Using OpenMP

Portable Shared Memory Parallel Programming

Barbara Chapman, Gabriele Jost, Ruud van der Pas

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in LaTeX by the authors and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Chapman, Barbara, 1954-
Using OpenMP : portable shared memory parallel programming / Barbara Chapman, Gabriele Jost, Ruud van der Pas.
p. cm. – (Scientific and engineering computation)
Includes bibliographical references and index.
ISBN-13: 978-0-262-53302-7 (paperback : alk. paper)
1. Parallel programming (Computer science) 2. Application program interfaces (Computer software) I. Jost, Gabriele. II. Pas, Ruud van der. III. Title.
QA76.642.C49 2007
005.2'75–dc22

2007026656

1 Introduction

OpenMP enables the creation of shared-memory parallel programs. In this chapter, we describe the evolution of computers that has led to the specification of OpenMP and that has made it relevant to mainstream computing. We put our subject matter into a broader context by giving a brief overview of parallel computing and the main approaches taken to create parallel programs. Our discussion of these topics is not intended to be comprehensive.

1.1 Why Parallel Computers Are Here to Stay

No matter how fast computers are, technology is being developed to make them even faster. Our appetite for compute power and memory seems insatiable. A more powerful machine leads to new kinds of applications, which in turn fuel our demand for yet more powerful systems. The result of this continued technological progress is nothing short of breathtaking: the laptops a couple of us are using to type this script would have been among the fastest machines on the planet just a decade ago, if they had been around at the time.

In order to achieve their breakneck speed, today's computer systems are highly complex [85]. They are made up of multiple components, or functional units, that may be able to operate simultaneously and have specific tasks, such as adding two integer numbers or determining whether a value is greater than zero. As a result, a computer might be able to fetch a datum from memory, multiply two floating-point numbers, and evaluate a branch condition all at the same time. This is a very low level of parallel processing and is often referred to as "instruction-level parallelism," or ILP. A processor that supports this is said to have a superscalar architecture. Nowadays it is a common feature in general-purpose microprocessors, even those used in laptops and PCs.

Careful reordering of these operations may keep the machine's components busy. The lion's share of the work of finding such a suitable ordering of operations is performed by the compiler (although it can be supported in hardware). To accomplish this, compiler writers developed techniques to determine dependences between operations and to find an ordering that efficiently utilizes the instruction-level parallelism and keeps many functional units and paths to memory busy with useful work. Modern compilers put considerable effort into this kind of instruction-level optimization. For instance, software pipelining may modify the sequence of instructions in a loop nest, often overlapping instructions from different iterations to ensure that as many instructions as possible complete every clock cycle. Unfortunately, several studies [95] showed that typical applications are not likely to contain more than three or four different instructions that can be fed to the computer at a time in this way. Thus, there is limited payoff for extending the hardware support for this kind of instruction-level parallelism.

Back in the 1980s, several vendors produced computers that exploited another kind of architectural parallelism.[1] They built machines consisting of multiple complete processors with a common shared memory. These shared-memory parallel, or multiprocessor, machines could work on several jobs at once, simply by parceling them out to the different processors. They could process programs with a variety of memory needs, too, and were thus suitable for many different workloads. As a result, they became popular in the server market, where they have remained important ever since. Both small and large shared-memory parallel computers (in terms of number of processors) have been built: at the time of writing, many of them have two or four CPUs, but there also exist shared-memory systems with more than a thousand CPUs in use, and the number that can be configured is growing. The technology used to connect the processors and memory has improved significantly since the early days [44]. Recent developments in hardware technology have made architectural parallelism of this kind important for mainstream computing.

In the past few decades, the components used to build both high-end and desktop machines have continually decreased in size. Shortly before 1990, Intel announced that the company had put a million transistors onto a single chip (the i860). A few years later, several companies had fit 10 million onto a chip. In the meantime, technological progress has made it possible to put billions of transistors on a single chip. As data paths became shorter, the rate at which instructions were issued could be increased. Raising the clock speed became a major source of advances in processor performance. This approach has inherent limitations, however, particularly with respect to power consumption and heat emission, which is increasingly hard to dissipate.

Recently, therefore, computer architects have begun to emphasize other strategies for increasing hardware performance and making better use of the available space on the chip. Given the limited usefulness of adding functional units, they have returned to the ideas of the 1980s: multiple processors that share memory are configured in a single machine and, increasingly, on a chip. This new generation of shared-memory parallel computers is inexpensive and is intended for general-purpose usage.

Some recent computer designs permit a single processor to execute multiple instruction streams in an interleaved way. Simultaneous multithreading, for example, interleaves instructions from multiple applications in an attempt to use more of the

[1]Actually, the idea was older than that, but it didn't take off until the 1980s.

hardware components at any given time. For instance, the computer might add two values from one set of instructions and, *at the same time*, fetch a value from memory that is needed to perform an operation in a different set of instructions. An example is Intel's hyperthreading^{TM} technology. Other recent platforms (e.g., IBM's Power5, AMD's Opteron and Sun's UltraSPARC IV, IV+, and T1 processors) go even further, replicating substantial parts of a processor's logic on a single chip and behaving much like shared-memory parallel machines. This approach is known as *multicore.* Simultaneous multithreading platforms, multicore machines, and shared-memory parallel computers all provide system support for the execution of multiple independent instruction streams, or *threads.* Moreover, these technologies may be combined to create computers that can execute high numbers of threads.

Given the limitations of alternative strategies for creating more powerful computers, the use of parallelism in general-purpose hardware is likely to be more pronounced in the near future. Some PCs and laptops are already multicore or multithreaded. Soon, processors will routinely have many cores and possibly the ability to execute multiple instruction streams within each core. In other words, multicore technology is going mainstream [159]. It is vital that application software be able to make effective use of the parallelism that is present in our hardware [171]. But despite major strides in compiler technology, the programmer will need to help, by describing the concurrency that is contained in application codes. In this book, we will discuss one of the easiest ways in which this can be done.

1.2 Shared-Memory Parallel Computers

Throughout this book, we will refer to shared-memory parallel computers as SMPs. Early SMPs included computers produced by Alliant, Convex, Sequent [146], Encore, and Synapse [10] in the 1980s. Larger shared-memory machines included IBM's RP3 research computer [149] and commercial systems such as the BBN Butterfly [23]. Later SGI's Power Challenge [65] and Sun Microsystem's Enterprise servers entered the market, followed by a variety of desktop SMPs.

The term SMP was originally coined to designate a symmetric multiprocessor system, a shared-memory parallel computer whose individual processors share memory (and I/O) in such a way that each of them can access any memory location with the same speed; that is, they have a *uniform memory access* (UMA) time. Many small shared-memory machines are symmetric in this sense. Larger shared-memory machines, however, usually do not satisfy this definition; even though the difference may be relatively small, some memory may be "nearer to" one or more of the processors and thus accessed faster by them. We say that such machines have *cache-coherent non-uniform memory access* (cc-NUMA). Early innovative attempts to build cc-NUMA shared-memory machines were undertaken by Kendall Square Research (KSR1 [62]) and Denelcor (the Denelcor HEP). More recent examples of large NUMA platforms with cache coherency are SGI's Origin and Altix series, HP's Exemplar, and Sun Fire E25K.

Today, the major hardware vendors all offer some form of shared-memory parallel computer, with sizes ranging from two to hundreds – and, in a few cases, thousands – of processors.

Conveniently, the acronym SMP can also stand for "shared-memory parallel computer," and we will use it to refer to *all* shared-memory systems, including cc-NUMA platforms. By and large, the programmer can ignore this difference, although techniques that we will explore in later parts of the book can help take cc-NUMA characteristics into account.

1.2.1 Cache Memory Is Not Shared

Somewhat confusing is the fact that even SMPs have some memory that is not shared. To explain why this is the case and what the implications for applications programming are, we present some background information. One of the major challenges facing computer architects today is the growing discrepancy in processor and memory speed. Processors have been consistently getting faster. But the more rapidly they can perform instructions, the quicker they need to receive the values of operands from memory. Unfortunately, the speed with which data can be read from and written to memory has not increased at the same rate. In response, the vendors have built computers with hierarchical memory systems, in which a small, expensive, and very fast memory called cache memory, or "cache" for short, supplies the processor with data and instructions at high rates [74]. Each processor of an SMP needs its own private cache if it is to be fed quickly; hence, not all memory is shared.

Figure 1.1 shows an example of a generic, cache-based dual-core processor. There are two levels of cache. The term *level* is used to denote how far away (in terms of access time) a cache is from the CPU, or core. The higher the level, the longer it takes to access the cache(s) at that level. At level 1 we distinguish a cache for data ("Data Cache"), one for instructions ("Instr. Cache"), and the "Translation-Lookaside Buffer" (or TLB for short). The last of these is an address cache. It is discussed in Section 5.2.2. These three caches are all private to a core: other core(s) cannot access them. Our figure shows only one cache at the second level. It is most likely bigger than each of the level-1 caches, and it is shared by both cores. It is also unified, which means that it contains instructions as well as data.

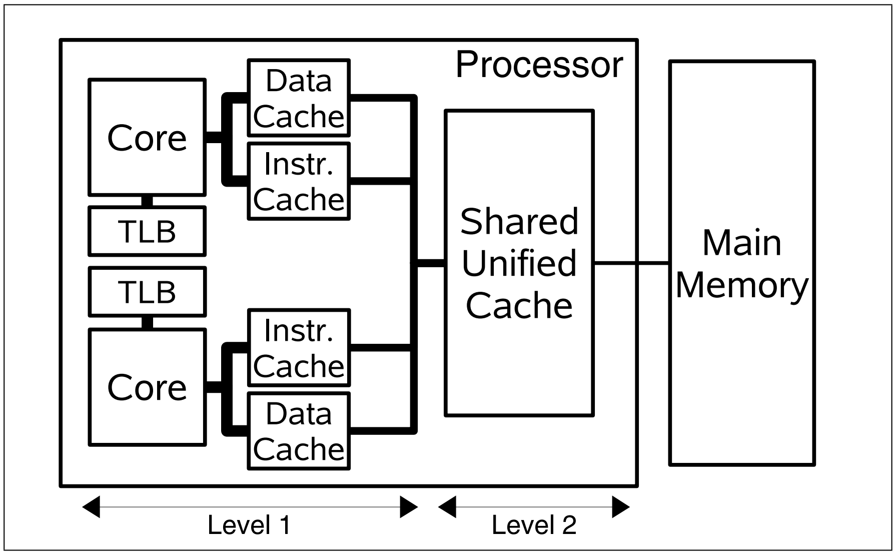

Figure 1.1: **Block diagram of a generic, cache-based dual core processor** – In this imaginary processor, there are two levels of cache. Those closest to the core are called "level 1." The higher the level, the farther away from the CPU (measured in access time) the cache is. The level-1 cache is private to the core, but the cache at the second level is shared. Both cores can use it to store and retrieve instructions, as well as data.

Data is copied into cache from main memory: blocks of consecutive memory locations are transferred at a time. Since the cache is very small in comparison to main memory, a new block may displace data that was previously copied in. An operation can be (almost) immediately performed if the values it needs are available in cache. But if they are not, there will be a delay while the corresponding data is retrieved from main memory. Hence, it is important to manage cache carefully. Since neither the programmer nor the compiler can directly put data into—or remove data from—cache, it is useful to learn how to structure program code to indirectly make sure that cache is utilized well.[2]

[2]The techniques developed to accomplish this task are useful for sequential programming, too. They are briefly covered in Section 5.2.3

1.2.2 Implications of Private Cache Memory

In a uniprocessor system, new values computed by the processor are written back to cache, where they remain until their space is required for other data. At that point, any new values that have not already been copied back to main memory are stored back there. This strategy does not work for SMP systems. When a processor of an SMP stores results of local computations in its private cache, the new values are accessible only to code executing on that processor. If no extra precautions are taken, they will not be available to instructions executing elsewhere on an SMP machine until after the corresponding block of data is displaced from cache. But it may not be clear when this will happen. In fact, since the old values might still be in other private caches, code executing on other processors might continue to use them even then.

This is known as the *memory consistency problem.* A number of strategies have been developed to help overcome it. Their purpose is to ensure that updates to data that have taken place on one processor are made known to the program running on other processors, and to make the modified values available to them if needed. A system that provides this functionality transparently is said to be *cache coherent.*

Fortunately, the OpenMP application developer does not need to understand how cache coherency works on a given computer. Indeed, OpenMP can be implemented on a computer that does not provide cache coherency, since it has its own set of rules on how data is shared among the threads running on different processors. Instead, the programmer must be aware of the *OpenMP memory model*, which provides for shared and private data and specifies when updated shared values are guaranteed to be available to all of the code in an OpenMP program.

1.3 Programming SMPs and the Origin of OpenMP

Once the vendors had the technology to build moderately priced SMPs, they needed to ensure that their compute power could be exploited by individual applications. This is where things got sticky. Compilers had always been responsible for adapting a program to make best use of a machine's internal parallelism. Unfortunately, it is very hard for them to do so for a computer with multiple processors or cores. The reason is that the compilers must then identify independent streams of instructions that can be executed in parallel. Techniques to extract such instruction streams from a sequential program do exist; and, for simple programs, it may be worthwhile trying out a compiler's automatic (shared-memory) parallelization options. However, the compiler often does not have enough information to decide whether it is possible to split up a program in this way. It also cannot make large-scale changes to code, such as replacing an algorithm that is not suitable for parallelization. Thus, most of the time the compiler will need some help from the user.

1.3.1 What Are the Needs?

To understand how programmers might express a code's parallelism, the hardware manufacturers looked carefully at existing technology. Beginning in the 1980s, scientists engaged in solving particularly tough computational problems attempted to exploit the SMPs of the day to speed up their code and to perform much larger computations than were possible on a uniprocessor. To get the multiple processors to collaborate to execute a *single* application, they looked for regions of code whose instructions could be shared among the processors. Much of the time, they focused on distributing the work in loop nests to the processors.

In most programs, code executed on one processor required results that had been calculated on another one. In principle, this was not a problem because a value produced by one processor could be stored in main memory and retrieved from there by code running on other processors as needed. However, the programmer needed to ensure that the value was retrieved after it had been produced, that is, that the accesses occurred in the required order. Since the processors operated independently of one another, this was a nontrivial difficulty: their clocks were not synchronized, and they could and did execute their portions of the code at slightly different speeds.

Accordingly, the vendors of SMPs in the 1980s provided special notation to specify how the work of a program was to be parceled out to the individual processors of an SMP, as well as to enforce an ordering of accesses by different threads to shared data. The notation mainly took the form of special instructions, or *directives*, that could be added to programs written in sequential languages, especially Fortran. The compiler used this information to create the actual code for execution by each processor. Although this strategy worked, it had the obvious deficiency that a program written for one SMP did not necessarily execute on another one.

1.3.2 A Brief History of Saving Time

Toward the end of the 1980s, vendors began to collaborate to improve this state of affairs. An informal industry group called the Parallel Computing Forum (PCF) agreed on a set of directives for specifying loop parallelism in Fortran programs; their work was published in 1991 [59]. An official ANSI subcommittee called X3H5 was set up to develop an ANSI standard based on PCF. A document for the new standard was drafted in 1994 [19], but it was never formally adopted. Interest in PCF and X3H5 had dwindled with the rise of other kinds of parallel computers that promised a scalable and more cost-effective approach to parallel programming. The X3H5 standardization effort had missed its window of opportunity.

But this proved to be a temporary setback. OpenMP was defined by the *OpenMP Architecture Review Board* (ARB), a group of vendors who joined forces during the latter half of the 1990s to provide a common means for programming a broad range of SMP architectures. OpenMP was based on the earlier PCF work. The first version, consisting of a set of directives that could be used with Fortran, was introduced to the public in late 1997. OpenMP compilers began to appear shortly thereafter. Since that time, bindings for C and C++ have been introduced, and the set of features has been extended. Compilers are now available for virtually all SMP platforms. The number of vendors involved in maintaining and further developing its features has grown. Today, almost all the major computer manufacturers, major compiler companies, several government laboratories, and groups of researchers belong to the ARB.

One of the biggest advantages of OpenMP is that the ARB continues to work to ensure that OpenMP remains relevant as computer technology evolves. OpenMP is under cautious, but active, development; and features continue to be proposed for inclusion into the application programming interface. Applications live vastly longer than computer architectures and hardware technologies; and, in general, application developers are careful to use programming languages that they believe will be supported for many years to come. The same is true for parallel programming interfaces.

1.4 What Is OpenMP?

OpenMP is a shared-memory application programming interface (API) whose features, as we have just seen, are based on prior efforts to facilitate shared-memory parallel programming. Rather than an officially sanctioned standard, it is an agreement reached between the members of the ARB, who share an interest in a portable, user-friendly, and efficient approach to shared-memory parallel programming. OpenMP is intended to be suitable for implementation on a broad range of SMP architectures. As multicore machines and multithreading processors spread in the marketplace, it might be increasingly used to create programs for uniprocessor computers also.

Like its predecessors, OpenMP is not a new programming language. Rather, it is notation that can be added to a sequential program in Fortran, C, or C++ to describe how the work is to be shared among threads that will execute on different processors or cores and to order accesses to shared data as needed. The appropriate insertion of OpenMP features into a sequential program will allow many, perhaps most, applications to benefit from shared-memory parallel architectures—often with minimal modification to the code. In practice, many applications have considerable parallelism that can be exploited.

The success of OpenMP can be attributed to a number of factors. One is its strong emphasis on structured parallel programming. Another is that OpenMP is comparatively simple to use, since the burden of working out the details of the parallel program is up to the compiler. It has the major advantage of being widely adopted, so that an OpenMP application will run on many different platforms.

But above all, OpenMP is timely. With the strong growth in deployment of both small and large SMPs and other multithreading hardware, the need for a shared-memory programming standard that is easy to learn and apply is accepted throughout the industry. The vendors behind OpenMP collectively deliver a large fraction of the SMPs in use today. Their involvement with this de facto standard ensures its continued applicability to their architectures.

1.5 Creating an OpenMP Program

OpenMP's *directives* let the user tell the compiler which instructions to execute in parallel and how to distribute them among the threads that will run the code. An OpenMP directive is an instruction in a special format that is understood by OpenMP compilers only. In fact, it looks like a comment to a regular Fortran compiler or a pragma to a C/C++ compiler, so that the program may run just as it did beforehand if a compiler is not OpenMP-aware. The API does not have many different directives, but they are powerful enough to cover a variety of needs. In the chapters that follow, we will introduce the basic idea of OpenMP and then each of the directives in turn, giving examples and discussing their main uses.

The first step in creating an OpenMP program from a sequential one is to identify the parallelism it contains. Basically, this means finding instructions, sequences of instructions, or even large regions of code that may be executed concurrently by different processors.

Sometimes, this is an easy task. Sometimes, however, the developer must reorganize portions of a code to obtain independent instruction sequences. It may even be necessary to replace an algorithm with an alternative one that accomplishes the same task but offers more exploitable parallelism. This can be a challenging problem. Fortunately, there are some typical kinds of parallelism in programs, and a variety of strategies for exploiting them have been developed. A good deal of knowledge also exists about algorithms and their suitability for parallel execution. A growing body of literature is being devoted to this topic [102, 60] and to the design of parallel programs [123, 152, 72, 34]. In this book, we will introduce some of these strategies by way of examples and will describe typical approaches to creating parallel code using OpenMP.

The second step in creating an OpenMP program is to express, using OpenMP, the parallelism that has been identified. A huge practical benefit of OpenMP is that it can be applied to *incrementally* create a parallel program from an existing sequential code. The developer can insert directives into a portion of the program and leave the rest in its sequential form. Once the resulting program version has been successfully compiled and tested, another portion of the code can be parallelized. The programmer can terminate this process once the desired speedup has been obtained.

Although creating an OpenMP program in this way can be easy, sometimes simply inserting directives is not enough. The resulting code may not deliver the expected level of performance, and it may not be obvious how to remedy the situation. Later, we will introduce techniques that may help improve a parallel program, and we will give insight into how to investigate performance problems. Armed with this information, one may be able to take a simple OpenMP program and make it run better, maybe even significantly better. It is essential that the resulting code be correct, and thus we also discuss the perils and pitfalls of the process. Finding certain kinds of bugs in parallel programs can be difficult, so an application developer should endeavor to prevent them by adopting best practices from the start.

Generally, one can quickly and easily create parallel programs by relying on the implementation to work out the details of parallel execution. This is how OpenMP directives work. Unfortunately, however, it is not always possible to obtain high performance by a straightforward, incremental insertion of OpenMP directives into a sequential program. To address this situation, OpenMP designers included several features that enable the programmer to specify more details of the parallel code. Later in the book, we will describe a completely different way of using OpenMP to take advantage of these features. Although it requires quite a bit more work, users may find that getting their hands downright dirty by creating the code for each thread can be a lot of fun. And, this may be the ticket to getting OpenMP to solve some very large problems on a very big machine.

1.6 The Bigger Picture

Many kinds of computer architectures have been built that exploit parallelism [55]. In fact, parallel computing has been an indispensable technology in many cutting-edge disciplines for several decades. One of the earliest kinds of parallel systems were the powerful and expensive vector computers that used the idea of pipelining instructions to apply the same operation to many data objects in turn (e.g., Cyber-205 [114], CRAY-1 [155], Fujitsu Facom VP-200 [135]). These systems dominated the high end of computing for several decades, and machines of this kind are still deployed. Other platforms were built that simultaneously applied the same operation to many data objects (e.g. CM2 [80], MasPar [140]). Many systems have been produced that connect multiple independent computers via a network; both proprietary and off-the-shelf networks have been deployed. Early products based on this approach include Intel's iPSC series [28] and machines built by nCUBE and Meiko [22]. Memory is associated with each of the individual computers in the network and is thus distributed across the machine. These distributed-memory parallel systems are often referred to as *massively parallel computers* (MPPs) because very large systems can be put together this way. Information on some of the fastest machines built during the past decade and the technology used to build them can be found at `http://www.top500.org`.

Many MPPs are in use today, especially for scientific computing. If distributed-memory computers are designed with additional support that enables memory to be shared between all the processors, they are also SMPs according to our definition. Such platforms are often called distributed shared-memory computers (DSMs) to emphasize the distinctive nature of this architecture (e.g., SGI Origin [106]). When distributed-memory computers are constructed by using standard workstations or PCs and an off-the-shelf network, they are usually called clusters [169]. Clusters, which are often composed of SMPs, are much cheaper to build than proprietary MPPs. This technology has matured in recent years, so that clusters are common in universities and laboratories as well as in some companies. Thus, although SMPs are the most widespread kind of parallel computer in use, there are many other kinds of parallel machines in the marketplace, particularly for high-end applications.

Figure 1.2 shows the difference in these architectures: in (a) we see a shared-memory system where processors share main memory but have their own private cache; (b) depicts an MPP in which memory is distributed among the processors, or nodes, of the system. The platform in (c) is identical to (b) except for the fact that the distributed memories are accessible to all processors. The cluster in (d) consists of a set of independent computers linked by a network.

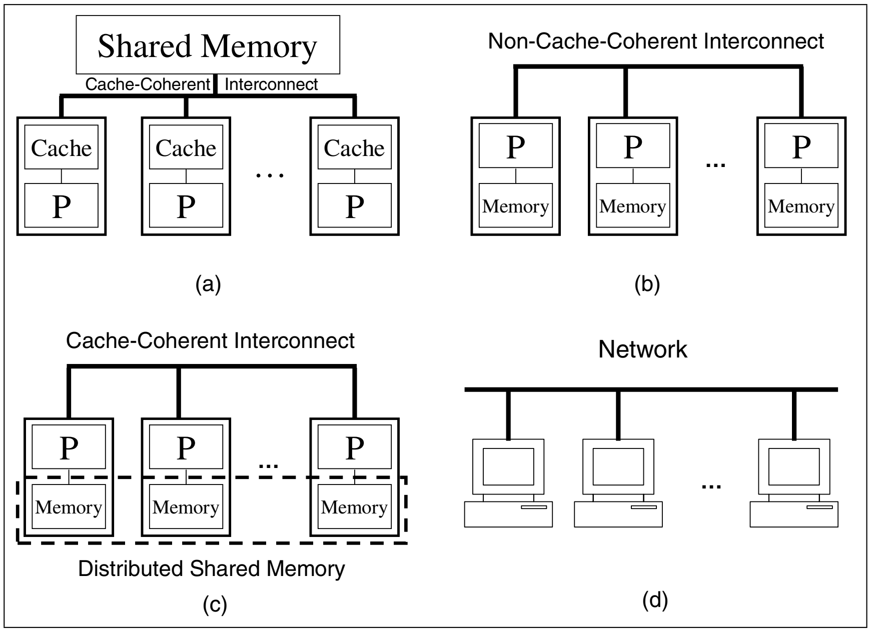
Shared Memory
Cache-Coherent Interconnect
Cache
P
Cache
P
Cache
P
...
(a)
Non-Cache-Coherent Interconnect
P
Memory
P
Memory
P
Memory
...
(b)
Cache-Coherent Interconnect
P
Memory
P
Memory
P
Memory
...
Distributed Shared Memory
(c)
Network
...
(d)

Figure 1.2: **Distributed- and shared-memory computers** – The machine in (a) has physically shared memory, whereas the others have distributed memory. However, the memory in (c) is accessible to all processors.

An equally broad range of applications makes use of parallel computers [61]. Very early adopters of this technology came from such disciplines as aeronautics, aerospace, and chemistry, where vehicles were designed, materials tested, and their properties evaluated long before they were constructed. Scientists in many disciplines have achieved monumental insights into our universe by running parallel programs that model real-world phenomena with high levels of accuracy. Theoretical results were confirmed in ways that could not be done via experimentation. Parallel computers have been used to improve the design and production of goods from automobiles to packaging for refrigerators and to ensure that the designs comply with pricing and behavioral constraints, including regulations. They have been used to study natural phenomena that we are unable to fully observe, such as the formation of galaxies and the interactions of molecules. But they are also routinely used in weather forecasting, and improvements in the accuracy of our daily forecasts are mainly the result of deploying increasingly fast (and large) parallel computers. More recently, they have been widely used in Hollywood and elsewhere to generate highly realistic film sequences and special effects. In this context, too, the ability to build bigger parallel computers has led to higher-quality results, here in the form of more realistic imagery. Of course, parallel computers are also used to digitally remaster old film and to perform many other tasks involving image processing. Other areas using substantial parallel computing include drug design, financial and economic forecasting, climate modeling, surveillance, and medical imaging. It is routine in many areas of engineering, chemistry, and physics, and almost all commercial databases are able to exploit parallel machines.

1.7 Parallel Programming Models

Just as there are several different classes of parallel hardware, so too are there several distinct models of parallel programming. Each of them has a number of concrete realizations. OpenMP realizes a shared-memory (or shared address space) programming model. This model assumes, as its name implies, that programs will be executed on one or more processors that share some or all of the available memory. Shared-memory programs are typically executed by multiple independent threads (execution states that are able to process an instruction stream); the threads share data but may also have some additional, private data. Shared-memory approaches to parallel programming must provide, in addition to a normal range of instructions, a means for starting up threads, assigning work to them, and coordinating their accesses to shared data, including ensuring that certain operations are performed by only one thread at a time [15].

A different programming model has been proposed for distributed-memory systems. Generically referred to as "message passing," this model assumes that programs will be executed by one or more processes, each of which has its own private address space [69]. Message-passing approaches to parallel programming must provide a means to initiate and manage the participating processes, along with operations for sending and receiving messages, and possibly for performing special operations across data distributed among the different processes. The pure message-passing model assumes that processes cooperate to exchange messages whenever one of them needs data produced by another one. However, some recent models are based on "single-sided communication." These assume that a process may interact directly with memory across a network to read and write data anywhere on a machine.

Various realizations of both shared- and distributed-memory programming models have been defined and deployed. An ideal API for parallel programming is expressive enough to permit the specification of many parallel algorithms, is easy to use, and leads to efficient programs. Moreover, the more transparent its implementation is, the easier it is likely to be for the programmer to understand how to obtain good performance. Unfortunately, there are trade-offs between these goals and parallel programming APIs differ in the features provided and in the manner and complexity of their implementation. Some are a collection of library routines with which the programmer may specify some or all of the details of parallel execution (e.g., GA [141] and Pthreads [108] for shared-memory programming and MPI for MPPs), while others such as OpenMP and HPF [101] take the form of additional instructions to the compiler, which is expected to utilize them to generate the parallel code.

1.7.1 Realization of Shared- and Distributed-Memory Models

Initially, vendors of both MPPs and SMPs provided their own custom sets of instructions for exploiting the parallelism in their machines. Application developers had to work hard to modify their codes when they were ported from one machine to another. As the number of parallel machines grew and as more and more parallel programs were written, developers began to demand standards for parallel programming. Fortunately, such standards now exist.

MPI, or the Message Passing Interface, was defined in the early 1990s by a group of researchers and vendors who based their work on existing vendor APIs [69, 137, 147]. It provides a comprehensive set of library routines for managing processes and exchanging messages. MPI is widely used in high-end computing, where problems are so large that many computers are needed to attack them. It is comparatively easy to implement on a broad variety of platforms and therefore provides excellent portability. However, the portability comes at a cost. Creating a parallel program based on this API typically requires a major reorganization of the original sequential code. The development effort can be large and complex compared to a compiler-supported approach such as that offered by OpenMP.

One can also combine some programming APIs. In particular, MPI and OpenMP may be used together in a program, which may be useful if a program is to be executed on MPPs that consist of multiple SMPs (possibly with multiple cores each). Reasons for doing so include exploiting a finer granularity of parallelism than possible with MPI, reducing memory usage, or reducing network communication. Various commercial codes have been programmed using both MPI and OpenMP. Combining MPI and OpenMP effectively is nontrivial, however, and in Chapter 6 we return to this topic and to the challenge of creating OpenMP codes that will work well on large systems.

1.8 Ways to Create Parallel Programs

In this section, we briefly compare OpenMP with the most important alternatives for programming shared-memory machines. Some vendors also provide custom APIs on their platforms. Although such APIs may be fast (this is, after all, the purpose of a custom API), programs written using them may have to be substantially rewritten to function on a different machine. We do not consider APIs that were not designed for broad use.

Automatic parallelization: Many compilers provide a flag, or option, for automatic program parallelization. When this is selected, the compiler analyzes the program, searching for independent sets of instructions, and in particular for loops whose iterations are independent of one another. It then uses this information to generate explicitly parallel code. One of the ways in which this could be realized is to generate OpenMP directives, which would enable the programmer to view and possibly improve the resulting code. The difficulty with relying on the compiler to detect and exploit parallelism in an application is that it may lack the necessary information to do a good job. For instance, it may need to know the values that will be assumed by loop bounds or the range of values of array subscripts: but this is often unknown ahead of run time. In order to preserve correctness, the compiler has to conservatively assume that a loop is not parallel whenever it cannot prove the contrary. Needless to say, the more complex the code, the more likely it is that this will occur. Moreover, it will in general not attempt to parallelize regions larger than loop nests. For programs with a simple structure, it may be worth trying this option.

MPI: The Message Passing Interface [137] was developed to facilitate portable programming for distributed-memory architectures (MPPs), where multiple processes execute independently and communicate data as needed by exchanging messages. The API was designed to be highly expressive and to enable the creation of efficient parallel code, as well as to be broadly implementable. As a result of its success in these respects, it is the most widely used API for parallel programming in the high-end technical computing community, where MPPs and clusters are common. Since most vendors of shared-memory systems also provide MPI implementations that leverage the shared address space, we include it here.

Creating an MPI program can be tricky. The programmer must create the code that will be executed by each process, and this implies a good deal of reprogramming. The need to restructure the entire program does not allow for incremental parallelization as does OpenMP. It can be difficult to create a single program version that will run efficiently on many different systems, since the relative cost of communicating data and performing computations varies from one system to another and may suggest different approaches to extracting parallelism. Care must be taken to avoid certain programming errors, particularly deadlock where two or more processes each wait in perpetuity for the other to send a message. A good introduction to MPI programming is provided in [69] and [147].

Since many MPPs consist of a collection of SMPs, MPI is increasingly mixed with OpenMP to create a program that directly matches the hardware. A recent revision of the standard, MPI-2 ([58]), facilitates their integration.

Pthreads: This is a set of threading interfaces developed by the IEEE (Institute of Electrical and Electronics Engineers) committees in charge of specifying a Portable Operating System Interface (POSIX). It realizes the shared-memory programming model via a collection of routines for creating, managing and coordinating a collection of threads. Thus, like MPI, it is a library. Some features were primarily designed for uniprocessors, where context switching enables a time-sliced execution of multiple threads, but it is also suitable for programming small SMPs. The Pthreads library aims to be expressive as well as portable, and it provides a fairly comprehensive set of features to create, terminate, and synchronize threads and to prevent different threads from trying to modify the same values at the same time: it includes mutexes, locks, condition variables, and semaphores. However, programming with Pthreads is much more complex than with OpenMP, and the resulting code is likely to differ substantially from a prior sequential program (if there is one). Even simple tasks are performed via multiple steps, and thus a typical program will contain many calls to the Pthreads library. For example, to execute a simple loop in parallel, the programmer must declare threading structures, create and terminate the threads individually, and compute the loop bounds for each thread. If interactions occur within loop iterations, the amount of thread-specific code can increase substantially. Compared to Pthreads, the OpenMP API directives make it easy to specify parallel loop execution, to synchronize threads, and to specify whether or not data is to be shared. For many applications, this is sufficient.

1.8.1 A Simple Comparison

The code snippets below demonstrate the implementation of a *dot product* in each of the programming APIs MPI, Pthreads, and OpenMP. We do not explain in detail the features used here, as our goal is simply to illustrate the flavor of each, although we will introduce those used in the OpenMP code in later chapters.

Sequential Dot-Product

```
int main(argc,argv)
int argc;
char *argv[];
{
 double sum;
 double a [256], b [256];
 int n;
 n = 256;
 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }
 sum = 0;
 for (i = 1; i <= n; i++ ) {
    sum = sum + a[i]*b[i];
 }
 printf ("sum = %f", sum);
}
```

The sequential program multiplies the individual elements of two arrays and saves the result in the variable `sum`; `sum` is a so-called reduction variable.

Dot-Product in MPI

```
int main(argc,argv)
int argc;
char *argv[];
{
 double sum, sum_local;
 double a [256], b [256];
 int n, numprocs, myid, my_first, my_last;

 n = 256;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 my_first = myid *  n/numprocs;
 my_last = (myid + 1) *  n/numprocs;

 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }
 sum_local = 0;
 for (i = my_first; i < my_last; i++) {
      sum_local = sum_local + a[i]*b[i];
 }
 MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM,
                                        MPI_COMM_WORLD);
 if (iam==0) printf ("sum = %f", sum);
}
```

Under MPI, all data is local. To implement the dot-product, each process builds a partial sum, the sum of its local data. To do so, each executes a portion of the original loop. Data and loop iterations are accordingly manually shared among processors by the programmer. In a subsequent step, the partial sums have to be communicated and combined to obtain the global result. MPI provides the global communication routine `MPI_Allreduce` for this purpose.

Dot-Product in Pthreads

```
#define NUMTHRDS 4
double sum;
double a [256], b [256];
int status;
int n=256;
pthread_t thd[NUMTHRDS];
pthread_mutex_t mutexsum;

int main(argc,argv)
int argc;
char *argv[];
{

 pthread_attr_t attr;
 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }

 thread_mutex_init(&mutexsum, NULL);
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

 for(i=0;i<NUMTHRDS;i++)
 {
   pthread_create( &thds[i], &attr, dotprod, (void *)i);
 }

 pthread_attr_destroy(&attr);

 for(i=0;i<NUMTHRDS;i++) {
   pthread_join( thds[i], (void **)&status);
 }

 printf ("sum =  %f \n", sum);
 pthread_mutex_destroy(&mutexsum);
 pthread_exit(NULL);
}


void *dotprod(void *arg)
{
  int myid, i, my_first, my_last;
  double sum_local;

  myid = (int)arg;
  my_first = myid *  n/NUMTHRDS;
  my_last = (myid + 1) *  n/NUMTHRDS;

  sum_local = 0;
  for (i = my_first; i <= my_last; i++) {
    sum_local = sum_local + a [i]*b[i];
  }
  pthread_mutex_lock (&mutex_sum);
  sum = sum + sum_local;
  pthread_mutex_unlock (&mutex_sum);

  pthread_exit((void*) 0);
}
```

In the Pthreads programming API, all data is shared but logically distributed among the threads. Access to globally shared data needs to be explicitly synchronized by the user. In the dot-product implementation shown, each thread builds a partial sum and then adds its contribution to the global sum. Access to the global sum is protected by a lock so that only one thread at a time updates this variable. We note that the implementation effort in Pthreads is as high as, if not higher than, in MPI.

Dot-Product in OpenMP

```
int main(argc,argv)
int argc; char *argv[];
{
 double sum;
 double a [256], b [256];
 int status;
 int n=256;

 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }

 sum = 0;
 #pragma omp for reduction(+:sum)
 for (i = 1; i <= n; i++ ) {
    sum = sum + a[i]*b[i];
 }
 printf ("sum =  %f \n", sum);
}
```

Under OpenMP, all data is shared by default. In this case, we are able to parallelize the loop simply by inserting a directive that tells the compiler to parallelize it, and identifying `sum` as a reduction variable. The details of assigning loop iterations to threads, having the different threads build partial sums and their accumulation into a global sum are left to the compiler. Since (apart from the usual variable declarations and initializations) nothing else needs to be specified by the programmer, this code fragment illustrates the simplicity that is possible with OpenMP.

1.9 A Final Word

Given the trend toward bigger SMPs and multithreading computers, it is vital that strategies and techniques for creating shared-memory parallel programs become widely known. Explaining how to use OpenMP in conjunction with the major programming languages Fortran, C, and C++ to write such parallel programs is the purpose of this book. Under OpenMP, one can easily introduce threading in such a way that the same program may run sequentially as well as in parallel. The application developer can rely on the compiler to work out the details of the parallel code or may decide to explicitly assign work to threads. In short, OpenMP is a very flexible medium for creating parallel code.

The discussion of language features in this book is based on the OpenMP 2.5 specification, which merges the previously separate specifications for Fortran and C/C++. At the time of writing, the ARB is working on the OpenMP 3.0 specification, which will expand the model to provide additional convenience and expressivity for the range of architectures that it supports. Further information on this, as well as up-to-date news, can be found at the ARB website `http://www.openmp.org` and at the website of its user community, `http://www.compunity.org`. The complete OpenMP specification can also be downloaded from the ARB website.