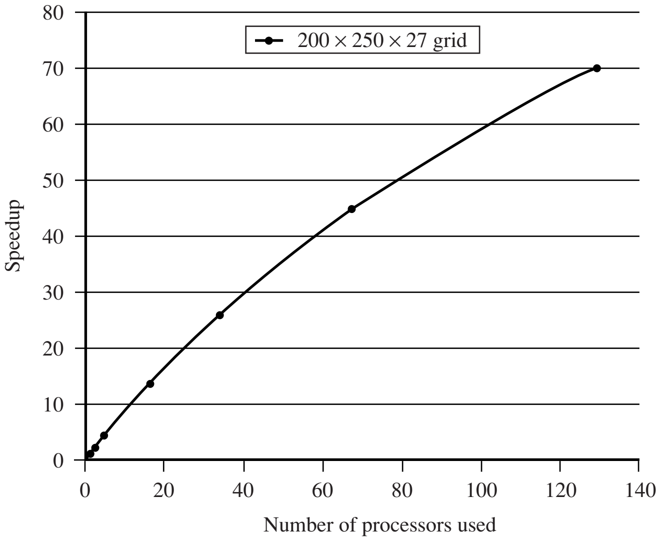


Figure 1.1 Performance of the MM5 weather code.

dramatic performance gains through parallel execution, it becomes possible to provide detailed and accurate weather forecasts in a timely fashion.

The MM5 application is a specific type of computational fluid dynamics (CFD) problem. CFD is routinely used to design both commercial and military aircraft and has an ever-increasing collection of nonaerospace applications as diverse as the simulation of blood flow in arteries [THZ 98] to the ideal mixing of ingredients during beer production. These simulations are very computationally expensive by the nature of the complex mathematical problem being solved. Like MM5, better and more accurate solutions require more detailed simulations, possible only if additional computational resources are available. The NAS Parallel Benchmarks [NASPB 91] are an industry standard series of performance benchmarks that emulate CFD applications as implemented for multiprocessor systems. The results from one of these benchmarks, known as APPLU, is shown in Figure 1.2 for a varying number of processors. Results are shown for both the OpenMP and MPI implementations of APPLU. The performance increases by a factor of more than 90 as we apply up to 64 processors to the same large simulation.[3]

[3] This seemingly impossible performance feat, where the application speeds up by a factor greater than the number of processors utilized, is called *superlinear* speedup and will be explained by cache memory effects, discussed in Chapter 6.

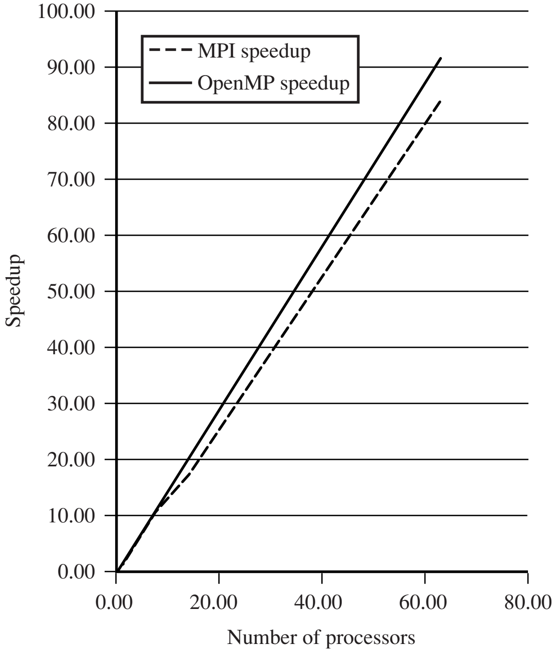


Figure 1.2 Performance of the NAS parallel benchmark, APPLU.

Clearly, parallel computing can have an enormous impact on application performance, and OpenMP facilitates access to this enhanced performance. Can any application be altered to provide such impressive performance gains and scalability over so many processors? It very likely can. How likely are such gains? It would probably take a large development effort, so it really depends on the importance of additional performance and the corresponding investment of effort. Is there a middle ground where an application can benefit from a modest number of processors with a correspondingly modest development effort? Absolutely, and this is the level at which most applications exploit parallel computer systems today. OpenMP is designed to support *incremental parallelization,* or the ability to parallelize an application a little at a time at a rate where the developer feels additional effort is worthwhile.

Automobile crash analysis is another application area that significantly benefits from parallel processing. Full-scale tests of car crashes are very expensive, and automobile companies as well as government agencies would like to do more tests than is economically feasible. Computational crash analysis applications have been proven to be highly accurate

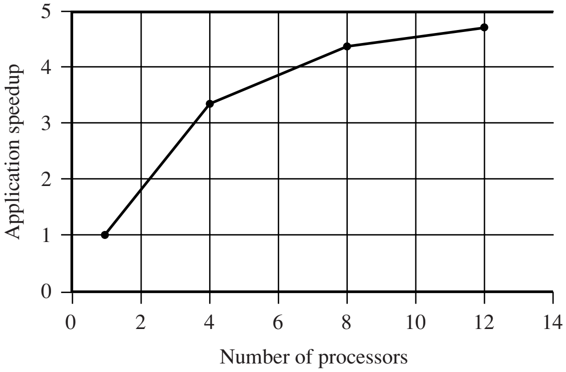


Figure 1.3 Performance of a leading crash code.

and much less expensive to perform than full-scale tests. The simulations are computationally expensive, with turnaround times for a single crash test simulation often measured in days even on the world's fastest supercomputers. This can directly impact the schedule of getting a safe new car design on the road, so performance is a key business concern. Crash analysis is a difficult problem class in which to realize the huge range of scalability demonstrated in the previous MM5 example. Figure 1.3 shows an example of performance from a leading parallel crash simulation code parallelized using OpenMP. The performance or speedup yielded by employing eight processors is close to 4.3 on the particular example shown [RAB 98]. This is a modest improvement compared to the weather code example, but a vitally important one to automobile manufacturers whose economic viability increasingly depends on shortened design cycles.

This crash simulation code represents an excellent example of incremental parallelism. The parallel version of most automobile crash codes evolved from the single-processor implementation and initially provided parallel execution in support of very limited but key functionality. Some types of simulations would get a significant advantage from parallelization, while others would realize little or none. As new releases of the code are developed, more and more parts of the code are parallelized. Changes range from very simple code modifications to the reformulation of central algorithms to facilitate parallelization. This incremental process helps to deliver enhanced performance in a timely fashion while following a conservative development path to maintain the integrity of application code that has been proven by many years of testing and verification.

1.2 A First Glimpse of OpenMP

Developing a parallel computer application is not so different from writing a sequential (i.e., single-processor) application. First the developer forms a clear idea of what the program needs to do, including the inputs to and the outputs from the program. Second, algorithms are designed that not only describe how the work will be done, but also, in the case of parallel programs, how the work can be distributed or decomposed across multiple processors. Finally, these algorithms are implemented in the application program or code. OpenMP is an implementation model to support this final step, namely, the implementation of parallel algorithms. It leaves the responsibility of designing the appropriate parallel algorithms to the programmer and/or other development tools.

OpenMP is not a new computer language; rather, it works in conjunction with either standard Fortran or C/C++. It is comprised of a set of compiler directives that describe the parallelism in the source code, along with a supporting library of subroutines available to applications (see Appendix A). Collectively, these directives and library routines are formally described by the application programming interface (API) now known as OpenMP.

The directives are instructional notes to any compiler supporting OpenMP. They take the form of source code comments (in Fortran) or *#pragmas* (in C/C++) in order to enhance application portability when porting to non-OpenMP environments. The simple code segment in Example 1.1 demonstrates the concept.

Example 1.1 Simple OpenMP program.

```
          program hello
          print *, "Hello parallel world from threads:"
!$omp parallel
          print *, omp_get_thread_num()
!$omp end parallel
          print *, "Back to the sequential world."
          end
```

The code in Example 1.1 will result in a single *Hello parallel world from threads:* message followed by a unique number for each thread started by the *!$omp parallel* directive. The total number of threads active will be equal to some externally defined degree of parallelism. The closing *Back to the sequential world* message will be printed once before the program terminates.

One way to set the degree of parallelism in OpenMP is through an operating system–supported environment variable named *OMP_NUM_THREADS*. Let us assume that this symbol has been previously set equal to 4. The program will begin execution just like any other program utilizing a single processor. When execution reaches the print statement bracketed by the *!$omp parallel/!$omp end parallel* directive pair, three additional copies of the print code are started. We call each copy a thread, or thread of execution. The OpenMP routine *omp_get_num_threads()* reports a unique thread identification number between 0 and *OMP_NUM_THREADS* – 1. Code after the *parallel* directive is executed by each thread independently, resulting in the four unique numbers from 0 to 3 being printed in some unspecified order. The order may possibly be different each time the program is run. The *!$omp end parallel* directive is used to denote the end of the code segment that we wish to run in parallel. At that point, the three extra threads are deactivated and normal sequential behavior continues. One possible output from the program, noting again that threads are numbered from 0, could be

```
Hello parallel world from threads:
1
3
0
2
Back to the sequential world.
```

This output occurs because the threads are executing without regard for one another, and there is only one screen showing the output. What if the digit of a thread is printed before the carriage return is printed from the previously printed thread number? In this case, the output could well look more like

```
Hello parallel world from threads:
13

02

Back to the sequential world.
```

Obviously, it is important for threads to cooperate better with each other if useful and correct work is to be done by an OpenMP program. Issues like these fall under the general topic of synchronization, which is addressed throughout the book, with Chapter 5 being devoted entirely to the subject.

This trivial example gives a flavor of how an application can go parallel using OpenMP with very little effort. There is obviously more to cover before useful applications can be addressed but less than one might think. By the end of Chapter 2 you will be able to write useful parallel computer code on your own!

Before we cover additional details of OpenMP, it is helpful to understand how and why OpenMP came about, as well as the target architecture for OpenMP programs. We do this in the subsequent sections.

1.3 The OpenMP Parallel Computer

OpenMP is primarily designed for shared memory multiprocessors. Figure 1.4 depicts the programming model or logical view presented to a programmer by this class of computer. The important aspect for our current purposes is that all of the processors are able to directly access all of the memory in the machine, through a logically direct connection. Machines that fall in this class include *bus-based* systems like the Compaq AlphaServer, all multiprocessor PC servers and workstations, the SGI Power Challenge, and the SUN Enterprise systems. Also in this class are distributed shared memory (DSM) systems. DSM systems are also known as *ccNUMA* (Cache Coherent Non-Uniform Memory Access) systems, examples of which include the SGI Origin 2000, the Sequent NUMA-Q 2000, and the HP 9000 V-Class. Details on how a machine provides the programmer with this logical view of a globally addressable memory are unimportant for our purposes at this time, and we describe all such systems simply as "shared memory."

The alternative to a shared configuration is distributed memory, in which each processor in the system is only capable of directly addressing memory physically associated with it. Figure 1.5 depicts the classic form of a distributed memory system. Here, each processor in the system can only address its own local memory, and it is always up to the programmer to manage the mapping of the program data to the specific memory system where data isto be physically stored. To access information in memory

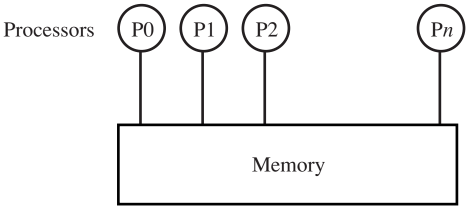


Figure 1.4 A canonical shared memory architecture.

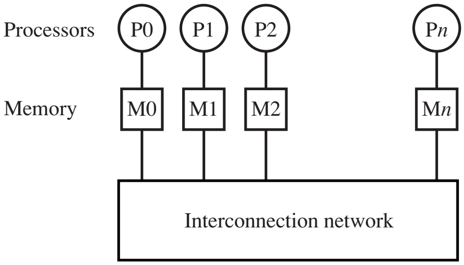


Figure 1.5 A canonical message passing (nonshared memory) architecture.

connected to other processors, the user must explicitly pass messages through some network connecting the processors. Examples of systems in this category include the IBM SP-2 and clusters built up of individual computer systems on a network, or networks of workstations (NOWs). Such systems are usually programmed with explicit message passing libraries such as Message Passing Interface (MPI) [PP96] and Parallel Virtual Machine (PVM). Alternatively, a high-level language approach such as High Performance Fortran (HPF) [KLS 94] can be used in which the compiler generates the required low-level message passing calls from parallel application code written in the language.

From this very simplified description one may be left wondering why anyone would build or use a distributed memory parallel machine. For systems with larger numbers of processors the shared memory itself can become a bottleneck because there is limited bandwidth capability that can be engineered into a single-memory subsystem. This places a practical limit on the number of processors that can be supported in a traditional shared memory machine, on the order of 32 processors with current technology. ccNUMA systems such as the SGI Origin 2000 and the HP 9000 V-Class have combined the logical view of a shared memory machine with physically distributed/globally addressable memory. Machines of hundreds and even thousands of processors can be supported in this way while maintaining the simplicity of the shared memory system model. A programmer writing highly scalable code for such systems must account for the underlying distributed memory system in order to attain top performance. This will be examined in Chapter 6.

1.4 Why OpenMP?

The last decade has seen a tremendous increase in the widespread availability and affordability of shared memory parallel systems. Not only have such multiprocessor systems become more prevalent, they also contain increasing numbers of processors. Meanwhile, most of the high-level, portable and/or standard parallel programming models are designed for distributed memory systems. This has resulted in a serious disconnect between the state of the hardware and the software APIs to support them. The goal of OpenMP is to provide a standard and portable API for writing shared memory parallel programs.

Let us first examine the state of hardware platforms. Over the last several years, there has been a surge in both the quantity and scalability of shared memory computer platforms. Quantity is being driven very quickly in the low-end market by the rapidly growing PC-based multiprocessor server/workstation market. The first such systems contained only two processors, but this has quickly evolved to four- and eight-processor systems, and scalability shows no signs of slowing. The growing demand for business/enterprise and technical/scientific servers has driven the quantity of shared memory systems in the medium- to high-end class machines as well. As the cost of these machines continues to fall, they are deployed more widely than traditional mainframes and supercomputers. Typical of these are bus-based machines in the range of 2 to 32 RISC processors like the SGI Power Challenge, the Compaq AlphaServer, and the Sun Enterprise servers.

On the software front, the various manufacturers of shared memory parallel systems have supported different levels of shared memory programming functionality in proprietary compiler and library products. In addition, implementations of distributed memory programming APIs like MPI are also available for most shared memory multiprocessors. Application portability between different systems is extremely important to software developers. This desire, combined with the lack of a standard shared memory parallel API, has led most application developers to use the message passing models. This has been true even if the target computer systems for their applications are all shared memory in nature. A basic goal of OpenMP, therefore, is to provide a portable standard parallel API specifically for programming shared memory multiprocessors.

We have made an implicit assumption thus far that shared memory computers and the related programming model offer some inherent advantage over distributed memory computers to the application developer. There are many pros and cons, some of which are addressed in Table 1.1. Programming with a shared memory model has been typically associated with ease of use at the expense of limited parallel scalability. Distributed memory programming on the other hand is usually regarded as more difficult but the only way to achieve higher levels of parallel scalability. Some of this common wisdom is now being challenged by the current genera-

Table 1.1 Comparing shared memory and distributed memory programming models.

Feature	*Shared Memory*	*Distributed Memory*
Ability to parallelize small parts of an application at a time	Relatively easy to do. Reward versus effort varies widely.	Relatively difficult to do. Tends to require more of an all-or-nothing effort.
Feasibility of scaling an application to a large number of processors	Currently, few vendors provide scalable shared memory systems (e.g., ccNUMA systems).	Most vendors provide the ability to cluster nonshared memory systems with moderate to high-performance interconnects.
Additional complexity over serial code to be addressed by programmer	Simple parallel algorithms are easy and fast to implement. Implementation of highly scalable complex algorithms is supported but more involved.	Significant additional overhead and complexity even for implementing simple and localized parallel constructs.
Impact on code quantity (e.g., amount of additional code required) and code quality (e.g., the readability of the parallel code)	Typically requires a small increase in code size (2–25%) depending on extent of changes required for parallel scalability. Code readability requires some knowledge of shared memory constructs, but is otherwise maintained as directives embedded within serial code.	Tends to require extra copying of data into temporary message buffers, resulting in a significant amount of message handling code. Developer is typically faced with extra code complexity even in non-performance-critical code segments. Readability of code suffers accordingly.
Availability of application development and debugging environments	Requires a special compiler and a runtime library that supports OpenMP. Well-written code will compile and run correctly on one processor without an OpenMP compiler. Debugging tools are an extension of existing serial code debuggers. Single memory address space simplifies development and support of a rich debugger functionality.	Does not require a special compiler. Only a library for the target computer is required, and these are generally available. Debuggers are more difficult to implement because a direct, global view of all program memory is not available.

tion of scalable shared memory servers coupled with the functionality offered by OpenMP.

There are other implementation models that one could use instead of OpenMP, including Pthreads [NBF 96], MPI [PP 96], HPF [KLS 94], and so on. The choice of an implementation model is largely determined by the type of computer architecture targeted for the application, the nature of the application, and a healthy dose of personal preference.

The message passing programming model has now been very effectively standardized by MPI. MPI is a portable, widely available, and accepted standard for writing message passing programs. Unfortunately, message passing is generally regarded as a difficult way to program. It requires that the program's data structures be explicitly partitioned, and typically the entire application must be parallelized in order to work with the partitioned data structures. There is usually no incremental path to parallelizing an application in this manner. Furthermore, modern multiprocessor architectures are increasingly providing hardware support for cache-coherent shared memory; therefore, message passing is becoming unnecessary and overly restrictive for these systems.

Pthreads is an accepted standard for shared memory in the low end. However it is not targeted at the technical or high-performance computing (HPC) spaces. There is little Fortran support for Pthreads, and even for many HPC class C and C++ language-based applications, the Pthreads model is lower level and awkward, being more suitable for task parallelism rather than data parallelism. Portability with Pthreads, as with any standard, requires that the target platform provide a standard-conforming implementation of Pthreads.

The option of developing new computer languages may be the cleanest and most efficient way to provide support for parallel processing. However, practical issues make the wide acceptance of a new computer language close to impossible. Nobody likes to rewrite old code to new languages. It is difficult to justify such effort in most cases. Also, educating and convincing a large enough group of developers to make a new language gain critical mass is an extremely difficult task.

A pure library approach was initially considered as an alternative for what eventually became OpenMP. Two factors led to rejection of a library-only methodology. First, it is far easier to write portable code using directives because they are automatically ignored by a compiler that does not support OpenMP. Second, since directives are recognized and processed by a compiler, they offer opportunities for compiler-based optimizations. Likewise, a pure directive approach is difficult as well: some necessary functionality is quite awkward to express through directives and ends up looking like executable code in directive syntax. Therefore, a small API defined by a mixture of directives and some simple library calls was chosen. The OpenMP API does address the portability issue of OpenMP library calls in non-OpenMP environments, as will be shown later.

1.5 History of OpenMP

Although OpenMP is a recently (1997) developed industry standard, it is very much an evolutionary step in a long history of shared memory programming models. The closest previous attempt at a standard shared memory programming model was the now dormant ANSI X3H5 standards effort [X3H5 94]. X3H5 was never formally adopted as a standard largely because interest waned as a wide variety of distributed memory machines came into vogue during the late 1980s and early 1990s. Machines like the Intel iPSC and the TMC Connection Machine were the platforms of choice for a great deal of pioneering work on parallel algorithms. The Intel machines were programmed through proprietary message passing libraries and the Connection Machine through the use of data parallel languages like CMFortran and C* [TMC 91]. More recently, languages such as High Performance Fortran (HPF) [KLS 94] have been introduced, similar in spirit to CMFortran.

All of the high-performance shared memory computer hardware vendors support some subset of the OpenMP functionality, but application portability has been almost impossible to attain. Developers have been restricted to using only the most basic common functionality that was available across all compilers, which most often limited them to parallelization of only single loops. Some third-party compiler products offered more advanced solutions including more of the X3H5 functionality. However, all available methods lacked direct support for developing highly scalable parallel applications like those examined in Section 1.1. This scalability shortcoming inherent in all of the support models is fairly natural given that mainstream scalable shared memory computer hardware has only become available recently.

The OpenMP initiative was motivated from the developer community. There was increasing interest in a standard they could reliably use to move code between the different parallel shared memory platforms they supported. An industry-based group of application and compiler specialists from a wide range of leading computer and software vendors came together as the definition of OpenMP progressed. Using X3H5 as a starting point, adding more consistent semantics and syntax, adding functionality known to be useful in practice but not covered by X3H5, and directly supporting scalable parallel programming, OpenMP went from concept to adopted industry standard from July 1996 to October 1997. Along the way, the OpenMP Architectural Review Board (ARB) was formed. For more information on the ARB, and as a great OpenMP resource in general, check out the Web site at *www.OpenMP.org*.

1.6 Navigating the Rest of the Book

This book is written to be introductory in nature while still being of value to those approaching OpenMP with significant parallel programming experience. Chapter 2 provides a general overview of OpenMP and is designed to get a novice up and running basic OpenMP-based programs. Chapter 3 focuses on the OpenMP mechanisms for exploiting loop-level parallelism. Chapter 4 presents the constructs in OpenMP that go beyond loop-level parallelism and exploit more scalable forms of parallelism based on parallel regions. Chapter 5 describes the synchronization constructs in OpenMP. Finally, Chapter 6 discusses the performance issues that arise when programming a shared memory multiprocessor using OpenMP.

CHAPTER 2

Getting Started with OpenMP

2.1 Introduction

A PARALLEL PROGRAMMING LANGUAGE MUST PROVIDE SUPPORT for the three basic aspects of parallel programming: specifying parallel execution, communicating between multiple threads, and expressing synchronization between threads. Most parallel languages provide this support through extensions to an existing sequential language; this has the advantage of providing parallel extensions within a familiar programming environment.

Different programming languages have taken different approaches to providing these extensions. Some languages provide additional constructs within the base language to express parallel execution, communication, and so on (e.g., the *forall* construct in Fortran-95 [ABM 97, MR 99]). Rather than designing additional language constructs, other approaches provide directives that can be embedded within existing sequential programs in the base language; this includes approaches such as HPF [KLS 94]. Finally, application programming interfaces such as MPI [PP 96] and various threads packages such as Pthreads [NBF 96] don't design new language constructs: rather, they provide support for expressing parallelism through calls to runtime library routines.

OpenMP takes a directive-based approach for supporting parallelism. It consists of a set of directives that may be embedded within a program written in a base language such as Fortran, C, or C++. There are two compelling benefits of a directive-based approach that led to this choice: The first is that this approach allows the same code base to be used for development on both single-processor and multiprocessor platforms; on the former, the directives are simply treated as comments and ignored by the language translator, leading to correct serial execution. The second related benefit is that it allows an incremental approach to parallelism—starting from a sequential program, the programmer can embellish the same existing program with directives that express parallel execution.

This chapter gives a high-level overview of OpenMP. It describes the basic constructs as well as the runtime execution model (i.e., the effect of these constructs when the program is executed). It illustrates these basic constructs with several examples of increasing complexity. This chapter will provide a bird's-eye view of OpenMP; subsequent chapters will discuss the individual constructs in greater detail.

2.2 OpenMP from 10,000 Meters

At its most elemental level, OpenMP is a set of compiler directives to express shared memory parallelism. These directives may be offered within any base language—at this time bindings have been defined for Fortran, C, and C++ (within the C/C++ languages, directives are referred to as "pragmas"). Although the basic semantics of the directives is the same, special features of each language (such as allocatable arrays in Fortran 90 or class objects in C++) require additional semantics over the basic directives to support those features. In this book we largely use Fortran 77 in our examples simply because the Fortran specification for OpenMP has existed the longest, and several Fortran OpenMP compilers are available.

In addition to directives, OpenMP also includes a small set of runtime library routines and environment variables (see Figure 2.1). These are typically used to examine and modify the execution parameters. For instance, calls to library routines may be used to control the degree of parallelism exploited in different portions of the program.

These three pieces—the directive-based language extensions, the runtime library routines, and the environment variables—taken together define what is called an *application programming interface,* or API. The OpenMP API is independent of the underlying machine/operating system. OpenMP compilers exist for all the major versions of UNIX as well as Windows NT. Porting a properly written OpenMP program from one system to another should simply be a matter of recompiling. Furthermore, C and

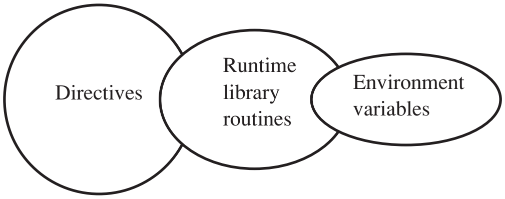


Figure 2.1 The components of OpenMP.

C++ OpenMP implementations provide a standard include file, called ***omp.h,*** that provides the OpenMP type definitions and library function prototypes. This file should therefore be included by all C and C++ OpenMP programs.

The language extensions in OpenMP fall into one of three categories: control structures for expressing parallelism, data environment constructs for communicating between threads, and synchronization constructs for coordinating the execution of multiple threads. We give an overview of each of the three classes of constructs in this section, and follow this with simple example programs in the subsequent sections. Prior to all this, however, we must present some sundry details on the syntax for OpenMP statements and conditional compilation within OpenMP programs. Consider it like medicine: it tastes bad but is good for you, and hopefully you only have to take it once.

2.2.1 OpenMP Compiler Directives or Pragmas

Before we present specific OpenMP constructs, we give an overview of the general syntax of directives (in Fortran) and pragmas (in C and C++).

Fortran source may be specified in either fixed form or free form. In fixed form, a line that begins with one of the following prefix keywords (also referred to as *sentinels*):

```
!$omp ...
c$omp ...
*$omp ...
```

and contains either a space or a zero in the sixth column is treated as an OpenMP directive by an OpenMP compiler, and treated as a comment (i.e., ignored) by a non-OpenMP compiler. Furthermore, a line that begins with one of the above sentinels and contains a character other than a space or a zero in the sixth column is treated as a continuation directive line by an OpenMP compiler.

In free-form Fortran source, a line that begins with the sentinel

```
!$omp ...
```

is treated as an OpenMP directive. The sentinel may begin in any column so long as it appears as a single word and is preceded only by white space. A directive that needs to be continued on the next line is expressed

```
!$opm <directive> &
```

with the ampersand as the last token on that line.

C and C++ OpenMP pragmas follow the syntax

```
#pragma omp ...
```

The *omp* keyword distinguishes the pragma as an OpenMP pragma, so that it is processed as such by OpenMP compilers and ignored by non-OpenMP compilers.

Since OpenMP directives are identified by a well-defined prefix, they are easily ignored by non-OpenMP compilers. This allows application developers to use the same source code base for building their application on either kind of platform—a parallel version of the code on platforms that support OpenMP, and a serial version of the code on platforms that do not support OpenMP. Furthermore, most OpenMP compilers provide an option to disable the processing of OpenMP directives. This allows application developers to use the same source code base for building both parallel and sequential versions of an application using just a compile-time flag.

Conditional Compilation

The selective disabling of OpenMP constructs applies only to directives, whereas an application may also contain statements that are specific to OpenMP. This could include calls to runtime library routines or just other code that should only be executed in the parallel version of the code. This presents a problem when compiling a serial version of the code (i.e., with OpenMP support disabled), such as calls to library routines that would not be available.

OpenMP addresses this issue though a *conditional compilation* facility that works as follows. In Fortran any statement that we wish to be included only in the parallel compilation may be preceded by a specific sentinel. Any statement that is prefixed with the sentinel *!$, c$,* or **$* starting in column one in fixed form, or the sentinel *!$* starting in any column but preceded only by white space in free form, is compiled only when OpenMP support is enabled, and ignored otherwise. These prefixes can therefore be used to mark statements that are relevant only to the parallel version of the program.

In Example 2.1, the line containing the call to *omp_get_thread_num* starts with the prefix *!$* in column one. As a result it looks like a normal Fortran comment and will be ignored by default. When OpenMP compilation is enabled, not only are directives with the *!$omp* prefix enabled, but the lines with the *!$* prefix are also included in the compiled code. The two characters that make up the prefix are replaced by white spaces at compile time. As a result only the parallel version of the program (i.e., with OpenMP enabled) makes the call to the subroutine. The serial version of the code ignores that entire statement, including the call and the assignment to *iam*.

Example 2.1 Using the conditional compilation facility.

```
      iam = 0

      ! The following statement is compiled only when
      ! OpenMP is enabled, and is ignored otherwise

!$    iam = omp_get_thread_num()
      ...

      ! The following statement is incorrect, since
      ! the sentinel is not preceeded by white space
      ! alone
      y = x !$ + offset
      ...
      ! This is the correct way to write the above
      ! statement. The right-hand side of the
      ! following assignment is x + offset with OpenMP
      ! enabled, and only x otherwise.
      y = x &
!$&   + offset
```

In C and C++ all OpenMP implementations are required to define the preprocessor macro name _OPENMP to the value of the year and month of the approved OpenMP specification in the form *yyyymm*. This macro may be used to selectively enable/disable the compilation of any OpenMP specific piece of code.

The conditional compilation facility should be used with care since the prefixed statements are not executed during serial (i.e., non-OpenMP) compilation. For instance, in the previous example we took care to initialize the ***iam*** variable with the value zero, followed by the conditional assignment of the thread number to the variable. The initialization to zero ensures that the variable is correctly defined in serial compilation when the subsequent assignment is ignored.

That completes our discussion of syntax in OpenMP. In the remainder of this section we present a high-level overview of the three categories of language extension comprising OpenMP: parallel control structures, data environment, and synchronization.

2.2.2 Parallel Control Structures

Control structures are constructs that alter the flow of control in a program. We call the basic execution model for OpenMP a fork/join model, and parallel control structures are those constructs that fork (i.e., start) new threads, or give execution control to one or another set of threads.

OpenMP adopts a minimal set of such constructs. Experience has shown that only a few control structures are truly necessary for writing most parallel applications. OpenMP includes a control structure only in those instances where a compiler can provide both functionality and performance over what a user could reasonably program.

OpenMP provides two kinds of constructs for controlling parallelism. First, it provides a directive to create multiple threads of execution that execute concurrently with each other. The only instance of this is the *parallel* directive: it encloses a block of code and creates a set of threads that each execute this block of code concurrently. Second, OpenMP provides constructs to divide work among an existing set of parallel threads. An instance of this is the *do* directive, used for exploiting loop-level parallelism. It divides the iterations of a loop among multiple concurrently executing threads. We present examples of each of these directives in later sections.

2.2.3 Communication and Data Environment

An OpenMP program always begins with a single thread of control that has associated with it an execution context or data environment (we will use the two terms interchangeably). This initial thread of control is referred to as the *master thread.* The execution context for a thread is the data address space containing all the variables specified in the program. This includes global variables, automatic variables within subroutines (i.e., allocated on the stack), as well as dynamically allocated variables (i.e., allocated on the heap).

The master thread and its execution context exist for the duration of the entire program. When the master thread encounters a parallel construct, new threads of execution are created along with an execution context for each thread. Let us now examine how the execution context for a parallel thread is determined.

Each thread has its own stack within its execution context. This private stack is used for stack frames for subroutines invoked by that thread. As a result, multiple threads may individually invoke subroutines and execute safely without interfering with the stack frames of other threads.

For all other program variables, the OpenMP *parallel* construct may choose to either share a single copy between all the threads or provide each thread with its own private copy for the duration of the parallel construct. This determination is made on a per-variable basis; therefore it is possible for threads to share a single copy of one variable, yet have a private per-thread copy of another variable, based on the requirements of the algorithms utilized. Furthermore, this determination of which variables are shared and which are private is made at each parallel construct, and may vary from one parallel construct to another.

This distinction between shared and private copies of variables during parallel constructs is specified by the programmer using OpenMP date scoping clauses (. . .) for individual variables. These clauses are used to determine the execution context for the parallel threads. A variable may have one of three basic attributes: *shared, private,* or *reduction.* These are discussed at some length in later chapters. At this early stage it is sufficient to understand that these scope clauses define the sharing attributes of an object.

A variable that has the ***shared*** scope clause on a parallel construct will have a single storage location in memory for the duration of that parallel construct. All parallel threads that reference the variable will always access the same memory location. That piece of memory is shared by the parallel threads. Communication between multiple OpenMP threads is therefore easily expressed through ordinary read/write operations on such shared variables in the program. Modifications to a variable by one thread are made available to other threads through the underlying shared memory mechanisms.

In contrast, a variable that has ***private*** scope will have multiple storage locations, one within the execution context of each thread, for the duration of the parallel construct. All read/write operations on that variable by a thread will refer to the private copy of that variable within that thread. This memory location is inaccessible to the other threads. The most common use of private variables is scratch storage for temporary results.

The *reduction* clause is somewhat trickier to understand, since reduction variables have both private and shared storage behavior. As the name implies, the reduction attribute is used on objects that are the target of an arithmetic reduction. Reduction operations are important to many applications, and the reduction attribute allows them to be implemented by the compiler efficiently. The most common example is the final summation of temporary local variables at the end of a parallel construct.

In addition to these three, OpenMP provides several other data scoping attributes. We defer a detailed discussion of these attributes until later chapters. For now, it is sufficient to understand the basic OpenMP mechanism: the data scoping attributes of individual variables may be controlled along with each OpenMP construct.

2.2.4 **Synchronization**

Multiple OpenMP threads communicate with each other through ordinary reads and writes to shared variables. However, it is often necessary to coordinate the access to these shared variables across multiple threads. Without any coordination between threads, it is possible that multiple threads may simultaneously attempt to modify the same variable, or that one thread may try to read a variable even as another thread is modifying that same variable. Such conflicting accesses can potentially lead to incorrect data values and must be avoided by explicit coordination between multiple threads. The term *synchronization* refers to the mechanisms by which a parallel program can coordinate the execution of multiple threads.

The two most common forms of synchronization are mutual exclusion and event synchronization. A mutual exclusion construct is used to control access to a shared variable by providing a thread exclusive access to a shared variable for the duration of the construct. When multiple threads are modifying the same variable, acquiring exclusive access to the variable before modifying it ensures the integrity of that variable. OpenMP provides mutual exclusion through a *critical* directive.

Event synchronization is typically used to signal the occurrence of an event across multiple threads. The simplest form of event synchronization is a *barrier.* A *barrier* directive in a parallel program defines a point where each thread waits for all other threads to arrive. Once all the threads arrive at that point, they can all continue execution past the barrier. Each thread is therefore guaranteed that all the code before the barrier has been completed across all other threads.

In addition to *critical* and *barrier,* OpenMP provides several other synchronization constructs. Some of these constructs make it convenient to express common synchronization patterns, while the others are useful in obtaining the highest performing implementation. These various constructs are discussed in greater detail in Chapter 5.

That completes our high-level overview of the language. Some of the concepts presented may not become meaningful until you have more experience with the language. At this point, however, we can begin presenting concrete examples and explain them using the model described in this section.

2.3 Parallelizing a Simple Loop

Enough of high-level concepts: let us look at a simple parallel program that shows how to execute a simple loop in parallel. Consider Example 2.2: a multiply-add, or *saxpy* loop as it is often called (for “single-precision *a*x* plus *y*”). In a true *saxpy* the variable *y* is an array, but for simplicity here we have *y* as just a scalar variable.

Example 2.2 A simple *saxpy*-like loop.

```
subroutine saxpy(z, a, x, y, n)
integer i, n
real z(n), a, x(n), y

do i = 1, n
   z(i) = a * x(i) + y
enddo
return
end
```

The loop in Example 2.2 has no *dependences.* In other words the result of one loop iteration does not depend on the result of any other iteration. This means that two different iterations could be executed simultaneously by two different processors. We parallelize this loop using the *parallel do* construct as shown in Example 2.3.

Example 2.3 The *saxpy*-like loop parallelized using OpenMP.

```
      subroutine saxpy(z, a, x, y, n)
      integer i, n
      real z(n), a, x(n), y
```

```
!$omp parallel do
      do i = 1, n
         z(i) = a * x(i) + y
      enddo
      return
      end
```

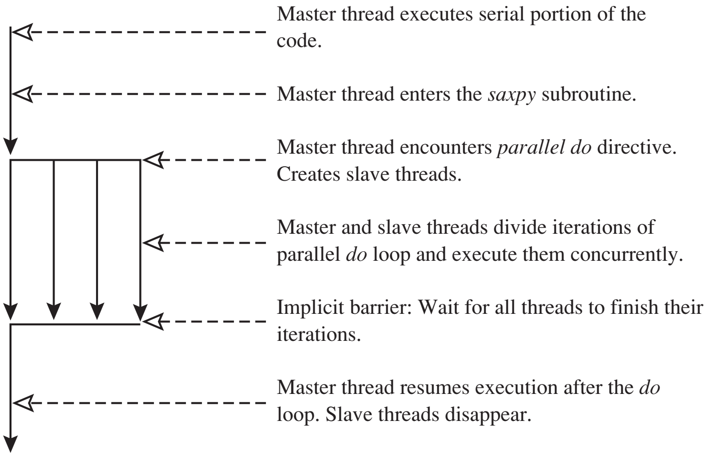
As we can see, the only change to the original program is the addition of the *parallel do* directive. This directive must be followed by a *do* loop construct and specifies that the iterations of the *do* loop be executed concurrently across multiple threads. An OpenMP compiler must create a set of threads and distribute the iterations of the *do* loop across those threads for parallel execution.

Before describing the runtime execution in detail, notice the minimal changes to the original sequential program. Furthermore, the original program remains "unchanged." When compiled using a non-OpenMP compiler, the *parallel do* directive is simply ignored, and the program continues to run serially and correctly.

2.3.1 Runtime Execution Model of an OpenMP Program

Let us examine what happens when the *saxpy* subroutine is invoked. This is most easily explained through the execution diagram depicted in Figure 2.2, which presents the execution of our example on four threads. Each vertical line represents a thread of execution. Time is measured along the vertical axis and increases as we move downwards along that axis. As we can see, the original program executes serially, that is, with a single thread, referred to as the "master thread" in an OpenMP program. This master thread invokes the *saxpy* subroutine and encounters the *parallel do* directive. At this point the master thread creates some additional threads (three in this example), and together with these additional threads (often referred to as "slave threads") forms a *team* of four parallel threads. These four threads divide the iterations of the *do* loop among themselves, with each thread executing a subset of the total number of iterations. There is an implicit barrier at the end of the *parallel do* construct. Therefore, after finishing its portions of the iterations, each thread waits for the remaining threads to finish their iterations. Once all the threads have finished and all iterations have been executed, the barrier condition is complete and all threads are released from the barrier. At this point, the slave threads disappear and the master thread resumes execution of the code past the *do* loop of the *parallel do* construct.

A "thread" in this discussion refers to an independent locus of control that executes within the same shared address space as the original sequen-



Master thread executes serial portion of the code.

Master thread enters the *saxpy* subroutine.

Master thread encounters *parallel do* directive. Creates slave threads.

Master and slave threads divide iterations of parallel *do* loop and execute them concurrently.

Implicit barrier: Wait for all threads to finish their iterations.

Master thread resumes execution after the *do* loop. Slave threads disappear.

Figure 2.2 Runtime execution of the *saxpy* parallel program.

tial program, with direct access to all of its variables. OpenMP does not specify the underlying execution vehicle; that is exclusively an implementation issue. An OpenMP implementation may choose to provide this abstraction in any of multiple ways—for instance, one implementation may map an OpenMP thread onto an operating system process, while another may map it onto a lightweight thread such as a Pthread. The choice of implementation does not affect the thread abstraction that we have just described.

A second issue is the manner in which iterations of the *do* loop are divided among the multiple threads. The iterations of the *do* loop are not replicated; rather, each thread is assigned a unique and distinct set of iterations to execute. Since the iterations of the *do* loop are assumed to be independent and can execute concurrently, OpenMP does not specify how the iterations are to be divided among the threads; this choice is left to the OpenMP compiler implementation. However, since the distribution of loop iterations across threads can significantly affect performance, OpenMP does supply additional attributes that can be provided with the *parallel do* directive and used to specify how the iterations are to be distributed across threads. These mechanisms are discussed later in Chapters 3 and 6.

2.3.2 Communication and Data Scoping

The shared memory model within OpenMP prescribes that multiple OpenMP threads execute within the same shared address space; we now examine the data references in the previous example within this memory model.

Each iteration of the loop reads the scalar variables *a* and *y,* as well as an element of the array *x;* it updates the corresponding element of the array *z.* What happens when multiple iterations of the loop execute concurrently on different threads? The variables that are being read—*a, y,* and *x*—can be read directly from shared memory since their values remain unchanged for the duration of the loop. Furthermore, since each iteration updates a distinct element of *z,* updates from different iterations are really to different memory locations and do not step over each other; these updates too can be made directly to shared memory. However, the loop index variable *i* presents a problem: since each iteration needs a distinct value for the value of the loop index variable, multiple iterations cannot use the same memory location for *i* and still execute concurrently.

This issue is addressed in OpenMP through the notion of *private* variables. For a *parallel do* directive, in OpenMP the default rules state that the *do* loop index variable is private to each thread, and all other variable references are shared. We illustrate this further in Figure 2.3. The top of the figure illustrates the data context during serial execution. Only the master thread is executing, and there is a single globally shared instance of each variable in the program as shown. All variable references made by the single master thread refer to the instance in the global address space.

The bottom of the figure illustrates the context during parallel execution. As shown, multiple threads are executing concurrently within the same global address space used during serial execution. However, along with the global memory as before, every thread also has a private copy of the variable *i* within its context. All variable references to variable *i* within a thread refer to the private copy of the variable within that thread; references to variables other than *i,* such as *a, x,* and so on, refer to the instance in global shared memory.

Let us describe the behavior of private variables in more detail. As shown in the figure, during parallel execution each thread works with a new, private instance of the variable *i.* This variable is newly created within each thread at the start of the parallel construct *parallel do*. Since a private variable gets a new memory location with each thread, its initial value is undefined within the parallel construct. Furthermore, once the *parallel do* construct has completed and the master thread resumes serial execution, all private instances of the variable *i* disappear, and the master thread continues execution as before, within the shared address space including *i.* Since the private copies of the variable *i* have disappeared and we revert back to the single global address space, the value of *i* is assumed to be undefined after the parallel construct.

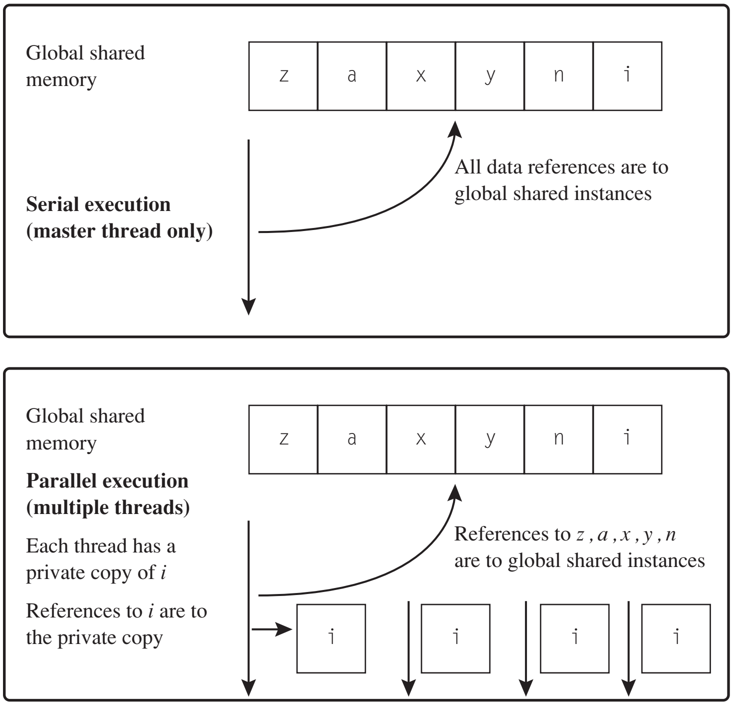

Global shared memory
z
a
x
y
n
i
All data references are to global shared instances
Serial execution (master thread only)
Global shared memory
z
a
x
y
n
i
Parallel execution (multiple threads)
Each thread has a private copy of *i*
References to *z* , *a* , *x* , *y* , *n* are to global shared instances
References to *i* are to the private copy
i
i
i
i

Figure 2.3 The behavior of private variables in an OpenMP program.

This example was deliberately chosen to be simple and does not need any explicit scoping attributes. The scope of each variable is therefore automatically determined by the default OpenMP rules. Subsequent examples in this chapter will present explicit data scoping clauses that may be provided by the programmer. These clauses allow the programmer to easily specify the sharing properties of variables in the program for a parallel construct, and depend upon the implementation to provide the desired behavior.

2.3.3 Synchronization in the Simple Loop Example

Our simple example did not include any explicit synchronization construct; let us now examine the synchronization requirements of the code and understand why it still executes correctly.

Synchronization is primarily used to control access to shared objects. There are two potential synchronization requirements in this example. First, the shared variable *z* is modified by multiple threads. However, recall that since each thread modifies a distinct element of *z,* there are no data conflicts and the updates can proceed concurrently without synchronization.

The second requirement is that all the values for the array *z* must have been updated when execution continues after the *parallel do* loop. Otherwise the master thread (recall that only the master thread executes the code after the *parallel do* construct) may not see the "latest" values of *z* in subsequent code because there may still be slave threads executing their sets of iterations. This requirement is met by the following property of the *parallel do* construct in OpenMP: the *parallel do* directive has an implicit *barrier* at its end—that is, each thread waits for all other threads to complete their set of iterations. In particular the master thread waits for all slave threads to complete, before resuming execution after the *parallel do* construct. This in turn guarantees that all iterations have completed, and all the values of *z* have been computed.

The default properties of the *parallel do* construct obviated the need for explicit synchronization in this example. Later in the chapter we present more complex codes that illustrate the synchronization constructs provided in OpenMP.

2.3.4 Final Words on the Simple Loop Example

The kind of parallelism exposed in this example is known as *loop-level* parallelism. As we saw, this type of parallelism is relatively easy to express and can be used to parallelize large codes in a straightforward manner simply by incrementally parallelizing individual loops, perhaps one at a time. However, loop-level parallelism does have its limitations. Applications that spend substantial portions of their execution time in noniterative (i.e., nonloop) constructs are less amenable to this form of parallelization. Furthermore, each parallel loop incurs the overhead for joining the threads at the end of the loop. As described above, each join is a synchronization point where all the threads must wait for the slowest one to arrive. This has a negative impact on performance and scalability since the program can only run as fast as the slowest thread in each parallel loop.

Nonetheless, if the goal is to parallelize a code for a modest-size multiprocessor, loop-level parallelism remains a very attractive approach. In a later section we will describe a more general method for parallelizing applications that can be used for nonloop constructs as well. First, however, we will consider a slightly more complicated loop and further investigate the issues that arise in parallelizing loops.

2.4 A More Complicated Loop

It would be nice if all loops were as simple as the one in Example 2.3, however, that is seldom the case. In this section we look at a slightly more complicated loop and how OpenMP is used to address the new issues that arise. The loop we examine is the outer loop in a Mandelbrot generator. A Mandelbrot generator is simply a program that determines which points in a plane belong to the Mandelbrot set. This is done by computing an iterative equation for each point we consider. The result of this calculation can then be used to color the corresponding pixel to generate the ubiquitous Mandelbrot image. This is a nice example because the Mandelbrot image is a visual representation of the underlying computation. However, we need not overly concern ourselves with the mechanics of computing the Mandelbrot set but rather focus on the structure of the loop.

Example 2.4 Mandelbrot generator: serial version.

```
real*8 x, y
integer i, j, m, n, maxiter
integer depth(*, *)
integer mandel_val


...
maxiter = 200
do i = 1, m
    do j = 1, n
        x = i/real(m)
        y = j/real(n)
        depth(j, i) = mandel_val(x, y, maxiter)
    enddo
enddo
```

Example 2.4 presents the sequential code for a Mandelbrot generator. For simplicity we have left out the code for evaluating the Mandelbrot equation (it is in the function *mandel_val*). The code we present is fairly straightforward: we loop for *m* points in the *i* direction and *n* points in the *j* direction, generate an *x* and *y* coordinate for each point (here restricted to the range (0,1]), and then compute the Mandelbrot equation for the given coordinates. The result is assigned to the two-dimensional array *depth*. Presumably this array will next be passed to a graphics routine for drawing the image.

Our strategy for parallelizing this loop remains the same as for the *saxpy* loop—we would like to use the *parallel do* directive. However, we must first convince ourselves that different iterations of the loop are actually independent and can execute in parallel, that is, that there are no data dependences in the loop from one iteration to another. Our brief description of a Mandelbrot generator would indicate that there are no dependences. In other words, the result for computing the Mandelbrot equation on a particular point does not depend on the result from any other point. This is evident in the code itself. The function *mandel_val* only takes *x, y,* and *maxiter* as arguments, so it can only know about the one point it is working on. If *mandel_val* included *i* or *j* as arguments, then there might be reason to suspect a dependence because the function could conceivably reference values for some other point than the one it is currently working on. Of course in practice we would want to look at the source code for *mandel_val* to be absolutely sure there are no dependences. There is always the possibility that the function modifies global structures not passed through the argument list, but that is not the case in this example. As a matter of jargon, a function such as this one that can safely be executed in parallel is referred to as being *thread-safe.* More interesting from a programming point of view, of course, are those functions that are not inherently thread-safe but must be made so through the use of synchronization.

Having convinced ourselves that there are no dependences, let us look more closely at what the loop is doing and how it differs from the *saxpy* loop of Example 2.3. The two loops differ in some fundamental ways. Additional complexities in this loop include a nested loop, three more scalar variables being assigned (*j, x,* and *y*), and a function call in the innermost loop. Let us consider each of these added complexities in terms of our runtime execution model.

Our understanding of the *parallel do/end parallel do* directive pair is that it will take the iterations of the enclosed loop, divide them among some number of parallel threads, and let each parallel thread execute its set of iterations. This does not change in the presence of a nested loop or a called function. Each thread will simply execute the nested loop and call the function in parallel with the other threads. So as far as control constructs are concerned, given that the called function is thread-safe, there is no additional difficulty on account of the added complexity of the loop.

As one may suspect, things are not so simple with the data environment. Any time we have variables being assigned values inside a parallel loop, we need to concern ourselves with the data environment. We know from the *saxpy* example that by default the loop index variable *i* will be private and everything else in the loop will be shared. This is appropriate for *m* and *n* since they are only read across different iterations and don't change in value from one iteration to the next. Looking at the other variables, though, the default rules are not accurate. We have a nested loop index variable `j,` and as a rule loop index variables should be private. The reason of course is that we want each thread to work on its own set of iterations. If *j* were shared, we would have the problem that there would be just one "global" value (meaning the same for all threads) for *j*. Consequently, each time a thread would increment *j* in its nested loop, it would also inadvertently modify the index variable for all other threads as well. So *j* must be a private variable within each thread. We do this with the private clause by simply specifying *private(j)* on the *parallel do* directive. Since this is almost always the desired behavior, loop index variables in Fortran are treated as having private scope by default, unless specified otherwise.[1]

What about the other variables, *x* and *y*? The same reasoning as above leads us to conclude that these variables also need to be private. Consider that *i* and *j* are private, and *x* and *y* are calculated based on the values of *i* and *j;* therefore it follows that *x* and *y* should be private. Alternatively, remember that *x* and *y* store the coordinates for the point in the plane for which we will compute the Mandelbrot equation. Since we wish each thread to work concurrently on a set of points in the plane, we need to have "parallel" (meaning here "multiple") storage for describing a point such that each thread can work independently. Therefore we must specify *x* and *y* to be private.

Finally we must consider synchronization in this loop. Recall that the main use of synchronization is to control access to shared objects. In the *saxpy* example, the only synchronization requirement was an implicit barrier at the close of the *parallel do.* This example is no different. There are two shared objects, *maxiter* and *depth.* The variable *maxiter* is only read and never written (we would need to check the *mandel_val* function to confirm this); consequently there is no reason or need to synchronize references to this shared variable. On the other hand, the array *depth* is modified in the loop. However, as with the *saxpy* example, the elements of the array are modified independently by each parallel thread, so the only synchronization necessary is at the end of the loop. In other words, we cannot allow the master thread to reference the *depth* array until all the array elements have been updated. This necessitates a barrier at the end of the loop, but we do not have to explicitly insert one since it is implicit in the *end parallel do* directive.

[1] This is actually a tricky area where the Fortran default rules differ from the C/C++ rules. In Fortran, the compiler will make *j* private by default, but in C/C++ the programmer must declare it private. Chapter 3 discusses the default rules for Fortran and C/C++ in greater detail.

At this point we are ready to present the parallelized version of Example 2.4. As with the *saxpy* example, we parallelized the loop in Example 2.5 with just the *parallel do* directive. We also see our first example of the *private* clause. There are many more ways to modify the behavior of the *parallel do* directive; these are described in Chapter 3.

Example 2.5 Mandelbrot generator: parallel OpenMP version.

```
      maxiter = 200
!$omp parallel do private(j, x, y)
      do i = 1, m
         do j = 1, n
            x = i/real(m)
            y = j/real(n)
            depth(j, i) = mandel_val(x, y, maxiter)
         enddo
      enddo
!$omp end parallel do
```

2.5 Explicit Synchronization

In Section 2.3 we introduced the concept of synchronization in the context of a simple parallel loop. We elaborated on this concept in Section 2.4 with a slightly more complicated loop in our Mandelbrot example. Both of those loops presented just examples of implicit synchronization. This section will extend the Mandelbrot example to show why explicit synchronization is sometimes needed, and how it is specified.

To understand this example we need to describe the *mandel_val* function in greater detail. The Mandelbrot equation (what is computed by *mandel_val*) is an iterative equation. This means that *mandel_val* iteratively computes the result for this equation until the result either has diverged or *maxiter* iterations have executed. The actual number of iterations executed is the value returned by *mandel_val* and assigned to the array *depth(i,j)*.

Imagine that, for whatever reason, it is important to know the total number of iterations executed by the Mandelbrot generator. One way to do this is to sum up the value for *depth* as it is computed. The sequential code to do this is trivial: we add a variable *total_iters,* which is initialized to zero, and we add into it each value of *depth* that we compute. The sequential code now looks like Example 2.6.

Example 2.6 Mandelbrot generator: computing the iteration count.

```
maxiter = 200
total_iters = 0
```

```
do i = 1, m
    do j = 1, n
        x = i/real(m)
        y = j/real(n)
        depth(j, i) = mandel_val(x, y, maxiter)
        total_iters = total_iters + depth(j, i)
    enddo
enddo
```

How does this change our parallel version? One might think this is pretty easy—we simply make *total_iters* a shared variable and leave everything else unchanged. This approach is only partially correct. Although *total_iters* needs to be shared, we must pay special attention to any shared variable that is modified in a parallel portion of the code. When multiple threads write to the same variable, there is no guaranteed order among the writes by the multiple threads, and the final value may only be one of a set of possible values. For instance, consider what happens when the loop executes with two threads. Both threads will simultaneously read the value stored in *total_iters,* add in their value of *depth(j,i)*, and write out the result to the single storage location for *total_iters*. Since the threads execute *asynchronously*, there is no guarantee as to the order in which the threads read or write the result. It is possible for both threads to read the same value of *total_iters*, in which case the final result will contain the increment from only one thread rather than both. Another possibility is that although *total_iters* is read by the threads one after the other, one thread reads an earlier (and likely smaller) value but writes its updated result later after the other thread. In this case the update executed by the other thread is lost and we are left with an incorrect result.

This phenomenon is known as a *race condition* on accesses to a shared variable. Such accesses to a shared variable must be controlled through some form of synchronization that allows a thread exclusive access to the shared variable. Having exclusive access to the variable enables a thread to atomically perform its read, modify, and update operation on the variable. This *mutual exclusion* synchronization is expressed using the *critical* construct in OpenMP.

The code enclosed by the *critical/end critical* directive pair in Example 2.7 can be executed by only one thread at a time. The first thread to reach the *critical* directive gets to execute the code. Any other threads wanting to execute the code must wait until the current thread exits the critical section. This means that only one thread at a time can update the value of *total_iters*, so we are assured that *total_iters* always stores the latest result and no updates are lost. On exit from the parallel loop, *total_iters* will store the correct result.

Example 2.7 Counting within a critical section.

```
!$omp critical
    total_iters = total_iters + depth(j, i)
!$omp end critical
```

The *critical/end critical* directive pair is an example of explicit synchronization. It must be specifically inserted by the programmer solely for synchronization purposes in order to control access to the shared variable *total_iters*. The previous examples of synchronization have been implicit because we did not explicitly specify them but rather the system inserted them for us.

Figure 2.4 shows a schematic for the parallel execution of this loop with the critical section. The schematic is somewhat simplified. It assumes that all threads reach the critical region at the same time and shows only one execution through the critical section. Nonetheless it should be clear that inserting a critical section into a parallel loop can have a very negative impact on performance because it may force other threads to temporarily halt their execution.

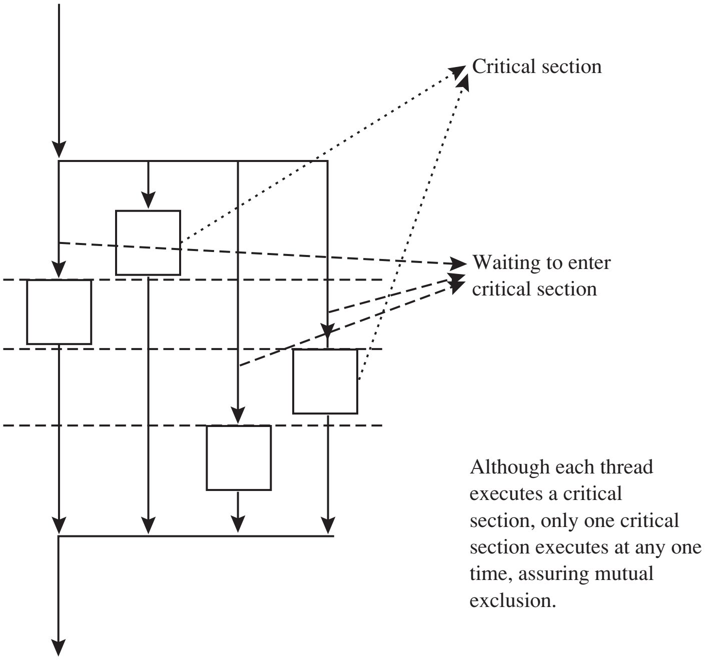

Critical section
Waiting to enter critical section
Although each thread executes a critical section, only one critical section executes at any one time, assuring mutual exclusion.

Figure 2.4 Parallel loop with a critical section.

OpenMP includes a rich set of synchronization directives as well as a full lock interface for more general kinds of locking than the simple mutual exclusion presented here. Synchronization is covered in greater detail in Chapter 5.

2.6 The *reduction* Clause

In Example 2.7 we saw a critical section being used to protect access to a shared variable. The basic operation we are executing on that variable is a sum reduction. Reductions are a sufficiently common type of operation that OpenMP includes a reduction data scope clause just to handle them. Using the *reduction* clause, we can rewrite Example 2.7 as shown in Example 2.8.

Example 2.8 Using the *reduction* clause.

```
      maxiter = 200
      total_iters = 0

!$omp parallel do private(j, x, y)
!$omp+ reduction(+:total_iters)
      do i = 1, m
         do j = 1, n
            x = i/real(m)
            y = j/real(n)
            depth(j, i) = mandel_val(x, y, maxiter)
            total_iters = total_iters + depth(j, i)
         enddo
      enddo
!$omp end parallel do
```

All we have done here is add the clause *reduction (+:total_iters)*, which tells the compiler that *total_iters* is the target of a sum reduction operation. The syntax allows for a large variety of reductions to be specified.

The compiler, in conjunction with the runtime environment, will implement the reduction in an efficient manner tailored for the target machine. The compiler can best address the various system-dependent performance considerations, such as whether to use a critical section or some other form of communication. It is therefore beneficial to use the reduction attribute where applicable rather than “rolling your own.”

The *reduction* clause is an actual data attribute distinct from either *shared* or *private*. The reduction variables have elements of both *shared* and *private*, but are really neither. In order to fully understand the reduction attribute we need to expand our runtime model, specifically the data environment part of it. We defer this discussion to Chapter 3, where the *reduction* clause is discussed in greater detail.

2.7 Expressing Parallelism with Parallel Regions

So far we have been concerned purely with exploiting loop-level parallelism. This is generally considered *fine-grained* parallelism. The term refers to the unit of work executed in parallel. In the case of loop-level parallelism, the unit of work is typically small relative to the program as a whole. Most programs involve many loops; if the loops are parallelized individually, then the amount of work done in parallel (before the parallel threads join with the master thread) is limited to the work of a single loop.

In this section we approach the general problem of parallelizing a program that needs to exploit coarser-grained parallelism than possible with the simple loop-level approach described in the preceding sections. To do this we will extend our now familiar example of the Mandelbrot generator.

Imagine that after computing the Mandelbrot set we wish to dither the *depth* array in order to soften the resulting image. We could extend our sequential Mandelbrot program as shown in Example 2.9. Here we have added a second array, *dith_depth(m,n)*, to store the result from dithering the *depth* array.

Example 2.9 Mandelbrot generator: dithering the image.

```
real x, y
integer i, j, m, n, maxiter
integer depth(*, *), dith_depth(*, *)
integer mandel_val

maxiter = 200
do i = 1, m
    do j = 1, n
        x = i/real(m)
        y = j/real(n)
        depth(j, i) = mandel_val(x, y, maxiter)
    enddo
enddo
do i = 1, m
    do j = 1, n
        dith_depth(j, i) = 0.5 * depth(j, i) +
$            0.25 * (depth(j - 1, i) + depth(j + 1, i))
    enddo
enddo
```

How would we parallelize this example? Applying our previous strategy, we would parallelize each loop nest individually using a *parallel do* directive. Since we are dithering values only along the *j* direction, and not along the *i* direction, there are no dependences on *i* and we could parallelize the outer dithering loop simply with a *parallel do* directive. However, this would unnecessarily force the parallel threads to synchronize, join, and fork again between the Mandelbrot loop and the dithering loop.

Instead, we would like to have each thread move on to dithering its piece of the depth array as soon as its piece has been computed. This requires that rather than joining the master thread at the end of the first parallel loop, each thread continues execution past the computation loop and onto its portion of the dithering phase. OpenMP supports this feature through the concept of a *parallel region* and the *parallel/end parallel* directives.

The *parallel* and *end parallel* directives define a parallel region. The block of code enclosed by the *parallel/end parallel* directive pair is executed in parallel by a team of threads initially forked by the *parallel* directive. This is simply a generalization of the *parallel do/end parallel do* directive pair; however, instead of restricting the enclosed block to a *do* loop, the *parallel/end parallel* directive pair can enclose any structured block of Fortran statements. The *parallel do* directive is actually just a *parallel* directive immediately followed by a *do* directive, which of course implies that one or more *do* directives (with corresponding loops) could appear anywhere in a parallel region. This distinction and its ramifications are discussed in later chapters.

Our runtime model from Example 2.3 remains unchanged in the presence of a parallel region. The *parallel/end parallel* directive pair is a control structure that forks a team of parallel threads with individual data environments to execute the enclosed code concurrently. This code is *replicated* across threads. Each thread executes the same code, although they do so asynchronously. The threads and their data environments disappear at the *end parallel* directive when the master thread resumes execution.

We are now ready to parallelize Example 2.9. For simplicity we consider execution with just two parallel threads. Generalizing to an arbitrary number of parallel threads is fairly straightforward but would clutter the example and distract from the concepts we are trying to present. Example 2.10 presents the parallelized code.

Example 2.10 Mandelbrot generator: parallel version.

```
      maxiter = 200
!$omp parallel
!$omp+ private(i, j, x, y)
!$omp+ private (my_width, my_thread, i_start, i_end)
```

```
      my_width = m/2
      my_thread = omp_get_thread_num()
      i_start = 1 + my_thread * my_width
      i_end = i_start + my_width - 1
      do i = i_start, i_end
         do j = 1, n
            x = i/real(m)
            y = j/real(n)
            depth(j, i) = mandel_val(x, y, maxiter)
         enddo
      enddo

      do i = i_start, i_end
         do j = 1, n
            dith_depth(j, i) = 0.5 * depth(j, i) + &
                 0.25 * (depth(j - 1, i) + depth(j + 1, i))
         enddo
      enddo
!$omp end parallel
```

Conceptually what we have done in Example 2.10 is divided the plane into two horizontal strips and forked a parallel thread for each strip. Each parallel thread first executes the Mandelbrot loop and then the dithering loop. Each thread works only on the points in its strip.

OpenMP allows users to specify how many threads will execute a parallel region with two different mechanisms: either through the *omp_set_num_threads()* runtime library procedure, or through the *OMP_NUM_THREADS* environment variable. In this example we assume the user has set the environment variable to the value 2.

The *omp_get_thread_num()* function is part of the OpenMP runtime library. This function returns a unique thread number (thread numbering begins with 0) to the caller. To generalize this example to an arbitrary number of threads, we also need the *omp_get_num_threads()* function, which returns the number of threads forked by the *parallel* directive. Here we assume for simplicity only two threads will execute the parallel region. We also assume the scalar *m* (the *do i* loop extent) is evenly divisible by 2. These assumptions make it easy to compute the width for each strip (stored in *my_width*).

A consequence of our coarse-grained approach to parallelizing this example is that we must now manage our own loop extents. This is necessary because the *do i* loop now indexes points in a thread's strip and not in the entire plane. To manage the indexing correctly, we compute new start and end values (*i_start* and *i_end*) for the loop that span only the width of a thread's strip. You should convince yourself that the example computes the *i_start* and *i_end* values correctly, assuming *my_thread* is numbered either 0 or 1.

With the modified loop extents we iterate over the points in each thread's strip. First we compute the Mandelbrot values, and then we dither the values of each row (*do j* loop) that we computed. Because there are no dependences along the *i* direction, each thread can proceed directly from computing Mandelbrot values to dithering without any need for synchronization.

You may have noticed a subtle difference in our description of parallelization with parallel regions. In previous discussions we spoke of threads working on a *set of iterations* of a loop, whereas here we have been describing threads as working on a *section* of an array. The distinction is important. The iterations of a loop have meaning only for the extent of the loop. Once the loop is completed, there are no more iterations to map to parallel threads (at least until the next loop begins). An array, on the other hand, will have meaning across many loops. When we map a section of an array to a parallel thread, that thread has the opportunity to execute all the calculations on its array elements, not just those of a single loop. In general, to parallelize an application using parallel regions, we must think of decomposing the problem in terms of the underlying data structures and mapping these to the parallel threads. Although this approach requires a greater level of analysis and effort from the programmer, it usually leads to better application scalability and performance. This is discussed in greater detail in Chapter 4.

2.8 Concluding Remarks

This chapter has presented a broad overview of OpenMP, beginning with an abstracted runtime model and subsequently using this model to parallelize a variety of examples. The OpenMP runtime abstraction has three elements: control structures, data environment, and synchronization. By understanding how these elements participate when executing an OpenMP program, we can deduce the runtime behavior of a parallel program and ultimately parallelize any sequential program.

Two different kinds of parallelization have been described. Fine-grained, or loop-level, parallelism is the simplest to expose but also has limited scalability and performance. Coarse-grained parallelism, on the other hand, demonstrates greater scalability and performance but requires more effort to program. Specifically, we must decompose the underlying data structures into parallel sections that threads can work on independently. OpenMP provides functionality for exposing either fine-grained or coarse-grained parallelism. The next two chapters are devoted to describing the OpenMP functionality for each of these two forms of parallelism.

2.9 Exercises

1. Write a program that breaks sequential semantics (i.e., it produces different results when run in parallel with a single thread versus run sequentially).
2. Parallelize Example 2.2 without using a *parallel do* directive.
3. In the early 1990s there was a popular class of parallel computer architecture known as the single-instruction multiple-data (SIMD) architecture. The classification applied to systems where a single instruction would act on parallel sets of data. In our runtime model, this would be akin to having a single thread with multiple data environments. How should Example 2.3 be changed to execute as an SIMD program?
4. Why does Example 2.4 store the result of *mandel_val* in *depth(j,i)* as opposed to *depth(i,j)*? Does it matter for correctness of the loop?
5. Parallelize Example 2.6 without using a critical section or a reduction attribute. To do this you will need to add a shared array to store the partial sums for each thread, and then add up the partial sums outside the parallel loop. Chapter 6 describes a performance problem known as *false sharing*. Is this problem going to affect the performance of the parallelized code?
6. Generalize Example 2.10 to run on an arbitrary number of threads. Make sure it will compute the correct loop extents regardless of the value of *m* or the number of threads. Include a print statement before the first loop that displays the number of threads that are executing. What happens if the *OMP_NUM_THREADS* environment variable is not set when the program is run?

CHAPTER 3

Exploiting Loop-Level Parallelism

3.1 Introduction

MANY TYPICAL PROGRAMS IN SCIENTIFIC AND ENGINEERING application domains spend most of their time executing loops, in particular, *do* loops in Fortran and *for* loops in C. We can often reduce the execution time of such programs by exploiting *loop-level* parallelism, by executing iterations of the loops concurrently across multiple processors. In this chapter we focus on the issues that arise in exploiting loop-level parallelism, and how they may be addressed using OpenMP.

OpenMP provides the *parallel do* directive for specifying that a loop be executed in parallel. Many programs can be parallelized successfully just by applying *parallel do* directives to the proper loops. This style of fine-grained parallelization is especially useful because it can be applied *incrementally:* as specific loops are found to be performance bottlenecks, they can be parallelized easily by adding directives and making small, localized changes to the source code. Hence the programmer need not rewrite the entire application just to parallelize a few performance-limiting loops. Because incremental parallelization is such an attractive technique, *parallel do* is one of the most important and frequently used OpenMP directives.

However, the programmer must choose carefully which loops to parallelize. The parallel version of the program generally must produce the same results as the serial version; in other words, the *correctness* of the program must be maintained. In addition, to maximize performance the execution time of parallelized loops should be as short as possible, and certainly not longer than the original serial version of the loops.

This chapter describes how to make effective use of the *parallel do* directive to parallelize loops in a program. We start off in Section 3.2 by discussing the syntactic form and usage of the *parallel do* directive. We give an overview of the various clauses that can be added to the directive to control the behavior of a parallel loop. We also identify the restrictions on the loops that may be parallelized using this directive. Section 3.3 reviews the runtime execution model of the *parallel do* directive. Section 3.4 describes the data scoping clauses in OpenMP. These scope clauses control the sharing behavior of program variables between multiple threads, such as whether a variable is shared by multiple threads or private to each thread. The first part of Section 3.5 shows how to determine whether it is safe to parallelize a loop, based upon the presence or absence of data dependences. Then, a number of techniques are presented for making loops parallelizable by breaking data dependences, using a combination of source code transformations and OpenMP directives. Finally, Section 3.6 describes two issues that affect the performance of parallelized loops—excessive parallel overhead and poor load balancing—and describes how these issues may be addressed by adding performance-tuning clauses to the *parallel do* directive.

By the end of the chapter, you will have gained a basic understanding of some techniques for speeding up a program by incrementally parallelizing its loops, and you will recognize some of the common pitfalls to be avoided.

3.2 Form and Usage of the *parallel do* Directive

Figure 3.1 shows a high-level view of the syntax of the *parallel do* directive in Fortran, while Figure 3.2 shows the corresponding syntax of the *parallel for* directive in C and C++. The square bracket notation ([. . .]) is used to identify information that is optional, such as the clauses or the *end parallel do* directive. Details about the contents of the clauses are presented in subsequent sections in this chapter.

```
!$omp parallel do [clause [,] [clause ...]]
    do index = first , last [, stride]
        body of the loop
    enddo
[!$omp end parallel do]
```

Figure 3.1 Fortran syntax for the *parallel do* directive.

```
#pragma omp parallel for [clause [clause ...]]
    for (index = first ; test_expr ; increment_expr) {
        body of the loop
    }
```

Figure 3.2 C/C++ syntax for the *parallel for* directive.

In Fortran, the *parallel do* directive parallelizes the loop that immediately follows it, which means there must be a statement following the directive, and that statement must be a *do* loop. Similarly, in C and C++ there must be a *for* loop immediately following the *parallel for* directive. The directive extends up to the end of the loop to which it is applied. In Fortran only, to improve the program's readability, the end of the loop may optionally be marked with an *end parallel do* directive.

3.2.1 Clauses

OpenMP allows the execution of a parallel loop to be controlled through optional clauses that may be supplied along with the *parallel do* directive. We briefly describe the various kinds of clauses here and defer a detailed explanation for later in this chapter:

- Scoping clauses (such as *private* or *shared*) are among the most commonly used. They control the sharing scope of one or more variables within the parallel loop. All the flavors of scoping clauses are covered in Section 3.4.
- The *schedule* clause controls how iterations of the parallel loop are distributed across the team of parallel threads. The choices for scheduling are described in Section 3.6.2.
- The *if* clause controls whether the loop should be executed in parallel or serially like an ordinary loop, based on a user-defined runtime test. It is described in Section 3.6.1.

- The *ordered* clause specifies that there is ordering (a kind of synchronization) between successive iterations of the loop, for cases when the iterations cannot be executed completely in parallel. It is described in Section 5.4.2.
- The *copyin* clause initializes certain kinds of *private* variables (called *threadprivate* variables) at the start of the parallel loop. It is described in Section 4.4.2.

Multiple scoping and *copyin* clauses may appear on a *parallel do;* generally, different instances of these clauses affect different variables that appear within the loop. The *if, ordered,* and *schedule* clauses affect execution of the entire loop, so there may be at most one of each of these. The section that describes each of these kinds of clauses also defines the default behavior that occurs when that kind of clause does not appear on the parallel loop.

3.2.2 Restrictions on Parallel Loops

OpenMP places some restrictions on the kinds of loops to which the *parallel do* directive can be applied. Generally, these restrictions make it easier for the compiler to parallelize loops as efficiently as possible. The basic principle behind them is that it must be possible to precisely compute the *trip-count,* or number of times a loop body is executed, without actually having to execute the loop. In other words, the trip-count must be computable at runtime based on the specified lower bound, upper bound, and stride of the loop.

In a Fortran program the *parallel do* directive must be followed by a *do* loop statement whose iterations are controlled by an index variable. It cannot be a *do-while* loop, a *do* loop that lacks iteration control, or an array assignment statement. In other words, it must start out like this:

```
DO index = lowerbound, upperbound [, stride]
```

In a C program, the statement following the *parallel for* pragma must be a *for* loop, and it must have *canonical shape,* so that the trip-count can be precisely determined. In particular, the loop must be of the form

```
for (index = start ; index < end ; increment_expr)
```

The *index* must be an integer variable. Instead of less-than (“ < ”), the comparison operator may be “ < = ”, “ > ”, or “ > = ”. The *start* and *end* values can be any numeric expression whose value does not change dur-

Table 3.1 Increment expressions for loops in C.

Operator	*Forms of* increment_expr
++	*index*++ or ++*index*
––	*index*–– or ––*index*
+=	*index* += *incr*
–=	*index* –= *incr*
=	*index* = *index* + *incr* or *index* = incr + *index* or *index* = *index* – incr

ing execution of the loop. The *increment_expr* must change the value of *index* by the same amount after each iteration, using one of a limited set of operators. Table 3.1 shows the forms it can take. In the table, *incr* is a numeric expression that does not change during the loop.

In addition to requiring a computable trip-count, the *parallel do* directive requires that the program complete all iterations of the loop; hence, the program cannot use any control flow constructs that exit the loop before all iterations have been completed. In other words, the loop must be a *block* of code that has a single entry point at the top and a single exit point at the bottom. For this reason, there are restrictions on what constructs can be used inside the loop to change the flow of control. In Fortran, the program cannot use an *exit* or *goto* statement to branch out of the loop. In C the program cannot use a *break* or *goto* to leave the loop, and in C++ the program cannot throw an exception from inside the loop that is caught outside. Constructs such as *cycle* in Fortran and *continue* in C that complete the current iteration and go on to the next are permitted, however. The program may also use a *goto* to jump from one statement inside the loop to another, or to raise a C++ exception (using *throw*) so long as it is caught by a *try* block somewhere inside the loop body. Finally, execution of the entire program can be terminated from within the loop using the usual mechanisms: a *stop* statement in Fortran or a call to *exit* in C and C++.

The other OpenMP directives that introduce parallel constructs share the requirement of the *parallel do* directive that the code within the lexical extent constitute a single-entry/single-exit block. These other directives are *parallel*, *sections*, *single*, *master*, *critical*, and *ordered*. Just as in the case of *parallel do*, control flow constructs may be used to transfer control to other points within the block associated with each of these parallel constructs, and a *stop* or *exit* terminates execution of the entire program, but control may not be transferred to a point outside the block.

3.3 Meaning of the *parallel do* Directive

Chapter 2 showed how to use directives for incremental parallelization of some simple examples and described the behavior of the examples in terms of the OpenMP runtime execution model. We briefly review the key features of the execution model before we study the *parallel do* directive in depth in this chapter. You may find it useful to refer back to Figures 2.2 and 2.3 now to get a concrete picture of the following execution steps of a parallel loop.

Outside of parallel loops a single master thread executes the program serially. Upon encountering a parallel loop, the master thread creates a team of parallel threads consisting of the master along with zero or more additional slave threads. This team of threads executes the parallel loop together. The iterations of the loop are divided among the team of threads; each iteration is executed only once as in the original program, although a thread may execute more than one iteration of the loop. During execution of the parallel loop, each program variable is either shared among all threads or private to each thread. If a thread writes a value to a shared variable, all other threads can read the value, whereas if it writes a value to a private variable, no other thread can access that value. After a thread completes all of its iterations from the loop, it waits at an implicit barrier for the other threads to complete their iterations. When all threads are finished with their iterations, the slave threads stop executing, and the master continues serial execution of the code following the parallel loop.

3.3.1 Loop Nests and Parallelism

When the body of one loop contains another loop, we say that the second loop is *nested* inside the first, and we often refer to the outer loop and its contents collectively as a *loop nest*. When one loop in a loop nest is marked by a *parallel do* directive, the directive applies only to the loop that immediately follows the directive. The behavior of all of the other loops remains unchanged, regardless of whether the loop appears in the serial part of the program or is contained within an outermore parallel loop: all iterations of loops not preceded by the *parallel do* are executed by each thread that reaches them.

Example 3.1 shows two important, common instances of parallelizing one loop in a multiple loop nest. In the first subroutine, the *j* loop is executed in parallel, and each iteration computes in *a(0, j)* the sum of elements from *a(1, j)* to *a(M, j)*. The iterations of the *i* loop are not partitioned or divided among the threads; instead, each thread executes all

M iterations of the *i* loop each time it reaches the *i* loop. In the second subroutine, this pattern is reversed: the outer *j* loop is executed serially, one iteration at a time. Within each iteration of the *j* loop a team of threads is formed to divide up the work within the inner *i* loop and compute a new column of elements *a(1:M, j)* using a simple smoothing function. This partitioning of work in the *i* loop among the team of threads is called *work-sharing*. Each iteration of the *i* loop computes an element *a(i, j)* of the new column by averaging the corresponding element *a(i, j – 1)* of the previous column with the corresponding elements *a(i – 1, j – 1)*and *a(i + 1, j – 1)* from the previous column.

Example 3.1 Parallelizing one loop in a nest.

```
      subroutine sums(a, M, N)
      integer M, N, a(0:M, N), i, j

!$omp parallel do
      do j = 1, N
         a(0, j) = 0
         do i = 1, M
            a(0, j) = a(0, j) + a(i, j)
         enddo
      enddo
      end

      subroutine smooth(a, M, N)
      integer M, N, a(0:M + 1, 0:N), i, j

      do j = 1, N
!$omp parallel do
         do i = 1, M
            a(i, j) = (a(i - 1, j - 1) + a(i, j - 1) + &
                       a(i + 1, j - 1))/3.0
         enddo
      enddo
      end
```

3.4 Controlling Data Sharing

Multiple threads within an OpenMP parallel program execute within the same shared address space and can share access to variables within this address space. Sharing variables between threads makes interthread communication very simple: threads send data to other threads by assigning values to shared variables and receive data by reading values from them.

In addition to sharing access to variables, OpenMP also allows a variable to be designated as private to each thread rather than shared among all threads. Each thread then gets a private copy of this variable for the duration of the parallel construct. Private variables are used to facilitate computations whose results are different for different threads.

In this section we describe the data scope clauses in OpenMP that may be used to control the sharing behavior of individual program variables inside a parallel construct. Within an OpenMP construct, every variable that is used has a *scope* that is either shared or private (the other scope clauses are usually simple variations of these two basic scopes). This kind of "scope" is different from the "scope" of accessibility of variable names in serial programming languages (such as local, file-level, and global in C, or local and common in Fortran). For clarity, we will consistently use the term "lexical scope" when we intend the latter, serial programming language sense, and plain "scope" when referring to whether a variable is shared between OpenMP threads. In addition, "scope" is both a noun and a verb: every variable used within an OpenMP construct has a scope, and we can explicitly scope a variable as shared or private on an OpenMP construct by adding a clause to the directive that begins the construct.

Although shared variables make it convenient for threads to communicate, the choice of whether a variable is to be shared or private is dictated by the requirements of the parallel algorithm and must be made carefully. Both the unintended sharing of variables between threads, or, conversely, the privatization of variables whose values need to be shared, are among the most common sources of errors in shared memory parallel programs. Because it is so important to give variables correct scopes, OpenMP provides a rich set of features for explicitly scoping variables, along with a well-defined set of rules for implicitly determining default scopes. These are all of the scoping clauses that can appear on a parallel construct:

- *shared* and *private* explicitly scope specific variables.
- *firstprivate* and *lastprivate* perform initialization and finalization of privatized variables.
- *default* changes the default rules used when variables are not explicitly scoped.
- *reduction* explicitly identifies reduction variables.

We first describe some general properties of data scope clauses, and then discuss the individual scope clauses in detail in subsequent sections.

3.4.1 General Properties of Data Scope Clauses

A data scope clause consists of the keyword identifying the clause (such as *shared* or *private*), followed by a comma-separated list of variables within parentheses. The data scoping clause applies to all the variables in the list and identifies the scope of these variables as either shared between threads or private to each thread.

Any variable may be marked with a data scope clause—automatic variables, global variables (in C/C++), *common* block variables or *module* variables (in Fortran), an entire *common* block (in Fortran), as well as formal parameters to a subroutine. However, a data scope clause does have several restrictions.

The first requirement is that the directive with the scope clause must be within the lexical extent of the declaration of each of the variables named within a scope clause; that is, there must be a declaration of the variable that encloses the directive.

Second, a variable in a data scoping clause cannot refer to a portion of an object, but must refer to the entire object. Therefore, it is not permitted to scope an individual array element or field of a structure—the variable must be either shared or private in its entirety. A data scope clause may be applied to a variable of type *struct* or *class* in C or C++, in which case the scope clause in turn applies to the entire structure including all of its subfields. Similarly, in Fortran, a data scope clause may be applied to an entire *common* block by listing the *common* block name between slashes ("/"), thereby giving an explicit scope to each variable within that *common* block.

Third, a directive may contain multiple *shared* or *private* scope clauses; however, an individual variable can appear on at most a single clause—that is, a variable may uniquely be identified as shared or private, but not both.

Finally, the data scoping clauses apply only to accesses to the named variables that occur in the code contained *directly* within the *parallel do/ end parallel do* directive pair. This portion of code is referred to as the *lexical extent* of the *parallel do* directive and is a subset of the larger *dynamic extent* of the directive that also includes the code contained within subroutines invoked from within the parallel loop. Data references to variables that occur within the lexical extent of the parallel loop are affected by the data scoping clauses. However, references from subroutines invoked from within the parallel loop are not affected by the scoping clauses in the dynamically enclosing parallel directive. The rationale for this is simple: References within the lexical extent are easily associated with the data scoping clause in the directly enclosing directive. However, this association is far less obvious for references that are outside the lexical scope, perhaps buried within a deeply nested chain of subroutine calls. Identifying the relevant data scoping clause would be extremely cumbersome and error prone in these situations. Example 3.2 shows a valid use of scoping clauses in Fortran.

Example 3.2 Sample scoping clauses in Fortran.

```
      COMMON /globals/ a, b, c
      integer i, j, k, count
      real a, b, c, x
      ...
!$omp parallel do private(i, j, k)
!$omp+ shared(count, /globals/)
!$omp+ private(x)
```

In C++, besides scoping an entire object, it is also possible to scope a static member variable of a class using a fully qualified name. Example 3.3 shows a valid use of scoping clauses in C++.

Example 3.3 Sample scoping clauses in C++.

```
class MyClass {
    ...
    static float x;
    ...
};
MyClass arr[N];
int j, k;
...
#pragma omp parallel for shared(MyClass::x, arr) \
    private(j, k)
```

For the rest of this section, we describe how to use each of the scoping clauses and illustrate the behavior of each clause through simple examples. Section 3.5 presents more realistic examples of how to use many of the clauses when parallelizing real programs.

3.4.2 The *shared* Clause

The *shared* scope clause specifies that the named variables should be shared by all the threads in the team for the duration of the parallel construct. The behavior of shared scope is easy to understand: even within the parallel loop, a reference to a shared variable from any thread continues to access the single instance of the variable in shared memory. All modifications to this variable update the global instance, with the updated value becoming available to the other threads.

Care must be taken when the *shared* clause is applied to a pointer variable or to a formal parameter that is passed by reference. A *shared* clause on a pointer variable will mark only the pointer value itself as shared, but will not affect the memory pointed to by the variable. Dereferencing a shared pointer variable will simply dereference the address value within the pointer variable. Formal parameters passed by reference behave in a similar fashion, with all the threads sharing the reference to the corresponding actual argument.

3.4.3 The *private* Clause

The *private* clause requires that each thread create a private instance of the specified variable. As we illustrated in Chapter 2, each thread allocates a private copy of these variables from storage within the private execution context of each thread; these variables are therefore private to each thread and not accessible by other threads. References to these variables within the lexical extent of the parallel construct are changed to read or write the private copy belonging to the referencing thread.

Since each thread has its own copy of private variables, this private copy is no longer storage associated with the original shared instance of the variable; rather, references to this variable access a distinct memory location within the private storage of each thread.

Furthermore, since private variables get new storage for the duration of the parallel construct, they are uninitialized upon entry to the parallel region, and their value is undefined. In addition, the value of these variables after the completion of the parallel construct is also undefined (see Example 3.4). This is necessary to maintain consistent behavior between the serial and parallel versions of the code. To see this, consider the serial instance of the code—that is, when the code is compiled without enabling OpenMP directives. The code within the loop accesses the single instance of variables marked private, and their final value is available after the parallel loop. However, the parallel version of this same code will access the private copy of these variables, so that modifications to them within the parallel loop will not be reflected in the copy of the variable after the parallel loop. In OpenMP, therefore, private variables have undefined values both upon entry to and upon exit from the parallel construct.

Example 3.4 Behavior of private variables.

```
      integer x

      x = ...
!$omp parallel do private(x)
      do i = 1, n
         ! Error! "x" is undefined upon entry,
         ! and must be defined before it can be used.
         ... = x
      enddo

      ! Error: x is undefined after the parallel loop,
      ! and must be defined before it can be used.
      ... = x
```

There are three exceptions to the rule that private variables are undefined upon entry to a parallel loop. The simplest instance is the loop control variable that takes on successive values in the iteration space during the execution of the loop. The second concerns C++ class objects (i.e., non-plain-old-data or non-POD objects), and the third concerns allocatable arrays in Fortran 90. Each of these languages defines an initial status for the types of variables mentioned above. OpenMP therefore attempts to provide the same behavior for the private instances of these variables that are created for the duration of a parallel loop.

In C++, if a variable is marked private and is a variable of class or struct type that has a constructor, then the variable must also have an accessible default constructor and destructor. Upon entry to the parallel loop when each thread allocates storage for the private copy of this variable, each thread also invokes this default constructor to construct the private copy of the object. Upon completion of the parallel loop, the private instance of each object is destructed using the default destructor. This correctly maintains the C++ semantics for construction/destruction of these objects when new copies of the object are created within the parallel loop.

In Fortran 90, if an allocatable array is marked private, then the serial copy before the parallel construct must be unallocated. Upon entry to the parallel construct, each thread gets a private unallocated copy of the array. This copy must be allocated before it can be used and must be explicitly deallocated by the program at the end of the parallel construct. The original serial copy of this array after the parallel construct is again unallocated. This preserves the general unallocated initial status of allocatable arrays. Furthermore, it means you must allocate and deallocate such arrays within the parallel loop to avoid memory leakage.

None of these issues arise with regard to shared variables. Since these variables are shared among all the threads, all references within the parallel code continue to access the single shared location of the variable as in the serial code. Shared variables therefore continue to remain available both upon entry to the parallel construct as well as after exiting the construct.

Since each thread needs to create a private copy of the named variable, it must be possible to determine the size of the variable based upon its declaration. In particular, in Fortran if a formal array parameter of adjustable size is specified in a private clause, then the program must also fully specify the bounds for the formal parameter. Similarly, in C/C++ a variable specified as private must not have an incomplete type. Finally, the private clause may not be applied to C++ variables of reference type; while the behavior of the data scope clauses is easily deduced for both ordinary variables and for pointer variables (see below), variables of reference type raise a whole set of complex issues and are therefore disallowed for simplicity.

Lastly, the *private* clause, when applied to a pointer variable, continues to behave in a consistent fashion. As per the definition of the *private* clause, each thread gets a private, uninitialized copy of a variable of the same type as the original variable, in this instance a pointer typed variable. This pointer variable is initially undefined and may be freely used to store memory addresses as usual within the parallel loop. Be careful that the scoping clause applies just to the pointer in this case; the sharing behavior of the storage pointed to is determined by the latter's scoping rules.

With regard to manipulating memory addresses, the only restriction imposed by OpenMP is that a thread is not allowed to access the private storage of another thread. Therefore a thread should not pass the address of a variable marked private to another thread because accessing the private storage of another thread can result in undefined behavior. In contrast, the heap is always shared among the parallel threads; therefore pointers to heap-allocated storage may be freely passed across multiple threads.

3.4.4 Default Variable Scopes

The default scoping rules in OpenMP state that if a variable is used within a parallel construct and is not scoped explicitly, then the variable is treated as shared. This is usually the desired behavior for variables that are read but not modified within the parallel loop—if a variable is assigned within the loop, then that variable may need to be explicitly scoped, or it may be necessary to add synchronization around statements that access the variable. In this section we first describe the general behavior of heap- and stack-allocated storage, and then discuss the behavior of different classes of variables under the default shared rule.

All threads share a single global heap in an OpenMP program. Heap-allocated storage is therefore uniformly accessible by all threads in a parallel team. On the other hand, each OpenMP thread has its own private stack that is used for subroutine calls made from within a parallel loop. Automatic (i.e., stack-allocated) variables within these subroutines are therefore private to each thread. However, automatic variables in the subroutine that contains the parallel loop continue to remain accessible by all the threads executing the loop and are treated as shared unless scoped otherwise. This is illustrated in Example 3.5.

Example 3.5 Illustrating the behavior of stack-allocated variables.

```
      subroutine f
      real a(N), sum

!$omp parallel do private (sum)
      do i = ...
         ! "a" is shared in the following reference
         ! while sum has been explicitly scoped as
         ! private

         a(i) = ...
         sum = 0
         call g (sum)
      enddo
      end

      subroutine g (s)
      real b (100), s
      integer i

      do i = ...
         ! "b" and "i" are local stack-allocated
         ! variables and are therefore private in
         ! the following references
         b(i) = ...
         s = s + b(i)
      enddo
      end
```

There are three exceptions to the rule that unscoped variables are made shared by default. We will first describe these exceptions, then present detailed examples in Fortran and C/C++ that illustrate the rules. First, certain loop index variables are made private by default. Second, in subroutines called within a parallel region, local variables and (in C and C++)

value parameters within the called subroutine are scoped as private. Finally, in C and C++), an *automatic* variable declared within the lexical extent of a parallel region is scoped as private. We discuss each of these in turn.

When executing a loop within a parallel region, if a loop index variable is shared between threads, it is almost certain to cause incorrect results. For this reason, the index variable of a loop to which a *parallel do* or *parallel for* is applied is scoped by default as private. In addition, in Fortran only, the index variable of a sequential (i.e., non-work-shared) loop that appears within the lexical extent of a parallel region is scoped as private. In C and C++, this is not the case: index variables of sequential *for* loops are scoped as shared by default. The reason is that, as was discussed in Section 3.2.2, the C *for* construct is so general that it is difficult for the compiler to figure out which variables should be privatized. As a result, in C the index variables of serial loops must explicitly be scoped as private.

Second, as we discussed above, when a subroutine is called from within a parallel region, then local variables within the called subroutine are private to each thread. However, if any of these variables are marked with the *save* attribute (in Fortran) or as *static* (in C/C++), then these variables are no longer allocated on the stack. Instead, they behave like globally allocated variables and therefore have shared scope.

Finally, C and C++ do not limit variable declarations to function entry as in Fortran; rather, variables may be declared nearly anywhere within the body of a function. Such nested declarations that occur within the lexical extent of a parallel loop are scoped as private for the parallel loop.

We now illustrate these default scoping rules in OpenMP. Examples 3.6 and 3.7 show sample parallel code in Fortran and C, respectively, in which the scopes of the variables are determined by the default rules. For each variable used in Example 3.6, Table 3.2 lists the scope, how that scope was determined, and whether the use of the variable within the parallel region is safe or unsafe. Table 3.3 lists the same information for Example 3.7.

Example 3.6 Default scoping rules in Fortran.

```
      subroutine caller(a, n)
      integer n, a(n), i, j, m

      m = 3
!$omp parallel do
      do i = 1, n
         do j = 1, 5
            call callee(a(i), m, j)
         enddo
      enddo
      end
      subroutine callee(x, y, z)
      common /com/ c
      integer x, y, z, c, ii, cnt
      save cnt

      cnt = cnt + 1
      do ii = 1, z
         x = y + c
      enddo
      end
```

Example 3.7 Default scoping rules in C.

```
void caller(int a[], int n)
{
    int i, j, m = 3;

    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        int k = m;

        for (j = 1; j ≤ 5; j++)
            callee(&a[i], &k, j);
    }
}

extern int c;

void callee(int *x, int *y, int z)
{
    int ii;
    static int cnt;

    cnt++;
    for (ii = 0; ii < z; i++)
        *x = *y + c;
}
```

3.4.5 Changing Default Scoping Rules

As we described above, by default, variables have shared scope within an OpenMP construct. If a variable needs to be private to each thread, then it must be explicitly identified with a *private* scope clause. If a construct requires that most of the referenced variables be private, then this default rule can be quite cumbersome since it may require a *private* clause

Table 3.2 Variable scopes for Fortran default scoping example.

Variable	*Scope*	*Is Use Safe?*	*Reason for Scope*
a	shared	yes	Declared outside parallel construct.
n	shared	yes	Declared outside parallel construct.
i	private	yes	Parallel loop index variable.
j	private	yes	Fortran sequential loop index variable.
m	shared	yes	Declared outside parallel construct.
x	shared	yes	Actual parameter is *a*, which is shared.
y	shared	yes	Actual parameter is *m*, which is shared.
z	private	yes	Actual parameter is *j*, which is private.
c	shared	yes	In a *common* block.
ii	private	yes	Local stack-allocated variable of called subroutine.
cnt	shared	no	Local variable of called subroutine with *save* attribute.

Table 3.3 Variable scopes for C default scoping example.

Variable	*Scope*	*Is Use Safe?*	*Reason for Scope*
a	shared	yes	Declared outside parallel construct.
n	shared	yes	Declared outside parallel construct.
i	private	yes	Parallel loop index variable.
j	shared	no	Loop index variable, but not in Fortran.
m	shared	yes	Declared outside parallel construct.
k	private	yes	Auto variable declared inside parallel construct.
x	private	yes	Value parameter.
**x*	shared	yes	Actual parameter is *a*, which is shared.
y	private	yes	Value parameter.
**y*	private	yes	Actual parameter is *k*, which is private.
z	private	yes	Value parameter.
c	shared	yes	Declared as *extern.*
ii	private	yes	Local stack-allocated variable of called subroutine.
cnt	shared	no	Declared as *static.*

for a large number of variables. As a convenience, therefore, OpenMP provides the ability to change the default behavior using the *default* clause on the parallel construct.

The syntax for this clause in Fortran is

```
default (shared | private | none)
```

while in C and C++, it is

```
default (shared | none)
```

In Fortran, there are three different forms of this clause: *default (shared), default(private),* and *default(none).* At most one *default* clause may appear on a parallel region. The simplest to understand is *default (shared),* because it does not actually change the scoping rules: it says that unscoped variables are still scoped as shared by default.

The clause *default(private)* changes the rules so that unscoped variables are scoped as private by default. For example, if we added a *default(private)* clause to the *parallel do* directive in Example 3.6, then *a, m,* and *n* would be scoped as private rather than shared. Scoping of variables in the called subroutine *callee* would not be affected because the subroutine is outside the lexical extent of the *parallel do.* The most common reason to use *default(private)* is to aid in converting a parallel application based on a distributed memory programming paradigm such as MPI, in which threads cannot share variables, to a shared memory OpenMP version. The clause *default(private)* is also convenient when a large number of scratch variables are used for holding intermediate results of a computation and must be scoped as private. Rather than listing each variable in an explicit *private* clause, *default(private)* may be used to scope all of these variables as private. Of course, when using this clause each variable that needs to be shared must be explicitly scoped using the *shared* clause.

The *default(none)* clause helps catch scoping errors. If *default(none)* appears on a parallel region, and any variables are used in the lexical extent of the parallel region but not explicitly scoped by being listed in a *private, shared, reduction, firstprivate,* or *lastprivate* clause, then the compiler issues an error. This helps avoid errors resulting from variables being implicitly (and incorrectly) scoped.

In C and C++, the clauses available to change default scoping rules are *default(shared)* and *default(none)*. There is no *default(private)* clause. This is because many C standard library facilities are implemented using macros that reference global variables. The standard library tends to be used pervasively in C and C++ programs, and scoping these globals as private is likely to be incorrect, which would make it difficult to write portable, correct OpenMP code using a *default(private)* scoping rule.

3.4.6 Parallelizing Reduction Operations

As discussed in Chapter 2, one type of computation that we often wish to parallelize is a reduction operation. In a reduction, we repeatedly apply a binary operator to a variable and some other value, and store the result back in the variable. For example, one common reduction is to compute the sum of the elements of an array:

```
      sum = 0
!$omp parallel do reduction(+ : sum)
      do i = 1, n
         sum = sum + a(i)
      enddo
```

and another is to find the largest (maximum) value:

```
x = a(1)
do i = 2, n
    x = max(x, a(i))
enddo
```

When computing the sum, we use the binary operator "+", and to find the maximum we use the *max* operator. For some operators (including "+" and *max*), the final result we get does not depend on the order in which we apply the operator to elements of the array. For example, if the array contained the three elements 1, 4, and 6, we would get the same sum of 11 regardless of whether we computed it in the order 1 + 4 + 6 or 6 + 1 + 4 or any other order. In mathematical terms, such operators are said to be commutative and associative.

When a program performs a reduction using a commutative-associative operator, we can parallelize the reduction by adding a *reduction* clause to the *parallel do* directive. The syntax of the clause is

```
reduction (redn_oper : var_list)
```

There may be multiple *reduction* clauses on a single work-sharing directive. The *redn_oper* is one of the built-in operators of the base language. Table 3.4 lists the allowable operators in Fortran, while Table 3.5 lists the operators for C and C++. (The other columns of the tables will be explained below.) The *var_list* is a list of scalar variables into which we are computing reductions using the *redn_oper.* If you wish to perform a reduction on an array element or field of a structure, you must create a scalar temporary with the same type as the element or field, perform the reduction on the temporary, and copy the result back into the element or field at the end of the loop.

Table 3.4 Reduction operators for Fortran.

Operator	*Data Types*	*Initial Value*
+	integer, floating point (complex or real)	0
*	integer, floating point (complex or real)	1
–	integer, floating point (complex or real)	0
.AND.	logical	.TRUE.
.OR.	logical	.FALSE.
.EQV.	logical	.TRUE.
.NEQV.	logical	.FALSE.
MAX	integer, floating point (real only)	smallest possible value
MIN	integer, floating point (real only)	largest possible value
IAND	integer	all bits on
IOR	integer	0
IEOR	integer	0

Table 3.5 Reduction operators for C/C++.

Operator	*Data Types*	*Initial Value*
+	integer, floating point	0
*	integer, floating point	1
–	integer, floating point	0
&	integer	all bits on
|	integer	0
^	integer	0
&&	integer	1
||	integer	0

For example, the parallel version of the sum reduction looks like this:

```
sum = 0
!$omp parallel do reduction(+ : sum)
do i = 1, n
    sum = sum + a(i)
enddo
```

At runtime, each thread performs a portion of the additions that make up the final sum as it executes its portion of the *n* iterations of the *parallel do* loop. At the end of the parallel loop, the threads combine their partial sums into a final sum. Although threads may perform the additions in an order that differs from that of the original serial program, the final result remains the same because of the commutative-associative property of the "+" operator (though as we will see shortly, there may be slight differences due to floating-point roundoff errors).

The behavior of the *reduction* clause, as well as restrictions on its use, are perhaps best understood by examining an equivalent OpenMP code that performs the same computation in parallel without using the reduction clause itself. The code in Example 3.8 may be viewed as a possible translation of the reduction clause by an OpenMP implementation, although implementations will likely employ other clever tricks to improve efficiency.

Example 3.8 Equivalent OpenMP code for parallelized reduction.

```
      sum = 0
!$omp parallel private(priv_sum) shared(sum)
      ! holds each thread's partial sum
      priv_sum = 0

!$omp do
      ! same as serial do loop
      ! with priv_sum replacing sum
      do i = 1, n
         ! compute partial sum
         priv_sum = priv_sum + a(i)
      enddo

      ! combine partial sums into final sum
      ! must synchronize because sum is shared
!$omp critical
      sum = sum + priv_sum
!$omp end critical
!$omp end parallel
```

As shown in Example 3.8, the code declares a new, private variable called *priv_sum*. Within the body of the *do* loop all references to the original reduction variable *sum* are replaced by references to this private variable. The variable *priv_sum* is initialized to zero just before the start of the loop and is used within the loop to compute each thread's partial sum. Since this variable is private, the *do* loop can be executed in parallel. After the *do* loop the threads may need to synchronize as they aggregate their partial sums into the original variable, *sum*.

The *reduction* clause is best understood in terms of the behavior of the above transformed code. As we can see, the user only need supply the reduction operator and the variable with the *reduction* clause and can leave the rest of the details to the OpenMP implementation. Furthermore, the reduction variable may be passed as a parameter to other subroutines that perform the actual update of the reduction variable; as we can see, the above transformation will continue to work regardless of whether the actual update is within the lexical extent of the directive or not. However, the programmer is responsible for ensuring that any modifications to the variable within the parallel loop are consistent with the reduction operator that was specified.

In Tables 3.4 and 3.5, the data types listed for each operator are the allowed types for reduction variables updated using that operator. For example, in Fortran and C, addition can be performed on any floating point or integer type. Reductions may only be performed on built-in types of the base language, not user-defined types such as a *record* in Fortran or *class* in C++.

In Example 3.8 the private variable *priv_sum* is initialized to zero just before the reduction loop. In mathematical terms, zero is the *identity value* for addition; that is, zero is the value that when added to any other value *x*, gives back the value *x*. In an OpenMP reduction, each thread's partial reduction result is initialized to the identity value for the reduction operator. The identity value for each reduction operator appears in the "Initial Value" column of Tables 3.4 and 3.5.

One caveat about parallelizing reductions is that when the type of the reduction variable is floating point, the final result may not be precisely the same as when the reduction is performed serially. The reason is that floating-point operations induce roundoff errors because floating-point variables have only limited precision. For example, suppose we add up four floating-point numbers that are accurate to four decimal digits. If the numbers are added up in this order (rounding off intermediate results to four digits):

```
((0.0004 + 1.000) + 0.0004) + 0.0002 = 1.000
```

we get a different result from adding them up in this ascending order:

```
((0.0002 + 0.0004) + 0.0004) + 1.000 = 1.001
```

For some programs, differences between serial and parallel versions resulting from roundoff may be unacceptable, so floating-point reductions in such programs should not be parallelized.

Finally, care must be exercised when parallelizing reductions that use subtraction ("–") or the C "&&" or "||" operators. Subtraction is in fact not a commutative-associative operator, so the code to update the reduction variable must be rewritten (typically replacing "–" by " + ") for the parallel reduction to produce the same result as the serial one. The C logical operators "&&" and "||" short-circuit (do not evaluate) their right operand if the result can be determined just from the left operand. It is therefore not desirable to have side effects in the expression that updates the reduction variable because the expression may be evaluated more or fewer times in the parallel case than in the serial one.

3.4.7 Private Variable Initialization and Finalization

Normally, each thread's copy of a variable scoped as private on a *parallel do* has an undefined initial value, and after the *parallel do* the master thread's copy also takes on an undefined value. This behavior has the advantage that it minimizes data copying for the common case in which we use the private variable as a temporary within the parallel loop. However, when parallelizing a loop we sometimes need access to the value that was in the master's copy of the variable just before the loop, and we sometimes need to copy the "last" value written to a private variable back to the master's copy at the end of the loop. (The "last" value is the value assigned in the last iteration of a sequential execution of the loop—this last iteration is therefore called "sequentially last.")

For this reason, OpenMP provides the *firstprivate* and *lastprivate* variants on the *private* clause. At the start of a *parallel do, firstprivate* initializes each thread's copy of a private variable to the value of the master's copy. At the end of a *parallel do, lastprivate* writes back to the master's copy the value contained in the private copy belonging to the thread that executed the sequentially last iteration of the loop.

The form and usage of *firstprivate* and *lastprivate* are the same as the *private* clause: each takes as an argument a list of variables. The variables in the list are scoped as private within the *parallel do* on which the clause appears, and in addition are initialized or finalized as described above. As was mentioned in Section 3.4.1, variables may appear in at most one scope clause, with the exception that a variable can appear in both *firstprivate* and *lastprivate,* in which case it is both initialized and finalized.

In Example 3.9, *x(1,1)* and *x(2,1)* are assigned before the parallel loop and only read thereafter, while *x(1,2)* and *x(2,2)* are used within the loop as temporaries to store terms of polynomials. Code after the loop uses the terms of the last polynomial, as well as the last value of the index variable *i*. Therefore *x* appears in a *firstprivate* clause, and both *x* and *i* appear in a *lastprivate* clause.

Example 3.9 Parallel loop with *firstprivate* and *lastprivate* variables.

```
      common /mycom/ x, c, y, z
      real x(n, n), c(n, n,), y(n), z(n)
      ...
      ! compute x(1, 1) and x(2, 1)
!$omp parallel do firstprivate(x) lastprivate(i, x)
      do i = 1, n
         x(1, 2) = c(i, 1) * x(1, 1)
         x(2, 2) = c(i, 2) * x(2, 1) ** 2
         y(i) = x(2, 2) + x(1, 2)
         z(i) = x(2, 2) - x(1, 2)
      enddo
      ...
      ! use x(1, 2), x(2, 2), and i
```

There are two important caveats about using these clauses. The first is that a *firstprivate* variable is initialized only once per thread, rather than once per iteration. In Example 3.9, if any iteration were to assign to *x(1,1)* or *x(2,1)*, then no other iteration is guaranteed to get the initial value if it reads these elements. For this reason *firstprivate* is useful mostly in cases like Example 3.9, where part of a privatized array is read-only. The second caveat is that if a *lastprivate* variable is a compound object (such as an array or structure), and only some of its elements or fields are assigned in the last iteration, then after the parallel loop the elements or fields that were not assigned in the final iteration have an undefined value.

In C++, if an object is scoped as *firstprivate* or *lastprivate,* the initialization and finalization are performed using appropriate member functions of the object. In particular, a *firstprivate* object is constructed by calling its copy constructor with the master thread's copy of the variable as its argument, while if an object is *lastprivate,* at the end of the loop the copy assignment operator is invoked on the master thread's copy, with the sequentially last value of the variable as an argument. (It is an error if a *firstprivate* object has no publicly accessible copy constructor, or a *lastprivate* object has no publicly accessible copy assignment operator.) Example 3.10 shows how this works. Inside the parallel loop, each private copy of *c1* is copy-constructed such that its *val* member has the value 2. On the last iteration, 11 is assigned to *c2.val*, and this value is copy-assigned back to the master thread's copy of *c2*.

Example 3.10 *firstprivate* and *lastprivate* objects in C++.

```
class C {
public:
    int val;
    // default constructor
    C() { val = 0; }

    C(int _val) { val = _val; }
    // copy constructor
    C(const C &c) { val = c.val; }
    // copy assignment operator
    C & operator = (const C &c) {
        val = c.val;
        return * this;
    }
};

void f () {
    C c1(2), c2(3);
    ...
    #pragma omp for firstprivate(c1) lastprivate(c2)
    for (int i = 0; i < 10; i++) {
        #pragma omp critical
        c2.val = c1.val + i;     // c1.val == 2
    }
    // after the loop, c2.val == 11
}
```

3.5 Removing Data Dependences

Up to this point in the chapter, we have concentrated on describing OpenMP's features for parallelizing loops. For the remainder of the chapter we will mostly discuss how to use these features to parallelize loops correctly and effectively.

First and foremost, when parallelizing loops it is necessary to maintain the program's correctness. After all, a parallel program is useless if it produces its results quickly but the results are wrong! The key characteristic of a loop that allows it to run correctly in parallel is that it must not contain any data dependences. In this section we will explain what data dependences are and what kinds of dependences there are. We will lay out a methodology for determining whether a loop contains any data dependences, and show how in many cases these dependences can be removed by transforming the program's source code or using OpenMP clauses.

This section provides only a brief glimpse into the topic of data dependences. While we will discuss the general rules to follow to deal correctly with dependences and show precisely what to do in many common cases, there are numerous special cases and techniques for handling them that we do not have space to address. For a more thorough introduction to this topic, and many pointers to further reading, we refer you to Michael Wolfe's book [MW 96]. In addition, a useful list of additional simple techniques for finding and breaking dependences appears in Chapter 5 of [SGI 99].

3.5.1 Why Data Dependences Are a Problem

Whenever one statement in a program reads or writes a memory location, and another statement reads or writes the same location, and at least one of the two statements writes the location, we say that there is a *data dependence* on that memory location between the two statements. In this context, a memory location is anything to which the program can assign a scalar value, such as an integer, character, or floating-point value. Each scalar variable, each element of an array, and each field of a structure constitutes a distinct memory location. Example 3.11 shows a loop that contains a data dependence: each iteration (except the last) writes an element of *a* that is read by the next iteration. Of course, a single statement can contain multiple memory references (reads and writes) to the same location, but it is usually the case that the references involved in a dependence occur in different statements, so we will assume this from now on. In addition, there can be dependences on data external to the program, such as data in files that is accessed using I/O statements, so if you wish to parallelize code that accesses such data, you must analyze the code for dependences on this external data as well as on data in variables.

Example 3.11 A simple loop with a data dependence.

```
do i = 2, N
    a(i) = a(i) + a(i - 1)
enddo
```

For purposes of parallelization, data dependences are important because whenever there is a dependence between two statements on some location, we cannot execute the statements in parallel. If we did execute them in parallel, it would cause what is called a *data race*. A parallel program contains a data race whenever it is possible for two or more statements to read or write the same memory location at the same time, and at least one of the statements writes the location.

In general, data races cause correctness problems because when we execute a parallel program that contains a data race, it may not produce the same results as an equivalent serial program. To see why, consider what might happen if we try to parallelize the loop in Example 3.11 by naively applying a *parallel do* directive. Suppose *n* is 3, so the loop iterates just twice, and at the start of the loop, the first three elements of *a* have been initialized to the values 1, 2, and 3. After a correct serial execution, the first three values are 1, 3, and 6. However, in a parallel execution it is possible for the assignment of the value 3 to *a(2)* in the first iteration to happen either before or after the read of *a(2)* in the second iteration (the two statements are "racing" each other). If the assignment happens after the read, *a(3)* receives an incorrect value of 5.

3.5.2 The First Step: Detection

Now that we have seen why data dependences are a problem, the first step in dealing with them is to detect any that are present in the loop we wish to parallelize. Since each iteration executes in parallel, but within a single iteration statements in the loop body are performed in sequence, the case that concerns us is a dependence between statements executed in different iterations of the loop. Such a dependence is called *loop-carried.*

Because dependences are always associated with a particular memory location, we can detect them by analyzing how each variable is used within the loop, as follows:

- Is the variable only read and never assigned within the loop body? If so, there are no dependences involving it.
- Otherwise, consider the memory locations that make up the variable and that are assigned within the loop. For each such location, is there exactly one iteration that accesses the location? If so, there are no dependences involving the variable. If not, there is a dependence.

To perform this analysis, we need to reason about each memory location accessed in each iteration of the loop. Reasoning about scalar variables is usually straightforward, since they uniquely identify the memory location being referenced. Reasoning about array variables, on the other hand, can be tricky because different iterations may access different locations due to the use of array subscript expressions that vary from one iteration to the next. The key is to recognize that we need to find two different values of the parallel loop index variable (call them i and i') that both lie within the bounds of the loop, such that iteration i assigns to some element of an array a, and iteration i' reads or writes that same element of a. If we can find suitable values for i and i', there is a dependence involving the array. If we can satisfy ourselves that there are no such values, there is no dependence involving the array. As a simple rule of thumb, a loop that meets all the following criteria has no dependences and can always be parallelized:

- All assignments are to arrays.
- Each element is assigned by at most one iteration.
- No iteration reads elements assigned by any other iteration.

When all the array subscripts are linear expressions in terms of *i* (as is often the case), we can use the subscript expressions and constraints imposed by the loop bounds to form a system of linear inequalities whose solutions identify all of the loop's data dependences. There are well-known general techniques for solving systems of linear inequalities, such as integer programming and Fourier-Motzkin projection. However, discussion of these techniques is beyond the scope of this book, so you should see [MW 96] for an introduction. Instead, in many practical cases the loop's bounds and subscript expressions are simple enough that we can find these loop index values *i* and *i′* just by inspection. Example 3.11 is one such case: each iteration *i* writes element a_i, while each iteration $i + 1$ reads a_i, so clearly there is a dependence between each successive pair of iterations.

Example 3.12 contains additional common cases that demonstrate how to reason about dependences and hint at some of the subtleties involved. The loop at line 10 is quite similar to that in Example 3.11, but in fact contains no dependence: Unlike Example 3.11, this loop has a stride of 2, so it writes every other element, and each iteration reads only elements that it writes or that are not written by the loop. The loop at line 20 also contains no dependences because each iteration reads only the element it writes plus an element that is not written by the loop since it has a subscript greater than *n*/2. The loop at line 30 is again quite similar to that at line 20, yet there is a dependence because the first iteration reads $a(n/2 + 1)$ while the last iteration writes this element. Finally, the loop at line 40 uses subscripts that are not linear expressions of the index variable *i*. In cases like this we must rely on whatever knowledge we have of the index expression. In particular, if the index array *idx* is known to be a *permutation* array—that is, if we know that no two elements of *idx* have the same value (which is frequently the case for index arrays used to represent linked-list structures)—we can safely parallelize this loop because each iteration will read and write a different element of *a*.

Example 3.12 Loops with nontrivial bounds and array subscripts.

```
10  do i = 2, n, 2
       a(i) = a(i) + a(i - 1)
    enddo
```

```
20  do i = 1, n/2
       a(i) = a(i) + a(i + n/2)
    enddo

30  do i = 1, n/2 + 1
       a(i) = a(i) + a(i + n/2)
    enddo

40  do i = 1, n
       a(idx(i)) = a(idx(i)) + b(idx(i))
    enddo
```

Of course, loop nests can contain more than one loop, and arrays can have more than one dimension. The three-deep loop nest in Example 3.13 computes the product of two matrices $C = A \times B$. For reasons we will explain later in the chapter, we usually want to parallelize the outermost loop in such a nest. For correctness, there must not be a dependence between any two statements executed in different iterations of the parallelized loop. However, there may be dependences between statements executed within a single iteration of the parallel loop, including dependences between different iterations of an inner, serial loop. In this matrix multiplication example, we can safely parallelize the *j* loop because each iteration of the *j* loop computes one column *c(1:n, j)* of the product and does not access elements of *c* that are outside that column. The dependence on *c(i, j)* in the serial *k* loop does not inhibit parallelization.

Example 3.13 Matrix multiplication.

```
do j = 1, n
    do i = 1, n
        c(i, j) = 0
        do k = 1, n
            c(i, j) = c(i, j) + a(i, k) * b(k, j)
        enddo
    enddo
enddo
```

It is important to remember that dependences must be analyzed not just within the lexical extent of the loop being parallelized, but within its entire dynamic extent. One major source of data race bugs is that subroutines called within the loop body may assign to variables that would have shared scope if the loop were executed in parallel. In Fortran, this problem is typically caused by variables in *common* blocks, variables with the *save* attribute, and *module* data (in Fortran 90); in C and C++ the usual culprits are global and static variables. Furthermore, we must also examine how subroutines called from a parallel loop use their parameters. There may be a dependence if a subroutine writes to a scalar output parameter, or if there is overlap in the portions of array parameters that are accessed by subroutines called from different iterations of the loop. In Example 3.14, the loop at line 10 cannot be parallelized because each iteration reads and writes the shared variable *cnt* in subroutine *add*. The loop at line 20 has a dependence due to an overlap in the portions of array argument *a* that are accessed in the call: subroutine *smooth* reads both elements of *a* that are adjacent to the element it writes, and *smooth* writes each element of *a* in parallel. Finally, the loop at line 30 has no dependences because subroutine *add_count* only accesses *a(i)* and only reads *cnt*.

Example 3.14 Loops containing subroutine calls.

```
    subroutine add(c, a, b)
    common /count/ cnt
    integer c, a, b, cnt

    c = a + b
    cnt = cnt + 1
    end

    subroutine smooth(a, n, i)
    integer n, a(n), i

    a(i) = (a(i) + a(i - 1) + a(i + 1))/3
    end

    subroutine add_count(a, n, i)
    common /count/ cnt
    integer n, a(n), i, cnt

    a(i) = a(i) + cnt
    end

10  do i = 1, n
       call add(c(i), a(i), b(i))
    enddo

20  do i = 2, n - 1
       call smooth(a, n, i)
    enddo

30  do i = 1, n
       call add_count(a, n, i)
    enddo
```

3.5.3 The Second Step: Classification

Once a dependence has been detected, the next step is to figure out what kind of dependence it is. This helps determine whether it needs to be removed, whether it can be removed, and, if it can, what technique to use to remove it. We will discuss two different classification schemes that are particularly useful for parallelization.

We mentioned in Section 3.5.2 that dependences may be classified based on whether or not they are loop-carried, that is, whether or not the two statements involved in the dependence occur in different iterations of the parallel loop. A non-loop-carried dependence does not cause a data race because within a single iteration of a parallel loop, each statement is executed in sequence, in the same way that the master thread executes serial portions of the program. For this reason, non-loop-carried dependences can generally be ignored when parallelizing a loop.

One subtle special case of non-loop-carried dependences occurs when a location is assigned in only some rather than all iterations of a loop. This case is illustrated in Example 3.15, where the assignment to *x* is controlled by the *if* statement at line 10. If the assignment were performed in every iteration, there would be just a non-loop-carried dependence between the assignment and the use of *x* at line 20, which we could ignore. But because the assignment is performed only in some iterations, there is in fact a loop-carried dependence between line 10 in one iteration and line 20 in the next. In other words, because the assignment is controlled by a conditional, *x* is involved in both a non-loop-carried dependence between lines 10 and 20 (which we can ignore) and a loop-carried dependence between the same lines (which inhibits parallelization).

Example 3.15 A loop-carried dependence caused by a conditional.

```
      x = 0
      do i = 1, n
10       if (switch_val(i)) x = new_val(i)
20       a(i) = x
      enddo
```

There is one other scheme for classifying dependences that is crucial for parallelization. It is based on the *dataflow* relation between the two dependent statements, that is, it concerns whether or not the two statements communicate values through the memory location. Let the statement performed earlier in a sequential execution of the loop be called S_1, and let the later statement be called S_2. The kind of dependence that is the most important and difficult to handle is when S_1 writes the memory location, S_2 reads the location, and the value read by S_2 in a serial execution is the same as that written by S_1. In this case the result of a computation by S_1 is communicated, or "flows," to S_2, so we call this kind a *flow dependence.* Because S_1 must execute first to produce the value that is consumed by S_2, in general we cannot remove the dependence and execute the two statements in parallel (hence this case is sometimes called a "true" dependence). However, we will see in Section 3.5.4 that there are some situations in which we can parallelize loops that contain flow dependences.

In this dataflow classification scheme, there are two other kinds of dependences. We can always remove these two kinds because they do not represent communication of data between S_1 and S_2, but instead are instances of reuse of the memory location for different purposes at different points in the program. In the first of these, S_1 reads the location, then S_2 writes it. Because this memory access pattern is the opposite of a flow dependence, this case is called an *anti dependence.* As we will see shortly, we can parallelize a loop that contains an anti dependence by giving each iteration a private copy of the location and initializing the copy belonging to S_1 with the value S_1 would have read from the location during a serial execution. In the second of the two kinds, both S_1 and S_2 write the location. Because only writing occurs, this is called an *output dependence.* Suppose we execute the loop serially, and give the name v to the last value written to the location. We will show below that we can always parallelize in the presence of an output dependence by privatizing the memory location and in addition copying v back to the shared copy of the location at the end of the loop.

To make all these categories of dependences clearer, the loop in Example 3.16 contains at least one instance of each. Every iteration of the loop is involved in six different dependences, which are listed in Table 3.6. For each dependence, the table lists the associated memory location and

Table 3.6 List of data dependences present in Example 3.16.

Memory Location	*Earlier Statement*			*Later Statement*			*Loop-carried?*	*Kind of Dataflow*
	Line	*Iteration*	*Access*	*Line*	*Iteration*	*Access*		
x	10	i	write	20	i	read	no	flow
x	10	i	write	10	$i + 1$	write	yes	output
x	20	i	read	10	$i + 1$	write	yes	anti
$a(i + 1)$	20	i	read	20	$i + 1$	write	yes	anti
$b(i)$	30	i	write	30	$i + 1$	read	yes	flow
$c(2)$	40	i	write	40	$i + 1$	write	yes	output

earlier and later dependent statements (S_1 and S_2), as well as whether the dependence is loop-carried and its dataflow classification. The two statements are identified by their line number, iteration number, and the kind of memory access they perform on the location. Although in reality every iteration is involved with every other iteration in an anti and an output dependence on x and an output dependence on *c(2)*, for brevity the table shows just the dependences between iterations i and $i + 1$. In addition, the loop index variable i is only read by the statements in the loop, so we ignore any dependences involving the variable i. Finally, notice that there are no dependences involving d because this array is read but not written by the loop.

Example 3.16 A loop containing multiple data dependences.

```
      do i = 2, N - 1
10        x = d(i) + i
20        a(i) = a(i + 1) + x
30        b(i) = b(i) + b(i - 1) + d(i - 1)
40        c(2) = 2 * i
      enddo
```

3.5.4 The Third Step: Removal

With a few exceptions, it is necessary to remove each loop-carried dependence within a loop that we wish to parallelize. Many dependences can be removed either by changing the scope of the variable involved in the dependence using a clause on the *parallel do* directive, or by transforming the program's source code in a simple manner, or by doing both. We will first present techniques for dealing with the easier dataflow categories of anti and output dependences, which can in principle always be removed, although this may sometimes be inefficient. Then we will discuss several special cases of flow dependences that we are able to remove, while pointing out that there are many instances in which removal of flow dependences is either impossible or requires extensive algorithmic changes.

When changing a program to remove one dependence in a parallel loop, it is critical that you not violate any of the other dependences that are present. In addition, if you introduce additional loop-carried dependences, you must remove these as well.

Removing Anti and Output Dependences

In an anti dependence, there is a race between statement S_1 reading the location and S_2 writing it. We can break the race condition by giving each thread or iteration a separate copy of the location. We must also ensure that S_1 reads the correct value from the location. If each iteration initializes the location before S_1 reads it, we can remove the dependence just by privatization. In Example 3.17, there is a non-loop-carried anti dependence on the variable *x* that is removed using this technique. On the other hand, the value read by S_1 may be assigned before the loop, as is true of the array element *a(i + 1)* read in line 10 of Example 3.17. To remove this dependence, we can make a copy of the array *a* before the loop (called *a2*) and read the copy rather than the original array within the parallel loop. Of course, creating a copy of the array adds memory and computation overhead, so we must ensure that there is enough work in the loop to justify the additional overhead.

Example 3.17 Removal of anti dependences.

Serial version containing anti dependences:

```
      ! Array a is assigned before start of loop.
      do i = 1, N - 1
         x = (b(i) + c(i))/2
 10      a(i) = a(i + 1) + x
      enddo
```

Parallel version with dependences removed:

```
!$omp parallel do shared(a, a2)
      do i = 1, N - 1
         a2(i) = a(i + 1)
      enddo

!$omp parallel do shared(a, a2) private(x)
      do i = 1, N - 1
         x = (b(i) + c(i))/2
 10      a(i) = a2(i) + x
      enddo
```

In Example 3.18, the last values assigned to *x* and *d(1)* within the loop and the value assigned to *d(2)* before the loop are read by a statement that follows the loop. We say that the values in these locations are *live-out* from the loop. Whenever we parallelize a loop, we must ensure that live-out locations have the same values after executing the loop as they would have if the loop were executed serially. If a live-out variable is scoped as shared on a parallel loop and there are no loop-carried output dependences on it (i.e., each of its locations is assigned by at most one iteration), then this condition is satisfied.

On the other hand, if a live-out variable is scoped as private (to remove a dependence on the variable) or some of its locations are assigned by more than one iteration, then we need to perform some sort of finalization to ensure that it holds the right values when the loop is finished. To parallelize the loop in Example 3.18, we must finalize both *x* (because it is scoped private to break an anti dependence on it) and *d* (because there is a loop-carried output dependence on *d(1)*). We can perform finalization on *x* simply by scoping it with a *lastprivate* rather than *private* clause. As we explained in Section 3.4.7, the *lastprivate* clause both scopes a variable as private within a parallel loop and also copies the value assigned to the variable in the last iteration of the loop back to the shared copy. It requires slightly more work to handle a case like the output dependence on *d(1)*: we cannot scope *d* as *lastprivate* because if we did, it would overwrite the live-out value in *d(2)*. One solution, which we use in this example, is to introduce a *lastprivate* temporary (called *d1*) to copy back the final value for *d(1)*.

Of course, if the assignment to a live-out location within a loop is performed only conditionally (such as when it is part of an *if* statement), the *lastprivate* clause will not perform proper finalization because the final value of the location may be assigned in some iteration other than the last, or the location may not be assigned at all by the loop. It is still possible to perform finalization in such cases: for example, we can keep track of the last value assigned to the location by each thread, then at the end of the loop we can copy back the value assigned in the highest-numbered iteration. However, this sort of finalization technique is likely to be much more expensive than the *lastprivate* clause. This highlights the fact that, in general, we can always preserve correctness when removing anti and output dependences, but we may have to add significant overhead to remove them.

Example 3.18 Removal of output dependences.

Serial version containing output dependences:

```
do i = 1, N
   x = (b(i) + c(i))/2
   a(i) = a(i) + x
   d(1) = 2 * x
enddo
y = x + d(1) + d(2)
```

Parallel version with dependences removed:

```
!$omp parallel do shared(a) lastprivate(x, d1)
      do i = 1, N
         x = (b(i) + c(i))/2
         a(i) = a(i) + x
         d1 = 2 * x
      enddo
      d(1) = d1

      y = x + d(1) + d(2)
```

Removing Flow Dependences

As we stated before, we cannot always remove a flow dependence and run the two dependent statements in parallel because the computation performed by the later statement, S_2, depends on the value produced by the earlier one, S_1. However, there are some special cases in which we can remove flow dependences, three of which we will now describe. We have in fact already seen the first case: reduction computations. In a reduction, such as that depicted in Example 3.19, the statement that updates the reduction variable also reads the variable, which causes a loop-carried flow dependence. But as we discussed in Section 3.4.6, we can remove this flow dependence and parallelize the reduction computation by scoping the variable with a *reduction* clause that specifies the operator with which to update the variable.

Example 3.19 Removing the flow dependence caused by a reduction.

Serial version containing a flow dependence:

```
      x = 0
      do i = 1, N
         x = x + a(i)
      enddo
```

Parallel version with dependence removed:

```
      x = 0
!$omp parallel do reduction(+: x)
      do i = 1, N
         x = x + a(i)
      enddo
```

If a loop updates a variable in the same fashion as a reduction, but also uses the value of the variable in some expression other than the one that computes the updated value (*idx, i_sum*, and *pow2* are updated and used in this way in Example 3.20), we cannot remove the flow dependence simply by scoping the variable with a *reduction* clause. This is because the values of a reduction variable during each iteration of a parallel execution differ from those of a serial execution. However, there is a special class of reduction computations, called *inductions,* in which the value of the reduction variable during each iteration is a simple function of the loop index variable. For example, if the variable is updated using multiplication by a constant, increment by a constant, or increment by the loop index variable, then we can replace uses of the variable within the loop by a simple expression containing the loop index. This technique is called *induction variable elimination,* and we use it in Example 3.20 to remove loop-carried flow dependences on *idx*, *i_sum*, and *pow2*. The expression for *i_sum* relies on the fact that

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

This kind of dependence often appears in loops that initialize arrays and when an induction variable is introduced to simplify array subscripts (*idx* is used in this way in the example).

Example 3.20 Removing flow dependences using induction variable elimination.

Serial version containing flow dependences:

```
      idx = N/2 + 1
      i_sum = 1
      pow2 = 2
      do i = 1, N/2
         a(i) = a(i) + a(idx)
         b(i) = i_sum
         c(i) = pow2
         idx = idx + 1
         i_sum = i_sum + i
         pow2 = pow2 * 2
      enddo
```

Parallel version with dependences removed:

```
!$omp parallel do shared(a, b, c)
      do i = 1, N/2
         a(i) = a(i) + a(i + N/2)
         b(i) = i * (i + 1)/2
         c(i) = 2 ** i
      enddo
```

The third technique we will describe for removing flow dependences is called *loop skewing*. The basic idea of this technique is to convert a loop-carried flow dependence into a non-loop-carried one. Example 3.21 shows a loop that can be parallelized by skewing it. The serial version of the loop has a loop-carried flow dependence from the assignment to *a(i)* at line 20 in iteration *i* to the read of *a(i – 1)* at line 10 in iteration *i* + 1. However, we can compute each element *a(i)* in parallel because its value does not depend on any other elements of *a*. In addition, we can shift, or "skew," the subsequent read of *a(i)* from iteration *i* + 1 to iteration *i,* so that the dependence becomes non-loop-carried. After adjusting subscripts and loop bounds appropriately, the final parallelized version of the loop appears at the bottom of Example 3.21.

Example 3.21 Removing flow dependences using loop skewing.

Serial version containing flow dependence:

```
      do i = 2, N
 10      b(i) = b(i) + a(i - 1)
 20      a(i) = a(i) + c(i)
      enddo
```

Parallel version with dependence removed:

```
      b(2) = b(2) + a(1)
!$omp parallel do shared(a, b, c)
      do i = 2, N - 1
 20      a(i) = a(i) + c(i)
 10      b(i + 1) = b(i + 1) + a(i)
      enddo
      a(N) = a(N) + c(N)
```

Dealing with Nonremovable Dependences

Although we have just seen several straightforward parallelization techniques that can remove certain categories of loop-carried flow dependences, in general this kind of dependence is difficult or impossible to remove. For instance, the simple loop in Example 3.22 is a member of a common category of computations called *recurrences*. It is impossible to parallelize this loop using a single *parallel do* directive and simple transformation of the source code because computing each element *a(i)* requires that we have the value of the previous element *a(i – 1)*. However, by using a completely different algorithm called a *parallel scan,* it is possible to compute this and other kinds of recurrences in parallel (see Exercise 3).

Example 3.22 A recurrence computation that is difficult to parallelize.

```
do i = 2, N
    a(i) = (a(i - 1) + a(i))/2
enddo
```

Even when we cannot remove a particular flow dependence, we may be able to parallelize some other part of the code that contains the dependence. We will show three different techniques for doing this. When applying any of these techniques, it is critical to make sure that we do not violate any of the other dependences that are present and do not introduce any new loop-carried dependences that we cannot remove.

The first technique is applicable only when the loop with the nonremovable dependence is part of a nest of at least two loops. The technique is quite simple: try to parallelize some other loop in the nest. In Example 3.23, the *j* loop contains a recurrence that is difficult to remove, so we can parallelize the *i* loop instead. As we will see in Section 3.6.1 and in Chapter 6, the choice of which loop in the nest runs in parallel can have a profound impact on performance, so the parallel loop must be chosen carefully.

Example 3.23 Parallelization of a loop nest containing a recurrence.

Serial version:

```
      do j = 1, N
         do i = 1, N
            a(i, j) = a(i, j) + a(i, j - 1)
         enddo
      enddo
```

Parallel version:

```
      do j = 1, N
!$omp parallel do shared(a)
         do i = 1, N
            a(i, j) = a(i, j) + a(i, j - 1)
         enddo
      enddo
```

The second technique assumes that the loop that contains the nonremovable dependence also contains other code that can be parallelized. By splitting, or fissioning, the loop into a serial and a parallel portion, we can achieve a speedup on at least the parallelizable portion of the loop. (Of course, when fissioning the loop we must not violate any dependences between the serial and parallel portions.) In Example 3.24, the recurrence computation using *a* in line 10 is hard to parallelize, so we fission it off into a serial loop and parallelize the rest of the original loop.

Example 3.24 Parallelization of part of a loop using fissioning.

Serial version:

```
      do i = 1, N
 10      a(i) = a(i) + a(i - 1)
 20      y = y + c(i)
      enddo
```

Parallel version:

```
      do i = 1, N
 10      a(i) = a(i) + a(i - 1)
      enddo

!$omp parallel do reduction(+: y)
      do i = 1, N
 20      y = y + c(i)
      enddo
```

The third technique also involves splitting the loop into serial and parallel portions. However, unlike fissioning this technique can also move non-loop-carried flow dependences from statements in the serial portion to statements in the parallel portion. In Example 3.25, there are loop-carried and non-loop-carried flow dependences on *y*. We cannot remove the loop-carried dependence, but we can parallelize the computation in line 20. The trick is that in iteration *i* of the parallel loop we must have available the value that is assigned to *y* during iteration *i* of a serial execution of the loop. To make it available, we fission off the update of *y* in line 10. Then we perform a transformation, called *scalar expansion,* that stores in a temporary array *y2* the value assigned to *y* during each iteration of the serial loop. Finally, we parallelize the loop that contains line 20 and replace references to *y* with references to the appropriate element of *y2*. A major disadvantage of this technique is that it introduces significant memory overhead due to the use of the temporary array, so you should use scalar expansion only when the speedup is worth the overhead.

Example 3.25 Parallelization of part of a loop using scalar expansion and fissioning.

Serial version:

```
      do i = 1, N
 10      y = y + a(i)
 20      b(i) = (b(i) + c(i)) * y
      enddo
```

Parallel version:

```
      y2(1) = y + a(1)
      do i = 2, N
 10      y2(i) = y2(i - 1) + a(i)
      enddo
      y = y2(N)

!$omp parallel do shared(b, c, y2)
      do i = 1, N
 20      b(i) = (b(i) + c(i)) * y2(i)
      enddo
```

3.5.5 Summary

To prevent data races in a loop we are parallelizing, we must remove each loop-carried dependence that is present. There is a loop-carried dependence whenever two statements in different iterations access a memory location, and at least one of the statements writes the location. Based upon the dataflow through the memory location between the two statements, each dependence may be classified as an anti, output, or flow dependence.

We can remove anti dependences by providing each iteration with an initialized copy of the memory location, either through privatization or by introducing a new array variable. Output dependences can be ignored unless the location is live-out from the loop. The values of live-out variables can often be finalized using the *lastprivate* clause.

We cannot always remove loop-carried flow dependences. However, we can parallelize a reduction, eliminate an induction variable, or skew a loop to make a dependence become non-loop-carried. If we cannot remove a flow dependence, we may instead be able to parallelize another loop in the nest, fission the loop into serial and parallel portions, or remove a dependence on a nonparallelizable portion of the loop by expanding a scalar into an array.

Whenever we modify a loop to remove a dependence, we should be careful that the memory or computational cost of the transformation does not exceed the benefit of parallelizing the loop. In addition, we must not violate any of the other data dependences present in the loop, and we must remove any new loop-carried dependences that we introduce.

The discussion in this section has emphasized data dependences that appear within parallel loops. However, the general rules for detecting, classifying, and removing dependences remain the same in the presence of other forms of parallelism, such as coarse-grained parallelism expressed using parallel regions.

3.6 Enhancing Performance

There is no guarantee that just because a loop has been correctly parallelized, its performance will improve. In fact, in some circumstances parallelizing the wrong loop can slow the program down. Even when the choice of loop is reasonable, some performance tuning may be necessary to make the loop run acceptably fast.

Two key factors that affect performance are parallel overhead and loop scheduling. OpenMP provides several features for controlling these factors, by means of clauses on the *parallel do* directive. In this section we will briefly introduce these two concepts, then describe OpenMP's performance-tuning features for controlling these aspects of a program's behavior. There are several other factors that can have a big impact on performance, such as synchronization overhead and memory cache utilization. These topics receive a full discussion in Chapters 5 and 6, respectively.

3.6.1 Ensuring Sufficient Work

From the discussion of OpenMP's execution model in Chapter 2, it should be clear that running a loop in parallel adds runtime costs: the master thread has to start the slaves, iterations have to be divided among the threads, and threads must synchronize at the end of the loop. We call these additional costs *parallel overhead.*

Each iteration of a loop involves a certain amount of work, in the form of integer and floating-point operations, loads and stores of memory locations, and control flow instructions such as subroutine calls and branches. In many loops, the amount of work per iteration may be small, perhaps just a few instructions (*saxpy* is one such case, as it performs just a load, a multiply, an add, and a store on each iteration). For these loops, the parallel overhead for the loop may be orders of magnitude larger than the average time to execute one iteration of the loop. Unfortunately, if we add a *parallel do* directive to the loop, it is always run in parallel each time. Due to the parallel overhead, the parallel version of the loop may run slower than the serial version when the trip-count is small.

Suppose we find out by measuring execution times that the smallest trip-count for which parallel *saxpy* is faster than the serial version is about 800. Then we could avoid slowdowns at low trip-counts by versioning the loop, that is, by creating separate serial and parallel versions. This is shown in the first part of Example 3.26.

Example 3.26 Avoiding parallel overhead at low trip-counts.

Using explicit versioning:

```
      if (n .ge. 800) then
!$omp parallel do
         do i = 1, n
            z(i) = a * x(i) + y
         enddo
      else
         do i = 1, n
            z(i) = a * x(i) + y
         enddo
      end if
```

Using the *if* clause:

```
!$omp parallel do if (n .ge. 800)
      do i = 1, n
         z(i) = a * x(i) + y
      enddo
```

Creating separate copies by hand is tedious, so OpenMP provides a feature called the *if* clause that versions loops more concisely. The *if* clause appears on the *parallel do* directive and takes a Boolean expression as an argument. Each time the loop is reached during execution, if the expression evaluates to true, the loop is run in parallel. If the expression is false, the loop is executed serially, without incurring parallel overhead.

Just as it may be unknown until runtime whether there is sufficient work in a loop to make parallel execution worthwhile, it may be unknown until runtime whether there are data dependences within the loop. The *if* clause can be useful in these circumstances to cause a loop to execute serially if a runtime test determines that there are dependences, and to execute in parallel otherwise.

When we want to speed up a loop nest, it is generally best to parallelize the loop that is as close as possible to being outermost. This is because of parallel overhead: we pay this overhead each time we reach a parallel loop, and the outermost loop in the nest is reached only once each time control reaches the nest, whereas inner loops are reached once per iteration of the loop that encloses them.

Because of data dependences, the outermost loop in a nest may not be parallelizable. In the first part of Example 3.27, the outer loop is a recurrence, while the inner loop is parallelizable. However, if we put a *parallel do* directive on the inner loop, we will incur the parallel overhead $n - 1$ times, that is, each time we reach the *do*=i loop nest. We can solve this problem by applying a source transformation called *loop interchange* that swaps the positions of inner and outer loops. (Of course, when we transform the loop nest in this way, we must respect its data dependences, so that it produces the same results as before.) The second part of the example shows the same loop nest, after interchanging the loops and parallelizing the now-outermost i loop.

Example 3.27 Reducing parallel overhead through loop interchange.

Original serial code:

```
      do j = 2, n           ! Not parallelizable.
         do i = 1, n        ! Parallelizable.
            a(i, j) = a(i, j) + a(i, j - 1)
         enddo
      enddo
```

Parallel code after loop interchange:

```
!$omp parallel do
      do i = 1, n
         do j = 2, n
            a(i, j) = a(i, j) + a(i, j - 1)
         enddo
      enddo
```

In this example, we have reduced the total amount of parallel overhead, but as we will see in Chapter 6, the transformed loop nest has worse utilization of the memory cache. This implies that transformations that improve one aspect of a program's performance may hurt another aspect; these trade-offs must be made carefully when tuning the performance of an application.

3.6.2 Scheduling Loops to Balance the Load

As we have mentioned before, the manner in which iterations of a parallel loop are assigned to threads is called the loop's *schedule*. Using the default schedule on most implementations, each thread executing a parallel loop performs about as many iterations as any other thread. When each iteration has approximately the same amount of work (such as in the *saxpy* loop), this causes threads to carry about the same amount of *load* (i.e, to perform about the same total amount of work) and to finish the loop at about the same time. Generally, when the load is *balanced* fairly equally among threads, a loop runs faster than when the load is unbalanced. So for a simple parallel loop such as *saxpy*, the default schedule is close to optimal.

Unfortunately it is often the case that different iterations have different amounts of work. Consider the code in Example 3.28. Each iteration of the loop may invoke either one of the subroutines *smallwork* or *bigwork*. Depending on the loop instance, therefore, the amount of work per iteration may vary in a regular way with the iteration number (say, increasing or decreasing linearly), or it may vary in an irregular or even random way. If the load in such a loop is unbalanced, there will be synchronization delays at some later point in the program, as faster threads wait for slower ones to catch up. As a result the total execution time of the program will increase.

Example 3.28 Parallel loop with an uneven load.

```
!$omp parallel do private(xkind)
      do i = 1, n
         xkind = f(i)
         if (xkind .lt. 10) then
            call smallwork(x[i])
         else
            call bigwork(x[i])
         endif
      enddo
```

By changing the schedule of a load-unbalanced parallel loop, it is possible to reduce these synchronization delays and thereby speed up the program. A schedule is specified by a *schedule* clause on the *parallel do* directive. In this section we will describe each of the options available for scheduling, concentrating on the mechanics of how each schedule assigns iterations to threads and on the overhead it imposes. Chapter 6 discusses the trade-offs between the different scheduling types and provides guidelines for selecting a schedule for the best performance.

3.6.3 Static and Dynamic Scheduling

One basic characteristic of a loop schedule is whether it is *static* or *dynamic:*

- In a *static* schedule, the choice of which thread performs a particular iteration is purely a function of the iteration number and number of threads. Each thread performs only the iterations assigned to it at the beginning of the loop.
- In a *dynamic* schedule, the assignment of iterations to threads can vary at runtime from one execution to another. Not all iterations are assigned to threads at the start of the loop. Instead, each thread requests more iterations after it has completed the work already assigned to it.

A dynamic schedule is more flexible: if some threads happen to finish their iterations sooner, more iterations are assigned to them. However, the OpenMP runtime system must coordinate these assignments to guarantee that every iteration gets executed exactly once. Because of this coordination, requests for iterations incur some synchronization cost. Static scheduling has lower overhead because it does not incur this cost, but it cannot compensate for load imbalances by shifting more iterations to less heavily loaded threads.

In both schemes, iterations are assigned to threads in contiguous ranges called *chunks.* The *chunk size* is the number of iterations a chunk contains. For example, if we executed the *saxpy* loop on an array with 100 elements using four threads and the default schedule, thread 1 might be assigned a single chunk of 25 iterations in which *i* varies from 26 to 50. When using a dynamic schedule, each time a thread requests more iterations from the OpenMP runtime system, it receives a new chunk to work on. For example, if *saxpy* were executed using a dynamic schedule with a fixed chunk size of 10, then thread 1 might be assigned three chunks with iterations 11 to 20, 41 to 50, and 81 to 90.

3.6.4 Scheduling Options

Each OpenMP scheduling option assigns chunks to threads either statically or dynamically. The other characteristic that differentiates the schedules is the way chunk sizes are determined. There is also an option for the schedule to be determined by an environment variable.

The syntactic form of a *schedule* clause is

```
schedule(type[, chunk])
```

The type is one of *static, dynamic, guided,* or *runtime.* If it is present, *chunk* must be a scalar integer value. The kind of schedule specified by the clause depends on a combination of the type and whether *chunk* is present, according to these rules, which are also summarized in Table 3.7:

- If type is *static* and *chunk* is not present, each thread is statically assigned one chunk of iterations. The chunks will be equal or nearly equal in size, but the precise assignment of iterations to threads depends on the OpenMP implementation. In particular, if the number of iterations is not evenly divisible by the number of threads, the runtime system is free to divide the remaining iterations among threads as it wishes. We will call this kind of schedule “simple static.”
- If type is *static* and *chunk* is present, iterations are divided into chunks of size *chunk* until fewer than *chunk* remain. How the remaining iterations are divided into chunks depends on the implementation. Chunks are statically assigned to processors in a round-robin fashion: the first thread gets the first chunk, the second thread gets the second chunk, and so on, until no more chunks remain. We will call this kind of schedule “interleaved.”
- If type is *dynamic*, iterations are divided into chunks of size *chunk*, similarly to an interleaved schedule. If *chunk* is not present, the size of all chunks is 1. At runtime, chunks are assigned to threads dynamically. We will call this kind of schedule “simple dynamic.”
- If type is *guided,* the first chunk is of some implementation-dependent size, and the size of each successive chunk decreases exponentially (it is a certain percentage of the size of the preceding chunk) down to a minimum size of *chunk*. (An example that shows what this looks like appears below.) The value of the exponent depends on the implementation. If fewer than *chunk* iterations remain, how the rest are divided into chunks also depends on the implementation. If *chunk* is not specified, the minimum chunk size is 1. Chunks are assigned to threads dynamically. Guided scheduling is sometimes also called “guided self-scheduling,” or “GSS.”
- If type is *runtime, chunk* must not appear. The schedule *type* is chosen at runtime based on the value of the environment variable *omp_schedule*. The environment variable should be set to a string that matches the parameters that may appear in parentheses in a *schedule*

clause. For example, if the program is run on a UNIX system, then performing the C shell command

```
setenv OMP_SCHEDULE "dynamic,3"
```

before executing the program would result in the loops specified as having a *runtime* schedule being actually executed with a dynamic schedule with chunk size 3. If *OMP_SCHEDULE* was not set, the choice of schedule depends on the implementation.

The syntax is the same for Fortran and C/C++. If a parallel loop has no schedule clause, the choice of schedule is left up to the implementation; typically the default is simple static scheduling.

A word of caution: It is important that the correctness of a program not depend on the schedule chosen for its parallel loops. A common way for such a dependence to creep into a program is if one iteration writes a value that is read by another iteration that occurs later in a sequential execution. If the loop is first parallelized using a schedule that happens to assign both iterations to the same thread, the program may get correct results at first, but then mysteriously stop working if the schedule is changed while tuning performance. If the schedule is dynamic, the program may fail only intermittently, making debugging even more difficult.

3.6.5 Comparison of Runtime Scheduling Behavior

To help make sense of these different scheduling options, Table 3.7 compares them in terms of several characteristics that affect performance. In the table, *N* is the trip-count of the parallel loop, *P* is the number of threads executing the loop, and *C* is the user-specified chunk size (if a schedule accepts a *chunk* argument but none was specified, *C* is 1 by default).

Determining the lower and upper bounds for each chunk of a loop's iteration space involves some amount of computation. While the precise cost of a particular schedule depends on the implementation, we can generally expect some kinds of schedules to be more expensive than others. For example, a guided schedule is typically the most expensive of all because it uses the most complex function to compute the size of each chunk. These relative computation costs appear in the "Compute Overhead" column of Table 3.7.

As we have said, dynamic schedules are more flexible in the sense that they assign more chunks to threads that finish their chunks earlier. As a result, these schedules can potentially balance the load better. To be

Table 3.7 Comparison of scheduling options.

Name	*type*	*chunk*	*Chunk Size*	*Number of Chunks*	*Static or Dynamic*	*Compute Overhead*
Simple static	*simple*	no	N/P	P	static	lowest
Interleaved	*simple*	yes	C	N/C	static	low
Simple dynamic	*dynamic*	optional	C	N/C	dynamic	medium
Guided	*guided*	optional	decreasing from N/P	fewer than N/C	dynamic	high
Runtime	*runtime*	no	varies	varies	varies	varies

more precise, if a simple dynamic schedule has a chunk size of C, it is guaranteed that at the end of the loop, the fastest thread never has to wait for the slowest thread to complete more than C iterations. So the quality of load balancing improves as C decreases. The potential for improved load balancing by dynamic schedules comes at the cost of one synchronization per chunk, and the number of chunks increases with decreasing C. This means that choosing a chunk size is a trade-off between the quality of load balancing and the synchronization and computation costs.

The main benefit of a guided schedule over a simple dynamic one is that guided schedules require fewer chunks, which lowers synchronization costs. A guided schedule typically picks an initial chunk size K_0 of about N/P, then picks successive chunk sizes using this formula:

$$K_i = \left(1 - \frac{1}{P}\right) \times K_{i-1}$$

For example, a parallel loop with a trip-count of 1000 executed by eight threads is divided into 41 chunks using a guided schedule when the minimum chunk size C is 1. A simple dynamic schedule would produce 1000 chunks. If C were increased to 25, the guided schedule would use 20 chunks, while a simple dynamic one would use 40. Because the chunk size of a guided schedule decreases exponentially, the number of chunks required increases only logarithmically with the number of iterations. So if the trip-count of our loop example were increased to 10,000, a simple dynamic schedule would produce 400 chunks when C is 25, but a guided schedule would produce only 38. For 100,000 iterations, the guided schedule would produce only 55.

In cases when it is not obvious which schedule is best for a parallel loop, it may be worthwhile to experiment with different schedules and measure the results. In other cases, the best schedule may depend on the input data set. The runtime scheduling option is useful in both of these situations because it allows the schedule type to be specified for each run of the program simply by changing an environment variable rather than by recompiling the program.

Which schedule is best for a parallel loop depends on many factors. This section has only discussed in general terms such properties of schedules as their load balancing capabilities and overheads. Chapter 6 contains specific advice about how to choose schedules based on the work distribution patterns of different kinds of loops, and also based upon performance considerations such as data locality that are beyond the scope of this chapter.

3.7 Concluding Remarks

In this chapter we have described how to exploit loop-level parallelism in OpenMP using the *parallel do* directive. We described the basic techniques used to identify loops that can safely be run in parallel and presented some of the factors that affect the performance of the parallel loop. We showed how each of these concerns may be addressed using the constructs in OpenMP.

Loop-level parallelism is one of the most common forms of parallelism found in programs, and is also the easiest to express. The *parallel do* construct is therefore among the most important of the OpenMP constructs. Having mastered loop-level parallelism, in the next chapter we will move on to exploiting more general forms of parallelism.

3.8 Exercises

1. Explain why each of the following loops can or cannot be parallelized with a *parallel do* (or *parallel for*) directive.

a)
```
     do i = 1, N
          if (x(i) .gt. maxval) goto 100
     enddo
100  continue
```

b)
```
     x(N/2:N) = a * y(N/2:N) + z(N/2:N)
```

c)
```
     do i = 1, N
          do j = 1, size(i)
               a(j, i) = a(j, i) + a(j + 1, i)
```

```
            enddo
        enddo
```

d)
```
     for (i = 0; i < N; i++) {
         if (weight[i] > HEAVY) {
             pid = fork();
             if (pid == -1) {
                 perror("fork");
                 exit(1);
             }
             if (pid == 0) {
                 heavy_task();
                 exit(1);
             }
         }
         else
             light_task();
     }
```

e)
```
     do i = 1, N
        a(i) = a(i) * a(i)
        if (fabs(a(i)) .gt. machine_max .or. &
           fabs(a(i)) .lt. machine_min) then
           print *, i
           stop
        endif
     enddo
```

2. Consider the following loop:

```
       x = 1
!$omp parallel do firstprivate(x)
       do i = 1, N
          y(i) = x + i
          x = i
       enddo
```

a) Why is this loop incorrect? (*Hint:* Does *y(i)* get the same result regardless of the number of threads executing the loop?)

b) What is the value of *i* at the end of the loop? What is the value of *x* at the end of the loop?

c) What would be the value of *x* at the end of the loop if it was scoped *shared*?

d) Can this loop be parallelized correctly (i.e., preserving sequential semantics) just with the use of directives?

3. There are at least two known ways of parallelizing the loop in Example 3.11, although not trivially with a simple *parallel do* directive. Implement one. (*Hint:* The simplest and most general method uses parallel regions, introduced in Chapter 2 and the focus of the next chapter. There is, however, a way of doing this using only *parallel do* directives, although it requires additional storage, takes $O(N \log N)$ operations, and it helps if N is a power of two. Both methods rely on partial sums.)
4. Write a parallel loop that benefits from dynamic scheduling.
5. Consider the following loop:

```
!$omp parallel do schedule(static, chunk)
      do i = 1, N
         x(i) = a * x(i) + b
      enddo
```

   Assuming the program is running on a cache-based multiprocessor system, what happens to the performance when we choose a chunk size of 1? 2? Experiment with chunk sizes that are powers of two, ranging up to 128. Is there a discontinuous jump in performance? Be sure to time only the loop and also make sure *x(i)* is initialized (so that the timings are not polluted with the cost of first mapping in *x(i)*). Explain the observed behavior.

CHAPTER 4

Beyond Loop-Level Parallelism: Parallel Regions

4.1 Introduction

THE PREVIOUS CHAPTER FOCUSED ON EXPLOITING loop-level parallelism using OpenMP. This form of parallelism is relatively easy to exploit and provides an incremental approach towards parallelizing an application, one loop at a time. However, since loop-level parallelism is based on local analysis of individual loops, it is limited in the forms of parallelism that it can exploit. A global analysis of the algorithm, potentially including multiple loops as well as other noniterative constructs, can often be used to parallelize larger portions of an application such as an entire phase of an algorithm. Parallelizing larger and larger portions of an application in turn yields improved speedups and scalable performance.

This chapter focuses on the support provided in OpenMP for moving beyond loop-level parallelism. This support takes two forms. First, OpenMP provides a generalized *parallel region* construct to express parallel execution. Rather than being restricted to a loop (as with the *parallel do* construct discussed in the previous chapter), this construct is attached to an arbitrary body of code that is executed concurrently by multiple threads. This form of replicated execution, with the body of code executing in a replicated fashion across multiple threads, is commonly referred to as "SPMD"-style parallelism, for "single-program multiple-data."

Second, within such a parallel body of code, OpenMP provides several constructs that *divide* the execution of code portions across multiple threads. These constructs are referred to as *work-sharing constructs* and are used to partition work across the multiple threads. For instance, one work-sharing construct is used to distribute iterations of a loop across multiple threads within a parallel region. Another work-sharing construct is used to assign distinct code segments to different threads and is useful for exploiting unstructured, noniterative forms of parallelism. Taken together, the parallel region construct and the work-sharing constructs enable us to exploit more general SPMD-style parallelism within an application.

The rest of this chapter proceeds as follows. We first describe the form and usage of the parallel region construct, along with the clauses on the construct, in Section 4.2. We then describe the behavior of the parallel region construct, along with the corresponding runtime execution model, in Section 4.3. We then describe the data scoping issues that are specific to the *parallel* directive in Section 4.4. Next, we describe the various ways to express work-sharing within OpenMP in Section 4.5 and present the restrictions on work-sharing constructs in Section 4.6. We describe the notion of orphaned work-sharing constructs in Section 4.7 and address nested parallel constructs in Section 4.8. Finally, we describe the mechanisms to query and control the runtime execution parameters (such as the number of parallel threads) in Section 4.9.

4.2 Form and Usage of the *parallel* Directive

The *parallel* construct in OpenMP is quite simple: it consists of a *parallel/end parallel* directive pair that can be used to enclose an arbitrary block of code. This directive pair specifies that the enclosed block of code, referred to as a *parallel region,* be executed in parallel by multiple threads.

The general form of the *parallel* directive in Fortran is

```
!$omp parallel [clause [,] [clause ...]]
    block
!$omp end parallel
```

In C and C++, the format is

```
#pragma omp parallel [clause [clause] ...]
    block
```

4.2.1 Clauses on the *parallel* Directive

The *parallel* directive may contain any of the following clauses:

```
PRIVATE (list)
SHARED (list)
DEFAULT (PRIVATE | SHARED | NONE)
REDUCTION ({op|intrinsic}:list)
IF (logical expression)
COPYIN (list)
```

The *private, shared, default, reduction,* and *if* clauses were discussed earlier in Chapter 3 and continue to provide exactly the same behavior for the *parallel* construct as they did for the *parallel do* construct. We briefly review these clauses here.

The *private* clause is typically used to identify variables that are used as scratch storage in the code segment within the parallel region. It provides a list of variables and specifies that each thread have a private copy of those variables for the duration of the parallel region.

The *shared* clause provides the exact opposite behavior: it specifies that the named variable be shared among all the threads, so that accesses from any thread reference the same shared instance of that variable in global memory. This clause is used in several situations. For instance, it is used to identify variables that are accessed in a read-only fashion by multiple threads, that is, only read and not modified. It may be used to identify a variable that is updated by multiple threads, but with each thread updating a distinct location within that variable (e.g., the *saxpy* example from Chapter 2). It may also be used to identify variables that are modified by multiple threads and used to communicate values between multiple threads during the parallel region (e.g., a shared error flag variable that may be used to denote a global error condition to all the threads).

The *default* clause is used to switch the default data-sharing attributes of variables: while variables are *shared* by default, this behavior may be switched to either *private* by default through the *default(private)* clause, or to unspecified through the *default(none)* clause. In the latter case, all variables referenced within the parallel region must be explicitly named in one of the above data-sharing clauses.

Finally, the *reduction* clause supplies a reduction operator and a list of variables, and is used to identify variables used in reduction operations within the parallel region.

The *if* clause dynamically controls whether a parallel region construct executes in parallel or in serial, based on a runtime test. We will have a bit more to say about this clause in Section 4.9.1.

Before we can discuss the *copyin* clause, we need to introduce the notion of *threadprivate* variables. This is the subject of Section 4.4.

4.2.2 Restrictions on the *parallel* Directive

The *parallel* construct consists of a *parallel/end parallel* directive pair that encloses a block of code. The section of code that is enclosed between the *parallel* and *end parallel* directives must be a *structured* block of code—that is, it must be a block of code consisting of one or more statements that is entered at the top (at the start of the parallel region) and exited at the bottom (at the end of the parallel region). Thus, this block of code must have a single entry point and a single exit point, with no branches into or out of any statement within the block. While branches within the block of code are permitted, branches to or from the block from without are not permitted.

Example 4.1 is not valid because of the presence of the *return* statement within the parallel region. The return statement is a branch out of the parallel region and therefore is not allowed.

Although it is not permitted to branch into or out of a parallel region, Fortran *stop* statements are allowed within the parallel region. Similarly, code within a parallel region in C/C++ may call the *exit* subroutine. If any thread encounters a *stop* statement, it will execute the *stop* statement and signal all the threads to stop. The other threads are signalled asynchronously, and no guarantees are made about the precise execution point where the other threads will be interrupted and the program stopped.

Example 4.1 Code that violates restrictions on parallel regions.

```
      subroutine sub(max)
      integer n

!$omp parallel
      call mypart(n)
      if (n .gt. max) return
!$omp end parallel

      return
      end
```

4.3 Meaning of the *parallel* Directive

The *parallel* directive encloses a block of code, a parallel region, and creates a team of threads to execute a copy of this block of code in parallel. The threads in the team concurrently execute the code in the parallel region in a replicated fashion.

We illustrate this behavior with a simple example in Example 4.2. This code fragment contains a parallel region consisting of the single print statement shown. Upon execution, this code behaves as follows (see Figure 4.1). Recall that by default an OpenMP program executes sequentially on a single thread (the master thread), just like an ordinary serial program. When the program encounters a construct that specifies parallel execution, it creates a parallel team of threads (the slave threads), with each thread in the team executing a copy of the body of code enclosed within the *parallel/end parallel* directive. After each thread has finished executing its copy of the block of code, there is an implicit barrier while the program waits for all threads to finish, after which the master thread (the original sequential thread) continues execution past the *end parallel* directive.

Example 4.2 A simple parallel region.

```
      ...
!$omp parallel
      print *, 'Hello world'
!$omp end parallel
      ...
```

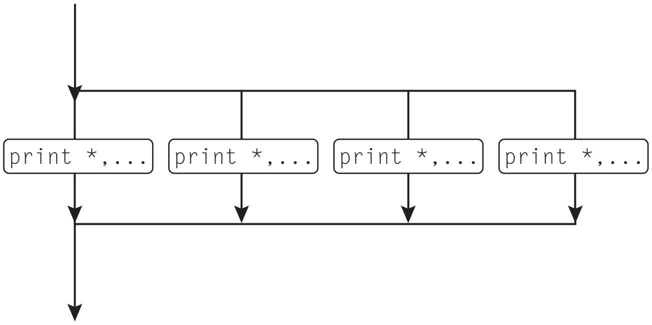


Figure 4.1 Runtime execution model for a parallel region.

Let us examine how the parallel region construct compares with the *parallel do* construct from the previous chapter. While the *parallel do* construct was associated with a loop, the parallel region construct can be associated with an arbitrary block of code. While the *parallel do* construct specified that multiple iterations of the *do* loop execute concurrently, the parallel region construct specifies that the block of code within the parallel region execute concurrently on multiple threads without any synchronization. Finally, in the *parallel do* construct, each thread executes a distinct iteration instance of the *do* loop; consequently, iterations of the *do* loop are divided among the team of threads. In contrast, the parallel region construct executes a *replicated* copy of the block of code in the parallel region on each thread.

We examine this final difference in more detail in Example 4.3. In this example, rather than containing a single print statement, we have a parallel region construct that contains a *do* loop of, say, 10 iterations. When this example is executed, a team of threads is created to execute a copy of the enclosed block of code. This enclosed block is a *do* loop with 10 iterations. Therefore, each thread executes 10 iterations of the *do* loop, printing the value of the loop index variable each time around. If we execute with a parallel team of four threads, a total of 40 print messages will appear in the output of the program (for simplicity we assume the print statements execute in an interleaved fashion). If the team has five threads, there will be 50 print messages, and so on.

Example 4.3 Replication of work with the parallel region directive.

```
!$omp parallel
      do i = 1, 10
         print *, 'Hello world', i
      enddo
!$omp end parallel
```

The *parallel do* construct, on the other hand, behaves quite differently. The construct in Example 4.4 executes a total of 10 iterations *divided* across the parallel team of threads. Regardless of the size of the parallel team (four threads, or more, or less), this program upon execution would produce a total of 10 print messages, with each thread in the team printing zero or more of the messages.

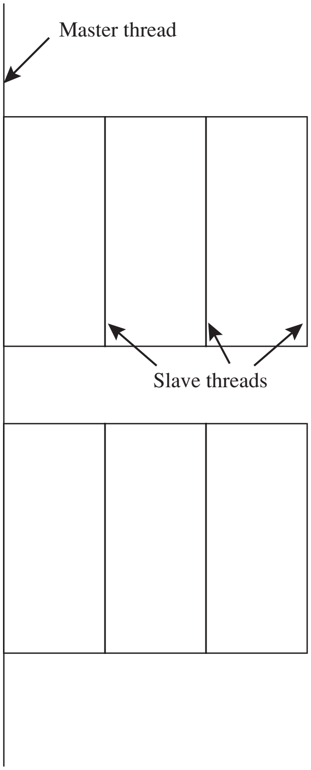
Example 4.4 Partitioning of work with the *parallel do* directive.

```
!$omp parallel do
      do i = 1, 10
         print *, 'Hello world', i
      enddo
```

These examples illustrate the difference between replicated execution (as exemplified by the parallel region construct) and work division across threads (as exemplified by the *parallel do* construct).

With replicated execution (and sometimes with the *parallel do* construct also), it is often useful for the programmer to query and control the number of threads in a parallel team. OpenMP provides several mechanisms to control the size of parallel teams; these are described later in Section 4.9.

Finally, an individual *parallel* construct invokes a team of threads to execute the enclosed code concurrently. An OpenMP program may encounter multiple *parallel* constructs. In this case each *parallel* construct individually behaves as described earlier—it gathers a team of threads to execute the enclosed construct concurrently, resuming serial execution once the *parallel* construct has completed execution. This process is repeated upon encountering another *parallel* construct, as shown in Figure 4.2.



```
      program main
      serial-region

!$omp parallel
      first parallel-region
!$omp end parallel

      serial-region

!$omp parallel
      second parallel-region
!$omp end parallel

      serial region

      end
```

Figure 4.2 Multiple parallel regions.

4.3.1 Parallel Regions and SPMD-Style Parallelism

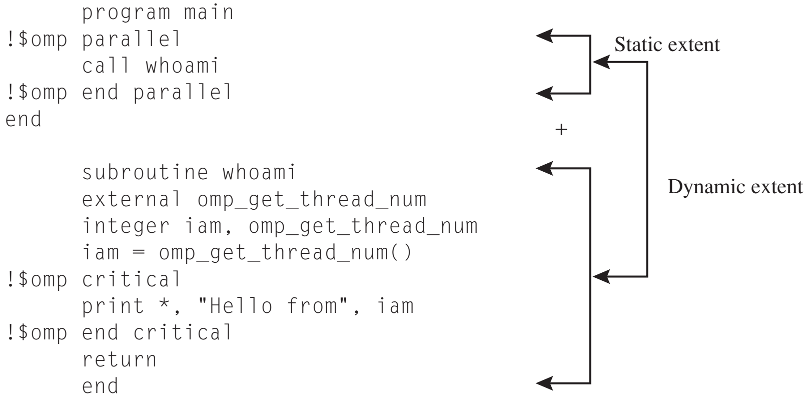
The *parallel* construct in OpenMP is a simple way of expressing parallel execution and provides replicated execution of the same code segment on multiple threads. It is most commonly used to exploit SPMD-style parallelism, where multiple threads execute the same code segments but on different data items. Subsequent sections in this chapter will describe different ways of distributing data items across threads, along with the specific constructs provided in OpenMP to ease this programming task.

4.4 *threadprivate* Variables and the *copyin* Clause

A parallel region encloses an arbitrary block of code, perhaps including calls to other subprograms such as another subroutine or function. We define the lexical or *static extent* of a parallel region as the code that is lexically within the *parallel/end parallel* directive. We define the *dynamic* extent of a parallel region to include not only the code that is directly between the *parallel* and *end parallel* directive (the static extent), but also to include all the code in subprograms that are invoked either directly or indirectly from within the parallel region. As a result the static extent is a subset of the statements in the dynamic extent of the parallel region.

Figure 4.3 identifies both the lexical (i.e., static) and the dynamic extent of the parallel region in this code example. The statements in the dynamic extent also include the statements in the lexical extent, along with the statements in the called subprogram *whoami*.

These definitions are important because the data scoping clauses described in Section 4.2.1 apply only to the lexical scope of a parallel



```
      program main
!$omp parallel
      call whoami
!$omp end parallel
end

      subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
!$omp critical
      print *, "Hello from", iam
!$omp end critical
      return
      end
```

Figure 4.3 A parallel region with a call to a subroutine.

region, and not to the entire dynamic extent of the region. For variables that are global in scope (such as *common* block variables in Fortran, or global variables in C/C++), references from within the lexical extent of a parallel region are affected by the data scoping clause (such as *private*) on the parallel directive. However, references to such global variables from the dynamic extent that are outside of the lexical extent are *not* affected by any of the data scoping clauses and always refer to the global shared instance of the variable.

Although at first glance this behavior may seem troublesome, the rationale behind it is not hard to understand. References within the lexical extent are easily associated with the data scoping clause since they are contained directly within the directive pair. However, this association is much less intuitive for references that are outside the lexical scope. Identifying the data scoping clause through a deeply nested call chain can be quite cumbersome and error-prone. Furthermore, the dynamic extent of a parallel region is not easily determined, especially in the presence of complex control flow and indirect function calls through function pointers (in C/C++). In general the dynamic extent of a parallel region is determined only at program runtime. As a result, extending the data scoping clauses to the full dynamic extent of a parallel region is extremely difficult and cumbersome to implement. Based on these considerations, OpenMP chose to avoid these complications by restricting data scoping clauses to the lexical scope of a parallel region.

Let us now look at an example to illustrate this issue further. We first present an incorrect piece of OpenMP code to illustrate the issue, and then present the corrected version.

Example 4.5 Data scoping clauses across lexical and dynamic extents.

```
      program wrong
      common /bounds/ istart, iend
      integer iarray(10000)

      N=10000
!$omp parallel private(iam, nthreads, chunk)
!$omp+ private (istart, iend)

      ! Compute the subset of iterations
      ! executed by each thread
      nthreads = omp_get_num_threads()
      iam = omp_get_thread_num()
      chunk = (N + nthreads - 1)/nthreads
      istart = iam * chunk + 1
      iend = min((iam + 1) * chunk, N)

      call work(iarray)
!$omp end parallel
      end
      subroutine work(iarray)

            ! Subroutine to operate on a thread's
            ! portion of the array "iarray"
            common /bounds/ istart, iend
            integer iarray(10000)

            do i = istart, iend
               iarray(i) = i * i
            enddo
            return
            end
```

In Example 4.5 we want to do some work on an array. We start a parallel region and make runtime library calls to fetch two values: *nthreads*, the number of threads in the team, and *iam*, the thread ID within the team of each thread. We calculate the portions of the array worked upon by each thread based on the thread id as shown. *istart* is the starting array index and *iend* is the ending array index for each thread. Each thread needs its own values of *iam*, *istart,* and *iend,* and hence we make them private for the parallel region. The subroutine *work* uses the values of *istart* and *iend* to work on a different portion of the array on each thread. We use a common block named *bounds* containing *istart* and *iend*, essentially containing the values used in both the main program and the subroutine.

However, this example will not work as expected. We correctly made *istart* and *iend* private, since we want each thread to have its own values of the index range for that thread. However, the private clause applies only to the references made from within the lexical scope of the parallel region. References to *istart* and *iend* from within the work subroutine are not affected by the private clause, and directly access the shared instances from the common block. The values in the common block are undefined and lead to incorrect runtime behavior.

Example 4.5 can be corrected by passing the values of *istart* and *iend* as parameters to the work subroutine, as shown in Example 4.6.

Example 4.6 Fixing data scoping through parameters.

```
            program correct
            common /bounds/ istart, iend
            integer iarray(10000)

            N = 10000
      !$omp parallel private(iam, nthreads, chunk)
      !$omp+ private(istart, iend)

            ! Compute the subset of iterations
            ! executed by each thread
      nthreads = omp_get_num_threads()
      iam = omp_get_thread_num()
      chunk = (N + nthreads - 1)/nthreads
      istart = iam * chunk + 1
      iend = min((iam + 1) * chunk, N)

      call work(iarray, istart, iend)
!$omp end parallel
      end

      subroutine work(iarray, istart, iend)

      ! Subroutine to operate on a thread's
      ! portion of the array "iarray"
      integer iarray(10000)

      do i = istart, iend
         iarray(i) = i * i
      enddo
      return
      end
```

By passing *istart* and *iend* as parameters, we have effectively replaced all references to these otherwise "global" variables to instead refer to the private copy of those variables within the parallel region. This program now behaves in the desired fashion.

4.4.1 The *threadprivate* Directive

While the previous example was easily fixed by passing the variables through the argument list instead of through the common block, it is often cumbersome to do so in real applications where the common blocks appear in several program modules. OpenMP provides an easier alternative that does not require modification of argument lists, using the *threadprivate* directive.

The *threadprivate* directive is used to identify a common block (or a global variable in C/C++) as being private to each thread. If a common block is marked as threadprivate using this directive, then a private copy of that entire common block is created for each thread. Furthermore, all references to variables within that common block anywhere in the entire program refer to the variable instance within the private copy of the common block in the executing thread. As a result, multiple references from within a thread, regardless of subprogram boundaries, always refer to the same private copy of that variable within that thread. Furthermore, threads cannot refer to the private instance of the common block belonging to another thread. As a result, this directive effectively behaves like a *private* clause except that it applies to the entire program, not just the lexical scope of a parallel region. (For those familiar with Cray systems, this directive is similar to the *taskcommon* specification on those machines.)

Let us look at how the *threadprivate* directive proves useful in our previous example. Example 4.7 contains a *threadprivate* declaration for the */bounds/*common block. As a result, each thread gets its own private copy of the entire common block, including the variables *istart* and *iend*. We make one further change to our original example: we no longer specify *istart* and *iend* in the private clause for the parallel region, since they are already private to each thread. In fact, supplying a private clause would be in error, since that would create a new private instance of these variables within the lexical scope of the parallel region, distinct from the threadprivate copy, and we would have had the same problem as in the first version of our example (Example 4.5). For this reason, the OpenMP specification does not allow threadprivate common block variables to appear in a *private* clause. With these changes, references to the variables *istart* and *iend* always refer to the private copy within that thread. Furthermore, references in both the main program as well as the work subroutine access the same threadprivate copy of the variable.

Example 4.7 Fixing data scoping using the *threadprivate* directive.

```
      program correct
      common /bounds/ istart, iend
!$omp threadprivate(/bounds/)
      integer iarray(10000)

      N = 10000
!$omp parallel private(iam, nthreads, chunk)

      ! Compute the subset of iterations
      ! executed by each thread
      nthreads = omp_get_num_threads()
      iam = omp_get_thread_num()
      chunk = (N + nthreads - 1)/nthreads
      istart = iam * chunk + 1
      iend = min((iam + 1) * chunk, N)

      call work(iarray)
!$omp end parallel
      end

      subroutine work(iarray)

      ! Subroutine to operate on a thread's
      ! portion of the array "iarray"
      common /bounds/ istart, iend
```

```
!$omp threadprivate(/bounds/)
      integer iarray(10000)

      do i = istart, iend
         iarray(i) = i * i
      enddo
      return
      end
```

Specification of the threadprivate *Directive*

The syntax of the *threadprivate* directive in Fortran is

```
!$omp threadprivate (/cb/[,/cb/]...)
```

where *cb1, cb2,* and so on are the names of common blocks to be made threadprivate, contained within slashes as shown. Blank (i.e., unnamed) common blocks cannot be made threadprivate. The corresponding syntax in C and C++ is

```
#pragma omp threadprivate (list)
```

where *list* is a list of named file scope or namespace scope variables.

The *threadprivate* directive must be provided *after* the declaration of the common block (or file scope or global variable in C/C++) within a subprogram unit. Furthermore, if a common block is threadprivate, then the threadprivate directive must be supplied after *every* declaration of the common block. In other words, if a common block is threadprivate, then it must be declared as such in all subprograms that use that common block: it is not permissible to have a common block declared threadprivate in some subroutines and not threadprivate in other subroutines.

Threadprivate common block variables must not appear in any other data scope clauses. Even the *default(private)* clause does not affect any threadprivate common block variables, which are always private to each thread. As a result, it is safe to use the *default(private)* clause even when threadprivate common block variables are being referenced in the parallel region.

A *threadprivate* directive has the following effect on the program: When the program begins execution there is only a single thread executing serially, the master thread. The master thread has its own private copy of the threadprivate common blocks.

When the program encounters a parallel region, a team of parallel threads is created. This team consists of the original master thread and some number of additional slave threads. Each slave thread has its own copy of the threadprivate common blocks, while the master thread continues to access its private copy as well. Both the initial copy of the master thread, as well as the copies within each of the slave threads, are initialized in the same way as the master thread's copy of those variables would be initialized in a serial instance of that program. For instance, in Fortran, a threadprivate variable would be initialized only if the program contained block data statements providing initial values for the common blocks. In C and C++, threadprivate variables are initialized if the program provided initial values with the definition of those variables, while objects in C++ would be constructed using the same constructor as for the master's copy. Initialization of each copy, if any, is done before the first reference to that copy, typically when the private copy of the threadprivate data is first created: at program startup time for the master thread, and when the threads are first created for the slave threads.

When the end of a parallel region is reached, the slave threads disappear, but they do not die. Rather, they park themselves on a queue waiting for the next parallel region. In addition, although the slave threads are dormant, they still retain their state, in particular their instances of the threadprivate common blocks. As a result, the contents of threadprivate data persist for each thread from one parallel region to another. When the next parallel region is reached and the slave threads are re-engaged, they can access their threadprivate data and find the values computed at the end of the previous parallel region. This persistence is guaranteed within OpenMP so long as the number of threads does not change. If the user modifies the requested number of parallel threads (say, through a call to a runtime library routine), then a new set of slave threads will be created, each with a freshly initialized set of threadprivate data.

Finally, during the serial portions of the program, only the master thread executes, and it accesses its private copy of the threadprivate data.

4.4.2 The *copyin* Clause

Since each thread has its own private copy of threadprivate data for the duration of the program, there is no way for a thread to access *another* thread's copy of such threadprivate data. However, OpenMP provides a limited facility for slave threads to access the master thread's copy of threadprivate data, through the *copyin* clause.

The *copyin* clause may be supplied along with a *parallel* directive. It can either provide a list of variables from within a threadprivate common block, or it can name an entire threadprivate common block. When a *copyin* clause is supplied with a parallel directive, the named threadprivate variables (or the entire threadprivate common block if so specified) within the private copy of each slave thread are initialized with the corresponding values in the master's copy. This propagation of values from the master to each slave thread is done at the start of the parallel region; subsequent to this initialization, references to the threadprivate variables proceed as before, referencing the private copy within each thread.

The *copyin* clause is helpful when the threadprivate variables are used for scratch storage within each thread but still need initial values that may either be computed by the master thread, or read from an input file into the master's copy. In such situations the *copyin* clause is an easy way to communicate these values from the master's copy to that of the slave threads.

The syntax of the *copyin* clause is

```
copyin (list)
```

where the list is a comma-separated list of names, with each name being either a threadprivate common block name, an individual threadprivate common block variable, or a file scope or global threadprivate variable in C/C++. When listing the names of threadprivate common blocks, they should appear between slashes.

We illustrate the *copyin* clause with a simple example. In Example 4.8 we have added another common block called *cm* with an array called *data*, and a variable *N* that holds the size of this data array being used as scratch storage. Although *N* would usually be a constant, in this example we are assuming that different threads use a different-sized subset of the data array. We therefore declare the *cm* common block as threadprivate. The master thread computes the value of *N* before the parallel region. Upon entering the parallel region, due to the *copyin* clause, each thread initializes its private copy of *N* with the value of *N* from the master thread.

Example 4.8 Using the *copyin* clause.

```
      common /bounds/ istart, iend
      common /cm/ N, data(1000)
!$omp threadprivate (/bounds/, /cm/)

      N = ...
!$omp parallel copyin(N)

      ! Each threadprivate copy of N is initialized
      ! with the value of N in the master thread.
      ! Subsequent modifications to N affect only
      ! the private copy in each thread
      ... = N
!$omp end parallel
      end
```

4.5 Work-Sharing in Parallel Regions

The *parallel* construct in OpenMP is a simple way of expressing parallel execution and provides replicated execution of the same code segment on multiple threads. Along with replicated execution, it is often useful to divide work among multiple threads—either by having different threads operate on different portions of a shared data structure, or by having different threads perform entirely different tasks. We now describe several ways of accomplishing this in OpenMP.

We present three different ways of accomplishing division of work across threads. The first example illustrates how to build a general parallel task queue that is serviced by multiple threads. The second example illustrates how, based on the id of each thread in a team, we can manually divide the work among the threads in the team. Together, these two examples are instances where the programmer manually divides work among a team of threads. Finally, we present some explicit OpenMP constructs to divide work among threads. Such constructs are termed *work-sharing* constructs.

4.5.1 A Parallel Task Queue

A parallel task queue is conceptually quite simple: it is a shared data structure that contains a list of work items or *tasks* to be processed. Tasks may range in size and complexity from one application to another. For instance, a task may be something very simple, such as processing an iteration (or a set of iterations) of a loop, and may be represented by just the loop index value. On the other hand, a complex task could consist of rendering a portion of a graphic image or scene on a display, and may be represented in a task list by a portion of an image and a rendering function. Regardless of their representation and complexity, however, tasks in a task queue typically share the following property: multiple tasks can be processed concurrently by multiple threads, with any necessary coordination expressed through explicit synchronization constructs. Furthermore, a given task may be processed by any thread from the team.

Parallelism is easily exploited in such a task queue model. We create a team of parallel threads, with each thread in the team repeatedly fetching and executing tasks from this shared task queue. In Example 4.9 we have a function that returns the index of the next task, and another subroutine that processes a given task. In this example we chose a simple task queue that consists of just an index to identify the task—the function *get_next_task* returns the next index to be processed, while the subroutine *process_task* takes an index and performs the computation associated with that index. Each thread repeatedly fetches and processes tasks, until all the tasks have been processed, at which point the parallel region completes and the master thread resumes serial execution.

Example 4.9 Implementing a task queue.

```
      ! Function to compute the next
      ! task index to be processed
      integer function get_next_task()
      common /mycom/ index
      integer index

!$omp critical
      ! Check if we are out of tasks
      if (index .eq. MAX) then
            get_next_task = -1
      else
            index = index + 1
            get_next_task = index
      endif
!$omp end critical
      return
      end

      program TaskQueue
      integer myindex, get_next_task

!$omp parallel private (myindex)
      myindex = get_next_task()
      do while (myindex .ne. -1)
         call process_task (myindex)
         myindex = get_next_task()
      enddo
!$omp end parallel
      end
```

Example 4.9 was deliberately kept simple. However, it does contain the basic ingredients of a task queue and can be generalized to more complex algorithms as needed.

4.5.2 Dividing Work Based on Thread Number

A parallel region is executed by a team of threads, with the size of the team being specified by the programmer or else determined by the implementation based on default rules. From within a parallel region, the number of threads in the current parallel team can be determined by calling the OpenMP library routine

```
integer function omp_get_num_threads()
```

Threads in a parallel team are numbered from 0 to number_of_threads – 1. This number constitutes a unique thread identifier and can be determined by invoking the library routine

```
integer function omp_get_thread_num()
```

The *omp_get_thread_num* function returns an integer value that is the identifier for the invoking thread. This function returns a different value when invoked by different threads. The master thread has the thread ID 0, while the slave threads have an ID ranging from 1 to *number_of_threads* – 1.

Since each thread can find out its thread number, we now have a way to divide work among threads. For instance, we can use the number of threads to divide up the work into as many pieces as there are threads. Furthermore, each thread queries for its thread number within the team and uses this thread number to determine its portion of the work.

Example 4.10 Using the thread number to divide work.

```
!$omp parallel private(iam)
      nthreads = omp_get_num_threads()
      iam = omp_get_thread_num()
      call work(iam, nthreads)
!$omp end parallel
```

Example 4.10 illustrates this basic concept. Each thread determines *nthreads* (the total number of threads in the team) and *iam* (its ID in this team of threads). Based on these two values, the subroutine *work* uses *iam* and *nthreads* to determine the portion of work assigned to the thread *iam* and executes that portion of the work. Each thread needs to have its own unique thread id; therefore we declare *iam* to be private to each thread.

We have seen this kind of manual work-sharing before, when dividing the iterations of a *do* loop among multiple threads.

Example 4.11 Dividing loop iterations among threads.

```
      program distribute_iterations
      integer istart, iend, chunk, nthreads, iam
      integer iarray(N)

!$omp parallel private(iam, nthreads, chunk)
!$omp+ private (istart, iend)
```

```
         ...
         ! Compute the subset of iterations
         ! executed by each thread
         nthreads = omp_get_num_threads()
         iam = omp_get_thread_num()
         chunk = (N + nthreads - 1)/nthreads
         istart = iam * chunk + 1
         iend = min((iam + 1) * chunk, N)
         do i = istart, iend
            iarray(i) = i * i
         enddo
!$omp end parallel
         end
```

In Example 4.11 we manually divide the iterations of a *do* loop among the threads in a team. Based on the total number of threads in the team, *nthreads,* and its own ID within that team, *iam,* each thread computes its portion of the iterations. This example performs a simple division of work—we try to divide the total number of iterations, *n*, equally among the threads, so that each thread gets “chunk” number of iterations. The first thread processes the first chunk number of iterations, the second thread the next chunk, and so on.

Again, this simple example illustrates a specific form of work-sharing, dividing the iterations of a parallel loop. This simple scheme can be easily extended to include more complex situations, such as dividing the iterations in a more complex fashion across threads, or dividing the iterations of multiple loops rather than just the single loop as in this example.

The next section introduces additional OpenMP constructs that substantially automate this task.

4.5.3 Work-Sharing Constructs in OpenMP

Example 4.11 presented the code to manually divide the iterations of a *do* loop among multiple threads. Although conceptually simple, it requires the programmer to code all the calculations for dividing iterations and rewrite the *do* loop from the original program. Compared with the *parallel do* construct from the previous chapter, this scheme is clearly primitive. The user could simply use a *parallel do* directive, leaving all the details of dividing and distributing iterations to the compiler/implementation; however, with a parallel region the user has to perform all these tasks manually. In an application with several parallel regions containing multiple *do* loops, this coding can be quite cumbersome.

This problem is addressed by the work-sharing directives in OpenMP. Rather than manually distributing work across threads (as in the previous examples), these directives allow the user to specify that portions of work should be divided across threads rather than executed in a replicated fashion. These directives relieve the programmer from coding the tedious details of work-sharing, as well as reduce the number of changes required in the original program.

There are three flavors of work-sharing directives provided within OpenMP: the *do* directive for distributing iterations of a *do* loop, the *sections* directive for distributing execution of distinct pieces of code among different threads, and the *single* directive to identify code that needs to be executed by a single thread only. We discuss each of these constructs next.

The do *Directive*

The work-sharing directive corresponding to loops is called the *do* work-sharing directive. Let us look at the previous example, written using the *do* directive. Compare Example 4.12 to the original code in Example 4.11. We start a parallel region as before, but rather than explicitly writing code to divide the iterations of the loop and parceling them out to individual threads, we simply insert the *do* directive before the *do* loop. The *do* directive does all the tasks that we had explicitly coded before, relieving the programmer from all the tedious bookkeeping details.

Example 4.12 Using the *do* work-sharing directive.

```
      program omp_do
      integer iarray(N)

!$omp parallel
       • • •
!$omp do
      do i = 1, N
         iarray(i) = i * i
      enddo
!$omp enddo
!$omp end parallel
      end
```

The *do* directive is strictly a work-sharing directive. It does not specify parallelism or create a team of parallel threads. Rather, within an existing team of parallel threads, it divides the iterations of a *do* loop across the parallel team. It is complementary to the parallel region construct. The parallel region directive spawns parallelism with replicated execution across a team of threads. In contrast, the *do* directive does not specify any parallelism, and rather than replicated execution it instead partitions the iteration space across multiple threads. This is further illustrated in Figure 4.4.

The precise syntax of the *do* construct in Fortran is

```
!$omp do [clause [,] [clause ...]]
      do i = ...
          ...
      enddo
!$omp enddo [nowait]
```

In C and C++ it is

```
#pragma omp for [clause [clause] ...]
    for-loop
```

where clause is one of the *private, firstprivate, lastprivate,* or *reduction* scoping clauses, or one of the *ordered* or *schedule* clauses. Each of these clauses has exactly the same behavior as for the *parallel do* directive discussed in the previous chapter.

By default, there is an implied barrier at the end of the *do* construct. If this synchronization is not necessary for correct execution, then the barrier may be avoided by the optional *nowait* clause on the *enddo* directive in Fortran, or with the *for* pragma in C and C++.

As illustrated in Example 4.13, the parallel region construct can be combined with the *do* directive to execute the iterations of a *do* loop in parallel. These two directives may be combined into a single directive, the familiar *parallel do* directive introduced in the previous chapter.

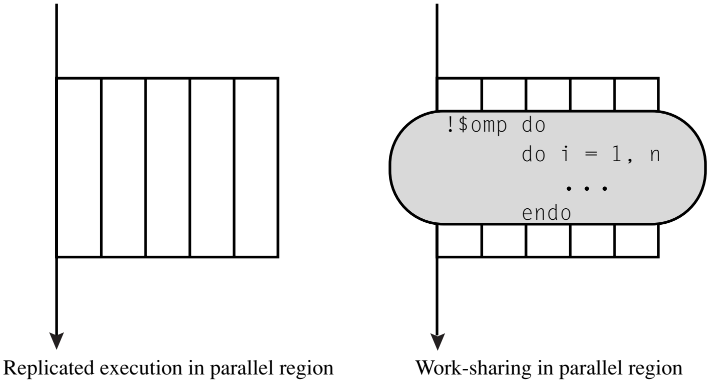

!$omp do
do i = 1, n
...
endo
Replicated execution in parallel region
Work-sharing in parallel region

Figure 4.4 Work-sharing versus replicated execution.

Example 4.13 Combining parallel region and work-sharing *do*.

```
!$omp parallel do
      do i = 1, N
         a(i) = a(i) **2
      enddo
!$omp end parallel do
```

This is the directive that exploits just loop-level parallelism, introduced in Chapter 3. It is essentially a shortened syntax for starting a parallel region followed by the *do* work-sharing directive. It is simpler to use when we need to run a loop in parallel. For more complex SPMD-style codes that contain a combination of replicated execution as well as work-sharing loops, we need to use the more powerful parallel region construct combined with the work-sharing *do* directive.

The *do* directive (and the other work-sharing constructs discussed in subsequent sections) enable us to easily exploit SPMD-style parallelism using OpenMP. With these directives, work-sharing is easily expressed through a simple directive, leaving the bookkeeping details to the underlying implementation. Furthermore, the changes required to the original source code are minimal.

Noniterative Work-Sharing: Parallel Sections

Thus far when discussing how to parallelize applications, we have been concerned primarily with splitting up the work of one task at a time among several threads. However, if the serial version of an application performs a sequence of tasks in which none of the later tasks depends on the results of the earlier ones, it may be more beneficial to assign different tasks to different threads. This is especially true in cases where it is difficult or impossible to speed up the individual tasks by executing them in parallel, either because the amount of work is too small or because the task is inherently serial. To handle such cases, OpenMP provides the *sections* work-sharing construct, which allows us to perform the entire sequence of tasks in parallel, assigning each task to a different thread.

The code for the entire sequence of tasks, or *sections,* begins with a *sections* directive and ends with an *end sections* directive. The beginning of each section is marked by a *section* directive, which is optional for the very first section. Another way to view it is that each section is separated from the one that follows by a *section* directive. The precise syntax of the *section* construct in Fortran is

```
!$omp section [clause [,] [clause ...]]
[!$omp section]
```

```
        code for the first section
    [!$omp section
        code for the second section
        ...
    ]
    !$omp end sections [nowait]
```

In C and C++ it is

```
    #pragma omp sections [clause [clause] ...]
        {
         [#pragma omp section]
            block
         [#pragma omp section
            block
        ...
        ...
        ]
       }
```

Each *clause* must be a *private, firstprivate, lastprivate,* or *reduction* scoping clause (C and C++ may also include the *nowait* clause on the pragma). The meaning of *private* and *firstprivate* is the same as for a *do* work-sharing construct. However, because a single thread may execute several sections, the value of a *firstprivate* variable can differ from that of the corresponding shared variable at the start of a section. On the other hand, if a variable *x* is made *lastprivate* within a *sections* construct, then the thread executing the section that appears last in the source code writes the value of its private *x* back to the corresponding shared copy of *x* after it has finished that section. Finally, if a variable *x* appears in a *reduction* clause, then after each thread finishes all sections assigned to it, it combines its private copy of *x* into the corresponding shared copy of *x* using the operator specified in the *reduction* clause.

The Fortran *end sections* directive must appear to mark the end, because it marks the end of the sequence of sections. Like the *do* construct, there is an implied barrier at the end of the *sections* construct, which may be avoided by adding the *nowait* clause; this clause may be added to the *end sections* directive in Fortran, while in C and C++ it is provided directly with the *omp sections* pragma.

This construct distributes the execution of the different sections among the threads in the parallel team. Each section is executed once, and each thread executes zero or more sections. A thread may execute more than one section if, for example, there are more sections than threads, or if a thread finishes one section before other threads reach the *sections* construct. It is generally not possible to determine whether one section will be executed before another (regardless of which came first in the program's source), or whether two sections will be executed by the same thread. This is because unlike the *do* construct, OpenMP provides no way to control how the different sections are scheduled for execution by the available threads. As a result, the output of one section generally should not serve as the input to another: instead, the section that generates output should be moved before the *sections* construct.

Similar to the combined *parallel do* construct, there is also a combined form of the sections construct that begins with the *parallel sections* directive and ends with the *end parallel sections* directive. The combined form accepts all the clauses that can appear on a *parallel* or *sections* construct.

Let us now examine an example using the *sections* directive. Consider a simulation program that performs several independent preprocessing steps after reading its input data but before performing the simulation. These preprocessing steps are

1. Interpolation of input data from irregularly spaced sensors into a regular grid required for the simulation step
2. Gathering of various statistics about the input data
3. Generation of random parameters for Monte Carlo experiments performed as part of the simulation

In this example we focus on parallelizing the preprocessing steps. Although the work within each is too small to benefit much from parallelism within a step, we can exploit parallelism across the multiple steps. Using the *sections* construct, we can execute all the steps concurrently as distinct sections. This code is presented in Example 4.14.

Example 4.14 Using the *sections* directive.

```
      real sensor_data(3, nsensors), grid(N, N)
      real stats(nstats), params(nparams)
      ...
!$omp parallel sections
      call interpolate(sensor_data, nsensors, &
                       grid, N, N)
!$omp section
      call compute_stats(sensor_data, nsensors, &
                         stats, nstats)
!$omp section
      call gen_random_params(params, nparams)
!$omp end parallel sections
```

Assigning Work to a Single Thread

The *do* and *sections* work-sharing constructs accelerate a computation by splitting it into pieces and apportioning the pieces among a team's threads. Often a parallel region contains tasks that should not be replicated or shared among threads, but instead must be performed just once, by any one of the threads in the team. OpenMP provides the *single* construct to identify these kinds of tasks that must be executed by just one thread.

The general form of the single construct in Fortran is

```
!$omp single [clause [,] [clause ...]]
    block of statements to be executed by just one
    thread
!$omp end single [nowait]
```

In C and C++ it is

```
#pragma omp single [clause [clause] ...]
    block
```

Each *clause* must be a *private* or *firstprivate* scoping clause (in C and C++ it may also be the *nowait* clause). The meaning of these clauses is the same as for a *parallel, do,* or *sections* construct, although only one private copy of each privatized variable needs to be created since only one thread executes the enclosed code. Furthermore, in C/C++ the *nowait* clause, if desired, is provided in the list of clauses supplied with the *omp single* pragma itself.

In Fortran the *end single* directive must be supplied since it marks the end of the single-threaded piece of code. Like all work-sharing constructs, there is an implicit barrier at the end of a *single* unless the *end single* directive includes the *nowait* clause (in C/C++ the *nowait* clause is supplied directly with the *single* pragma). There is no implicit barrier at the start of the *single* construct—if one is needed, it must be provided explicitly in the program. Finally, there is no combined form of the directive because it makes little sense to define a parallel region that must be executed by only one thread.

Example 4.15 illustrates the *single* directive. A common use of *single* is when performing input or output within a parallel region that cannot be successfully parallelized and must be executed sequentially. This is often the case when the input/output operations must be performed in the same strict order as in the serial program. In this situation, although any thread can perform the desired I/O operation, it must be executed by just one thread. In this example we first read some data, then all threads perform some computation on this data in parallel, after which the intermediate results are printed out to a file. The I/O operations are enclosed by the *single* directive, so that one of the threads that has finished the computation performs the I/O operation. The other threads skip around the *single* construct and move on to the code after the *single* directive.

Example 4.15 Using the *single* directive.

```
      integer len
      real in(MAXLEN), out(MAXLEN), scratch(MAXLEN)

      ...
!$omp parallel shared (in, out, len)
       ...
!$omp single
      call read_array(in, len)
!$omp end single

!$omp do private(scratch)
      do j = 1, len
         call compute_result(out(j), in, len, scratch)
      enddo

!$omp single
      call write_array(out, len)
!$omp end single nowait
!$omp end parallel
```

At the beginning of the parallel region a single thread reads the shared input array *in*. The particular thread that performs the *single* section is not specified: an implementation may choose any heuristic, such as the first thread to reach the construct or always select the master thread. Therefore the correctness of the code must not depend on the choice of the particular thread. The remaining threads wait for the *single* construct to finish and the data to be read in at the implicit barrier at the *end single* directive, and then continue execution.

After the array has been read, all the threads compute the elements of the output array *out* in parallel, using a work-sharing *do*. Finally, one thread writes the output to a file. Now the threads do not need to wait for output to complete, so we use the *nowait* clause to avoid synchronizing after writing the output.

The *single* construct is different from other work-sharing constructs in that it does not really divide work among threads, but rather assigns all the work to a single thread. However, we still classify it as a work-sharing construct for several reasons. Each piece of work within a *single* construct is performed by exactly one thread, rather than performed by all threads as is the case with replicated execution. In addition, the *single* construct shares the other characteristics of work-sharing constructs as well: it must be reached by all the threads in a team and each thread must reach all work-sharing constructs (including *single*) in the same order. Finally, the *single* construct also shares the implicit barrier and the *nowait* clause with the other work-sharing constructs.

4.6 Restrictions on Work-Sharing Constructs

There are a few restrictions on the form and use of work-sharing constructs that we have glossed over up to this point. These restrictions involve the syntax of work-sharing constructs, how threads may enter and exit them, and how they may nest within each other.

4.6.1 Block Structure

In the syntax of Fortran executable statements, there is a notion of a *block,* which consists of zero or more complete consecutive statements, each at the same level of nesting. Each of these statements is an assignment, a *call*, or a control construct such as *if* or *do* that contains one or more blocks at a nesting level one deeper. The directives that begin and end an OpenMP work-sharing construct must be placed so that all the executable statements between them form a valid Fortran block.

All the work-sharing examples presented so far follow this rule. For instance, when writing a *do* construct without an *enddo* directive, it is still easy to follow this rule because the *do* loop is a single statement and therefore is also a block.

Code that violates this restriction is shown in Example 4.16. The *single* construct includes only part of the *if* statement, with the result that statement 10 is from a shallower level of nesting than statement 20. Assuming that *b* has shared scope, we can correct this problem by moving the *end single* right after the *end if.*

Example 4.16 Code that violates the block structure requirement.

```
!$omp single
 10   x = 1
      if (z .eq. 3) then
 20      a(1) = 4
!$omp end single
         b(1) = 6
      end if
```

An additional restriction on the block of code within a construct is that it is not permissible to branch into the block from outside the construct, and it is not permissible to branch out of the construct from within the block of code. Therefore no thread may enter or leave the block of statements that make up a work-sharing construct using a control flow construct such as *exit*, *goto*, or *return*. Each thread must instead enter the work-sharing construct "at the top" and leave "out the bottom." However, a *goto* within a construct that transfers control to another statement also within the construct is permitted, since it does not leave the block of code.

4.6.2 Entry and Exit

Because work-sharing constructs divide work among all the threads in a team, it is an OpenMP requirement that all threads participate in each work-sharing construct that is executed (lazy threads are not allowed to shirk their fair share of work). There are three implications of this rule. First, if any thread reaches a work-sharing construct, then all the threads in the team must also reach that construct. Second, whenever a parallel region executes multiple work-sharing constructs, all the threads must reach all the executed work-sharing constructs in the same order. Third, although a region may contain a work-sharing construct, it does not have to be executed, so long as it is skipped by all the threads.

We illustrate these restrictions through some examples. For instance, the code in Example 4.17 is invalid, since thread 0 will not encounter the *do* directive. All threads need to encounter work-sharing constructs.

Example 4.17 Illustrating the restrictions on work-sharing directives.

```
      ...
!$omp parallel private(iam)
      iam = omp_get_thread_num()
      if (iam .ne. 0) then
!$omp do
         do i = 1, n
            ...
         enddo
!$omp enddo
      endif
!$omp end parallel
```

In Example 4.17, we had a case where one of the threads did not encounter the work-sharing directive. It is not enough for all threads to encounter a work-sharing construct either. Threads must encounter the same work-sharing construct. In Example 4.18 all threads encounter a work-sharing construct, but odd-numbered threads encounter a different work-sharing construct than the even-numbered ones. As a result, the code is invalid. It's acceptable for all threads to skip a work-sharing construct though.

Example 4.18 All threads must encounter the same work-sharing contructs.

```
      ...
!$omp parallel private(iam)
      iam = omp_get_thread_num()
      if (mod(iam, 2) .eq. 0) then
!$omp do
         do j = 1, n
            ...
         enddo
      else
!$omp do
         do j = 1, n
            ...
         enddo
      end if
!$omp end parallel
```

In Example 4.19 the *return* statement from the work-shared *do* loop causes an invalid branch out of the block.

Example 4.19 Branching out from a work-sharing construct.

```
      subroutine test(n, a)
      real a(n)
!$omp do
      do i = 1, n
         if(a(i) .lt. 0) return
         a(i) = sqrt(a(i))
      enddo
!$omp enddo
      return
      end
```

Although it is not permitted to branch into or out of a block that is associated with a work-sharing directive, it is possible to branch within the block. In Example 4.20 the *goto* statement is legal since it does not cause a branch out of the block associated with the *do* directive. It is not a good idea to use *goto* statements as in our example. We use it here only to illustrate the branching rules.

Example 4.20 Branching within a work-sharing directive.

```
      subroutine test(n, a)
      real a(n)

!$omp do
      do i = 1, n
         if (a(i) .lt. 0) goto 10
         a(i) = sqrt(a(i))
         goto 20
 10      a(i) = 0
 20      continue
      enddo
      return
      end
```

4.6.3 Nesting of Work-Sharing Constructs

OpenMP does not allow a work-sharing construct to be nested; that is, if a thread, while in the midst of executing a work-sharing construct, encounters another work-sharing construct, then the program behavior is undefined. We illustrate this in Example 4.21. This example violates the nesting requirement since the outermost *do* directive contains an inner *do* directive.

Example 4.21 Program with illegal nesting of work-sharing constructs.

```
!$omp parallel
!$omp do
      do i = 1, M
         ! The following directive is illegal
!$omp do
         do j = 1, N
            ...
         enddo
      enddo
!$omp end parallel
```

The rationale behind this restriction is that a work-sharing construct divides a piece of work among a team of parallel threads. However, once a thread is executing within a work-sharing construct, it is the only thread executing that code (e.g., it may be executing one section of a *sections* construct); there is no team of threads executing that specific piece of code anymore, so it is nonsensical to attempt to further divide a portion of work using a work-sharing construct. Nesting of work-sharing constructs is therefore illegal in OpenMP.

It is possible to parallelize a loop nest such as this such that iterations of both the *i* and *j* loops are executed in parallel. The trick is to add a third, outermost parallel loop that iterates over all the threads (a *static* schedule will ensure that each thread executes precisely one iteration of this loop). Within the body of the outermost loop, we manually divide the iterations of the *i* and *j* loops such that each thread executes a different subset of the *i* and *j* iterations.

Although it is a synchronization rather than work-sharing construct, a *barrier* also requires the participation of all the threads in a team. It is therefore subject to the following rules: either all threads or no thread must reach the barrier; all threads must arrive at multiple *barrier* constructs in the same order; and a barrier cannot be nested within a work-sharing construct. Based on these rules, a *do* directive cannot contain a *barrier* directive.

4.7 Orphaning of Work-Sharing Constructs

All the examples that we have presented so far contain the work-sharing constructs lexically enclosed within the parallel region construct. However, it is easy to imagine situations where this might be rather restrictive, and we may wish to exploit work-sharing within a subroutine called from inside a parallel region.

Example 4.22 Work-sharing outside the lexical scope.

```
      subroutine work
      integer a(N)

!$omp parallel
      call initialize(a, N)
      ...
!$omp end parallel
      ...
      end

      subroutine initialize (a, N)
      integer i, N, a(N)

      ! Iterations of the following do loop may be
      ! executed in parallel
      do i = 1, N
         a(i) = 0
      enddo
      end
```

In Example 4.22 the *work* subroutine contains a parallel region to do some computation in parallel: it first initializes the elements of array *a* and then performs the real computation. In this instance the initialization happens to be performed within a separate subroutine, *initialize*. Although the *do* loop that initializes the array is trivially parallelizable, it is contained outside the lexical scope of the parallel region. Furthermore, it is possible that *initialize* may be called from within the parallel region (as in subroutine *work*) as well as from serial code in other portions of the program.

OpenMP does not restrict work-sharing directives to be within the lexical scope of the parallel region; they can occur within a subroutine that is invoked, either directly or indirectly, from inside a parallel region. Such work-sharing constructs are referred to as *orphaned,* so named because they are no longer enclosed within the lexical scope of the parallel region.

When an orphaned work-sharing construct is encountered from within a parallel region, its behavior is identical (almost) to that of a similar work-sharing construct directly enclosed within the parallel region. The differences in behavior are small and relate to the scoping of variables, and are discussed later in this section. However, the basic behavior in terms of dividing up the enclosed work among the parallel team of threads is the same as that of directives lexically within the parallel region.

We illustrate this by rewriting Example 4.22 to use an orphaned work-sharing construct, as shown in Example 4.23. The only change is the *do* directive attached to the loop in the *initialize* subroutine. With this change the parallel construct creates a team of parallel threads. Each thread invokes the *initialize* subroutine, encounters the *do* directive, and computes a portion of the iterations from the *do i* loop. At the end of the *do* directive, the threads gather at the implicit barrier, and then return to replicated execution with the *work* subroutine. The *do* directive therefore successfully divides the *do* loop iterations across the threads.

Example 4.23 Work-sharing outside the lexical scope.

```
      subroutine work
      integer a(N)

!$omp parallel
      call initialize(a, N)
      ...
!$omp end parallel
      end

      subroutine initialize (a, N)
      integer i, N, a(N)
      ! Iterations of this do loop are
      ! now executed in parallel
!$omp do
      do i = 1, N
         a(i) = 0
      enddo
      end
```

Let us now consider the scenario where the *initialize* subroutine is invoked from a serial portion of the program, leaving the *do* directive exposed without an enclosing parallel region. In this situation OpenMP specifies that the single serial thread behave like a parallel team of threads that consists of only one thread. As a result of this rule, the work-sharing construct assigns all its portions of work to this single thread. In this instance all the iterations of the *do* loop are assigned to the single serial thread, which executes the *do* loop in its entirety before continuing. The behavior of the code is almost as if the directive did not exist—the differences in behavior are small and relate to the data scoping of variables, described in the next section. As a result of this rule, a subroutine containing orphaned work-sharing directives can safely be invoked from serial code, with the directive being essentially ignored.

To summarize, an orphaned work-sharing construct encountered from within a parallel region behaves almost as if it had appeared within the lexical extent of the *parallel* construct. An orphaned work-sharing construct encountered from within the serial portion of the program behaves almost as if the work-sharing directive had not been there at all.

4.7.1 Data Scoping of Orphaned Constructs

Orphaned and nonorphaned work-sharing constructs differ in the way variables are scoped within them. Let us examine their behavior for each variable class. Variables in a *common* block (global variables in C/C++) are *shared* by default in an orphaned work-sharing construct, regardless of the scoping clauses in the enclosing parallel region. Automatic variables in the subroutine containing the orphaned work-sharing construct are always *private,* since each thread executes within its own stack. Automatic variables in the routine containing the parallel region follow the usual scoping rules for a parallel region—that is, *shared* by default unless specified otherwise in a data scoping clause. Formal parameters to the subroutine containing the orphaned construct have their sharing behavior determined by that of the corresponding actual variables in the calling routine's context.

Data scoping for orphaned and non-orphaned constructs is similar in other regards. For instance, the *do* loop index variable is private by default for either case. Furthermore, both kinds of work-sharing constructs disallow the *shared* clause. As a result, a variable that is private in the enclosing context (based on any of the other scoping rules) can no longer be made shared across threads for the work-sharing construct. Finally, both kinds of constructs support the *private* clause, so that any variables that are *shared* in the surrounding context can be made private for the scope of the work-sharing construct.

4.7.2 Writing Code with Orphaned Work-Sharing Constructs

Before we leave orphaned work-sharing constructs, it bears repeating that care must be exercised in using orphaned constructs. OpenMP tries to provide reasonable behavior for orphaned OpenMP constructs regardless of whether the code is invoked from within a serial or parallel region. However, if a subroutine contains an orphaned work-sharing construct, then this property cannot be considered encapsulated within that subroutine. Rather, it must be treated as part of the interface to that subroutine and exposed to the callers of the subroutine.

While subroutines containing orphaned work-sharing constructs behave as expected when invoked from a serial code, they can cause nasty surprises if they are accidentally invoked from within a parallel region. Rather than executing the code within the work-sharing construct in a replicated fashion, this code ends up being divided among multiple threads. Callers of routines with orphaned constructs must therefore be aware of the orphaned constructs in those routines.

4.8 Nested Parallel Regions

We have discussed at length the behavior of work-sharing constructs contained within a parallel region. However, by now you probably want to know what happens in an OpenMP program with *nested parallelism,* where a parallel region is contained within another parallel region.

Parallel regions and nesting are fully orthogonal concepts in OpenMP. The OpenMP programming model allows a program to contain parallel regions nested within other parallel regions (keep in mind that the *parallel do* and the *parallel sections* constructs are shorthand notations for a parallel region containing either the *do* or the *sections* construct). The basic semantics of a parallel region is that it creates a team of threads to execute the block of code contained within the parallel region construct, returning to serial execution at the end of the parallel construct. This behavior is followed regardless of whether the parallel region is encountered from within serial code or from within an outer level of parallelism.

Example 4.24 illustrates nested parallelism. This example consists of a subroutine *taskqueue* that contains a parallel region implementing task-queue-based parallelism, similar to that in Example 4.9. However, in this example we provide the routine to process a task (called *process_task*). The task index passed to this subroutine is a column number in a two-dimensional shared matrix called *grid*. Processing a task for the interior (i.e., nonboundary) columns involves doing some computation on each element of the given column of this matrix, as shown by the *do* loop within the *process_task* subroutine, while the boundary columns need no processing. The *do* loop to process the interior columns is a parallel loop with multiple iterations updating distinct rows of the *myindex* column of the matrix. We can therefore express this additional level of parallelism by providing the *parallel do* directive on the *do* loop within the *process_task* subroutine.

Example 4.24 A program with nested parallelism.

```
      subroutine TaskQueue
      integer myindex, get_next_task

!$omp parallel private (myindex)
      myindex = get_next_task()
      do while (myindex .ne. -1)
         call process_task (myindex)
         myindex = get_next_task()
      enddo
!$omp end parallel
      end

      subroutine process_task (myindex)
      integer myindex
      common /MYCOM/ grid(N, M)

      if (myindex .gt. 1 .AND myindex .lt. M) then
!$omp parallel do
         do i = 1, N
            grid(i, myindex) = ...
         enddo
      endif
      return
      end
```

When this program is executed, it will create a team of threads in the *taskqueue* subroutine, with each thread repeatedly fetching and processing tasks. During the course of processing a task, a thread may encounter the *parallel do* construct (if it is processing an interior column). At this point this thread will create an additional, brand-new team of threads, of which it will be the master, to execute the iterations of the *do* loop. The execution of this *do* loop will proceed in parallel with this newly created team, just like any other parallel region. After the *parallel do* loop is over, this new team will gather at the implicit barrier, and the original thread will return to executing its portion of the code. The slave threads of the now defunct team will become dormant. The nested parallel region therefore simply provides an additional level of parallelism and semantically behaves just like a nonnested parallel region.

It is sometimes tempting to confuse work-sharing constructs with the parallel region construct, so the distinctions between them bear repeating. A parallel construct (including each of the *parallel, parallel do,* and *parallel sections* directives) is a complete, encapsulated construct that attempts to speed up a portion of code through parallel execution. Because it is a self-contained construct, there are no restrictions on where and how often a parallel construct may be encountered.

Work-sharing constructs, on the other hand, are *not* self-contained but instead depend on the surrounding context. They work in tandem with an enclosing parallel region (invocation from serial code is like being invoked from a parallel region but with a single thread). We refer to this as a binding of a work-sharing construct to an enclosing parallel region. This *binding* may be either lexical or dynamic, as is the case with orphaned work-sharing constructs. Furthermore, in the presence of nested parallel constructs, this binding of a work-sharing construct is to the *closest* enclosing parallel region.

To summarize, the behavior of a work-sharing construct depends on the surrounding context; therefore there are restrictions on the usage of work-sharing constructs—for example, all (or none) of the threads must encounter each work-sharing construct. A parallel construct, on the other hand, is fully self-contained and can be used without any such restrictions. For instance, as we show in Example 4.24, only the threads that process an interior column encounter the nested *parallel do* construct.

Let us now consider a parallel region that happens to execute serially, say, due to an *if* clause on the parallel region construct. There is absolutely no effect on the semantics of the parallel construct, and it executes exactly as if in parallel, except on a team consisting of only a single thread rather than multiple threads. We refer to such a region as a *serialized parallel region.* There is no change in the behavior of enclosed work-sharing constructs—they continue to bind to the serialized parallel region as before. With regard to synchronization constructs, the *barrier* construct also binds to the closest dynamically enclosing parallel region and has no effect if invoked from within a serialized parallel region. Synchronization constructs such as *critical* and *atomic* (presented in Chapter 5), on the other hand, synchronize relative to all other threads, not just those in the current team. As a result, these directives continue to function even when invoked from within a serialized parallel region. Overall, therefore, the only perceptible difference due to a serialized parallel region is in the performance of the construct.

Unfortunately there is little reported practical experience with nested parallelism. There is only a limited understanding of the performance and implementation issues with supporting multiple levels of parallelism, and even less experience with the needs of applications programs and its implication for programming models. For now, nested parallelism continues to be an area of active research. Because many of these issues are not well understood, by default OpenMP implementations support nested parallel constructs but serialize the implementation of nested levels of parallelism. As a result, the program behaves correctly but does not benefit from additional degrees of parallelism.

You may change this default behavior by using either the runtime library routine

```
call omp_set_nested (.TRUE.)
```

or the environment variable

```
setenv OMP_NESTED TRUE
```

to enable nested parallelism; you may use the value *false* instead of *true* to disable nested parallelism. In addition, OpenMP also provides a routine to query whether nested parallelism is enabled or disabled:

```
logical function omp_get_nested()
```

As of the date of this writing, however, all OpenMP implementations only support one level of parallelism and serialize the implementation of further nested levels. We expect this to change over time with additional experience.

4.8.1 Directive Nesting and Binding

Having described work-sharing constructs as well as nested parallel regions, we now summarize the OpenMP rules with regard to the nesting and binding of directives.

All the work-sharing constructs (each of the *do, sections,* and *single* directives) bind to the closest enclosing *parallel* directive. In addition, the synchronization constructs *barrier* and *master* (see Chapter 5) also bind to the closest enclosing *parallel* directive. As a result, if the enclosing parallel region is serialized, these directives behave as if executing in parallel with a team of a single thread. If there is no enclosing parallel region currently being executed, then each of these directives has no effect. Other synchronization constructs such as *critical* and *atomic* (see Chapter 5) have a global effect across all threads in all teams, and execute regardless of the enclosing parallel region.

Work-sharing constructs are not allowed to contain other work-sharing constructs. In addition, they are not allowed to contain the *barrier* synchronization construct, either, since the latter makes sense only in a parallel region.

The synchronization constructs *critical*, *master,* and *ordered* (see Chapter 5) are not allowed to contain any work-sharing constructs, since the latter require that either all or none of the threads arrive at each instance of the construct.

Finally, a parallel directive inside another parallel directive logically establishes a new nested parallel team of threads, although current implementations of OpenMP are physically limited to a team size of a single thread.

4.9 Controlling Parallelism in an OpenMP Program

We have thus far focused on specifying parallelism in an OpenMP parallel program. In this section we describe the mechanisms provided in OpenMP for controlling parallel execution during program runtime. We first describe how parallel execution may be controlled at the granularity of an individual parallel construct. Next we describe the OpenMP mechanisms to query and control the degree of parallelism exploited by the program. Finally, we describe a dynamic thread's mechanism that adjusts the degree of parallelism based on the available resources, helping to extract the maximum throughput from a system.

4.9.1 Dynamically Disabling the *parallel* Directives

As we discussed in Section 3.6.1, the choice of whether to execute a piece of code in parallel or serially is often determined by runtime factors such as the amount of work in the parallel region (based on the input data set size, for instance) or whether we chose to go parallel in some other portion of code or not. Rather than requiring the user to create multiple versions of the same code, with one containing parallel directives and the other remaining unchanged, OpenMP instead allows the programmer to supply an optional *if* clause containing a general logical expression with the *parallel* directive. When the program encounters the parallel region at runtime, it first evaluates the logical expression. If it yields the value *true*, then the corresponding parallel region is executed in parallel; otherwise it is executed serially on a team of one thread only.

In addition to the *if* clause, OpenMP provides a runtime library routine to query whether the program is currently executing within a parallel region or not:

```
logical function omp_in_parallel
```

This function returns the value *true* when called from within a parallel region executing in parallel on a team of multiple threads. It returns the value *false* when called from a serial portion of the code or from a serialized parallel region (a parallel region that is executing serially on a team of only one thread). This function is often useful for programmers and library writers who may need to decide whether to use a parallel algorithm or a sequential algorithm based on the parallelism in the surrounding context.

4.9.2 Controlling the Number of Threads

In addition to specifying parallelism, OpenMP programmers may wish to control the size of parallel teams during the execution of their parallel program. The degree of parallelism exploited by an OpenMP program need not be determined until program runtime. Different executions of a program may therefore be run with different numbers of threads. Moreover, OpenMP allows the number of threads to change during the execution of a parallel program as well. We now describe these OpenMP mechanisms to query and control the number of threads used by the program.

OpenMP provides two flavors of control. The first is through an environment variable that may be set to a numerical value:

```
setenv OMP_NUM_THREADS 12
```

If this variable is set when the program is started, then the program will execute using teams of *omp_num_threads* parallel threads (12 in this case) for the parallel constructs.

The environment variable allows us to control the number of threads only at program start-up time, for the duration of the program. To adjust the degree of parallelism at a finer granularity, OpenMP also provides a runtime library routine to change the number of threads during program runtime:

```
call omp_set_num_threads(16)
```

This call sets the desired number of parallel threads during program execution for subsequent parallel regions encountered by the program. This adjustment is not possible while the program is in the middle of executing a parallel region; therefore, this call may only be invoked from the serial portions of the program. There may be multiple calls to this routine in the program, each of which changes the desired number of threads to the newly supplied value.

Example 4.25 Dynamically adjusting the number of threads.

```
      call omp_set_num_threads(64)
!$omp parallel private (iam)
      iam = omp_get_thread_num()
      call workon(iam)
!$omp end parallel
```

In Example 4.25 we ask for 64 threads before the parallel region. This parallel region will therefore execute with 64 threads (or rather, *most likely* execute with 64 threads, depending on whether dynamic threads is enabled or not—see Section 4.9.3). Furthermore, all subsequent parallel regions will also continue to use teams of 64 threads unless this number is changed yet again with another call to *omp_set_num_threads*.

If neither the environment variable nor the runtime library calls are used, then the choice of number of threads is implementation dependent. Systems may then just choose a fixed number of threads or use heuristics such as the number of available processors on the machine.

In addition to controlling the number of threads, OpenMP provides the query routine

```
integer function omp_get_num_threads()
```

This routine returns the number of threads being used in the currently executing parallel team. Consequently, when called from a serial portion or from a serialized parallel region, the routine returns 1.

Since the choice of number of threads is likely to be based on the size of the underlying parallel machine, OpenMP also provides the call

```
integer function omp_get_num_procs()
```

This routine returns the number of processors in the underlying machine available for execution to the parallel program. To use all the available processors on the machine, for instance, the user can make the call

```
omp_set_num_threads(omp_get_num_procs())
```

Even when using a larger number of threads than the number of available processors, or while running on a loaded machine with few available processors, the program will continue to run with the requested number of threads. However, the implementation may choose to map multiple threads in a time-sliced fashion on a single processor, resulting in correct execution but perhaps reduced performance.

4.9.3 Dynamic Threads

In a multiprogrammed environment, parallel machines are often used as shared compute servers, with multiple parallel applications running on the machine at the same time. In this scenario it is possible for all the parallel applications running together to request more processors than are actually available. This situation, termed *oversubscription,* leads to contention for computing resources, causing degradations in both the performance of an individual application as well as in overall system throughput. In this situation, if the number of threads requested by each application could be chosen to match the number of available processors, then the operating system could improve overall system utilization. Unfortunately, the number of available processors is not easily determined by a user; furthermore, this number may change during the course of execution of a program based on other jobs on the system.

To address this issue, OpenMP allows the implementation to automatically adjust the number of active threads to match the number of available processors for that application based on the system load. This feature is called *dynamic threads* within OpenMP. On behalf of the application, the OpenMP runtime implementation can monitor the overall load on the system and determine the number of processors available for the application. The number of parallel threads executing within the application may then be adjusted (i.e., perhaps increased or decreased) to match the number of available processors. With this scheme we can avoid oversubscription of processing resources and thereby deliver good system throughput. Furthermore, this adjustment in the number of active threads is done automatically by the implementation and relieves the programmer from having to worry about coordinating with other jobs on the system.

It is difficult to write a parallel program if parallel threads can choose to join or leave a team in an unpredictable manner. Therefore OpenMP requires that the number of threads be adjusted only during serial portions of the code. Once a parallel construct is encountered and a parallel team has been created, then the size of that parallel team is guaranteed to remain unchanged for the duration of that parallel construct. This allows all the OpenMP work-sharing constructs to work correctly. For manual division of work across threads, the suggested programming style is to query the number of threads upon entry to a parallel region and to use that number for the duration of the parallel region (it is assured of remaining unchanged). Of course, subsequent parallel regions may use a different number of threads.

Finally, if a user wants to be assured of a known number of threads for either a phase or even the entire duration of a parallel program, then this feature may be disabled through either an environment variable or a runtime library call. The environment variable

```
setenv OMP_DYNAMIC {TRUE, FALSE}
```

can be used to enable/disable this feature for the duration of the parallel program. To adjust this feature at a finer granularity during the course of the program (say, for a particular phase), the user can insert a call to the runtime library of the form

```
call omp_set_dynamic ({.TRUE.}, {.FALSE.})
```

The user can also query the current state of dynamic threads with the call

```
logical omp_get_dynamic ()
```

The default—whether dynamic threads is enabled or disabled—is implementation dependent.

We have given a brief overview here of the dynamic threads feature in OpenMP and discuss this issue further in Chapter 6.

4.9.4 Runtime Library Calls and Environment Variables

In this section we give an overview of the runtime library calls and environment variables available in OpenMP to control the execution parameters of a parallel application. Table 4.1 gives a list and a brief description of the library routines in C/C++ and Fortran. The behavior of the routines is the same in all of the languages. Prototypes for the C and C++ OpenMP library calls are available in the include file *omp.h.* Missing from the list here are the library routines that provide lock operations—these are described in Chapter 5.

The behavior of most of these routines is straightforward. The *omp_set_num_threads* routine is used to change the number of threads used for parallel constructs during the execution of the program, perhaps from one parallel construct to another. Since this function changes the number of threads to be used, it must be called from within the serial portion of the program; otherwise its behavior is undefined.

In the presence of dynamic threads, the number of threads used for a parallel construct is determined only when the parallel construct begins executing. The *omp_get_num_threads* call may therefore be used to determine how many threads are being used in the currently executing region; this number will no longer change for the rest of the current parallel construct.

The *omp_get_max_threads* call returns the maximum number of threads that may be used for a parallel construct, independent of the dynamic adjustment in the number of active threads.

The *omp_get_thread_num* routine returns the thread number within the currently executing team. When invoked from either serial code or from within a serialized parallel region, this routine returns 0, since there is only a single executing master thread in the current team.

Finally, the *omp_in_parallel* call returns the value *true* when invoked from within the dynamic extent of a parallel region actually executing in parallel. Serialized parallel regions do not count as executing in parallel. Furthermore, this routine is not affected by nested parallel regions that may have been serialized; if there is an outermore parallel region, then the routine will return *true*.

Table 4.2 lists the environment variables that may be used to control the execution of an OpenMP program. The default values of most of these variables are implementation dependent, except for *OMP_NESTED,* which must be *false* by default. The *OMP_SCHEDULE* environment variable applies only to the *do/for* constructs and to the *parallel do/parallel for* constructs that have the *runtime* schedule type. Similar to the schedule clause,

Table 4.1 Summary of OpenMP runtime library calls in C/C++ and Fortran.

Library Routine (C and C++)	*Library Routine (Fortran)*	*Description*
void omp_set_num_threads (int)	call omp_set_num_threads (integer)	Set the number of threads to use in a team.
int omp_get_num_threads (void)	integer omp_get_num_threads ()	Return the number of threads in the currently executing parallel region. Return 1 if invoked from serial code or a serialized parallel region.
int omp_get_max_threads (void)	integer omp_get_max_threads ()	Return the maximum value that calls to *omp_get_num_threads* may return. This value changes with calls to *omp_set_num_threads.*
int omp_get_thread_num (void)	integer omp_get_thread_num ()	Return the thread number within the team, a value within the inclusive interval 0 and *omp_get_num_threads()* – 1. Master thread is 0.
int omp_get_num_procs (void)	integer omp_get_num_procs ()	Return the number of processors available to the program.
void omp_set_dynamic (int)	call omp_set_dynamic (logical)	Enable/disable dynamic adjustment of number of threads used for a parallel construct.
int omp_get_dynamic (void)	logical omp_get_dynamic ()	Return .TRUE. if dynamic threads is enabled, and .FALSE. otherwise.
int omp_in_parallel (void)	logical omp_in_parallel ()	Return .TRUE. if this function is invoked from within the dynamic extent of a parallel region, and .FALSE. otherwise.
void omp_set_nested (int)	call omp_set_nested (logical)	Enable/disable nested parallelism. If disabled, then nested parallel regions are serialized.
int omp_get_nested (void)	logical omp_get_nested ()	Return .TRUE. if nested parallelism is enabled, and .FALSE. otherwise.

Table 4.2 Summary of the OpenMP environment variables.

Variable	*Example Value*	*Description*
`OMP_SCHEDULE`	`"dynamic, 4"`	Specify the schedule type for parallel loops with a *runtime* schedule.
`OMP_NUM_THREADS`	`16`	Specify the number of threads to use during execution.
`OMP_DYNAMIC`	`TRUE` or `FALSE`	Enable/disable dynamic adjustment of threads.
`OMP_NESTED`	`TRUE` or `FALSE`	Enable/disable nested parallelism.

the environment variable must specify the schedule type and may provide an optional chunk size as shown. These environment variables are examined only when the program begins execution; subsequent modifications, once the program has started executing, are ignored. Finally, the OpenMP runtime parameters controlled by these environment variables may be modified during the execution of the program by invoking the library routines described above.

4.10 Concluding Remarks

In conclusion, it is worthwhile to compare loop-level parallelism exemplified by the *parallel do* directive with SPMD-style parallelism exemplified by the parallel regions construct.

As with any parallel construct, both of these directive sets require that the programmer have a complete understanding of the code contained within the dynamic extent of the parallel construct. The programmer must examine all variable references within the parallel construct and ensure that scope clauses and explicit synchronization constructs are supplied as needed. However, the two styles of parallelism have some basic differences. Loop-level parallelism only requires that iterations of a loop be independent, and executes different iterations concurrently across multiple processors. SPMD-style parallelism, on the other hand, consists of a combination of replicated execution complemented with work-sharing constructs that divide work across threads. In fact, loop-level parallelism is just a special case of the more general parallel region directive containing only a single loop-level work-sharing construct.

Because of the simpler model of parallelism, loop-level parallelism is easier to understand and use, and lends itself well to incremental parallelization one loop at a time. SPMD-style parallelism, on the other hand, has a more complex model of parallelism. Programmers must explicitly concern themselves with the replicated execution of code and identify those code segments where work must instead be divided across the parallel team of threads.

This additional effort can have significant benefits. Whereas loop-level parallelism is limited to loop-level constructs, and that to only one loop at a time, SPMD-style parallelism is more powerful and can be used to parallelize the execution of more general regions of code. As a result it can be used to parallelize coarser-grained, and therefore larger regions of code (compared to loops). Identifying coarser regions of parallel code is essential in parallelizing greater portions of the overall program. Furthermore, it can reduce the overhead of spawning threads and synchronization that is necessary with each parallel construct. As a result, although SPMD-style parallelism is more cumbersome to use, it has the potential to yield better parallel speedups than loop-level parallelism.

4.11 Exercises

1. Parallelize the following recurrence using parallel regions:

```
do i = 2, N
    a(i) = a(i) + a(i - 1)
enddo
```

2. What happens in Example 4.7 if we include *istart* and *iend* in the *private* clause? What storage is now associated with the names *istart* and *iend* inside the parallel region? What about inside the subroutine *work*?
3. Rework Example 4.7 using an orphaned *do* directive and eliminating the *bounds* common block.
4. Do you have to do anything special to Example 4.7 if you put *iarray* in a common block? If you make this common block *threadprivate,* what changes do you have to make to the rest of the program to keep it correct? Does it still make sense for *iarray* to be declared of size 10,000? If not, what size should it be, assuming the program always executes with four threads. How must you change the program now to keep it correct? Why is putting *iarray* in a *threadprivate* common block a silly thing to do?

5. Integer sorting is a class of sorting where the keys are integers. Often the range of possible key values is smaller than the number of keys, in which case the sorting can be very efficiently accomplished using an array of "buckets" in which to count keys. Below is the bucket sort code for sorting the array *key* into a new array, *key2*. Parallelize the code using parallel regions.

```
integer bucket(NUMBUKS), key(NUMKEYS),
key2(NUMKEYS)

do i = 1, NUMBUKS
    bucket(i) = 0
enddo
do i = 1, NUMKEYS
    bucket(key(i)) = bucket(key(i)) + 1
enddo
do i = 2, NUMBUKS
    bucket(i) = bucket(i) + bucket(i - 1)
enddo
do i = 1, NUMKEYS
    key2(bucket(key(i))) = key(i)
    bucket(key(i)) = bucket(key(i)) - 1
enddo
```

This Page Intentionally Left Blank

CHAPTER 5

Synchronization

5.1 Introduction

Shared memory architectures enable implicit communication between threads through references to variables in the globally shared memory. Just as in a regular uniprocessor program, multiple threads in an OpenMP program can communicate through regular read/write operations on variables in the shared address space. The underlying shared memory architecture is responsible for transparently managing all such communication and maintaining a coherent view of memory across all the processors.

Although communication in an OpenMP program is implicit, it is usually necessary to coordinate the access to shared variables by multiple threads to ensure correct execution. This chapter describes the synchronization constructs of OpenMP in detail. It introduces the various kinds of data conflicts that arise in a parallel program and describes how to write a correct OpenMP program in the presence of such data conflicts.

We first motivate the need for synchronization in shared memory parallel programs in Section 5.2. We then present the OpenMP synchronization constructs for mutual exclusion including critical sections, the atomic directive, and the runtime library lock routines, all in Section 5.3. Next we present the synchronization constructs for event synchronization such as barriers and ordered sections in Section 5.4. Finally we describe some of the issues that arise when building custom synchronization using shared memory variables, and how those are addressed in OpenMP, in Section 5.5.

5.2 Data Conflicts and the Need for Synchronization

OpenMP provides a shared memory programming model, with communication between multiple threads trivially expressed through regular read/write operations on variables. However, as Example 5.1 illustrates, these accesses must be coordinated to ensure the integrity of the data in the program.

Example 5.1 Illustrating a data race.

```
      ! Finding the largest element in a list of numbers
      cur_max = MINUS_INFINITY
!$omp parallel do
      do i = 1, n
         if (a(i) .gt. cur_max) then
            cur_max = a(i)
         endif
      enddo
```

Example 5.1 finds the largest element in a list of numbers. The original uniprocessor program is quite straightforward: it stores the current maximum (initialized to *minus_infinity*) and consists of a simple loop that examines all the elements in the list, updating the maximum if the current element is larger than the largest number seen so far.

In parallelizing this piece of code, we simply add a *parallel do* directive to run the loop in parallel, so that each thread examines a portion of the list of numbers. However, as coded in the example, it is possible for the program to yield incorrect results. Let's look at a possible execution trace of the code. For simplicity, assume two threads, P0 and P1. Furthermore, assume that *cur_max* is 10, and the elements being examined by P0 and P1 are 11 and 12, respectively. Example 5.2 is a possible interleaved sequence of instructions executed by each thread.

Example 5.2 Possible execution fragment from Example 5.1.

```
         Thread 0                                       Thread 1
read a(i)    (value = 12)              read a(j)    (value = 11)
read cur_max (value = 10)              read cur_max (value = 10)
if (a(i) > cur_max) (12 > 10)
cur_max = a(i) (i.e. 12)

                                       if (a(j) > cur_max) (11 > 10)
                                       cur_max = a(j) (i.e. 11)
```

As we can see in the execution sequence shown in Example 5.2, although each thread correctly executes its portion of the code, the final value of *cur_max* is incorrectly set to 11. The problem arises because a thread is reading *cur_max* even as some other thread is modifying it. Consequently a thread may (incorrectly) decide to update *cur_max* based upon having read a stale value for that variable. Concurrent accesses to the same variable are called *data races* and must be coordinated to ensure correct results. Such coordination mechanisms are the subject of this chapter.

Although all forms of concurrent accesses are termed data races, synchronization is required only for those data races where at least one of the accesses is a write (i.e., modifies the variable). If all the accesses just read the variable, then they can proceed correctly without any synchronization. This is illustrated in Example 5.3.

Example 5.3 A data race without conflicts: multiple reads.

```
!$omp parallel do shared(a, b)
      do i = 1, n
         a(i) = a(i) + b
      enddo
```

In Example 5.3 the variable *b* is declared *shared* in the parallel loop and read concurrently by multiple threads. However, since *b* is not modified, there is no conflict and therefore no need for synchronization between threads. Furthermore, even though array *a* is being updated by all the threads, there is no data conflict since each thread modifies a distinct element of *a*.

5.2.1 Getting Rid of Data Races

When parallelizing a program, there may be variables that are accessed by all threads but are not used to communicate values between them; rather they are used as scratch storage *within* a thread to compute some values used subsequently within the context of that same thread. Such variables can be safely declared to be private to each thread, thereby avoiding any data race altogether.

Example 5.4 Getting rid of a data race with privatization.

```
!$omp parallel do shared(a) private(b)
      do i = 1, n
         b = func(i, a(i))
         a(i) = a(i) + b
      enddo
```

Example 5.4 is similar to Example 5.3, except that each thread computes the value of the scalar *b* to be added to each element of the array *a*. If *b* is declared *shared,* then there is a data race on *b* since it is both read and modified by multiple threads. However, *b* is used to compute a new value within each iteration, and these values are *not* used across iterations. Therefore *b* is not used to communicate between threads and can safely be marked *private* as shown, thereby getting rid of the data conflict on *b*.

Although this example illustrates the privatization of a scalar variable, this technique can also be applied to more complex data structures such as arrays and structures. Furthermore, in this example we assumed that the value of *b* was not used after the loop. If indeed it is used, then *b* would need to be declared *lastprivate* rather than *private,* and we would still successfully get rid of the data race.

Example 5.5 Getting rid of a data race using reductions.

```
      cur_max = MINUS_INFINITY
!$omp parallel do reduction(MAX:cur_max)
      do i = 1, n
         cur_max = max (a(i), cur_max)
      enddo
```

A variation on this theme is to use reductions. We return to our first example (Example 5.1) to find the largest element in a list of numbers. The parallel version of this code can be viewed as if we first find the largest element within each thread (by only examining the elements within each thread), and then find the largest element across all the threads (by only examining the largest element within each thread computed in the previous step). This is essentially a reduction on *cur_max* using the *max* operator, as shown in Example 5.5. This avoids the data race for *cur_max* since each thread now has a private copy of *cur_max* for computing the local maxima (i.e., within its portion of the list). The subsequent phase, finding the largest element from among each thread's maxima, does require synchronization, but is implemented automatically as part of the *reduction* clause. Thus the *reduction* clause can often be used successfully to overcome data races.

5.2.2 Examples of Acceptable Data Races

The previous examples showed how we could avoid data races altogether in some situations. We now present two examples, each with a data race, but a race that is acceptable and does not require synchronization.

Example 5.6 tries to determine whether a given item occurs within a list of numbers. An item is allowed to occur multiple times within the list—that is, the list may contain duplicates. Furthermore, we are only interested in a *true/false* answer, depending on whether the item is found or not.

Example 5.6 An acceptable data race: multiple writers that write the same value.

```
      foundit = .FALSE.
!$omp parallel do
      do i = 1, n
         if (a(i) .eq. item) then
            foundit = .TRUE.
         endif
      enddo
```

The code segment in this example consists of a *do* loop that scans the list of numbers, searching for the given item. Within each iteration, if the item is found, then a global flag, *foundit*, is set to the value *true*. We execute the loop in parallel using the *parallel do* directive. As a result, iterations of the *do* loop run in parallel and the *foundit* variable may potentially be modified in multiple iterations. However, the code contains no synchronization and allows multiple threads to update *foundit* simultaneously. Since all the updates to *foundit* write the *same* value into the variable, the final value of *foundit* will be *true* even in the presence of multiple updates. If, of course, the item is not found, then no iteration will update the *foundit* variable and it will remain *false,* which is exactly the desired behavior.

Example 5.7 presents a five-point SOR (successive over relaxation) kernel. This algorithm contains a two-dimensional matrix of elements, where each element in the matrix is updated by taking its average along with the four nearest elements in the grid. This update step is performed for each element in the grid and is then repeated over a succession of time steps until the values converge based on some physical constraints within the algorithm.

Example 5.7 An acceptable data race: multiple writers, but a relaxed algorithm.

```
!$omp parallel do
      do j = 2, n - 1
         do i = 2, n - 1
            a(i, j) = (a(i, j) + a(i, j - 1) + a(i, j + 1) &
                       + a(i - 1, j) + a(i + 1, j))/5
         enddo
      enddo
```

The code shown in Example 5.7 consists of a nest of two *do* loops that scan all the elements in the grid, computing the average of each element as the code proceeds. With the *parallel do* directive as shown, we execute the *do j* loop in parallel, thereby processing the columns of the grid concurrently across multiple threads. However, this code contains a data dependence from one iteration of the *do j* loop to another. While iteration *j* computes *a(i,j)*, iteration *j* + 1 is using this same value of *a(i,j)* in computing the average for the value at *a(i,j + 1)*. As a result, this memory location may simultaneously be read in one iteration while it is being modified in another iteration.

Given the loop-carried data dependence in this example, the parallel code contains a data race and may behave differently from the original, serial version of the code. However, this code can provide acceptable behavior in spite of this data race. As a consequence of this data race, it is possible that an iteration may occasionally use an older value of *a(i,j – 1)* or *a(i,j + 1)* from a preceding or succeeding column, rather than the latest value. However, the algorithm is tolerant of such errors and will still converge to the desired solution within an acceptable number of time-step iterations. This class of algorithms are called *relaxed algorithms,* where the parallel version does not implement the strict constraints of the original serial code, but instead carefully relaxes some of the constraints to exploit parallelism. As a result, this code can also successfully run in parallel without any synchronization and benefit from additional speedup.

A final word of caution about this example. We have assumed that at any time *a(i,j)* either has an old value or a new value. This assumption is in turn based on the implicit assumption that *a(i,j)* is of primitive type such as a floating-point number. However, imagine a scenario where instead the type of *a(i,j)* is no longer primitive but instead is complex or structured. It is then possible that while *a(i,j)* is in the process of being updated, it has neither an old nor a new value, but rather a value that is somewhere in between. While one iteration updates the new value of *a(i,j)* through multiple write operations, another thread in a different iteration may be reading *a(i,j)* and obtaining a mix of old and new partial values, violating our assumption above. Tricks such as this, therefore, must be used carefully with an understanding of the granularity of primitive write operations in the underlying machine.

5.2.3 Synchronization Mechanisms in OpenMP

Having understood the basics of data races and some ways of avoiding them, we now describe the synchronization mechanisms provided in

OpenMP. Synchronization mechanisms can be broadly classified into two categories: mutual exclusion and event synchronization.

Mutual exclusion synchronization is typically used to acquire exclusive access to shared data structures. When multiple threads need to access the same data structure, then mutual exclusion synchronization can be used to guarantee that (1) only one thread has access to the data structure for the duration of the synchronization construct, and (2) accesses by multiple threads are interleaved at the granularity of the synchronization construct.

Event synchronization is used to signal the completion of some event from one thread to another. For instance, although mutual exclusion provides exclusive access to a shared object, it does not provide any control over the *order* in which threads access the shared object. Any desired ordering between threads is implemented using event synchronization constructs.

5.3 Mutual Exclusion Synchronization

OpenMP provides three different constructs for mutual exclusion. Although the basic functionality provided is similar, the constructs are designed *hierarchically* and range from restrictive but easy to use at one extreme, to powerful but requiring more skill on the part of the programmer at the other extreme. We present these constructs in order of increasing power and complexity.

5.3.1 The Critical Section Directive

The general form of the critical section in Fortran is

```
!$omp critical [(name)]
    block
!$omp end critical [(name)]
```

In C and C++ it is

```
#pragma omp critical [(name)]
    block
```

In this section we describe the basic critical section construct. We discuss critical sections with the optional *name* argument in the subsequent section.

We illustrate critical sections using our earlier example to find the largest element in a list of numbers. This example as originally written

(Example 5.1) had a data conflict on the variable *cur_max*, with multiple threads potentially reading and modifying the variable. This data conflict can be addressed by adding synchronization to acquire exclusive access while referencing the *cur_max* variable. This is shown in Example 5.8.

In Example 5.8 the critical section construct consists of a matching begin/end directive pair that encloses a structured block of code. Upon encountering the *critical* directive, a thread waits until no other thread is executing within a critical section, at which point it is granted exclusive access. At the *end critical* directive, the thread surrenders its exclusive access, signals a waiting thread (if any), and resumes execution past the end of the critical section. This code will now execute correctly, since only one thread can examine/update *cur_max* at any one time.

Example 5.8 Using a critical section.

```
      cur_max = MINUS_INFINITY
!$omp parallel do
      do i = 1, n
!$omp critical
         if (a(i) .gt. cur_max) then
            cur_max = a(i)
         endif
!$omp end critical
      enddo
```

The critical section directive provides mutual exclusion relative to *all* critical section directives in the program—that is, only one critical section is allowed to execute at one time anywhere in the program. Conceptually this is equivalent to a global lock in the program.

The begin/end pair of directives must enclose a structured block of code, so that a thread is assured of encountering both the directives in order. Furthermore, it is illegal to branch into or jump out of a critical section, since either of those would clearly violate the desired semantics of the critical section.

There is no guarantee of fairness in granting access to a critical section. In this context, "fairness" refers to the assurance that when multiple threads request the critical section, then they are granted access to the critical section in some equitable fashion, such as in the order in which requests are made (first come, first served) or in a predetermined order such as round-robin. OpenMP does not provide any such guarantee. However, OpenMP does guarantee forward progress, that is, at least one of the waiting threads will get access to the critical section. Therefore, in the presence of contention, while it is possible for a thread to be starved while other threads get repeated access to a critical section, an eligible thread will always be assured of acquiring access to the critical section.

Refining the Example

The code in Example 5.8 with the added synchronization will now execute correctly, but it no longer exploits any parallelism. With the critical section as shown, the execution of this parallel loop is effectively serialized since there is no longer any overlap in the work performed in different iterations of the loop. However, this need not be so. We now refine the code to also exploit parallelism.

Example 5.9 Using a critical section.

```
      cur_max = MINUS_INFINITY
!$omp parallel do
      do i = 1, n
         if (a(i) .gt. cur_max) then
!$omp critical
            if (a(i) .gt. cur_max) then
               cur_max = a(i)
            endif
!$omp end critical
         endif
      enddo
```

The new code in Example 5.9 is based on three key observations. The first is that the value of *cur_max* increases *monotonically* during the execution of the loop. Therefore, if an element of the list is less than the present value of *cur_max*, it is going to be less than all subsequent values of *cur_max* during the loop. The second observation is that in general most iterations of the loop only examine *cur_max*, but do not actually update it (of course, this is not always true—for instance, imagine a list that is sorted in increasing order). Finally, we are assuming that each element of the array is a scalar type that can be written atomically by the underlying architecture (e.g., a 4-byte or 8-byte integer or floating-point number). This ensures that when a thread examines *cur_max*, it reads either an old value or a new value, but never a value that is partly old and partly new, and therefore an invalid value for *cur_max* (such might be the case with a number of type complex, for instance). Based on these observations, we have rewritten the code to first examine *cur_max* without any synchronization. If the present value of *cur_max* is already greater than *a(i)*, then we need proceed no further. Otherwise *a(i)* may be the largest element seen so far, and we enter the critical section. Once inside the critical section we examine *cur_max* again. This is necessary because we previously examined *cur_max* outside the critical section so it could have changed in the meantime. Examining it again within the critical section guarantees the thread exclusive access to *cur_max*, and an update based on the fresh comparison is assured of being correct. Furthermore, this new code will (hopefully) enter the critical section only infrequently and benefit from increased parallelism.

A preliminary test such as this before actually executing the synchronization construct is often helpful in parallel programs to avoid unnecessary overhead and exploit additional parallelism.

Named Critical Sections

The critical section directive presented above provides *global* synchronization—that is, only one critical section can execute at one time, anywhere in the program. Although this is a simple model, it can be overly restrictive when we need to protect access to distinct data structures. For instance, if a thread is inside a critical section updating an object, it unnecessarily prevents another thread from entering another critical section to update a different object, thereby reducing the parallelism exploited in the program. OpenMP therefore allows critical sections to be *named,* with the following behavior: a named critical section must synchronize with other critical sections of the same name but can execute concurrently with critical sections of a different name, and unnamed critical sections synchronize only with other unnamed critical sections.

Conceptually, if an unnamed critical section is like a global lock, named critical sections are like lock variables, with distinct names behaving like different lock variables. Distinct critical sections are therefore easily specified by simply providing a name with the critical section directive. The names for critical sections are in a distinct namespace from program variables and therefore do not conflict with those names, but they are in the same global namespace as subroutine names or names of *common* blocks (in Fortran). Conflicts with the latter lead to undefined behavior and should be avoided.

Example 5.10 Using named critical sections.

```
      cur_max = MINUS_INFINITY
      cur_min = PLUS_INFINITY
!$omp parallel do
      do i = 1, n

         if (a(i) .gt. cur_max) then
!$omp critical (MAXLOCK)
            if (a(i) .gt. cur_max) then
               cur_max = a(i)
            endif
!$omp end critical (MAXLOCK)
         endif

         if (a(i) .lt. cur_min) then
!$omp critical (MINLOCK)
            if (a(i) .lt. cur_min) then
               cur_min = a(i)
            endif
!$omp end critical (MINLOCK)
         endif
      enddo
```

Example 5.10 illustrates how named critical sections may be used. This extended example finds both the smallest and the largest element in a list of numbers, and therefore updates both a *cur_min* along with a *cur_max* variable. Each of these variables must be updated within a critical section, but using a distinct critical section for each enables us to exploit additional parallelism in the program. This is easily achieved using named critical sections.

Nesting Critical Sections

Nesting of critical sections—that is, trying to enter one critical section while already executing within another critical section—must be done carefully! OpenMP does not provide any protection against deadlock, a condition where threads are waiting for synchronization events that will never happen. A common scenario is one where a thread executing within a critical section invokes a subroutine that acquires a critical section of the same name (or both critical sections could be unnamed), leading to deadlock. Since the nested critical section is in another subroutine, this nesting is not immediately apparent from an examination of the code, making such errors harder to catch.

Why doesn't OpenMP allow a thread to enter a critical section if it has already acquired exclusive access to that critical section? Although this feature is easy to support, it incurs additional execution overhead, adding to the cost of entering and leaving a critical section. Furthermore, this overhead must be incurred at nearly all critical sections (only some simple cases could be optimized), regardless of whether the program actually contains nested critical sections or not. Since most programs tend to use simple synchronization mechanisms and do not contain nested critical sections, this overhead is entirely unnecessary in the vast majority of programs. The OpenMP designers therefore chose to optimize for the common case, leaving the burden of handling nested critical sections to the programmer. There is one concession for nested mutual exclusion: along with the regular version of the runtime library lock routines (described in Section 5.3.3), OpenMP provides a nested version of these same routines as well.

If the program must nest critical sections, be careful that all threads try to acquire the multiple critical sections in the same order. For instance, suppose a program has two critical sections named *min* and *max*, and threads need exclusive access to both critical sections simultaneously. All threads must acquire the two critical sections in the same order (either *min* followed by *max*, or *max* followed by *min*). Otherwise it is possible that a thread has entered *min* and is waiting to enter *max*, while another thread has entered *max* and is trying to enter *min*. Neither thread will be able to succeed in acquiring the second critical section since it is already held by the other thread, leading to deadlock.

5.3.2 The *atomic* Directive

Most modern shared memory multiprocessors provide hardware support for atomically updating a single location in memory. This hardware support typically consists of special machine instructions, such as the load-linked store-conditional (LL/SC) pair of instructions on the MIPS processor or the compare-and-exchange (CMPXCHG) instruction on the Intel x86 processors, that allow the processor to perform a read, modify, and write operation (such as an increment) on a memory location, all in an atomic fashion. These instructions use hardware support to acquire exclusive access to this single location for the duration of the update. Since the synchronization is integrated with the update of the shared location, these primitives avoid having to acquire and release a separate lock or critical section, resulting in higher performance.

The *atomic* directive is designed to give an OpenMP programmer access to these efficient primitives in a portable fashion. Like the *critical* directive, the *atomic* directive is just another way of expressing mutual exclusion and does not provide any additional functionality. Rather, it comes with a set of restrictions that allow the directive to be implemented using the hardware synchronization primitives. The first set of restrictions applies to the form of the critical section. While the *critical* directive encloses an arbitrary block of code, the *atomic* directive can be applied only if the critical section consists of a single assignment statement that updates a scalar variable. This assignment statement must be of one of the following forms (including their commutative variations):

```
!$omp atomic
      x = x operator expr
```

```
          ...
!$omp atomic
          x = intrinsic (x, expr)
```

where *x* is a scalar variable of intrinsic type, *operator* is one of a set of predefined operators (including most arithmetic and logical operators), *intrinsic* is one of a set of predefined intrinsics (including min, max, and logical intrinsics), and *expr* is a scalar expression that does not reference *x*. Table 5.1 provides the complete list of operators in the language. The corresponding syntax in C and C++ is

```
#pragma omp atomic
x < binop >= expr
...
#pragma omp atomic
/* One of */
x++, ++x, x--, or --x
```

As you might guess, these restrictions on the form of the *atomic* directive ensure that the assignment can be translated into a sequence of machine instructions to atomically read, modify, and write the memory location for *x*.

The second set of restrictions concerns the synchronization used for different accesses to this location in the program. The user must ensure that all conflicting accesses to this location are consistently protected using the same synchronization directive (either atomic or critical). Since the atomic and critical directive use quite different implementation mechanisms, they cannot be used simultaneously and still yield correct results. Nonconflicting accesses, on the other hand, can safely use different synchronization directives since only one mechanism is active at any one time. Therefore, in a program with distinct, nonoverlapping phases it is permissible for one phase to use the *atomic* directive for a particular variable and for the other phase to use the *critical* directive for protecting accesses to the same variable. Within a phase, however, we must consistently use either one or the other directive for a particular variable. And, of course, different mechanisms can be used for different variables within

Table 5.1 List of accepted operators for the *atomic* directive.

Language	*Operators*
Fortran	+, *, -, /, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, IOR, IEOR
C/C++	+, *, -, /, &, ^, |, <<, >>

the same phase of the program. This also explains why this transformation cannot be performed automatically (i.e., examining the code within a critical construct and implementing it using the synchronization hardware) and requires explicit user participation.

Finally, the *atomic* construct provides exclusive access only to the location being updated (e.g., the variable *x* above). References to variables and side effect expressions within *expr* must avoid data conflicts to yield correct results.

We illustrate the *atomic* construct with Example 5.11, building a histogram. Imagine we have a large list of numbers where each number takes a value between 1 and 100, and we need to count the number of instances of each possible value. Since the location being updated is an integer, we can use the *atomic* directive to ensure that updates of each element of the histogram are atomic. Furthermore, since the *atomic* directive synchronizes using the location being updated, multiple updates to different locations can proceed concurrently,[1] exploiting additional parallelism.

Example 5.11 Building a histogram using the *atomic* directive.

```
      integer histogram(100)

!$omp parallel do
      do i = 1, n
!$omp atomic
         histogram(a(i)) = histogram(a(i)) + 1
      enddo
```

How should a programmer evaluate the performance trade-offs between the *critical* and the *atomic* directives (assuming, of course, that both are applicable)? A critical section containing a single assignment statement is usually more efficient with the *atomic* directive, and never worse. However, a critical section with multiple assignment statements cannot be transformed to use a single *atomic* directive, but rather requires an *atomic* directive for each statement within the critical section (assuming that each statement meets the criteria listed above for atomic, of course). The performance trade-offs for this transformation are nontrivial—a single *critical* has the advantage of incurring the overhead for just a single synchronization construct, while multiple *atomic* directives have the benefits of (1) exploiting additional overlap and (2) smaller overhead for each synchronization construct. A general guideline is to use the *atomic* directive when updating

[1] If multiple shared variables lie within a cache line, then performance can be adversely affected. This phenomenon is called *false sharing* and is discussed further in Chapter 6.

either a single location or a few locations, and to prefer the *critical* directive when updating several locations.

5.3.3 Runtime Library Lock Routines

In addition to the *critical* and *atomic* directives, OpenMP provides a set of lock routines within a runtime library. These routines are listed in Table 5.2 (for Fortran and C/C++) and perform the usual set of operations on locks such as acquire, release, or test a lock, as well as allocation and deallocation. Example 5.12 illustrates the familiar code to find the largest element using these library routines rather than a critical section.

Example 5.12 Using the lock routines.

```
      real*8 maxlock
      call omp_init_lock(maxlock)

      cur_max = MINUS_INFINITY
!$omp parallel do
      do i = 1, n
         if (a(i) .gt. cur_max) then
            call omp_set_lock(maxlock)
            if (a(i) .gt. cur_max) then
               cur_max = a(i)
            endif
            call omp_unset_lock(maxlock)
         endif
      enddo
      call omp_destroy_lock(maxlock)
```

Lock routines are another mechanism for mutual exclusion, but provide greater flexibility in their use as compared to the *critical* and *atomic* directives. First, unlike the *critical* directive, the set/unset subroutine calls do not have to be block-structured. The user can place these calls at arbitrary points in the code (including in different subroutines) and is responsible for matching the invocations of the set/unset routines. Second, each lock subroutine takes a lock variable as an argument. In Fortran this variable must be sized large enough to hold an address[2] (e.g., on 64-bit systems the variable must be sized to be 64 bits), while in C and C++ it must be the address of a location of the predefined type *omp_lock_t* (in C and

[2] This allows an implementation to treat the lock variable as a pointer to lock data structures allocated and maintained within an OpenMP implementation.

Table 5.2 List of runtime library routines for lock access.

Language	*Routine Name*	*Description*

Fortran
C/C++
	`omp_init_lock(var)`
	`void omp_init_lock(omp_lock_t*)`
		Allocate and initialize the lock.

Fortran
C/C++
	`omp_destroy_lock(var)`
	`void omp_destroy_lock(omp_lock_t*)`
		Deallocate and free the lock.

Fortran
C/C++
	`omp_set_lock(var)`
	`void omp_set_lock(omp_lock_t*)`
		Acquire the lock, waiting until it becomes available, if necessary.

Fortran
C/C++
	`omp_unset_lock(var)`
	`void omp_unset_lock(omp_lock_t*)`
		Release the lock, resuming a waiting thread (if any).

Fortran
C/C++
	`logical omp_test_lock(var)`
	`int omp_test_lock(omp_lock_t*)`
		Try to acquire the lock, return success (*true*) or failure (*false*).

C++ this type, and in fact the prototypes of all the OpenMP runtime library functions, may be found in the standard OpenMP include file *omp.h*). The actual lock variable can therefore be determined dynamically, including being passed around as a parameter to other subroutines. In contrast, even named critical sections are determined statically based on the name supplied with the directive. Finally, the *omp_test_lock(. . .)* routine provides the ability to write nondeterministic code—for instance, it enables a thread to do other useful work while waiting for a lock to become available.

Attempting to reacquire a lock already held by a thread results in deadlock, similar to the behavior of nested critical sections discussed in Section 5.3.1. However, support for nested locks is sometimes useful: For instance, consider a recursive subroutine that must execute with mutual exclusion. This routine would deadlock with either critical sections or the lock routines, although it could, in principle, execute correctly without violating any of the program requirements.

OpenMP therefore provides another set of library routines that may be nested—that is, reacquired by the same thread without deadlock. The interface to these routines is very similar to the interface for the regular routines, with the additional keyword *nest,* as shown in Table 5.3.

These routines behave similarly to the regular routines, with the difference that they support nesting. Therefore an *omp_set_nest_lock* operation that finds that the lock is already set will check whether the lock is

Table 5.3 List of runtime library routines for nested lock access.

Language	*Subroutine*	*Description*

Fortran
C/C++
	`omp_init_nest_lock (var)`
	`void omp_init_nest_lock(omp_lock_t*)`
		Allocate and initialize the lock.

Fortran
C/C++
	`omp_destroy_nest_lock (var)`
	`void omp_destroy_nest_lock(omp_lock_t*)`
		Deallocate and free the lock.

Fortran
C/C++
	`omp_set_nest_lock (var)`
	`void omp_set_nest_lock(omp_lock_t*)`
		Acquire the lock, waiting until it becomes available, if necessary. If lock is already held by the same thread, then access is automatically assured.

Fortran
C/C++
	`omp_unset_nest_lock (var)`
	`void omp_unset_nest_lock(omp_lock_t*)`
		Release the lock. If this thread no longer has a pending lock acquire, then resume a waiting thread (if any).

Fortran
C/C++
	`logical omp_test_nest_lock (var)`
	`int omp_test_nest_lock(omp_lock_t*)`
		Try to acquire the lock, return success (*true*) or failure (*false*). If this thread already holds the lock, then acquisition is assured, and return *true*.

held by the requesting thread. If so, then the *set* operation is successful and continues execution. Otherwise, of course, it waits for the thread that holds the lock to exit the critical section.

5.4 Event Synchronization

As we discussed earlier in the chapter, the constructs for mutual exclusion provide exclusive access but do not impose any order in which the critical sections are executed by different threads. We now turn to the OpenMP constructs for ordering the execution between threads—these include barriers, ordered sections, and the *master* directive.

5.4.1 Barriers

The *barrier* directive in OpenMP provides classical barrier synchronization between a group of threads. Upon encountering the *barrier* directive, each thread waits for all the other threads to arrive at the barrier.

Only when all the threads have arrived do they continue execution past the *barrier* directive.

The general form of the *barrier* directive in Fortran is

```
!$omp barrier
```

In C and C++ it is

```
#pragma omp barrier
```

Barriers are used to synchronize the execution of multiple threads within a parallel region. A barrier directive ensures that all the code occurring before the barrier has been completed by all the threads, before any thread can execute any of the code past the *barrier* directive. This is a simple directive that can be used to ensure that a piece of work (e.g., a computational phase) has been completed before moving on to the next phase.

We illustrate the *barrier* directive in Example 5.13. The first portion of code within the parallel region has all the threads generating and adding work items to a list, until there are no more work items to be generated. Work items are represented by an index in this example. In the second phase, each thread fetches and processes work items from this list, again until all the items have been processed. To ensure that the first phase is complete before the start of the second phase, we add a *barrier* directive at the end of the first phase. This ensures that all threads have finished generating all their tasks before any thread moves on to actually processing the tasks.

A barrier in OpenMP synchronizes *all* the threads executing within the parallel region. Therefore, like the other work-sharing constructs, all threads executing the parallel region must execute the *barrier* directive, otherwise the program will deadlock. In addition, the *barrier* directive synchronizes only the threads within the current parallel region, not any threads that may be executing within other parallel regions (other parallel regions are possible with nested parallelism, for instance). If a parallel region gets serialized, then it effectively executes with a team containing a single thread, so the barrier is trivially complete when this thread arrives at the barrier. Furthermore, a barrier cannot be invoked from within a work-sharing construct (synchronizing an arbitrary set of iterations of a parallel loop would be nonsensical); rather it must be invoked from within a parallel region as shown in Example 5.13. It may, of course, be orphaned. Finally, work-sharing constructs such as the *do* construct or the *sections* construct have an implicit barrier at the end of the construct, thereby ensuring that the work being divided among the threads has completed before moving on to the next piece of work. This implicit barrier may be overridden by supplying the *nowait* clause with the end directive of the work-sharing construct (e.g., *end do* or *end sections*).

Example 5.13 Illustrating the *barrier* directive.

```
!$omp parallel private(index)
      index = Generate_Next_Index()
      do while (index .ne. 0)
         call Add_Index (index)
         index = Generate_Next_Index()
      enddo

      ! Wait for all the indices to be generated
!$omp barrier
      index = Get_Next_Index()
      do while (index .ne. 0)
         call Process_Index (index)
         index = Get_Next_Index()
      enddo
!$omp end parallel
```

5.4.2 Ordered Sections

The ordered section directive is used to impose an order across the iterations of a parallel loop. As described earlier, iterations of a parallel loop are assumed to be independent of each other and execute concurrently without synchronization. With the ordered section directive, however, we can identify a portion of code within each loop iteration that must be executed in the original, sequential order of the loop iterations. Instances of this portion of code from different iterations execute in the same order, one after the other, as they would have executed if the loop had not been parallelized. These sections are only ordered relative to each other and execute concurrently with other code outside the ordered section.

The general form of an ordered section in Fortran is

```
!$omp ordered
    block
!$omp end ordered
```

In C and C++ it is

```
$pragma omp ordered
    block
```

Example 5.14 illustrates the usage of the ordered section directive. It consists of a parallel loop where each iteration performs some complex computation and then prints out the value of the array element. Although the core computation is fully parallelizable across iterations of the loop, it is necessary that the output to the file be performed in the original, sequential order. This is accomplished by wrapping the code to do the output within the ordered section directive. With this directive, the computation across multiple iterations is fully overlapped, but before entering the ordered section each thread waits for the ordered section from the previous iteration of the loop to be completed. This ensures that the output to the file is maintained in the original order. Furthermore, if most of the time within the loop is spent computing the necessary values, then this example will exploit substantial parallelism as well.

Example 5.14 Using an ordered section.

```
!$omp parallel do ordered
      do i = 1, n
         a(i) = ... complex calculations here ...

         ! wait until the previous iteration has
         ! finished its ordered section
!$omp ordered
         print *, a(i)
         ! signal the completion of ordered
         !from this iteration
!$omp end ordered
      enddo
```

The *ordered* directive may be *orphaned*—that is, it does not have to occur within the lexical scope of the parallel loop; rather, it can be encountered anywhere within the dynamic extent of the loop. Furthermore, the *ordered* directive may be combined with any of the different schedule types on a parallel *do* loop. In the interest of efficiency, however, the *ordered* directive has the following two restrictions on its usage. First, if a parallel loop contains an *ordered* directive, then the parallel loop directive itself must contain the *ordered* clause. A parallel loop with an *ordered* clause does not have to encounter an ordered section. This restriction enables the implementation to incur the overhead of the *ordered* directive only when it is used, rather than for all parallel loops. Second, an iteration of a parallel loop is allowed to encounter at most one ordered section (it can safely encounter no ordered section). Encountering more than one ordered section will result in undefined behavior. Taken together, these restrictions allow OpenMP to provide both a simple and efficient model for ordering a portion of the iterations of a parallel loop in their original serial order.

5.4.3 The *master* Directive

A parallel region in OpenMP is a construct with SPMD-style execution—that is, all threads execute the code contained within the parallel region in a replicated fashion. In this scenario, the OpenMP *master* construct can be used to identify a block of code within the parallel region that must be executed by the master thread of the executing parallel team of threads.

The precise form of the *master* construct in Fortran is

```
!$omp master
    block
!$omp end master
```

In C and C++ it is

```
#pragma omp master
    block
```

The code contained within the *master* construct is executed only by the master thread in the team. This construct is distinct from the other work-sharing constructs presented in Chapter 4—all threads are not required to reach this construct, and there is no implicit barrier at the end of the construct. If another thread encounters the construct, then it simply skips past this block of code onto the subsequent code.

Example 5.15 illustrates the *master* directive. In this example all threads perform some computation in parallel, after which the intermediate results must be printed out. We use the *master* directive to ensure that the I/O operations are performed serially, by the master thread

Example 5.15 Using the *master* directive.

```
!$omp parallel

!$omp do
      do i = 1, n
         ... perform computation ...
      enddo

!$omp master
      print *, intermediate_results
!$omp end master

      ... continue next computation phase ...
!$omp end parallel
```

This construct may be used to restrict I/O operations to the master thread only, to access the master's copy of threadprivate variables, or perhaps just as a more efficient instance of the *single* directive.

5.5 Custom Synchronization: Rolling Your Own

As we have described in this chapter, OpenMP provides a wide variety of synchronization constructs. In addition, since OpenMP provides a shared memory programming model, it is possible for programmers to build their own synchronization constructs using ordinary load/store references to shared memory locations. We now discuss some of the issues in crafting such custom synchronization and how they are addressed within OpenMP.

We illustrate the issues with a simple producer/consumer-style application, where a producer thread repeatedly produces a data item and signals its availability, and a consumer thread waits for a data value and then consumes it. This coordination between the producer and consumer threads can easily be expressed in a shared memory program without using any OpenMP construct, as shown in Example 5.16.

Example 5.16 A producer/consumer example.

```
Producer Thread                Consumer Thread
data = ...
flag = 1                       do while (flag .eq. 0)
                               ... = data
```

Although the code as written is basically correct, it assumes that the various read/write operations to the memory locations *data* and *flag* are performed in strictly the same order as coded. Unfortunately this assumption is all too easily violated on modern systems—for instance, a compiler may allocate either one (or both) of the variables in a register, thereby delaying the update to the memory location. Another possibility is that the compiler may reorder either the modifications to *data* and *flag* in the producer, or conversely reorder the reads of *data* and *flag* in the consumer. Finally, the architecture itself may cause the updates to *data* and *flag* to be observed in a different order by the consumer thread due to reordering of memory transactions in the memory interconnect. Any of these factors can cause the consumer to observe the memory operations in the wrong order, leading to incorrect results (or worse yet, deadlock).

While it may seem that the transformations being performed by the compiler/architecture are incorrect, these transformations are commonplace in modern computer systems—they cause no such correctness problems in sequential (i.e., nonparallel) programs and are absolutely essential to obtaining high performance in those programs. Even for parallel programs the vast majority of code portions do not contain synchronization through memory references and can safely be optimized using the transformations above. For a detailed discussion of these issues, see [AG 96] and [KY 95]. Given the importance of these transformations to performance, the approach taken in OpenMP is to selectively identify the code portions that synchronize directly through shared memory, thereby disabling these troublesome optimizations for those code portions.

5.5.1 The *flush* Directive

OpenMP provides the *flush* directive, which has the following form:

```
!$omp flush [(list)]                (in Fortran)
#pragma omp flush [(list)]          (in C and C++)
```

where list is an optional list of variable names.

The *flush* directive in OpenMP may be used to identify a synchronization point in the program. We define a synchronization point as a point in the execution of the program where the executing thread needs to have a *consistent* view of memory. A consistent view of memory has two requirements. The first requirement is that all memory operations, including read and write operations, that occur before the *flush* directive in the source program must be performed *before* the synchronization point in the executing thread. In a similar fashion, the second requirement is that all memory operations that occur after the *flush* directive in the source code must be performed only *after* the synchronization point in the executing thread. Taken together, a synchronization point imposes a strict order upon the memory operations within the executing thread: at a synchronization point all previous read/write operations must have been performed, and none of the subsequent memory operations should have been initiated. This behavior of a synchronization point is often called a *memory fence,* since it inhibits the movement of memory operations across that point.

Based on these requirements, an implementation (i.e., the combination of compiler and processor/architecture) cannot retain a variable in a register/hardware buffer across a synchronization point. If the variable is modified by the thread before the synchronization point, then the updated value must be written out to memory before the synchronization point; this will make the updated value visible to other threads. For instance, a compiler must restore a modified value from a register to memory, and hardware must flush write buffers, if any. Similarly, if the variable is read by the thread after the synchronization point, then it must be retrieved from memory before the first use of that variable past the synchronization point; this will ensure that the subsequent read fetches the latest value of the variable, which may have changed due to an update by another thread.

At a synchronization point, the compiler/architecture is required to flush only those shared variables that might be accessible by another thread. This requirement does not apply to variables that cannot be accessed by another thread, since those variables could not be used for communication/synchronization between threads. A compiler, for instance, can often prove that an automatic variable cannot be accessed by any other thread. It may then safely allocate that variable in a register across the synchronization point identified by a *flush* directive.

By default, a *flush* directive applies to all variables that could potentially be accessed by another thread. However, the user can also choose to provide an optional list of variables with the *flush* directive. In this scenario, rather than applying to *all* shared variables, the *flush* directive instead behaves like a memory fence for only the variables named in the *flush* directive. By carefully naming just the necessary variables in the *flush* directive, the programmer can choose to have the memory fence for those variables while simultaneously allowing the optimization of other shared variables, thereby potentially gaining additional performance.

The *flush* directive makes only the executing thread's view of memory consistent with global shared memory. To achieve a globally consistent view across all threads, each thread must execute a *flush* operation.

Finally, the *flush* directive does not, by itself, perform any synchronization. It only provides memory consistency between the executing thread and global memory, and must be used in combination with other read/write operations to implement synchronization between threads.

Let us now illustrate how the *flush* directive would be used in the previous producer/consumer example. For correct execution the program in Example 5.17 requires that the producer first make all the updates to *data,* and then set the *flag* variable. The first *flush* directive in the producer ensures that all updates to *data* are flushed to memory before touching *flag.* The second *flush* ensures that the update of *flag* is actually flushed to memory rather than, for instance, being allocated into a register for the rest of the subprogram. On the consumer side, the consumer must keep reading the memory location for *flag*—this is ensured by the first *flush* directive, requiring *flag* to be reread in each iteration of the *while* loop. Finally, the second *flush* assures us that any references to data are made only after the correct value of *flag* has been read. Together, these constraints provide a correctly running program that can, at the same time, be highly optimized in code portions without synchronization, as well as in code portions contained between synchronization (i.e., flush) points in the program.

Example 5.17 A producer/consumer example using the *flush* directive.

```
Producer Thread                     Consumer Thread
    data = ...
!$omp flush (data)
    flag = 1                        do
!$omp flush (flag)                  !$omp flush (flag)
                                    while (flag .eq. 0)
                                !$omp flush (data)
                                     ... = data
```

Experienced programmers reading the preceding description about the *flush* directive have probably encountered this problem in other shared memory programming models. Unfortunately this issue has either been ignored or addressed in an ad hoc fashion in the past. Programmers have resorted to tricks such as inserting a subroutine call (perhaps to a dummy subroutine) at the desired synchronization point, hoping to thereby prevent compiler optimizations across that point. Another trick is to declare some of the shared variables as volatile and hope that a reference to a volatile variable at a synchronization point would also inhibit the movement of all other memory operations across that synchronization point. Unfortunately these tricks are not very reliable. For instance, modern compilers may inline a called subroutine or perform interprocedural analyses that enable optimizations across a call. In a similar vein, the interpretation of the semantics of volatile variable references and their effect on other, nonvolatile variable references has unfortunately been found to vary from one compiler to another. As a result these tricks are generally unreliable and not portable from one platform to another (sometimes not even from one compiler release to the next). With the *flush* directive, therefore, OpenMP provides a standard and portable way of writing custom synchronization through regular shared memory operations.

5.6 Some Practical Considerations

We now discuss a few practical issues that arise related to synchronization in parallel programs. The first issue concerns the behavior of library routines in the presence of parallel execution, the second has to do with waiting for a synchronization request to be satisfied, while the last issue examines the impact of cache lines on the performance of synchronization constructs.

Most programs invoke library facilities—whether to manage dynamically allocated storage through *malloc/free* operations, to perform I/O and file operations, or to call mathematical routines such as a random number generator or a transcendental function. Questions then arise: What assumptions can the programmer make about the behavior of these library facilities in the presence of parallel execution? What happens when these routines are invoked from within a parallel piece of code, so that multiple threads may be invoking multiple instances of the same routine concurrently?

The desired behavior, clearly, is that library routines continue to work correctly in the presence of parallel execution. The most natural behavior for concurrent library calls is to behave as if they were invoked one at a time, although in some nondeterministic order that may vary from one execution to the next. For instance, the desired behavior for *malloc* and *free* is to continue to allocate and deallocate storage, maintaining the integrity of the heap in the presence of concurrent calls. Similarly, the desired behavior for I/O operations is to perform the I/O in such a fashion that an individual I/O request is perceived to execute *atomically,* although multiple requests may be interleaved in some order. For other routines, such as a random number generator, it may even be sufficient to simply generate a random number on a per-thread basis rather than coordinating the random number generation across multiple threads.

Such routines that continue to function correctly even when invoked concurrently by multiple threads are called *thread-safe* routines. Although most routines don't start out being thread-safe, they can usually be made thread-safe through a variety of schemes. Routines that are "pure"—that is, those that do not contain any state but simply compute and return a value based on the value of the input parameters—are usually automatically thread-safe since there is no contention for any shared resources across multiple invocations. Most other routines that do share some state (such as I/O and *malloc/free*) can usually be made thread-safe by adding a trivial lock/unlock pair around the entire subroutine (to be safe, we would preferably add a *nest_lock/nest_unlock*). The lock/unlock essentially makes the routine single-threaded—that is, only one invocation can execute at any one time, thereby maintaining the integrity of each individual call. Finally, for some routines we can often rework the underlying algorithm to enable multiple invocations to execute concurrently and provide the desired behavior; we suggested one such candidate in the random number generator routine.

Although thread safety is highly desirable, the designers of OpenMP felt that although they could prescribe the behavior of OpenMP constructs, they could not dictate the behavior of all library routines on a system.

Thread safety, therefore, has been left as a vendor-specific issue. It is the hope and indeed the expectation that over time libraries on most vendors will become thread-safe and function correctly. Meanwhile, you the programmer will have to determine the behavior provided on your favorite platform.

The second issue is concerned with the underlying implementation, rather than the semantics of OpenMP itself. This issue relates to the mechanism used by a thread to wait for a synchronization event. What does a thread do when it needs to wait for an event such as a lock to become available or for other threads to arrive at a barrier? There are several implementation options in this situation. At one extreme the thread could simply busy-wait for the synchronization event to occur, perhaps by spinning on a flag in shared memory. Although this option will inform the thread nearly immediately once the synchronization is signaled, the waiting thread can adversely affect the performance of other processes on the system. For instance, it will consume processor cycles that could have been usefully employed by another thread, and it may cause contention for system resources such as the bus, network, and/or the system memory. At the other extreme the waiting thread could immediately surrender the underlying processor to some other runnable process and block, waiting to resume execution only when the synchronization event has been signaled. This option avoids the waste in system resources, but may incur some delay before the waiting thread is informed of the synchronization event. Furthermore, it will incur the context switch overhead in blocking and then unblocking the waiting thread. Finally, there is a range of intermediate options as well, such as busy-waiting for a while followed by blocking, or busy-waiting combined with an exponential back-off scheme.

This is clearly an implementation issue, so that the OpenMP programmer need never be concerned about it, particularly from a correctness or semantic perspective. It does have some impact on performance, so the advanced programmer would do well to be aware of these implementation issues on their favorite platform.

Finally, we briefly mention the impact of the underlying cache on the performance of synchronization constructs. Since all synchronization mechanisms fundamentally coordinate the execution of multiple threads, they require the communication of variables between multiple threads. At the same time, cache-based machines communicate data between processors at the granularity of a cache line, typically 64 to 128 bytes in size. If multiple variables happen to be located within the same cache line, then accesses to one variable will cause all the other variables to also be moved around from one processor to another. This phenomenon is called *false* *sharing* and is discussed in greater detail in Chapter 6. For now it is sufficient to mention that since synchronization variables are likely to be heavily communicated, we would do well to be aware of this accidental sharing and avoid it by padding heavily communicated variables out to a cache line.

5.7 Concluding Remarks

This chapter gave a detailed description of the various synchronization constructs in OpenMP. Although OpenMP provides a range of such constructs, there is a deliberate attempt to provide a *hierarchy* of mechanisms. The goal in OpenMP has been to make it easy to express simple synchronization requirements—for instance, a critical section or a barrier requires only a single directive, yet at the same time also provides powerful mechanisms for more complex situations—such as the *atomic* and *flush* directives. Taken together, this hierarchy of control tries to provide flexible constructs for the needs of complex situations without cluttering the constructs required by the more common and simpler programs.

5.8 Exercises

1. Rewrite Example 5.8 using the *atomic* directive instead of critical sections.
2. Rewrite Example 5.8 using locks instead of critical sections.
3. Most modern microprocessors include special hardware to test and modify a variable in a single instruction cycle. In this manner a shared variable can be modified without interrupt, and this forms the basis of most compiler implementations of critical sections. It is, however, possible to implement critical sections for two threads in software simply using the *flush* directive, a status array (either "locked" or "unlocked" for a thread), and a "turn" variable (to store the thread id whose turn it is to enter the critical section). Rewrite Example 5.8 using a software implementation of critical sections for two threads. How would you modify your code to accommodate named critical sections?
4. Generalize your software implementation of critical sections from Exercise 3 to arbitrary numbers of threads. You will have to add another status besides locked and unlocked.

5. Consider the following highly contrived example of nested, unnamed critical sections (the example computes a sum reduction and histograms the sum as it gets generated):

```
integer sum, a(N), histogram(m)
!$omp parallel do
do i = 1, N
    !$omp critical
    sum = sum + a(i)
    call histo(sum, histogram)
    !$omp end critical
enddo
end

subroutine histo(sum, histogram)
integer sum, histogram(m)
!$omp critical
histogram(sum) = histogram(sum) + 1
!$omp end critical
return
end
```

As you know, nesting of unnamed (or same-name) critical sections is not allowed in OpenMP. Rewrite this example using your own software implementation for critical sections (see Exercise 3). Did you have to change your implementation in any way to still make it work correctly? Can you think of a different, less contrived situation where you might want to nest critical sections of the same name?

This Page Intentionally Left Blank

CHAPTER 6

Performance

6.1 Introduction

The previous chapters in this book have concentrated on explaining the OpenMP programming language. By now, you should be proficient in writing parallel programs using the different constructs available in the language. This chapter takes a moment to step back and ask why we would want to write a parallel program. Writing a program in OpenMP does not offer esthetic advantages over writing serial programs. It does not provide algorithmic capabilities that are not already available in straight Fortran, C, or C++. The reason to program in OpenMP is performance. We want to utilize the power of multiple processors to solve problems more quickly.

Some problems lend themselves naturally to a programming style that will run efficiently on multiple processors. Assuming access to a good compiler and sufficiently large data sets, it is difficult, for example, to program a matrix multiply routine that will not run well in parallel. On the other hand, other problems are easy to code in ways that will run even slower in parallel than the original serial code. Modern machines are quite complicated, and parallel programming adds an extra dimension of complexity to coding.

In this chapter, we attempt to give an overview of factors that affect parallel performance in general and also an overview of modern machine characteristics that affect parallelism. The variety of parallel machines developed over the years is quite large. The performance characteristics of a vector machine such as the SGI Cray T90 is very different from the performance characteristics of a bus-based multiprocessor such as the Compaq ProLiant 7000, and both are very different from a multithreaded machine such as the Tera MTA. Tuning a program for one of these machines might not help, and might even hinder, performance on another. Tuning for all possible machines is beyond the scope of this chapter.

Although many styles of machines still exist, in recent years a large percentage of commercially available shared memory multiprocessors have shared fundamental characteristics. In particular, they have utilized standard microprocessors connected together via some type of network, and they have contained caches close to the processor to minimize the time spent accessing memory. PC-derived machines, such as those available from Compaq, as well as workstation-derived machines, such as those available from Compaq, HP, IBM, SGI, and Sun, all share these fundamental characteristics. This chapter will concentrate on getting performance on these types of machines. Although the differences between them can be large, the issues that affect performance on each one are remarkably similar. In order to give concrete examples with real-world numbers, we will choose one machine, the SGI Origin 2000, to generate sample numbers. While the exact numbers will differ, other cache-based microprocessor machines will have similar characteristics. The Origin 2000 we use contains sixteen 195 Mhz MIPS R10000 microprocessors, each containing a 32 KB on-chip data cache and a 4 MB external cache. All codes were compiled with version 7.2.1 of the MIPSPro Fortran compilers.

A few core issues dominate parallel performance: coverage, granularity, load balancing, locality, and synchronization. Coverage is the percentage of a program that is parallel. Granularity refers to how much work is in each parallel region. Load balancing refers to how evenly balanced the work load is among the different processors. Locality and synchronization refer to the cost to communicate information between different processors on the underlying system. The next section (Section 6.2) will cover these key issues. In order to understand locality and synchronization, some knowledge of the machine architecture is required. We will by necessity, then, be forced to digress a bit into a discussion of caches.

Once we have covered the core issues, Section 6.3 will discuss methodologies for performance tuning of application codes. That section will discuss both tools and general guidelines on how to approach the problem of tuning a particular code. Finally we will have two sections covering more advanced topics: dynamic threads in Section 6.4 and NUMA (Non-Uniform Memory Access) considerations in Section 6.5.

6.2 Key Factors That Impact Performance

The key attributes that affect parallel performance are coverage, granularity, load balancing, locality, and synchronization. The first three are fundamental to parallel programming on any type of machine. The concepts are also fairly straightforward to understand. The latter two are highly tied in with the type of hardware we are considering—cache-based microprocessor systems. Their effects are often more surprising and harder to understand, and their impact can be huge.

6.2.1 Coverage and Granularity

In order to get good performance on a parallel code, it is necessary to parallelize a sufficiently large portion of the code. This is a fairly obvious concept, but what is less obvious and what is perhaps even counterintuitive is that as the number of processors is increased, the performance of the application can become dominated by the serial portions of the program, even when those portions were relatively unimportant in the serial execution of the program. This idea is captured in Amdahl's law, named after the computer architect Gene Amdahl. If F is the fraction of the code that is parallelized and S_p is the speedup achieved in the parallel sections of the code, the overall speedup S is given by

$$S = \frac{1}{(1-F) + \frac{F}{S_p}}$$

The formula can easily be derived as follows. In a serial execution, the program will execute in T_s time. In the parallel execution, the serial portion $(1 - F)$ of the code will execute in time $(1 - F)T_s$, while the parallel portion (F) will execute in time

$$\frac{FT_s}{S_p}$$

Adding these two numbers together gives us a parallel execution time of

$$(1-F)T_s + \frac{FT_s}{S_p}$$

Dividing the serial execution time by the parallel execution time gives us Amdahl's law.

The key insight in Amdahl's law is that no matter how successfully we parallelize the parallel portion of a code and no matter how many processors we use, eventually the performance of the application will be completely limited by *F*, the proportion of the code that we are able to parallelize. If, for example, we are only able to parallelize code for half of the application's runtime, the overall speedup can never be better than two because no matter how fast the parallel portion runs, the program will spend half the original serial time in the serial portion of the parallel code. For small numbers of processors, Amdahl's law has a moderate effect, but as the number of processors increase, the effect becomes surprisingly large. Consider the case where we wish to achieve within 10% of linear speedup, and the parallel portion of the code does speed up linearly. With two processors, plugging in the formula for the case that the overall speedup is 1.8, we get that

$$1.8 = \frac{1}{\left((1-F)+\frac{F}{2}\right)}$$

or $F = 89\%$. Doing the same calculation for 10 processors and an overall speedup of nine, we discover that $F = 99\%$. Portions of the code that took a meaningless 1% of the execution time when run on one processor become extremely relevant when run on 10 processors.

Through Amdahl's law, we have shown that it is critical to parallelize the large majority of a program. This is the concept of coverage. High coverage by itself is not sufficient to guarantee good performance. Granularity is another issue that effects performance. Every time the program invokes a parallel region or loop, it incurs a certain overhead for going parallel. Work must be handed off to the slaves and, assuming the *nowait* clause was not used, all the threads must execute a barrier at the end of the parallel region or loop. If the coverage is perfect, but the program invokes a very large number of very small parallel loops, then performance might be limited by granularity. The exact cost of invoking a parallel loop is actually quite complicated. In addition to the costs of invoking the parallel loop and executing the barrier, cache and synchronization effects can greatly increase the cost. These two effects will be discussed in Sections 6.2.3 and 6.2.4. In this section, we are concerned with the minimum cost to invoke parallelism, ignoring these effects. We therefore measured the overhead to invoke an empty *parallel do* (a *parallel do* loop containing no work—i.e., an empty body; see Example 6.1) given different numbers of processors on the SGI Origin 2000. The time, in cycles, is given in Table 6.1.

Table 6.1 Time in cycles for empty *parallel do*.

Processors	*Cycles*
1	1800
2	2400
4	2900
8	4000
16	8000

Example 6.1 An empty *parallel do* loop.

```
!$omp parallel do
 do ii = 1, 16
 enddo
!$omp end parallel do
```

In general, one should not parallelize a loop or region unless it takes significantly more time to execute than the parallel overhead. Therefore it may be worthwhile to determine the corresponding numbers for your specific platform.

Making these type of measurements brings to mind once again the issue of loop-level parallelism versus domain decomposition. Can we improve the parallel overhead by doing a coarse-grained parallel region? To check, we measured the amount of time needed to perform just an empty *!$omp do,* contained within an outer parallel region (Example 6.2). The time, in cycles, is given in Table 6.2. The *!$omp do* scales much better than the *!$omp parallel do*. So, we can significantly decrease our overhead by using the coarse-grained approach.

Example 6.2 An empty *!$omp do*.

```
!$omp do
 do j = 1, 16
 enddo
!$omp enddo
```

6.2.2 Load Balance

A chain is only as strong as its weakest link. The analog in parallel processing is that a parallel loop (assuming no *nowait*) will not complete until the last processor completes its iterations. If some processors have

Table 6.2 Time in cycles for empty *!$omp do*.

Processors	*Cycles*
1	2200
2	1700
4	1700
8	1800
16	2900

more work to do than other processors, performance will suffer. As previously discussed, OpenMP allows iterations of loops to be divided among the different processors in either a static or dynamic scheme (based on the discussion in Chapter 3, guided self-scheduling, or GSS, is really a special form of dynamic). With static schemes, iterations are divided among the processors at the start of the parallel region. The division is only dependent on the number of iterations and the number of processors; the amount of time spent in each iteration does not affect the division. If the program spends an equal amount of time in every iteration, both static and dynamic schemes will exhibit good load balancing. Simple domain decompositions using non-sparse algorithms often have these properties. In contrast, two classes of codes might suffer from load balancing problems: codes where the amount of work is dependent on the data and codes with triangular loops.

Consider the problem of scaling the elements of a sparse matrix—that is, multiplying each element by a constant. Assume the matrix is stored as an array of pointers to row vectors. Each row vector is an array of column positions and values. A code fragment to execute this algorithm is given in Example 6.3. Since the code manipulates pointers to arrays, it is more easily expressed in C++.

Example 6.3 Scaling a sparse matrix.

```
struct SPARSE_MATRIX {
  double *element; /* actual elements of array */
  int    *index;   /* column position of elements */
  int    num_cols; /* number of columns in this row */
} *rows;
int num_rows;

...
for (i = 0; i<num_rows; i++) {
    SPARSE_MATRIX tmp = rows[i];
```

```
        for (j = 0; j < tmp.num_cols; j++) {
            tmp.element[j] = c * tmp.element[j];
        }
    }
```

A straightforward parallelization scheme would put a *parallel for* pragma on the outer, *i*, loop. If the data is uniformly distributed, a simple static schedule might perform well, but if the data is not uniformly distributed, if some rows have many more points than others, load balancing can be a problem with static schedules. One thread might get many more points than another, and it might then finish much later than the other. For these types of codes, the load balancing problem can be solved by using a dynamic schedule. With a schedule type of *(dynamic,1)*, each thread will take one row at a time and will only get another row when it finishes the current row. When the first thread finishes all its work, the slowest thread can have at most one row left to process. As long as a single row does not have a significant fraction of the total work, all the threads will finish in about the same time.

You might ask why not always use a dynamic scheduling algorithm. If load balancing is the most important contributor to performance in the algorithm, perhaps we should, but there are also costs to using dynamic schedules. The first cost is a synchronization cost. With a *(dynamic,1)*, each thread must go to the OpenMP runtime system after *each* iteration and ask for another iteration to execute. The system must take care to hand out different iterations to each processor. That requires synchronization among the processors. We will discuss synchronization in Section 6.2.4. The amount of synchronization can be alleviated by using a larger chunk size in the dynamic clause. Rather than handing out work one iteration at a time, we can hand out work several iterations at a time. By choosing a sufficiently large chunk size, we can eliminate most or all of the synchronization cost. The trade-off is that as we increase the chunk size, we may get back the load balancing problem. In an extreme case, we could choose the chunk size to be *num_rows/P*, where *P* is the number of threads. We would have the same work distribution as in the static case and therefore the same load balancing problem. The hope is that there is a happy medium—a chunk size large enough to minimize synchronization yet small enough to preserve load balancing. Whether such a point exists depends on the particular piece of code.

The second downside to using dynamic schedules is data locality. We will discuss this issue in more detail in the next section. For now, keep in mind that locality of data references may be an issue depending on the algorithm. If locality is an issue, there is a good chance that the problem cannot be overcome by using larger chunk sizes. Locality versus load balancing is perhaps the most important trade-off in parallel programming. Managing this trade-off can be the key to achieving good performance.

The sparse matrix example is a case where the amount of work per iteration of a loop varies in an unpredictable, data-dependent manner. There are also examples where the amount of work varies but varies in a predictable manner. Consider the simple example of scaling a dense, triangular matrix in Example 6.4.

Example 6.4 Scaling a dense, triangular matrix.

```
for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
      a[i][j] = c * a[i][j];
    }
}
```

We could parallelize this loop by adding a *!$omp parallel for* pragma to the *i* loop. Each iteration has a different amount of work, but the amount of work varies regularly. Each successive iteration has a linearly decreasing amount of work. If we use a static schedule without a chunk size, we will have a load balance problem. The first thread will get almost twice as much work as the average thread. As with the sparse matrix example, we could use a dynamic schedule. This could solve the load balancing problem but create synchronization and locality problems. For this example, though, we do not need a dynamic schedule to avoid load balancing problems. We could use a static schedule with a relatively small chunk size. This type of schedule will hand off iterations to the threads in a round-robin, interleaved manner. As long as *n* is not too small relative to the number of threads, each thread will get almost the same amount of work. Since the iterations are partitioned statically, there are no synchronization costs involved. As we shall discuss in the next section, there are potential locality issues, but they are likely to be much smaller than the locality problems with dynamic schedules.

Based on the previous discussion, we can see that a static schedule can usually achieve good load distribution in situations where the amount of work per iteration is uniform or if it varies in a predictable fashion as in the triangular loop above. When the work per iteration varies in an unpredictable manner, then a dynamic or GSS schedule is likely to achieve the best load balance. However, we have implicitly assumed that all threads arrive at the parallel loop at the same time, and therefore distributing the work uniformly is the best approach. If the parallel loop is instead preceded by a *do* or a *sections* construct with a *nowait* clause, then the threads may arrive at the *do* construct at different times. In this scenario a static distribution may not be the best approach even for regular loops, since it would assign each thread the same amount of work, and the threads that finish their portion of the iterations would simply have to wait for the other threads to arrive and catch up. Instead, it may be preferable to use a dynamic or a GSS schedule—this way the earliest arriving threads would have an opportunity to keep processing some of the pending loop iterations if some other threads are delayed in the previous construct, thereby speeding up the execution of the parallel loop.

We have thus far focused on distributing the work uniformly across threads so that we achieve maximal utilization of the available threads and thereby minimize the loop completion time. In the next section we address the other crucial factor affecting the desired loop schedule—namely, data locality.

6.2.3 Locality

On modern cache-based machines, locality is often the most critical factor affecting performance. This is the case with single-processor codes; it is even more true with multiprocessor codes. To understand how locality affects performance, we need to have some basic understanding of how caches work. We give a basic overview of caches in the following subsection. For a detailed discussion of these issues, see [HP 90]. If you already understand caches, you can safely skip the subsection.

Caches

The programmer can view a computer as a system comprising processing elements and memory. Given a Fortran statement *a(i) = b(i) + c(i)*, the program can view the system as bringing in from memory to the processor the values present in locations *b(i)* and c(i), computing the sum inside the processor, and returning the result to the memory location reserved for *a(i)*. Programmers might like to view memory as monolithic. They would like there to be no fundamental difference between the memory location for one variable, *a(i),* and another, *b(i).* On uniprocessor systems (we will discuss multiprocessors a bit later in this subsection), from a correctness point of view, memory really is monolithic. From a performance point of view, though, memory is not monolithic. It might take more time to bring *a(i)* from memory into the processor than it does to bring in *b(i),* and bringing in *a(i)* at one point in time of the program's execution might take longer than bringing it in at a later point in time.

Two factors lead to the design of machines with varying memory latencies. The first factor is that it is both cheaper and more feasible to build faster memory if that memory is small. Multiple technologies exist for building memory. Faster technologies tend to be more expensive (presumably one can also devise slow and expensive technology, but no one will use it). That means that for a given budget, we can pay for a small amount of faster memory and a larger amount of slower memory. We cannot replace the slower memory with the faster one without raising costs. Even ignoring costs, it is technologically more feasible to make accesses to a smaller memory faster than accesses to a larger one. Part of the time required to bring data from memory into a processor is the time required to find the data, and part of the time is the time required to move the data. Data moves using electrical signals that travel at a finite velocity. That means that the closer a piece of data is to the processor, the faster it can be brought into the processor. There is a limit to how much space exists within any given distance from the processor. With microprocessors, memory that can be put on the same chip as the processor is significantly faster to access than memory that is farther away on some other chip, and there is a limit to how much memory can be put on the processor chip.

The fact that we can build smaller amounts of faster memory would not necessarily imply that we should. Let's say, for example, that we can build a system with 10 times as much slow memory that is 10 times less expensive. Let's say that memory references are distributed randomly between the fast and slow memory. Since there is more slow memory, 90% of the references will be to the slow memory. Even if the fast memory is infinitely fast, we will only speed up the program by about 10% (this percentage can be derived using another variation of Amdahl's law). The costs, on the other hand, have increased by a factor of two. It is unlikely that this trade-off is worthwhile.

The second factor that leads to the design of machines with varying memory latencies, and that makes it worthwhile to build small but fast memories, is the concept of locality. Locality is the property that if a program accesses a piece of memory, there is a much higher than random probability that it will again access the same location "soon." A second aspect of locality is that if a program accesses a piece of memory, there is a much higher than random probability that it will access a nearby location soon. The first type of locality is called *temporal locality;* the second is called *spatial.* Locality is not a law of nature. It is perfectly possible to write a program that has completely random or systematically nonlocal memory behavior. Locality is just an empirical rule. Many programs naturally have locality, and many others will be written to have locality if the prevalent machines are built to take advantage of locality.

Let us consider some examples that illustrate locality. Let us say that we wish to zero a large array. We can zero the elements in any order, but the natural way to zero them is in lexical order, that is, first *a(0),* then *a(1),* then *a(2),* and so on. Zeroing an array this way exhibits spatial locality. Now let's assume that we want to apply some type of convolution filter to an image (e.g., a simple convolution may update each element to be a weighted average of its neighboring values in a grid). Each element in the image will be touched a number of times equal to the size of the filter. If the filter is moved lexically through the image, all accesses to a given point will occur close in time. This type of algorithm has both spatial and temporal locality.

The prevalence of locality allows us to design a small and fast memory system that can greatly impact performance. Consider again the case where 10% of the memory is fast and 90% is slow. If accesses are random and only 10% of the memory accesses are to the fast memory, the cost of the fast memory is probably too large to be worthwhile. Caches, on the other hand, provide a way to exploit locality to insure that a much larger percentage of your memory references are to faster memory.

Caches essentially work as follows. All information is kept in the larger slower memory. Every time the system brings a memory reference into the processor, it "caches" a copy of the data in the smaller, faster memory, the cache. For simplicity, we will use the generic term "memory" to refer to the slower, bigger memory and "cache" to refer to the faster, smaller memory. Before performing a memory reference, the system checks to see if the data is inside the cache. If the data is found, the system does not go to memory; it just retrieves the copy of the data found in the cache. Only when the data is not in the cache does the system go to memory. Since the cache is smaller than the memory, it cannot contain a copy of all the data in the memory. In other words, the cache can fill up. In such cases, the cache must remove some other data in order to make room for the new data.

Whenever the user writes some data (i.e., assigns some variable), the system has two options. In a write-through cache, the data is immediately written back to memory. In such a system, memory and cache are always consistent; the cache always contains the same data found in the memory. Write-through caches are not as bad as they might seem. No computation depends on a write completing so the system can continue to compute while the write to memory is occurring. On the other hand, write-through caches do require moving large amounts of data from the processor to memory. This increases the system overhead and makes it harder to load other data from memory concurrently.

The alternative to a write-through cache is a write-back cache. In write-back caches, the system does not write the data to memory immediately. The cache is not kept consistent with memory. Instead, a bit is added to each entry in the cache indicating whether the memory is dirty, that is, different from the value in main memory. When the system evicts an entry from the cache (in order to make room for other data), the system checks the bit. If it is dirty, main memory is updated at that point. The advantage of the write-back scheme is that a processor can potentially write the same location multiple times before having to write a value back to memory. In today's systems, most caches that are not on the same chip as the processor are write-back caches. For on-chip caches, different systems use write-back or write-through.

Let us look in more detail at caches by considering an example of a simple cache. Assume that memory is referenced via 32-bit addresses. Each address refers to a byte in memory. That allows for up to 4 GB of memory. Assume a 64 KB cache organized as 8192 entries of 8 bytes (1 double word) each. One way to build the cache is to make a table as in Figure 6.1.

We use bits 3 through 15 of a memory address to index into the table, and use the three bits 0 through 2 to access the byte within the 8 bytes in that table entry. Each address maps into only one location in the cache.[1] Each entry in the cache contains the 8 bytes of data, 1 bit to say whether the entry is dirty, and a 16-bit tag. Many addresses map into the same location in the cache, in fact all the addresses with the same value for bits 3 through 15. The system needs some way of knowing which address is present in a given cache entry. The 16-bit tag contains the upper 16 bits of the address present in the given cache entry. These 16 bits, along with the 13 bits needed to select the table entry and the 3 bits to select the byte within a table entry, completely specify the original 32-bit address. Every time the processor makes a memory reference, the system takes bits 3 through 15 of the address to index into the cache. Given the cache entry, it compares the tag with the upper 16 bits of the memory address. If they match, we say that there is a "cache hit." Bits 0 through 2 are used to determine the location within the cache entry. The data is taken from cache, and memory is never accessed. If they do not match, we say there is a "cache miss." The system goes to memory to find the data. That data replaces the data currently present in the entry of the cache. If the current data is dirty, it must first be written out to memory. If it is not dirty, the system can simply throw it out. This cache allows us to exploit temporal

[1] Such caches are called *direct mapped caches.* In contrast, *set associative caches* allow one memory address to map into more than one table entry. For simplicity, we will stick to direct mapped caches although most real-world caches are set associative.

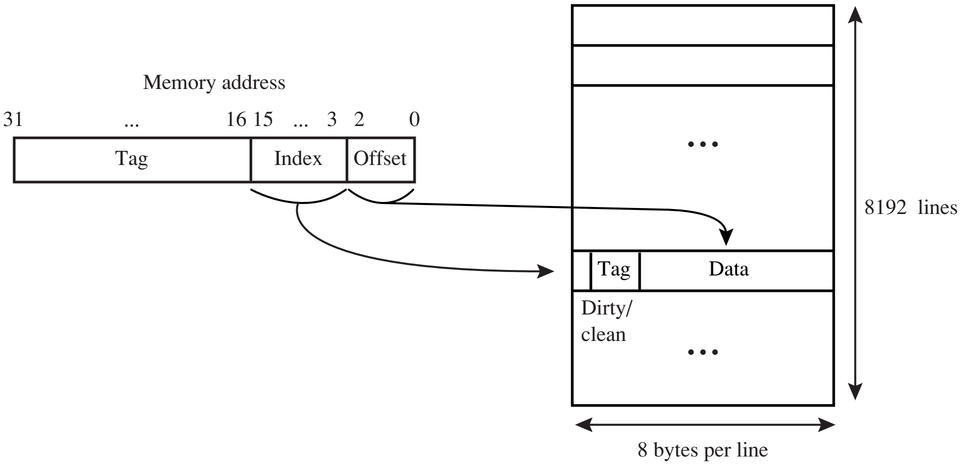


Figure 6.1 Simplified cache.

locality. If we reference a memory location twice and in between the two references the processor did not issue another reference with the same value of the index bits, the second reference will hit in the cache. Real caches are also designed to take advantage of spatial locality. Entries in a cache are usually not 8 bytes. Instead they are larger, typically anywhere from 32 to 256 contiguous bytes. The set of entries in a single cache location is called the *cache line.* With longer cache lines, if a reference, *a(i),* is a cache miss, the system will bring into the cache not only *a(i)* but also *a(i + 1)*, a*(i + 2),* and so on.

We have described a single-level cache. Most systems today use multiple levels of cache. With a two-level cache system, one can view the first cache as a cache for the second cache, and the second cache as a cache for memory. The Origin 2000, for example, has a 32 KB primary data cache on the MIPS R10000 chip and a 1–4 MB secondary cache off chip. Given a memory reference, the system attempts to find the value in the smaller primary cache. If there is a miss there, it attempts to find the reference in the secondary cache. If there is a miss there, it goes to memory.

Multiprocessors greatly complicate the issue of caches. Given that OpenMP is designed for shared memory multiprocessors, let us examine the memory architecture of these machines. On a shared memory machine, each processor can directly access any memory location in the entire system. The issue then arises whether to have a single large cache for the entire system or multiple smaller caches for each processor. If there is only one cache, that cache by definition will be far away from some processors since a single cache cannot be close to all processors. While that cache might be faster than memory, it will still be slower to access memory in a faraway cache than to access memory in a nearby cache. Consequently, most systems take the approach of having a cache for every processor.

In an architecture with per-processor caches, if a processor references a particular location, the data for that location is placed in the cache of the processor that made the reference. If multiple processors reference the same location, or even different locations in the same cache line, the data is placed in multiple caches. As long as processors only read data, there is no problem in putting data in multiple caches, but once processors write data, maintaining the latest correct value of the data becomes an issue; this is referred to as *cache coherence*. Let us say that processor 1 reads a location. That data is put inside its cache. Now, let's say that processor 2 writes the same location. If we are not careful, the data inside processor 1's cache will be old and invalid. Future reads by processor 1 might return the wrong value. Two solutions can be used to avoid this problem. In an *update-based* protocol, when processor 2 writes the location, it must broadcast the new value to all the caches that hold the data. The more common protocol, though, is an *invalidation-based* protocol. When processor 2 writes a location, it asks for exclusive access to the cache line. All other caches containing this line must invalidate their entries. Once a processor has exclusive access, it can continue to write the line as often as it likes. If another processor again tries to read the line, the other processor needs to ask the writing processor to give up exclusive access and convert the line back to shared state.

Caches and Locality

In the previous subsection, we gave a basic overview of caches on multiprocessor systems. In this subsection, we go into detail on how caches can impact the performance of OpenMP codes. Caches are designed to exploit locality, so the performance of a parallel OpenMP code can be greatly improved if the code exhibits locality. On the Origin 2000, for example, the processor is able to do a load or a store every cycle if the data is already in the primary cache. If the data is in neither the primary nor the secondary cache, it can take about 70 cycles [HLK 97].[2] Thus, if the program has no

[2] The approximation of 70 cycles is a simplified number to give the reader a feel for the order of magnitude of time. In reality, other factors greatly complicate the estimate. First, the R10000 is an out-of-order machine that can issue multiple memory references in parallel. If your code exhibits instruction-level parallelism, this effect can cut the perceived latency significantly. On the other hand, the Origin 2000 is a ccNUMA machine, meaning that some memory is closer to a processor than other memory. The 70-cycles figure assumes you are accessing the closest memory.

locality, it can slow down by a factor of 70. Spatial locality is often easier to achieve than temporal locality. Stride-1 memory references[3] have perfect spatial locality. The cache lines on the secondary cache of the Origin are 128 bytes, or 16 double words. If the program has perfect spatial locality, it will only miss every 16 references. Even so, without temporal locality, the code will still slow down by about a factor of four (since we will miss once every 16 references, and incur a cost of 70 cycles on that miss). Even on uniprocessors, there is much that can be done to improve locality. Perhaps the classic example on uniprocessors for scientific codes is loop interchange to make references stride-1. Consider the following loop nest:

```
do i = 1, n
   do j = 1, n
      a(i, j) = 0.0
   enddo
enddo
```

Arrays in Fortran are stored in column-major order, meaning that the columns of the array are laid out one after the other in memory. As a result, elements from successive rows within the same column of an array are laid out adjacently in memory; that is, the address of *a(i,j)* is just before the address of *a(i+1,j)* but is *n* elements away from the address of *a(i,j+1)*, where *n* is the size of the first dimension of *a*. In the code example, successive iterations of the inner loop do not access successive locations in memory. Now, there is spatial locality in the code; given a reference to *a(i,j)*, we will eventually access *a(i+1,j)*. The problem is that it will take time, in fact *n* iterations. During that time, there is some chance that the line containing *a(i,j)* and *a(i+1,j)* will be evicted from the cache. Thus, we might not be able to exploit the spatial locality. If we interchange the two loops as follows:

```
do j = 1, n
   do i = 1, n
      a(i, j) = 0.0
   enddo
enddo
```

[3] *Stride* refers to the distance in memory between successive references to a data structure. Stride-1 therefore implies that multiple references are to successive memory locations, while stride-*k* implies that multiple references are to locations *k* bytes apart.

the array references are stride-1. Successive references in time are adjacent in memory. There is no opportunity for the cache line to be evicted, and we are able to fully exploit spatial locality.

Many compilers will automatically interchange loops for the user, but sometimes more complicated and global transformations are needed to improve locality. Discussing uniprocessor cache optimizations in detail is beyond the scope of this book. Multiprocessor caches add significant complications. With uniprocessors, the user need only worry about locality. With multiprocessors, the user must worry about restricting locality to a single processor. Accessing data that is in some other processor's cache is usually no faster than accessing data that is in memory. In fact, on the Origin 2000, accessing data that is in someone else's cache is often more expensive than accessing data that is in memory. With uniprocessors, the user needs to insure that multiple references to the same or nearby locations happen close to each other in time. With multiprocessors, the user must also insure that other processors do not touch the same cache line.

Locality and Parallel Loop Schedules

The effect of locality and multiprocessors can perhaps be best seen in the interaction between loop schedules and locality. Consider again our example of scaling a sparse matrix. Imagine that this scaling is part of an interactive image-processing algorithm where the user might scale the same matrix multiple times. Assume that the total size of the matrix is small enough to fit in the aggregate caches of the processors. In other words, each processor's portion of the matrix is small enough to fit in its cache. After one scaling of the matrix, the matrix will be in the aggregate caches of the processors; each processor's cache will contain the portion of the matrix scaled by the particular processor. Now when we do a second scaling of the matrix, if a processor receives the same portions of the matrix to scale, those portions will be in its cache, and the scaling will happen very fast. If, on the other hand, a processor receives different portions of the matrix to scale, those portions will be in some other processor's cache, and the second scaling will be slow. If we had parallelized the code with a static schedule, every invocation of the scaling routine would divide the iterations of the loop the same way; the processor that got iteration *i* on the first scaling would get the same iteration on the second scaling. Since every iteration *i* always touches the same data, every processor would scale the same portions of the matrix. With a dynamic schedule, there is no such guarantee. Part of the appeal of dynamic schedules is that iterations are handed out to the first processor that needs more work. There is no guarantee that there will be any correlation between how work was handed out in one invocation versus another.

How bad is the locality effect? It depends on two different factors. The first is whether each processor's portion of the data is small enough to fit in cache. If the data fits in cache, a bad schedule means that each processor must access data in some other cache rather than its own. If the data is not small enough, most or all of the data will be in memory, regardless of the schedule. In such cases, dynamic schedules will have minimal impact on performance. To measure the effect, we parallelized a dense version of the matrix scaling algorithm applied repeatedly to a piece of data:

```
do i = 1, n
    do j = 1, n
        a(j, i) = 2.0 * a(j, i)
    enddo
enddo
```

We experimented with three different data set sizes; 400 by 400, 1000 by 1000 and 4000 by 4000. We ran each data size with both a static and a dynamic schedule on both one and eight processors. The dynamic schedule was chosen with a chunk size sufficiently large to minimize synchronization costs. The sizes were chosen so that the 400 by 400 case would fit in the cache of a single processor, the 1000 by 1000 case would be bigger than one processor's cache but would fit in the aggregate caches of the eight processors, and the 4000 by 4000 case would not fit in the aggregate caches of the eight processors.

We can see several interesting results from the data in Table 6.3. For both cases where the data fits in the aggregate caches, the static case is about a factor of 10 better than the dynamic case. The penalty for losing locality is huge. In the 400 by 400 case, the dynamic case is even slower than running on one processor. This shows that processing all the data from one's own cache is faster than processing one-eighth of the data from some other processor's cache. In the 1000 by 1000 case, the static schedule

Table 6.3 Static versus dynamic schedule for scaling.

Size	*Static Speedup*	*Dynamic Speedup*	*Ratio: Static/ Dynamic*
400 x 400	6.2	0.6	9.9
1000 x 1000	18.3	1.8	10.3
4000 x 4000	7.5	3.9	1.9

speeds up superlinearly, that is, more than by the number of processors. Processing one-eighth of the data from one's own cache is more than eight times faster than processing all the data from memory. For this data set size, the dynamic case does speed up a little bit over the single-processor case. Processing one-eighth of the data from someone else's cache is a little bit faster than processing all of the data from memory. Finally, in the large 4000 by 4000 case, the static case is faster than the dynamic, but not by such a large amount.[4]

We mentioned that there were two factors that influence how important locality effects are to the choice of schedule. The first is the size of each processor's data set. Very large data sets are less influenced by the interaction between locality and schedules. The second factor is how much reuse is present in the code processing a chunk of iterations. In the scaling example, while there is spatial locality, there is no temporal locality within the parallel loop (the only temporal locality is across invocations of the parallel scaling routine). If, on the other hand, there is a large amount of temporal reuse within a parallel chunk of work, scheduling becomes unimportant. If the data is going to be processed many times, where the data lives at the start of the parallel loop becomes mostly irrelevant. Consider, as an example, matrix multiply. Matrix multiplication does $O(n^3)$ computation on $O(n^2)$ data. There is, therefore, a lot of temporal locality as well as spatial locality. We timed a 1000 by 1000 matrix multiply on an eight-processor Origin 2000 using both static and dynamic schedules. The static schedule achieved a speedup of 7.5 while the dynamic achieved a speedup of 7.1.

From the scaling and matrix multiply examples we have given, one might conclude that it is always better to use static schedules, but both examples are cases with perfect load balance. Locality can make such a large difference that for cases such as the in-cache scaling example, it is probably always better to use a static schedule, regardless of load balance. On the other hand, for cases such as matrix multiply, the dynamic penalty is fairly small. If load balancing is an issue, dynamic scheduling would probably be better. There is a fundamental trade-off between load balancing and locality. Dealing with the trade-off is unfortunately highly dependent on your specific code.

[4] On some machines, dynamic and static schedules for the large data set might yield almost identical performance. The Origin 2000 is a NUMA machine, and on NUMA machines locality can have an impact on memory references as well as cache references. NUMA will be described in more detail at the end of the chapter.

False Sharing

Even with static schedules, good locality is not guaranteed. There are a few common, easy-to-fix errors that can greatly hurt locality. One of the more common is false sharing. Consider counting up the even and odd elements of an array in Example 6.5.

Every processor accumulates its portion of the result into a local portion of an array, *local_s.* At the end of the work-shared parallel loop, each processor atomically increments the shared array, *is,* with its portion of the local array, *local_s.* Sounds good, no?

Example 6.5 Counting the odd and even elements of an array.

```
      integer local_s(2, MAX_NUM_THREADS)

!$omp parallel private(my_id)
      my_id = omp_get_thread_num() + 1

!$omp do schedule(static) private(index)
      do i = 1, n
         index = MOD(ia(i), 2) + 1
         local_s(index, my_id) = local_s(index, my_id) + 1
      enddo

!$omp atomic
      is(1) = is(1) + local_s(1, my_id)
!$omp atomic
      is(2) = is(2) + local_s(2, my_id)
!$omp end parallel
```

The problem comes with how we create *local_s.* Every cache line (entry in the table) contains multiple contiguous words, in the case of the Origin 2000, 16 eight-byte words. Whenever there is a cache miss, the entire cache line needs to be brought into the cache. Whenever a word is written, every other address in the same cache line must be invalidated in all of the other caches in the system. In this example, the different processors' portions of *local_s* are contiguous in memory. Since each processor's portion is significantly smaller than a cache line, each processor's portion of *local_s* shares a cache line with other processors' portions. Each time a processor updates its portion, it must first invalidate the cache line in all the other processors' caches. The cache line is likely to be dirty in some other cache. That cache must therefore send the data to the new processor before that processor updates its local portion. The cache line will thus ping-pong among the caches of the different processors, leading to poor performance. We call such behavior *false sharing* because a cache line is being shared among multiple processors even though the different processors are accessing distinct data. We timed this example on an Origin using eight processors and a 1,000,000-element data array, *ia*. The parallel code slows down by a factor of 2.3 over the serial code. However, we can modify the code.

Example 6.6 Counting the odd and even elements of an array using distinct cache lines.

```
      integer local_s(2)

!$omp parallel private(local_s)
      local_s(1) = 0
      local_s(2) = 0
!$omp do schedule(static) private(index)
      do i = 1, n
         index = MOD(ia(i), 2) + 1
         local_s(index) = local_s(index) + 1
      enddo
!$omp atomic
      is(1) = is(1) + local_s(1)
!$omp atomic
      is(2) = is(2) + local_s(2)
!$omp end parallel
```

Instead of using a shared array indexed by the processor number to hold the local results, we use the *private* clause as shown in Example 6.6. The system insures that each processor's *local_s* array is on different cache lines. The code speeds up by a factor of 7.5 over the serial code, or a factor of 17.2 over the previous parallel code.

Other types of codes can also exhibit false sharing. Consider zeroing an array:

```
!$omp parallel do schedule(static)
      do i = 1, n
         do j = 1, n
            a(i, j) = 0.0
         enddo
      enddo
```

We have divided the array so that each processor gets a contiguous set of rows, as shown in Figure 6.2. Fortran, though, is a column major language, with elements of a column allocated in contiguous memory locations. Every column will likely use distinct cache lines, but multiple consecutive rows of the same column will use the same cache line.

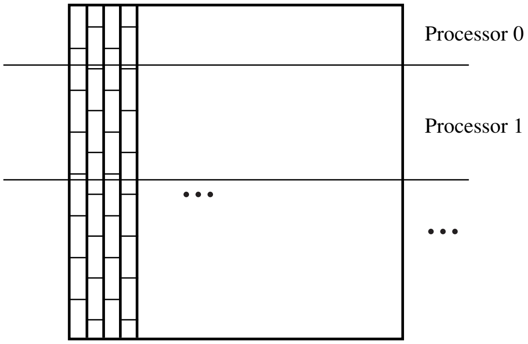


Figure 6.2 Dividing rows across processors.

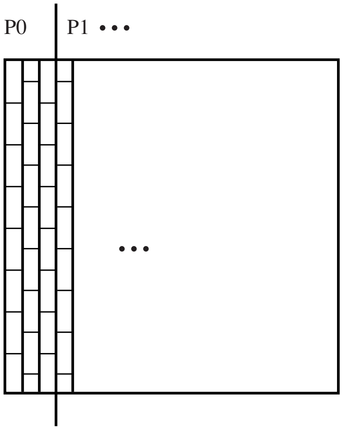


Figure 6.3 Dividing columns across processors.

If we parallelize the *i* loop, we divide the array among the processors by rows. In every single column, there will be cache lines that are falsely shared among the processors. If we instead interchange the loops, we divide the array among the processors by columns, as shown in Figure 6.3. For any consecutive pair of processors, there will be at most one cache line that is shared between the two. The vast majority of the false sharing will be eliminated.

Inconsistent Parallelization

Another situation that can lead to locality problems is inconsistent parallelization. Imagine the following set of two loops:

```
do i = 1, N
    a(i) = b(i)
enddo
do i = 1, N
    a(i) = a(i) + a(i - 1)
enddo
```

The first loop can be trivially parallelized. The second loop cannot easily be parallelized because every iteration depends on the value of *a(i–1)* written in the previous iteration. We might have a tendency to parallelize the first loop and leave the second one sequential. But if the arrays are small enough to fit in the aggregate caches of the processors, this can be the wrong decision. By parallelizing the first loop, we have divided the *a* matrix among the caches of the different processors. Now the serial loop starts and all the data must be brought back into the cache of the master processor. As we have seen, this is potentially very expensive, and it might therefore have been better to let the first loop run serially.

6.2.4 Synchronization

The last key performance factor that we will discuss is synchronization. We will consider two types of synchronization: barriers and mutual exclusion.

Barriers

Barriers are used as a global point of synchronization. A typical use of barriers is at the end of every parallel loop or region. This allows the user to consider a parallel region as an isolated unit and not have to consider dependences between one parallel region and another or between one parallel region and the serial code that comes before or after the parallel region. Barriers are very convenient, but on a machine without special support for them, barriers can be very expensive. To measure the time for a barrier, we timed the following simple loop on our Origin 2000:

```
!$omp parallel
      do i = 1, 1000000
!$omp barrier
      enddo
!$omp end parallel
```

On an eight-processor system, it took approximately 1000 cycles per barrier. If the program is doing significantly more than 1000 cycles of work in between barriers, this time might be irrelevant, but if we are parallelizing very small regions or loops, the barrier time can be quite significant. Note also that the time for the actual barrier is not the only cost to using barriers. Barriers synchronize all the processors. If there is some load imbalance and some processors reach the barrier later than other processors, all the processors have to wait for the slow ones. On some codes, this time can be the dominant effect.

So, how can we avoid barriers? First, implicit barriers are put at the end of all work-sharing constructs. The user can avoid these barriers by use of the *nowait* clause. This allows all threads to continue processing at the end of a work-sharing construct without having to wait for all the threads to complete. Of course, the user must insure that it is safe to eliminate the barrier in this case. Another technique to avoiding barriers is to coalesce multiple parallel loops into one. Consider Example 6.7.

Example 6.7 Code with multiple adjacent parallel loops.

```
!$omp parallel do
      do i = 1, n
         a(i) = ...
      enddo

!$omp parallel do
      do i = 1, n
         b(i) = a(i) + ...
      enddo
```

There is a dependence between the two loops, so we cannot simply run the two loops in parallel with each other, but the dependence is only within corresponding iterations. Iteration *i* of the second loop can not proceed until iteration *i* of the first loop is finished, but all other iterations of the second loop do not depend on iteration *i*. We can eliminate a barrier (and also the overhead cost for starting a parallel loop) by fusing the two loops together as follows in Example 6.8.

Example 6.8 Coalescing adjacent parallel loops.

```
!$omp parallel do
      do i = 1, n
         a(i) = ...
         b(i) = a(i) + ...
      enddo
```

Barriers are an all-points form of synchronization. Every processor waits for every other processor to finish a task. Sometimes, this is excessive; a processor only needs to wait for one other processor. Consider a simple two-dimensional recursive convolution:

```
do j = 2, n - 1
    do i = 2, n - 1
        a(i, j) = 0.5 * a(i, j) + 0.125 * (a(i - 1, j) +
                                     a(i + 1, j) + a(i, j - 1) +
                                     a(i, j + 1))
    enddo
enddo
```

Neither loop is parallel by itself, but all the points on a diagonal can be run in parallel. One potential parallelization technique is to skew the loop so that each inner loop covers a diagonal of the original:

```
do j = 4, 2 * n - 2
    do i = max(2, j - n + 1), min(n - 1, j - 2)
        a(i, j - i) = 0.5 * a(i, j - i) + 0.125 *
                    (a(i - 1, j - i) + a(i + 1, j - i) +
                     a(i, j - i - 1) + a(i, j - i + 1))
    enddo
enddo
```

After this skew, we can put a *parallel do* directive on the inner loop. Unfortunately, that puts a barrier in every iteration of the outer loop. Unless *n* is very large, this is unlikely to be efficient enough to be worth it. Note than a full barrier is not really needed. A point on the diagonal does not need to wait for all of the previous diagonal to be finished; it only needs to wait for points in lower-numbered rows and columns. For example, the processor processing the first element of each diagonal does not need to wait for any other processor. The processor processing the second row need only wait for the first row of the previous diagonal to finish.

We can avoid the barrier by using an alternative parallelization scheme. We divide the initial iteration space into blocks, handing *n/p* columns to each processor. Each processor's portion is not completely parallel. A processor cannot start a block until the previous processor has finished its corresponding block. We add explicit, point-to-point synchronization to insure that no processor gets too far ahead. This is illustrated in Figure 6.4, where each block spans *n/p* columns, and the height of each block is one row.

Parallelizing this way has not increased the amount of parallelism. In fact, if the height of each of the blocks is one row, the same diagonal will execute in parallel. We have, though, made two improvements. First with the previous method (skewing), each processor must wait for all the other processors to finish a diagonal; with this method (blocking), each processor only needs to wait for the preceding processor. This allows a much cheaper form of synchronization, which in turn allows early processors to proceed more quickly. In fact, the first processor can proceed without any

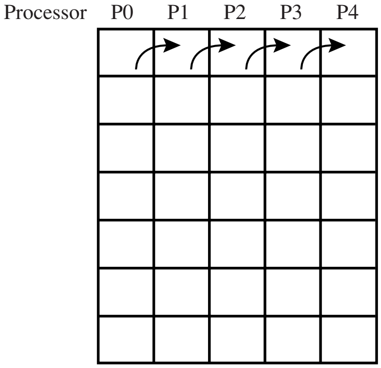


Figure 6.4 Point-to-point synchronization between blocks.

synchronization. The second advantage is that we are able to trade off synchronization for parallelism. With the skewing method, we have a barrier after every diagonal. With this method, we have a point-to-point synchronization after each block. At one extreme we can choose the block size to be one row. Each processor will synchronize n times, just as with the skewing method. At the other extreme, we make each block n rows, and the entire code will proceed sequentially. By choosing a size in the middle, we can trade off load balancing for synchronization.

Mutual Exclusion

Another common reason for synchronization is mutual exclusion. Let us say, for example, that multiple processors are entering data into a binary tree. We might not care which processor enters the data first, and it might also be fine if multiple processors enter data into different parts of the tree simultaneously, but if multiple processors try to enter data to the same part of the tree at the same time, one of the entries may get lost or the tree might become internally inconsistent. We can use mutual exclusion constructs, either locks or critical sections, to avoid such problems. By using a critical section to guard the entire tree, we can insure that no two processors update any part of the tree simultaneously. By dividing the tree into sections and using different locks for the different sections, we can ensure that no two processors update the same part of the tree simultaneously, while still allowing them to update different portions of the tree at the same time.

As another example, consider implementing a parallel sum reduction. One way to implement the reduction, and the way used by most compilers implementing the *reduction* clause, is to give each processor a local sum and to have each processor add its local sum into the global sum at the end of the parallel region. We cannot allow two processors to update the global sum simultaneously. If we do, one of the updates might get lost. We can use a critical section to guard the global update.

How expensive is a critical section? To check, we timed the program in Example 6.9. The times in cycles per iteration for different processor counts are given in Table 6.4.

Example 6.9 Measuring the contention for a critical section.

```
!$omp parallel
      do i = 1, 1000000
!$omp critical
!$omp end critical
      enddo
!$omp end parallel
```

It turns out that the time is very large. It takes about 10 times as long for eight processors to do a critical section each than it takes for them to execute a barrier (see the previous subsection). As we increase the number of processors, the time increases quadratically.

Why is a critical section so expensive compared to a barrier? On the Origin, a barrier is implemented as *P* two-way communications followed by one broadcast. Each slave writes a distinct memory location telling the master that it has reached the barrier. The master reads the *P* locations. When all are set, it resets the locations, thereby telling all the slaves that every slave has reached the barrier. Contrast this with a critical section. Critical sections on the Origin and many other systems are implemented using a pair of hardware instructions called LLSC (load-linked, store conditional). The load-linked instruction operates like a normal load. The store conditional instruction conditionally stores a new value to the location only if no other processor has updated the location since the load. These two instructions together allow the processor to atomically update memory locations. How is this used to implement critical sections? One

Table 6.4 Time in cycles for doing *P* critical sections.

Processors	*Cycles*
1	100
2	400
4	2500
8	11,000

processor holds the critical section and all the others try to obtain it. The processor releases the critical section by writing a memory location using a normal store. In the meantime, all the other processors are spinning, waiting for the memory location to be written. These processors are continuously reading the memory location, waiting for its value to change. Therefore, the memory location is in every processor's cache. Now, the processor that holds the critical section decides to release it. It writes the memory location. Before the write can succeed, the location must be invalidated in the cache of every other processor. Now the write succeeds. Every processor immediately reads the new value using the load-linked instruction. One processor manages to update the value using the store conditional instruction, but to do that it must again invalidate the location in every other cache. So, in order to acquire and release a critical section, two messages must be sent from one processor to all the others. In order for every processor to acquire and release a critical section, 2*P* messages must be sent from one processor to all the others. This is a very expensive process.

How can we avoid the large expense of a critical section? First, a critical section is only really expensive if it is heavily contended by multiple processors. If one processor repeatedly attempts to acquire a critical section, and no other processor is waiting, the processor can read and write the memory location directly in its cache. There is no communication. If multiple processors are taking turns acquiring a critical section, but at intervals widely enough spaced in time so that when one processor acquires the critical section no other processors are spinning, the processor that acquires the critical section need only communicate with the last processor to have the critical section, not every other processor. In either case, a key improvement is to lower the contention for the critical section. When updating the binary tree, do not lock the entire tree. Just lock the section that you are interested in. Multiple processors will be able to lock multiple portions of the tree without contending with each other.

Consider the following modification of Example 6.9, where instead every processor locks and unlocks one of 100 locks rather than one critical section:

```
!$omp parallel
    do i = 1, 1000000
        call omp_set_lock(lock(mod(i, 100) + 1))
        call omp_unset_lock(lock(mod(i, 100) + 1))
    enddo
!$omp end parallel
```

If only one lock was used, the performance would be exactly equivalent to the critical section, but by using 100 locks, we have greatly reduced the contention for any one lock. While every locking and unlocking still requires communication, it typically requires a two-way communication rather than a *P*-way communication. The time to execute this sequence on eight processors is about 500 cycles per iteration, much better than the 10,000 cycles we saw for the contended critical section.

Another approach to improving the efficiency of critical sections is using the *atomic* directive. Consider timing a series of atomic increments:

```
      do i = 1, 1000000
!$omp critical
         isum = isum + i
!$omp end critical
      enddo
```

On eight processors, this takes approximately the same 11,000 cycles per iteration as the empty critical section. If we instead replace the critical section with an *atomic* directive, the time decreases to about 5000 cycles per iteration. The critical section version of the increment requires three separate communications while the *atomic* only requires one. As we mentioned before, an empty critical section requires two communications, one to get the critical section and one to release it. Doing an increment adds a third communication. The actual data, *isum*, must be moved from one processor to another. Using the *atomic* directive allows us to update *isum* with only one communication. We can implement the *atomic* in this case with a load-linked from *isum,* followed by an increment, followed by a store conditional to *isum* (repeated in a loop for the cases that the store conditional fails). The location *isum* is used both to synchronize the communication and to hold the data.

The use of *atomic* has the additional advantage that it leads to the minimal amount of contention. With a critical section, multiple variables might be guarded by the same critical section, leading to unnecessary contention. With *atomic,* only accesses to the same location are synchronized. There is no excessive contention.

6.3 Performance-Tuning Methodology

We have spent the bulk of this chapter discussing characteristics of cache-based shared memory multiprocessors and how these characteristics interact with OpenMP to affect the performance of parallel programs. Now we are going to shift gears a bit and talk in general about how to go about improving the performance of parallel codes. Specific approaches to tuning depend a lot on the performance tools available on a specific platform, and there are large variations in the tools supported on different platforms. Therefore, we will not go into too much specific detail, but instead shall outline some general principles.

As discussed in earlier chapters, there are two common styles of programming in the OpenMP model: loop-level parallelization and domain decomposition. Loop-level parallelization allows for an incremental approach to parallelization. It is easier to start with a serial program and gradually parallelize more of the code as necessary. When using this technique, the first issue to worry about is coverage. Have you parallelized enough of your code? And if not, where should you look for improvements? It is important not to waste time parallelizing a piece of code that does not contribute significantly to the application's execution time. Parallelize the important parts of the code first.

Most (if not all) systems provide profiling tools to determine what fraction of the execution time is spent in different parts of the source code. These profilers tend to be based on one of two approaches, and many systems provide both types. The first type of profiler is based on *pc-sampling*. A clock-based interrupt is set to go off at fixed time intervals (typical intervals are perhaps 1–10 milliseconds). At each interrupt, the profiler checks the current program counter, *pc,* and increments a counter. A postprocessing step uses the counter values to give the user a statistical view of how much time was spent in every line or subroutine.

The second type of profiler is based on instrumentation. The executable is modified to increment a counter at every branch (goto, loop, etc.) or label. The instrumentation allows us to obtain an exact count of how many times every instruction was executed. The tools melds this count with a static estimate of how much time an instruction or set of instructions takes to execute. Together, this provides an estimate for how much time is spent in different parts of the code. The advantage of the instrumentation approach is that it is not statistical. We can get an exact count of how many times an instruction is executed. Nonetheless, particularly with parallel codes, be wary of instrumentation-based profilers; pc-sampling usually gives more accurate information.

There are two problems with instrumentation-based approaches that particularly affect parallel codes. The first problem is that the tool must make an estimate of how much time it takes to execute a set of instructions. As we have seen, the amount of time it takes to execute an instruction can be highly context sensitive. In particular, a load or store might take widely varying amounts of time depending on whether the data is in cache, in some other processor's cache, or in memory. The tool typically does not have the context to know, so typically instrumentation-based profilers assume that all memory references hit in the cache. Relying on such information might lead us to make the wrong trade-offs. For example, using such a tool you might decide to use a dynamic schedule to minimize load imbalance while completely ignoring the more important locality considerations.

The second problem with instrumentation-based profilers is that they are intrusive. In order to count events, the tool has changed the code. It is quite possible that most of the execution time goes towards counting events rather than towards the original computation. On uniprocessor codes this might not be an important effect. While instrumenting the code might increase the time it takes to run the instrumented executable, it does not change the counts of what is being instrumented. With multiprocessor codes, though, this is not necessarily the case. While one processor is busy counting events, another processor might be waiting for the first processor to reach a barrier. The more time the first processor spends in instrumentation code, the more time the second processor spends at the barrier. It is possible that an instrumented code will appear to have a load imbalance problem while the original, real code does not.

Using a pc-sampling-based profiler, we can discover which lines of code contribute to execution time in the parallel program. Often, profilers in OpenMP systems give an easy way to distinguish the time spent in a parallel portion of the code from the time spent in serial portions. Usually, each parallel loop or region is packaged in a separate subroutine. Looking at a profile, it is therefore very easy to discover what time is spent in serial portions of the code and in which serial portions. For loop-based parallelization approaches, this is the key to figuring out how much of the program is actually parallelized and which additional parts of the program we should concentrate on parallelizing.

With domain decomposition, coverage is less of an issue. Often, the entire program is parallelized. Sometimes, though, coverage problems can exist, and they tend to be more subtle. With domain decomposition, portions of code might be replicated; that is, every processor redundantly computes the same information. Looking for time spent in these replicated regions is analogous to looking for serial regions in the loop-based codes.

Regardless of which parallelization strategy, loop-level parallelization or domain decomposition, is employed, we can also use a profiler to measure load imbalance. Many profilers allow us to generate a per-thread, per-line profile. By comparing side by side the profiles for multiple threads, we can find regions of the code where some threads spend more time than others. Profiling should also allow us to detect synchronization issues. A profile should be able to tell us how much time we are spending in critical sections or waiting at barriers.

Many modern microprocessors provide hardware counters that allow us to measure interesting hardware events. Using these counters to measure cache misses can provide an easy way to detect locality problems in an OpenMP code. The Origin system provides two interfaces to these counters. The *perfex* utility provides an aggregate count of any or all of the hardware counters. Using it to measure cache misses on both a serial and parallel execution of the code can give a quick indication of whether cache misses are an issue in the code and whether parallelization has made cache issues worse. The second interface (Speedshop) is integrated together with the profiler. It allows us to find out how many cache misses are in each procedure or line of the source code. If cache misses are a problem, this tool allows us to find out what part of the code is causing the problem.

6.4 Dynamic Threads

So far this chapter has considered the performance impact of how you code and parallelize your algorithm, but it has considered your program as an isolated unit. There has been no discussion of how the program interacts with other programs in a computer system. In some environments, you might have the entire computer system to yourself. No other application will run at the same time as yours. Or, you might be using a batch scheduler that will insure that every application runs separately, in turn. In either of these cases, it might be perfectly reasonable to look at the performance of your program in isolation. On some systems and at some times, though, you might be running in a multiprogramming environment. Some set of other applications might be running concurrently with your application. The environment might even change throughout the execution time of your program. Some other programs might start to run, others might finish running, and still others might change the number of processors they are using.

OpenMP allows two different execution models. In one model, the user specifies the number of threads, and the system gives the user exactly that number of threads. If there are not enough processors or there are not enough free processors, the system might choose to multiplex those threads over a smaller number of processors, but from the program's point of view the number of threads is constant. So, for example, in this mode running the following code fragment:

```
      call omp_set_num_threads(4)
!$omp parallel
!$omp critical
      print *, 'Hello'
!$omp end critical
!$omp end parallel
```

will always print "Hello" four times, regardless of how many processors are available. In the second mode, called dynamic threads, the system is free at each parallel region or *parallel do* to lower the number of threads. With dynamic threads, running the above code might result in anywhere from one to four "Hello"s being printed. Whether or not the system uses dynamic threads is controlled either via the environment variable *OMP_DYNAMIC* or the runtime call *omp_set_dynamic*. The default behavior is implementation dependent. On the Origin 2000, dynamic threads are enabled by default. If the program relies on the exact number of threads remaining constant, if, for example, it is important that "Hello" is printed exactly four times, dynamic threads cannot be used. If, on the other hand, the code does not depend on the exact number of threads, dynamic threads can greatly improve performance when running in a multiprogramming environment (dynamic threads should have minimal impact when running stand-alone or under batch systems).

Why can using dynamic threads improve performance? To understand, let's first consider the simple example of trying to run a three-thread job on a system with only two processors. The system will need to multiplex the jobs among the different processors. Every so often, the system will have to stop one thread from executing and give the processor to another thread. Imagine, for example, that the stopped, or *preempted*, thread was in the middle of executing a critical section. Neither of the other two threads will be able to proceed. They both need to wait for the first thread to finish the critical section. If we are lucky, the operating system might realize that the threads are waiting on a critical section and might immediately preempt the waiting threads in favor of the previously preempted thread. More than likely, however, the operating system will not know, and it will let the other two threads spin for a while waiting for the critical section to be released. Only after a fixed time interval will some thread get preempted, allowing the holder of the critical section to complete. The entire time interval is wasted.

One might argue that the above is an implementation weakness. The OpenMP system should communicate with the operating system and inform it that the other two threads are waiting for a critical section. The problem is that communicating with the operating system is expensive. The system would have to do an expensive communication every critical section to improve the performance in the unlikely case that a thread is preempted at the wrong time. One might also argue that this is a degenerate case: critical sections are not that common, and the likelihood of a thread being preempted at exactly the wrong time is very small. Perhaps that is true, but barriers are significantly more common. If the duration of a parallel loop is smaller than the interval used by the operating system to multiplex threads, it is likely that at every parallel loop the program will get stuck waiting for a thread that is currently preempted.

Even ignoring synchronization, running with fewer processors than threads can hinder performance. Imagine again running with three threads on a two-processor system. Assume that thread 0 starts running on processor 0 and that thread 1 starts running on processor 1. At some point in time, the operating system preempts one of the threads, let us say thread 0, and gives processor 0 to thread 2. Processor 1 continues to execute thread 1. After some more time, the operating system decides to restart thread 0. On which processor should it schedule thread 0? If it schedules it on processor 0, both thread 0 and thread 2 will get fewer CPU cycles than thread 1. This will lead to a load balancing problem. If it schedules it on processor 1, all processors will have equal amounts of CPU cycles, but we may have created a locality problem. The data used by thread 0 might still reside in cache 0. By scheduling thread 0 on processor 1, we might have to move all its data from cache 0 to cache 1.

We have given reasons why using more threads than the number of processors can lead to performance problems. Do the same problems occur when each parallel application uses less threads than processors, but the set of applications running together in aggregate uses more threads than processors? Given, for example, two applications, each requesting all the processors in the system, the operating system could give each application half the processors. Such a scheduling is known as *space sharing*. While space sharing is very effective for applications using dynamic threads, without dynamic threads each application would likely have more threads than allotted processors, and performance would suffer greatly. Alternatively, the operating system can use *gang scheduling*—scheduling the machine so that an application is only run when all of its threads can be scheduled. Given two applications, each wanting all the processors in the system, a gang scheduler would at any given time give all the processors to one of the applications, alternating over time which of the applications gets the processors. For programs run without dynamic threads, gang scheduling can greatly improve performance over space sharing, but using dynamic threads and space sharing can improve performance compared to no dynamic threads and gang scheduling.

There are at least three reasons why gang scheduling can be less effective than space sharing with dynamic threads. First, gang scheduling can lead to locality problems. Each processor, and more importantly each cache, is being shared by multiple applications. The data used by the multiple applications will interfere with each other, leading to more cache misses. Second, gang scheduling can create packaging problems. Imagine, for example, a 16-processor machine running two applications, each using nine threads. Both applications cannot run concurrently using gang scheduling, so the system will alternate between the two applications. Since each one only uses nine threads, seven of the processors on the system will be idle. Finally, many applications' performance does not scale linearly with the number of processors. For example, consider running two applications on a two-processor machine. Assume that each one individually gets a speedup of 1.6 on two processors. With gang scheduling, each application will only get the processors half the time. In the best case, each application will get a speedup of 0.8 (i.e., a slowdown of 20%) compared to running serially on an idle machine. With space sharing, each application will run as a uniprocessor application, and it is possible that each application will run as fast as it did running serially on an idle machine.

6.5 Bus-Based and NUMA Machines

Many traditional shared memory machines, such as the Sun UE10000, Compaq's AlphaServer 8400, and SGI's Power Challenge, consist of a bus-based design as shown in Figure 6.5. On one side of the bus sit all the processors and their associated caches. On the other side of the bus sits all the memory. Processors and memory communicate with each other exclusively through the bus.

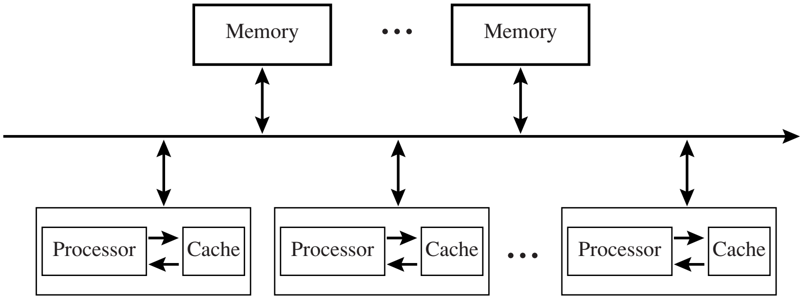


Figure 6.5 A bus-based multiprocessor.

From the programmer's point of view, bus-based multiprocessors have the desirable property that all memory can be accessed equally quickly from any processor. The problem is that the bus can become a single point of contention. It will usually not have enough bandwidth to support the worst case where every processor is suffering cache misses at the same time. This can lead to a performance wall. Once the bus is saturated, using more processors, either on one application or on many, does not buy any additional performance. A second performance limitation of bus-based machines is memory latency. In order to build a bus powerful enough to support simultaneous accesses from many processors, the system may slow down the access time to memory of any single processor.

An alternative type of system design is based on the concept of NUMA. The HP Exemplar [BA 97], SGI Origin 2000 [LL 97], Sequent NUMA-Q 2000 [LC 96], and the Stanford DASH machine [DLG 92] are all examples of NUMA machines. The architecture can be viewed as in Figure 6.6.

Memory is associated with each processor inside a node. Nodes are connected to each other through some type of, possibly hierarchical, network. These are still shared memory machines. The system ensures than any processor can access data in any memory, but from a performance point of view, accessing memory that is in the local node or a nearby node can be faster than accessing memory from a remote node.

There are two potential performance implications to using NUMA machines. The first is that locality might matter on scales significantly larger than caches. Recall our dense matrix scaling example from earlier in the

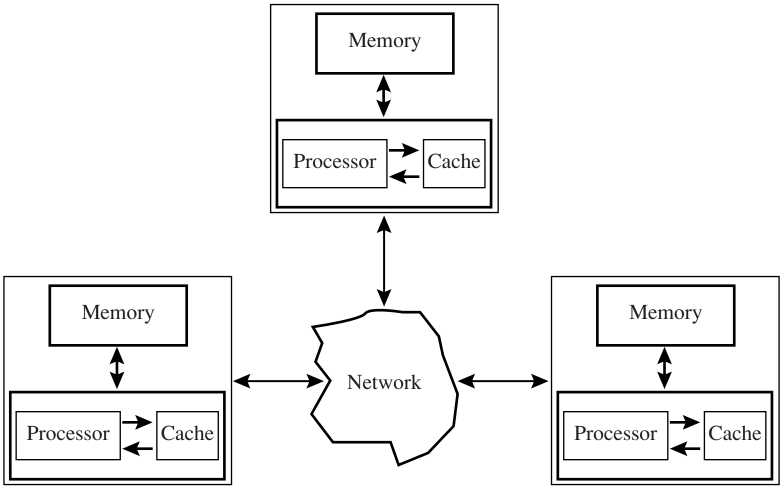


Figure 6.6 A NUMA multiprocessor.

chapter. When scaling a 4000 by 4000 element matrix, we said that cache locality was not that important since each processor's portion of the matrix was larger than its cache. Regardless of which processor scaled which portion of the matrix, the matrix would have to be brought into the processor's cache from memory. While a 4000 by 4000 element matrix is bigger than the aggregate cache of eight processors, it is not necessarily bigger than the memory in the nodes of the processors. Using a static scheduling of the iterations, it is possible that each processor will scale the portion of the matrix that resides in that processor's local memory. Using a dynamic schedule, it is not.

The second performance implication is that a user might worry about where in memory a data structure lives. For some data structures, where different processors process different portions of the data structure, the user might like for the data structure itself to be distributed across the different processors. Some OpenMP vendors supply directive-based extensions to allow users control over the distribution of their data structures. The exact mechanisms are highly implementation dependent and are beyond the scope of this book.

One might get the impression that NUMA effects are as critical to performance as cache effects, but that would be misleading. There are several differences between NUMA and cache effects that minimize the NUMA problems. First, NUMA effects are only relevant when the program has cache misses. If the code is well structured to deal with locality and almost all the references are cache hits, the actual home location of the cache line is completely irrelevant. A cache hit will complete quickly regardless of NUMA effects.

Second, there is no NUMA equivalent to false sharing. With caches, if two processors repeatedly write data in the same cache line, that data will ping-pong between the two caches. One can create a situation where a uniprocessor code is completely cache contained while the multiprocessor version suffers tremendously from cache issues. The same issue does not come up with memory. When accessing data located in a remote cache, that data is moved from the remote cache to the local cache. Ping-ponging occurs when data is repeatedly shuffled between multiple caches. In contrast, when the program accesses remote data on a NUMA machine, the data is not moved to the memory of the local node. If multiple processors are writing data on the same page,[5] the only effect is that some processors, the nonlocal ones, will take a little longer to complete the write. In fact, if

[5] The unit of data in a cache is a *cache line.* The analogous unit of data in memory is a *page.*

the two processors are writing data on different cache lines, both writes might even be cache contained; there will be no NUMA effects at all.

Finally, on modern systems, the difference between a cache hit and a cache miss is very large, a factor of 70 on the Origin 2000. The differences between local and remote memory accesses on the Origin are much smaller. On a 16-processor system, the worst-case latency goes up by a factor of 2.24. In terms of bandwidth, the difference is even smaller. The amount of bandwidth available from a processor to the farthest memory node is only 15% less than the amount of bandwidth available to local memory. Of course, the ratio of remote to local memory latency varies across different machines; for larger values of this ratio, the NUMA effect and data distribution optimizations can become increasingly important.

6.6 Concluding Remarks

We hope that we have given an overview of different factors that affect performance of shared memory parallel programs. No book, however, can teach you all you need to know. As with most disciplines, the key is practice. Write parallel programs. See how they perform. Gain experience. Happy programming.

6.7 Exercises

1. Implement the blocked version of the two-dimensional recursive, convolutional filter described in Section 6.2.4. To do this you will need to implement a point-to-point synchronization. You may wish to reread Section 5.5 to understand how this can be done in OpenMP.
2. Consider the following matrix transpose example:

```
real*8 a(N, N), b(N, N)
do i = 1, N
    do j = 1, N
        a(j, i) = b(i, j)
    enddo
enddo
```

   a) Parallelize this example using a *parallel do* and measure the parallel performance and scalability. Make sure you measure just the time to do the transpose and not the time to page in the memory. You also should make sure you are measuring the performance on a "cold" cache. To do this you may wish to define an additional array that is larger than the combined cache sizes in your system, and initialize this array after initializing *a* and *b* but before doing the transpose. If you do it right, your timing measurements will be repeatable (i.e., if you put an outer loop on your initialization and transpose procedures and repeat it five times, you will measure essentially the same time for each transpose). Measure the parallel speedup from one to as many processors as you have available with varying values of *N*. How do the speedup curves change with different matrix sizes? Can you explain why? Do you observe a limit on scalability for a fixed value of *N*? Is there a limit on scalability if you are allowed to grow *N* indefinitely?

b) One way to improve the sequential performance of a matrix transpose is to do it in place. The sequential code for an in-place transpose is

```
do i = 1, N
    do j = i + 1, N
        swap = a(i, j)
        a(i, j) = a(j, i)
        a(j, i) = swap
    enddo
enddo
```

Parallelize this example using a *parallel do* and include it in the timing harness you developed for Exercise 2a. Measure the performance and speedup for this transpose. How does it compare with the copying transpose of Exercise 2a? Why is the sequential performance better but the speedup worse?

c) It is possible to improve both the sequential performance and parallel speedup of matrix transpose by doing it in blocks. Specifically, you use the copying transpose of Exercise 2a but call it on subblocks of the matrix. By judicious choice of the subblock size, you can keep more of the transpose cache-contained, thereby improving performance. Implement a parallel blocked matrix transpose and measure its performance and speedup for different subblock sizes and matrix sizes.

d) The SGI MIPSPro Fortran compilers [SGI 99] include some extensions to OpenMP suitable for NUMA architectures. Of specific interest here are the data placement directives *distribute* and *distribute_reshape*. Read the documentation for these directives and apply them to your transpose program. How does the performance compare to that of the default page placement you measured in Exercise 2a–c? Would you expect any benefit from these directives on a UMA architecture? Could they actually hurt performance on a UMA system?

This Page Intentionally Left Blank

APPENDIX A

A Quick Reference to OpenMP

Table A.1 **OpenMP directives.**

Fortran	*C/C++*
Syntax	
sentinel directive-name [clause] ... Fixed form | Free form Sentinel: !$omp \| c$omp | *$omp — !$omp Continuation: !$omp+ — Trailing & Conditional compilation: !$ | c$ | *$ — !$	#pragma omp directive-name [clause] ... Continuation: Trailing \ Conditional compilation: #ifdef _OPENMP ... #endif
Parallel region construct	
!$omp parallel [clause] ... structured-block !$omp end parallel	#pragma omp parallel [clause] ... structured-block
Work-sharing constructs	
!$omp do [clause] ... do-loop !$omp enddo [nowait]	#pragma omp for [clause] ... for-loop

!$omp sections [clause] ...
[!$omp section
structured-block] ...
!$omp end sections [nowait]
	#pragma omp sections [clause] ...
	{
	[#pragma omp section
	structured-block] ...
	}

!$omp single [clause] ... structured-block !$omp end single [nowait]	#pragma omp single [clause] ... structured-block

Table A.1 *(Continued)*

Fortran	*C/C++*
Combined parallel work-sharing constructs	
!$omp parallel do [clause] ... do-loop [!$omp end parallel do]	#pragma omp parallel for [clause] ... for-loop

!$omp parallel sections [clause] ...
[!$omp section
    structured-block] ...
!$omp end parallel sections
	#pragma omp parallel sections [clause] ...
	{
	[#pragma omp section
	    structured-block] ...
	}

Synchronization constructs	
!$omp master structured-block !$end master	#pragma omp master structured-block
!$omp critical [(name)] structured-block !$omp end critical [(name)]	#pragma omp critical [(name)] structured-block
!$omp barrier	#pragma omp barrier
!$omp atomic expression-statement	#pragma omp atomic expression-statement
!$omp flush [(list)]	#pragma omp flush [(list)]
!$omp ordered structured-block !$omp end ordered	#pragma omp ordered structured-block
Data environment	
!$omp threadprivate (/c1/, /c2/)	#pragma omp threadprivate (list)

Table A.2 **OpenMP directive clauses.**

	Fortran						*C/C++*					
Clause	Parallel region	DO	Sections	Single	Parallel DO	Parallel sections	Parallel region	for	Sections	Single	Parallel for	Parallel sections
`shared(list)`	y				y	y	y				y	y
`private(list)`	y	y	y	y	y	y	y	y	y	y	y	y
`firstprivate(list)`	y	y	y	y	y	y	y	y	y	y	y	y
`lastprivate(list)`		y	y		y	y		y	y		y	y
`default(private | shared | none)`	y				y	y						
`default(shared | none)`							y				y	y
`reduction (operator | intrinsic : list)`	y	y	y		y	y	y	y	y		y	y
`copyin (list)`	y				y	y	y				y	y
`if (expr)`	y				y	y	y				y	y
`ordered`		y			y			y			y	
`schedule(type[,chunk])`		y			y			y			y	
`nowait`		y	y	y				y	y	y		

Table A.3 **OpenMP runtime library routines.**

Fortran	*C/C++*	*Description*
`call omp_set_num_threads (integer)`	`void omp_set_num_threads (int)`	Set the number of threads to use in a team.
`integer omp_get_num_threads ()`	`int omp_get_num_threads (void)`	Return the number of threads in the currently executing parallel region.
`integer omp_get_max_threads ()`	`int omp_get_max_threads (void)`	Return the maximum value that omp_get_num_threads may return.
`integer omp_get_thread_num ()`	`int omp_get_thread_num (void)`	Return the thread number within the team.
`integer omp_get_num_procs ()`	`int omp_get_num_procs (void)`	Return the number of processors available to the program.
`call omp_set_dynamic (logical)`	`void omp_set_dynamic (int)`	Control the dynamic adjustment of the number of parallel threads.
`logical omp_get_dynamic ()`	`int omp_get_dynamic (void)`	Return .TRUE. if dynamic threads is enabled, .FALSE. otherwise.
`logical omp_in_parallel ()`	`int omp_in_parallel (void)`	Return .TRUE. for calls within a parallel region, .FALSE. otherwise.
`call omp_set_nested (logical)`	`void omp_set_nested (int)`	Enable/disable nested parallelism.
`logical omp_get_nested ()`	`int omp_get_nested (void)`	Return .TRUE. if nested parallelism is enabled, .FALSE. otherwise.

Table A.4 **OpenMP lock routines.**

Fortran	*C/C++*	*Description*
`omp_init_lock (var)`	`void omp_init_lock(omp_lock_t*)`	Allocate and initialize the lock.
`omp_destroy_lock (var)`	`void omp_destroy_lock(omp_lock_t*)`	Deallocate and free the lock.
`omp_set_lock(var)`	`void omp_set_lock(omp_lock_t*)`	Acquire the lock, waiting until it becomes available, if necessary.
`omp_unset_lock (var)`	`void omp_unset_lock(omp_lock_t*)`	Release the lock, resuming a waiting thread (if any).
`logical omp_test_ lock(var)`	`int omp_test_lock(omp_lock_t*)`	Try to acquire the lock, return success (TRUE) or failure (FALSE).

Table A.5 **OpenMP environment variables.**

Variable	*Example*	*Description*
`OMP_SCHEDULE`	`"dynamic, 4"`	Specify the schedule type for parallel loops with a RUNTIME schedule.
`OMP_NUM_THREADS`	`16`	Specify the number of threads to use during execution.
`OMP_DYNAMIC`	`TRUE` or `FALSE`	Enable/disable dynamic adjustment of threads.
`OMP_NESTED`	`TRUE` or `FALSE`	Enable/disable nested parallelism.

Table A.6 **OpenMP reduction operators.**

Fortran	+	*	–	.AND.	.OR.	.EQV.	.NEQV.	MAX	MIN	IAND	IOR	IEOR
C/C++	+	*	–	&	|	^	&&	||				

Table A.7 **OpenMP atomic operators.**

Fortran	+	*	–	/	.AND.	.OR.	.EQV.	.NEQV.	MAX	MIN	IAND	IOR	IEOR
C/C++	++	––	+	*	–	/	&	^	<<	>>	|		

References

[ABM 97] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 95 Handbook.* MIT Press. June 1997.

[AG 96] S. V. Adve and K. Gharachorloo. “Shared Memory Consistency Model: A Tutorial.” WRL Research Report 95/7, DEC, September 1995.

[BA 97] Tony Brewer and Greg Astfalk. “The Evolution of the HP/Convex Exemplar.” In *Proceedings of the COMPCON Spring 1997 42nd IEEE Computer Society International Conference,* February 1997, pp. 81–86.

[BS 97] Bjarne Stroustrup. *The C++ Programming Language,* third edition. Addison-Wesley. June 1997.

[DLG 92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. “The Stanford DASH Multiprocessor.” *IEEE Computer,* Volume 25, No. 3, March 1992, pp. 63–79.

[DM 98] L. Dagum and R. Menon. “OpenMP: An Industry-Standard API for Shared Memory Programming.” *Computational Science and Engineering,* Vol. 5, No. 1, January–March 1998.

[GDS 95] Georg Grell, Jimmy Dudhia, and David Stauffer. *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5).* NCAR Technical Note TN-398 + STR. National Center for Atmospheric Research, Boulder, Colorado. June 1995.

[HLK 97] Cristina Hristea, Daniel Lenoski, and John Keen. “Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks.” In *Proceedings of Supercomputing 97,* November 1997.

[HP 90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann. 1990.

[KLS 94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook,* Scientific and Engineering Computation. MIT Press. January 1994.

[KR 88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language,* second edition. Prentice Hall. 1988.

[KY 95] Arvind Krishnamurthy and Katherine Yelick. “Optimizing Parallel Programs with Explicit Synchronization.” In *Proceedings of the Conference on Program Language Design and Implementation (PLDI),* June 1995, pp. 196–204.

[LC 96] Tom Lovett and and Russell Clapp. “STiNG: A CC-NUMA Computer System for the Commercial Marketplace.” In *Proceedings of the 23rd Annual International Symposium on Computer Architecture,* May 1996, pp. 308–317.

[LL 97] James Laudon and Daniel Lenoski. “The SGI Origin: A ccNUMA Highly Scalable Server.” In *Proceedings of the 24th Annuual International Symposium on Computer Architecture,* June 1997, pp. 241–251.

[MR 99] Michael Metcalf and John Ker Reid. *Fortran 90/95 Explained.* Oxford University Press. August 1999.

[MW 96] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley. 1996.

[NASPB 91] D. H. Baily, J. Barton, T. A. Lasinski, and H. Simon. *The NAS Parallel Benchmarks.* NAS Technical Report RNR-91-002. 1991.

[NBF 96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O’Reilly & Associates. September 1996.

[PP 96] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann. October 1996.

[RAB 98] Joseph Robichaux, Alexander Akkerman, Curtis Bennett, Roger Jia, and Hans Leichtl. “LS-DYNA 940 Parallelism on the Compaq Family of Systems.” In *Proceedings of the Fifth International LS-DYNA Users’ Conference.* August 1998.

[SGI 99] Silicon Graphics Inc. *Fortran 77 Programmer’s Guide.* SGI Technical Publications Document Number 007-0711-060. Viewable at *techpubs.sgi.com,* 1999.

[THZ 98] C. A. Taylor, T. J. R. Hughes, and C. K. Zarins. “Finite Element Modeling of Blood Flow in Arteries.” *Computer Methods in Applied Mechanics and Engineering.* Vol. 158, Nos. 1–2, pp. 155–196, 1998.

[TMC 91] Thinking Machines Corporation. *The Connection Machine CM5 Technical Summary.* 1991.

[X3H5 94] American National Standards Institute. *Parallel Extensions for Fortran.* Technical Report X3H5/93-SD1-Revision M. Accredited Standards Committee X3. April 1994.

Index

Related Titles from Morgan Kaufmann Publishers

Parallel Computer Architecture: A Hardware/Software Approach
David E. Culler and Jaswinder Pal Singh with Anoop Gupta

Industrial Strength Parallel Computing
Edited by Alice E. Koniges

Parallel Programming with MPI
Peter S. Pacheco

Distributed Algorithms
Nancy A. Lynch

Forthcoming

Implicit Parallel Programming in pH
Arvind and Rishiyur S. Nikhil

Practical IDL Programming
Liam E. Gumley

Advanced Compilation for Vector and Parallel Computers
Randy Allen and Ken Kennedy

Parallel I/O for High Performance Computing
John M. May

The CRPC Handbook of Parallel Computing
Edited by Jack Dongarra, Ian Foster, Geoffrey Fox, Ken Kennedy,
Linda Torczon, and Andy White

This Page Intentionally Left Blank