Using OpenMP

PORTABLE SHARED MEMORY PARALLEL PROGRAMMING

BARBARA CHAPMAN,
GABRIELE JOST,
AND RUUD VAN DER PAS

foreword by
DAVID J. KUCK

2 Overview of OpenMP

In this chapter we give an overview of the OpenMP programming interface and compare it with other approaches to parallel programming for SMPs.

2.1 Introduction

The OpenMP Application Programming Interface (API) was developed to enable portable shared memory parallel programming. It aims to support the parallelization of applications from many disciplines. Moreover, its creators intended to provide an approach that was relatively easy to learn as well as apply. The API is designed to permit an incremental approach to parallelizing an existing code, in which portions of a program are parallelized, possibly in successive steps. This is a marked contrast to the all-or-nothing conversion of an entire program in a single step that is typically required by other parallel programming paradigms. It was also considered highly desirable to enable programmers to work with a single source code: if a single set of source files contains the code for both the sequential and the parallel versions of a program, then program maintenance is much simplified. These goals have done much to give the OpenMP API its current shape, and they continue to guide the OpenMP Architecture Review Board (ARB) as it works to provide new features.

2.2 The Idea of OpenMP

A thread is a runtime entity that is able to independently execute a stream of instructions. OpenMP builds on a large body of work that supports the specification of programs for execution by a collection of cooperating threads [15]. The operating system creates a process to execute a program: it will allocate some resources to that process, including pages of memory and registers for holding values of objects. If multiple threads collaborate to execute a program, they will share the resources, including the address space, of the corresponding process. The individual threads need just a few resources of their own: a program counter and an area in memory to save variables that are specific to it (including registers and a stack). Multiple threads may be executed on a single processor or core via context switches; they may be interleaved via simultaneous multithreading. Threads running simultaneously on multiple processors or cores may work concurrently to execute a parallel program.

Multithreaded programs can be written in various ways, some of which permit complex interactions between threads. OpenMP attempts to provide ease of programming and to help the user avoid a number of potential programming errors,

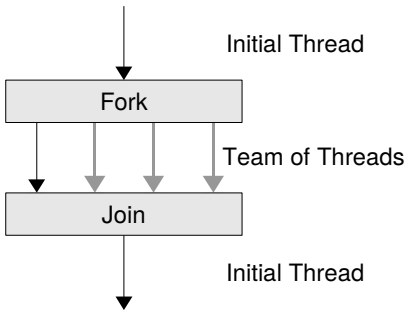

Figure 2.1: **The fork-join programming model supported by OpenMP** – The program starts as a single thread of execution, the initial thread. A team of threads is forked at the beginning of a parallel region and joined at the end.

by offering a structured approach to multithreaded programming. It supports the so-called fork-join programming model [48], which is illustrated in Figure 2.1. Under this approach, the program starts as a single thread of execution, just like a sequential program. The thread that executes this code is referred to as the *initial thread*. Whenever an OpenMP parallel construct is encountered by a thread while it is executing the program, it creates a team of threads (this is the *fork*), becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct. At the end of the construct, only the original thread, or master of the team, continues; all others terminate (this is the *join*). Each portion of code enclosed by a parallel construct is called a parallel region.

OpenMP expects the application developer to give a high-level specification of the parallelism in the program and the method for exploiting that parallelism. Thus it provides notation for indicating the regions of an OpenMP program that should be executed in parallel; it also enables the provision of additional information on how this is to be accomplished. The job of the OpenMP implementation is to sort out the low-level details of actually creating independent threads to execute the code and to assign work to them according to the strategy specified by the programmer.

2.3 The Feature Set

The OpenMP API comprises a set of compiler directives, runtime library routines, and environment variables to specify shared-memory parallelism in Fortran and C/C++ programs. An OpenMP directive is a specially formatted comment or pragma that generally applies to the executable code immediately following it in the program. A directive or OpenMP routine generally affects only those threads that encounter it. Many of the directives are applied to a *structured block* of code, a sequence of executable statements with a single entry at the top and a single exit at the bottom in Fortran programs, and an executable statement in C/C++ (which may be a compound statement with a single entry and single exit). In other words, the program may not branch into or out of blocks of code associated with directives. In Fortran programs, the start and end of the applicable block of code are explicitly marked by OpenMP directives. Since the end of the block is explicit in C/C++, only the start needs to be marked.

OpenMP provides means for the user to

- create teams of threads for parallel execution,
- specify how to share work among the members of a team,
- declare both shared and private variables, and
- synchronize threads and enable them to perform certain operations exclusively (i.e., without interference by other threads).

In the following sections, we give an overview of the features of the API. In subsequent chapters we describe these features and show how they can be used to create parallel programs.

2.3.1 Creating Teams of Threads

A team of threads is created to execute the code in a parallel region of an OpenMP program. To accomplish this, the programmer simply specifies the parallel region by inserting a `parallel` directive immediately before the code that is to be executed in parallel to mark its start; in Fortran programs, the end is also marked by an `end parallel` directive. Additional information can be supplied along with the `parallel` directive. This is mostly used to enable threads to have private copies of some data for the duration of the parallel region and to initialize that data. At the end of a parallel region is an implicit barrier synchronization: this means that no thread can progress until all other threads in the team have reached that point in

the program. Afterwards, program execution continues with the thread or threads that previously existed. If a team of threads executing a parallel region encounters another `parallel` directive, each thread in the current team creates a new team of threads and becomes its master. Nesting enables the realization of multilevel parallel programs.

OpenMP is commonly used to incrementally parallelize an existing sequential code, and this task is most easily accomplished by creating parallel regions one at a time.

2.3.2 Sharing Work among Threads

If the programmer does not specify how the work in a parallel region is to be shared among the executing threads, they will each redundantly execute all of the code. This approach does not speed up the program. The OpenMP work-sharing directives are provided for the programmer to state how the computation in a structured block of code is to be distributed among the threads. Unless explicitly overridden by the programmer, an implicit barrier synchronization also exists at the end of a work-sharing construct. The choice of work-sharing method may have a considerable bearing on the performance of the program.

Work Sharing and Loops Probably the most common work-sharing approach is to distribute the work in a `DO` (Fortran) or `for` (C/C++) loop among the threads in a team. To accomplish this, the programmer inserts the appropriate directive immediately before each loop within a parallel region that is to be shared among threads. Work-sharing directives cannot be applied to all kinds of loops that occur in C/C++ code. Many programs, especially scientific applications, spend a large fraction of their time in loops performing calculations on array elements and so this strategy is widely applicable and often very effective.

All OpenMP strategies for sharing the work in loops assign one or more disjoint sets of iterations to each thread. The programmer may specify the method used to partition the iteration set. The most straightforward strategy assigns one contiguous *chunk of iterations* to each thread. More complicated strategies include dynamically computing the next chunk of iterations for a thread. If the programmer does not provide a strategy, then an implementation-defined default will be used.

When Can the Work Be Shared? Not every loop can be parallelized in this fashion. It must be possible to determine the number of iterations in the loop

upon entry, and this number may not change while the loop is executing. Loops implemented via a `while` construct, for example, may not satisfy this condition. Furthermore, a loop is suitable for sharing among threads only if its iterations are independent. By this, we mean that the order in which the iterations are performed has no bearing on the outcome. The job of the OpenMP programmer is to determine whether this is the case, and sometimes a loop needs to be modified somewhat to accomplish this.

Consider the nested loop in Fortran below which is typical for applications from the field of computational fluid dynamics.

```
!$OMP PARALLEL
!$OMP DO
   do 10 j = 1, jmax
      i = 1
      vv (i,j) = v (i,j,m)
      do 60 i = 2, imax-1
         vv (i,j) = vv (i-1,j) + b(i,j)
 60   continue
      i = imax
      vv (i,j) = vv(i-1,j)
      do 100 i = imax-1, 1, -1
         vv (i,j) = vv (i+1,j) + a (i,j)
 100  continue
 10  continue
!$OMP END DO
!$OMP END PARALLEL
```

Loops 60 and 100 are not suitable for parallelization via work sharing. In loop 60, the `i`th iteration writes array element `vv(i,j)` and in the `i+1`th iteration this value is used. Thus iteration `i+1` depends on the outcome of iteration `i`, and the order in which they are executed must be maintained. A similar situation occurs with regard to loop 100 where, for example, the second iteration writes `vv(imax-2,j)`. But this calculation uses the value of `vv(imax-1,j)` that was computed in the previous iteration. All is not lost, however. With some thought one can see that the outermost `j` loop is independent. Thus we can share the iterations of this loop among the executing threads. The two comment lines at the start and end of the example are all that is needed to achieve this in OpenMP.

The data reuse patterns illustrated in the inner loops are examples of *data dependence* [25]. Data dependences prevent us from parallelizing a loop when a value

that is written in one loop iteration is either written or read in another iteration. The more complex the loop, the harder it can be to decide whether such dependences exist. Keep in mind that the correct placement of the OpenMP directives is the user's responsibility and that on occasion some insight into the application is needed to decide whether there really is a dependence.

Other Work-Sharing Strategies Other approaches may be used to assign work to threads within a parallel region. One approach consists of giving distinct pieces of work to the individual threads. This approach is suitable when independent computations are to be performed and the order in which they are carried out is irrelevant. It is straightforward to specify this by using the corresponding OpenMP directive. Just as before, the programmer must ensure that the computations can truly be executed in parallel. It is also possible to specify that just one thread should execute a block of code in a parallel region.

Moreover, the OpenMP API contains a directive for sharing the work in Fortran 90 array statements among threads. It also works for the Fortran 95 `forall` construct. Such operations on an entire array contain considerable exploitable parallelism. The method of distributing the work to threads will be determined by the compiler in this case.

2.3.3 The OpenMP Memory Model

OpenMP is based on the shared-memory model; hence, by default, data is shared among the threads and is visible to all of them. Sometimes, however, one needs variables that have thread-specific values. When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, we say that the variable is *private*. For example, when a team of threads executes a parallel loop, each thread needs its own value of the iteration variable. This case is so important that the compiler enforces it; in other cases the programmer must determine which variables are shared and which are private. Data can be declared to be shared or private with respect to a parallel region or work-sharing construct.[1]

The use of private variables can be beneficial in several ways. They can reduce the frequency of updates to shared memory. Thus, they may help avoid hot spots, or competition for access to certain memory locations, which can be expensive if large numbers of threads are involved. They can also reduce the likelihood of remote data accesses on cc-NUMA platforms and may remove the need for some

[1] OpenMP also allows private data objects to persist between parallel regions.

synchronizations. The downside is that they will increase the program's memory footprint.

Threads need a place to store their private data at run time. For this, each thread has its own special region in memory known as the thread stack. The application developer may safely ignore this detail, with one exception: most compilers give the thread stack a default size. But sometimes the amount of data that needs to be saved there can grow to be quite large, as the compiler will use it to store other information, too. Then the default size may not be large enough. Fortunately, the application developer is usually provided with a means to increase the size of the thread stack (for details, the manual should be consulted). Dynamically declared data and persistent data objects require their own storage area. For those who would like to know a little more about this, we describe how data is stored and retrieved at run time in Chapter 8.

In Section 1.2.1 of Chapter 1, we explained that each processor of an SMP has a small amount of private memory called cache. In Section 1.2.2 we showed that this could potentially lead to trouble where shared data is concerned, as the new value of a shared object might be in a cache somewhere on the system instead of main memory, where other threads can access it. Fortunately, the OpenMP programmer does not need to know how a specific system deals with this problem, since OpenMP has its own rules about when shared data is visible to (or accessible by) all threads. These rules state that the values of shared objects must be made available to all threads at synchronization points. (We explain what these are in the next section.) Between the synchronization points, threads may temporarily keep their updated values in local cache. As a result, threads may *temporarily* have different values for some shared objects. If one thread needs a value that was created by another thread, then a synchronization point must be inserted into the code.

OpenMP also has a feature called `flush`, discussed in detail in Section 4.9.2, to synchronize memory. A `flush` operation makes sure that the thread calling it has the same values for shared data objects as does main memory. Hence, new values of any shared objects updated by that thread are written back to shared memory, and the thread gets any new values produced by other threads for the shared data it reads. In some programming languages, a `flush` is known as a *memory fence*, since reads and writes of shared data may not be moved relative to it.

2.3.4 Thread Synchronization

Synchronizing, or coordinating the actions of, threads is sometimes necessary in order to ensure the proper ordering of their accesses to shared data and to prevent

data corruption. Many mechanisms have been proposed to support the synchronization needs of a variety of applications [51, 50, 81]. OpenMP has a rather small set of fairly well understood synchronization features.

Ensuring the required thread coordination is one of the toughest challenges of shared-memory programming. OpenMP attempts to reduce the likelihood of synchronization errors, and to make life easier for the programmer, provides for implicit synchronization. By default, OpenMP gets threads to wait at the end of a worksharing construct or parallel region until all threads in the team executing it have finished their portion of the work. Only then can they proceed. This is known as a barrier. Synchronizing the actions of a subset of threads is harder to accomplish in OpenMP and requires care in programming because there is no explicit support for this.

Sometimes a programmer may need to ensure that only one thread at a time works on a piece of code. OpenMP has several mechanisms that support this kind of synchronization. For example, if a thread attempts to execute code that is protected by a such a feature, and it is already being executed by another thread, then the former will have to wait for its turn. Alternatively, it may be possible for it to carry out other work while waiting. If it suffices to protect updates to an individual variable (more precisely, a memory location), it may be more efficient to employ the atomic update feature provided by OpenMP.

Synchronization points are those places in the code where synchronization has been specified, either explicitly or implicitly. They have an additional function in OpenMP code, as we have just learned: at these places in the code, the system ensures that threads have consistent values of shared data objects. OpenMP's synchronization points include explicit and implicit barriers, the start and end of critical regions, points where locks are acquired or released, and anywhere the programmer has inserted a *flush* directive.

2.3.5 Other Features to Note

Procedures Subroutines and functions can complicate the use of parallel programming APIs. In order to accommodate them, major changes to a program may sometimes be needed. One of the innovative features of OpenMP is the fact that directives may be inserted into the procedures that are invoked from within a parallel region. These have come to be known as *orphan directives*, a term that indicates that they are not in the routine in which the parallel region is specified. (For compiler buffs, this means that they are not within the lexical extent of the parallel construct.)

Number of Threads and Thread Numbers For some applications, it can be important to control the number of threads that execute a parallel region. OpenMP lets the programmer specify this number prior to program execution via an environment variable, after the computation has begun via a library routine, or at the start of a parallel region. If this is not done, then the implementation must choose the number of threads that will be used. Some programs need to use the number of threads in a team to set the values of certain variables. Other programs may want to assign computation to a specific thread. OpenMP assigns consecutive numbers, starting from 0, to each thread in a team in order to identify them. There are library routines for retrieving the number of threads as well as for enabling a thread to access its own thread number.

OpenMP has a couple of features that may affect the number of threads in a team. First, it is possible to permit the execution environment to dynamically vary the number of threads, in which case a team may have fewer than the specified number of threads, possibly as a result of other demands made on the system's resources. The default behavior in this regard is implementation defined. Once the number of threads in a given team has been created, however, that number will not change. Second, an implementation is permitted to disallow nested parallelism, or it may be disabled by the programmer.

2.4 OpenMP Programming Styles

OpenMP encourages structured parallel programming and relies heavily on distributing the work in loops among threads. But sometimes the amount of loop-level parallelism in an application is limited. Sometimes parallelization using OpenMP directives leads to unacceptable overheads. Particularly when the application or the number of threads to be used is relatively large, an alternative method of using OpenMP may be beneficial. One can also write OpenMP programs that do not rely on work-sharing directives but rather *assign work explicitly to different threads* using their thread numbers. This approach can lead to highly efficient code. However, the programmer must then insert synchronization manually to ensure that accesses to shared data are correctly controlled. In this mode, programming errors such as deadlock (when all threads wait for each other in perpetuity) may occur and must be avoided via careful code design.

This approach can help solve a broad variety of programming problems. For instance, it may be used to give threads slightly different amounts of work if one knows that the operating system will take up some cycles on a processor. A particularly popular style of programming that can achieve high efficiency based on this

approach is suitable for parallelizing programs working on a computational domain that can be subdivided. With it, the user explicitly creates the subdomains (a strategy sometimes called domain decomposition) and assigns them to the threads. Each thread then works on its portion of the data. This strategy is often referred to as SPMD (single program muliple data) programming.

Those approaches that require manual assignment of work to threads and that need explicit synchronization are often called "low-level programming." This style of programming can be very effective and it is broadly applicable, but it requires much more development effort and implies more care on the part of the developer to ensure program correctness. OpenMP provides sufficient features to permit such a low-level approach to parallel program creation.

2.5 Correctness Considerations

One of the major difficulties of shared-memory parallel programming is the effort required to ensure that a program is correct. In addition to all the sources of errors that may arise in a sequential program, shared-memory programs may contain new, and sometimes devious, bugs. Fortunately, the use of directives and a structured programming style is sufficient to prevent many problems; when the programmer adopts a low-level style of programming, however, more care is needed.

One kind of error in particular, a *data race condition,*[2] can be extremely difficult to detect and manifests itself in a shared-memory parallel code through silent data corruption. Unfortunately, the runtime behavior of a program with a data race condition is not reproducible: erroneous data may be produced by one program run, but the problem may not show up the next time it is executed. The problem arises when two or more threads access the same shared variable without any synchronization to order the accesses, and at least one of the accesses is a write. Since it is relatively easy, for example, to create a parallel version of a loop nest without noticing that multiple iterations reference the same array element, the programmer must be aware of the impact that this may have. In general, the more complex the code, the harder it is to guarantee that no such errors have been introduced.

The order of operations actually observed in a code with a data race condition depends on the load on a system and the relative timing of the threads involved. Since threads may execute their instructions at slightly different speeds, and the work of the operating system sometimes affects the performance of one or more threads, the order in which they reach certain code is observed to vary from one

[2]This is sometimes also referred to as "data race" or "race condition."

run to another. In some instances, the problem occurs only for a specific number of threads. As a result, a data race bug may escape detection during testing and even for some time when a program is in production use. Thus, the importance of avoiding this problem by paying careful attention during program development can not be overemphasized. Tools may help by pointing out potential problems.

One can identify other potential causes of errors in OpenMP programs. For instance, if the programmer has relied on program execution by a certain number of threads and if a different number is used to execute the code, perhaps because insufficient resources are available, then results may be unexpected. OpenMP generally expects the programmer to check that the execution environment is just what was required, and provides runtime routines for such queries.

Incorrect use of synchronization constructs leads to problems that may not be readily apparent. To avoid them, one must carefully think through the logic of explicit synchronization in a program and must exercise special care with the use of low-level synchronization constructs, such as locks.

2.6 Performance Considerations

How much reduction of the execution time can be expected from OpenMP parallelization and, indeed, by shared-memory parallelization? If we denote by T_1 the execution time of an application on 1 processor, then in an ideal situation, the execution time on P processors should be T_1/P. If T_P denotes the execution time on P processors, then the ratio

$$S = T_1/T_P \tag{2.1}$$

is referred to as the parallel speedup and is a measure for the success of the parallelization. However, a number of obstacles usually have to be overcome before perfect speedup, or something close to it, is achievable. Virtually all programs contain some regions that are suitable for parallelization and other regions that are not. By using an increasing number of processors, the time spent in the parallelized parts of the program is reduced, but the sequential section remains the same. Eventually the execution time is completely dominated by the time taken to compute the sequential portion, which puts an upper limit on the expected speedup. This effect, known as *Amdahl's law*, can be formulated as

$$S = \frac{1}{(f_{par}/P + (1 - f_{par}))}, \tag{2.2}$$

where f_{par} is the parallel fraction of the code and P is the number of processors. In the ideal case when all of the code runs in parallel, $f_{par} = 1$, the expected speedup

is equal to the number of processors. If only 80 percent of the code runs in parallel ($f_{par} = 0.8$), the maximal speedup one can expect on 16 processors is 4 and on 32 processors is 4.4. Frustrating, isn't it? It is thus important to parallelize as much of the code as possible, particularly if large numbers of processors are to be exploited.

Other obstacles along the way to perfect linear speedup are the overheads introduced by forking and joining threads, thread synchronization, and memory accesses. On the other hand, the ability to fit more of the program's data into cache may offset some of the overheads. A measure of a program's ability to decrease the execution time of the code with an increasing number of processors is referred to as *parallel scalability*.

Note that if the application developer has a specific goal in terms of required performance improvement, it may be possible to select several regions for parallelization, and to ignore the rest, which remains sequential. Note, too, that parallel speedup is often defined as the improvement in performance relative to the "best" sequential algorithm for the problem at hand. This measure indicates whether parallelization provides benefits that could not be obtained by choosing a different approach to solving the problem.

2.7 Wrap-Up

In this chapter, we have given a brief overview of the features that are available in the OpenMP API. In the following chapters, we discuss each of these features in detail, giving their purpose, syntax, examples of their use, and, where necessary, further rules on how they may be applied.

4 OpenMP Language Features

In this chapter we introduce the constructs and user-level functions of OpenMP, giving syntax and information on their usage.

4.1 Introduction

OpenMP provides directives, library functions, and environment variables to create and control the execution of parallel programs. In Chapter 3 we informally introduced a few of its most important constructs. In this chapter, we give a fairly extensive overview of the language features, including examples that demonstrate their syntax, usage, and, where relevant, behavior. We show how these OpenMP constructs and clauses are used to tackle some programming problems.

A large number of applications can be parallelized by using relatively few constructs and one or two of the functions. Those readers familiar with MPI will be aware that, despite the relatively large number of features provided by that parallel programming API, just half a dozen of them are really indispensable [69]. OpenMP is a much smaller API than MPI, so it is not all that difficult to learn the entire set of features; but it is similarly possible to identify a short list of constructs that a programmer really should be familiar with. We begin our overview by presenting and discussing a limited set that suffices to write many different programs in Sections 4.3, 4.4, and 4.5. This set comprises the following constructs, some of the clauses that make them powerful, and (informally) a few of the OpenMP library routines:

- Parallel Construct
- Work-Sharing Constructs
    1. Loop Construct
    2. Sections Construct
    3. Single Construct
    4. Workshare Construct (Fortran only)
- Data-Sharing, No Wait, and Schedule Clauses

Next, we introduce the following features, which enable the programmer to orchestrate the actions of different threads, in Section 4.6:

- Barrier Construct

- Critical Construct
- Atomic Construct
- Locks
- Master Construct

The OpenMP API also includes library functions and environment variables that may be used to control the manner in which a program is executed; these are presented in Section 4.7. The remaining clauses for parallel and work-sharing constructs are reviewed in Section 4.8. In Section 4.9, we complete our presentation of the API with a discussion of a few more specialized features.

Where relevant, we comment on practical matters related to the constructs and clauses, but not all details are covered. It is not our objective to duplicate the OpenMP specification, which can be downloaded from [2]. At times, the wording in this chapter is less formal than that typically found in the official document. Our intent is to stay closer to the terminology used by application developers.

4.2 Terminology

Several terms are used fairly often in the OpenMP standard, and we will need them here also. The definitions of *directive* and *construct* from Section 1.2.2 of the OpenMP 2.5 document are cited verbatim for convenience:

- *OpenMP Directive* - In C/C++, a `#pragma` and in Fortran, a comment, that specifies OpenMP program behavior.
- *Executable directive* - An OpenMP directive that is not declarative; that is, it may be placed in an executable context.[1]
- *Construct* - An OpenMP executable directive (and, for Fortran, the paired `end` directive, if any) and the associated statement, loop, or structured block, if any, not including the code in any called routines, that is, the lexical extent of an executable directive.

OpenMP requires well-structured programs, and, as can be seen from the above, constructs are associated with statements, loops, or structured blocks. In C/C++ a "structured block" is defined to be an executable statement, possibly a compound

[1] All directives except the `threadprivate` directive are executable directives.

statement, with a single entry at the top and a single exit at the bottom. In Fortran, it is defined as a block of executable statements with a single entry at the top and a single exit at the bottom. For both languages, the point of entry cannot be a labeled statement, and the point of exit cannot be a branch of any type.

In C/C++ the following additional rules apply to a structured block:

- The point of entry cannot be a call to `setjmp()`.
- `longjmp()` and `throw()` (C++ only) must not violate the entry/exit criteria.
- Calls to `exit()` are allowed in a structured block.
- An expression statement, iteration statement, selection statement, or try block is considered to be a structured block if the corresponding compound statement obtained by enclosing it in { and } would be a structured block.

In Fortran the following applies:

- `STOP` statements are allowed in a structured block.

Another important concept in OpenMP is that of a *region* of code. This is defined as follows by the standard: "An OpenMP region consists of all code encountered during a specific instance of the execution of a given OpenMP construct or library routine. A region includes any code in called routines, as well as any implicit code introduced by the OpenMP implementation." In other words, a region encompasses all the code that is in the *dynamic* extent of a construct.

Most OpenMP directives are clearly associated with a region of code, usually the dynamic extent of the structured block or loop nest immediately following it. A few (`barrier` and `flush`) do not apply to any code. Some features affect the behavior or use of threads. For these, the notion of a *binding thread set* is introduced. In particular, some of the runtime library routines have an effect on the thread that invokes them (or return information pertinent to that thread only), whereas others are relevant to a team of threads or to all threads that execute the program. We will discuss binding issues only in those few places where it is important or not immediately clear what the binding of a feature is.

4.3 Parallel Construct

Before embarking on our description of the other basic features of OpenMP we introduce the most important one of all. The *parallel construct* plays a crucial role

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

Figure 4.1: **Syntax of the parallel construct in C/C++** – The parallel region implicitly ends at the end of the structured block. This is a closing curly brace (}) in most cases.

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

Figure 4.2: **Syntax of the parallel construct in Fortran** – The terminating `!$omp end parallel` directive is mandatory for the parallel region in Fortran.

in OpenMP: a program without a parallel construct will be executed sequentially. Its C/C++ syntax is given in Figure 4.1; the Fortran syntax is given in Figure 4.2.

This construct is used to specify the computations that should be executed in parallel. Parts of the program that are not enclosed by a parallel construct will be executed serially. When a thread encounters this construct, a team of threads is created to execute the associated parallel region, which is the code dynamically contained within the parallel construct. But although this construct ensures that computations are performed in parallel, it does not distribute the work of the region among the threads in a team. In fact, if the programmer does not use the appropriate syntax to specify this action, the work will be replicated. At the end of a parallel region, there is an implied *barrier* that forces all threads to wait until the work inside the region has been completed. Only the initial thread continues execution after the end of the parallel region. For more information on barriers, we refer to Section 4.6.1 on page 84.

The thread that encounters the parallel construct becomes the *master* of the new team. Each thread in the team is assigned a unique thread number (also referred to as the "thread id") to identify it. They range from zero (for the master thread) up to one less than the number of threads within the team, and they can be accessed by the programmer. Although the parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution. One way to achieve this is to exploit the thread numbers. We give a simple example in Figure 4.3.

Here, the OpenMP library function `omp_get_thread_num()` is used to obtain the number of each thread executing the parallel region.[2] Each thread will execute

[2]We discuss the library functions in Section 4.7; this routine appears in several examples.

```
#pragma omp parallel
   {
     printf("The parallel region is executed by thread %d\n",
        omp_get_thread_num());

     if ( omp_get_thread_num() == 2 ) {
        printf("  Thread %d does things differently\n",
               omp_get_thread_num());
     }
   }  /*-- End of parallel region --*/
```

Figure 4.3: **Example of a parallel region** – All threads execute the first `printf` statement, but only the thread with thread number 2 executes the second one.

all code in the parallel region, so that we should expect each to perform the first print statement. However, only one thread will actually execute the second print statement (assuming there are at least three threads in the team), since we used the thread number to control its execution. The output in Figure 4.4 is based on execution of this code by a team with four threads.[3] Note that one cannot make any assumptions about the order in which the threads will execute the first `printf` statement. When the code is run again, the order of execution could be different.

```
The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
  Thread 2 does things differently
The parallel region is executed by thread 1
```

Figure 4.4: **Output of the code shown in Figure 4.3** – Four threads are used in this example.

We list in Figure 4.5 the clauses that may be used along with the parallel construct. They are discussed in Sections 4.5 and 4.8.

There are several restrictions on the parallel construct and its clauses:

[3]This is an incomplete OpenMP code fragment and requires a wrapper program before it can be executed. The same holds for all other examples throughout this chapter.

if*(scalar-expression)*	(C/C++)
if*(scalar-logical-expression)*	(Fortran)
num_threads*(integer-expression)*	(C/C++)
num_threads*(scalar-integer-expression)*	(Fortran)
private*(list)*	
firstprivate*(list)*	
shared*(list)*	
default(none|shared)	(C/C++)
default(none|shared|private)	(Fortran)
copyin*(list)*	
reduction*(operator:list)*	(C/C++)
reduction*({operator|intrinsic_procedure_name}:list)*	(Fortran)

Figure 4.5: **Clauses supported by the parallel construct** – Note that the default(private) clause is not supported on C/C++.

- A program that branches into or out of a parallel region is nonconforming. In other words, if a program does so, then it is *illegal*, and the behavior is undefined.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive or on any side effects of the evaluations of the clauses.
- At most one `if` clause can appear on the directive.
- At most one `num_threads` clause can appear on the directive. The expression for the clause must evaluate to a positive integer value.

In C++ there is an additional constraint. A `throw` inside a parallel region must cause execution to resume within the same parallel region, *and* it must be caught by the same thread that threw the exception. In Fortran, unsynchronized use of I/O statements by multiple threads on the *same* unit has unspecified behavior.

Section 4.7 explains how the programmer may specify how many threads should be in the team that executes a parallel region. This number cannot be modified once the team has been created. Note that under exceptional circumstances, for example, a lack of hardware resources, an implementation is permitted to provide fewer than the requested number of threads. Thus, the application may need to check on the number actually assigned for its execution.

The OpenMP standard distinguishes between an *active* parallel region and an *inactive* parallel region. A parallel region is active if it is executed by a team of

threads consisting of more than one thread. If it is executed by one thread only, it has been serialized and is considered to be inactive. For example, one can specify that a parallel region be conditionally executed, in order to be sure that it contains enough work for this to be worthwhile (see Section 4.8.1 on page 100). If the condition does not hold at run time, then the parallel region will be inactive. A parallel region may also be inactive if it is nested within another parallel region and this feature is either disabled or not provided by the implementation (see Section 4.7 and Section 4.9.1 for details).

4.4 Sharing the Work among Threads in an OpenMP Program

OpenMP's work-sharing constructs are the next most important feature of OpenMP because they are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. Fortran has one more. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active `parallel` region in order to have an effect. If a work-sharing directive is encountered in an inactive `parallel` region or in the sequential part of the program, it is simply ignored. Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others.

The work-sharing constructs are listed in Figure 4.6. For the sake of readability, the clauses have been omitted. These are discussed in Section 4.5 and Section 4.8.

Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations over the threads	**#pragma omp for**	**!$omp do**
Distribute independent work units	**#pragma omp sections**	**!$omp sections**
Only one thread executes the code block	**#pragma omp single**	**!$omp single**
Parallelize array-syntax		**!$omp workshare**

Figure 4.6: **OpenMP work-sharing constructs** – These constructs are simple, yet powerful. Many applications can be parallelized by using just a parallel region and one or more of these constructs, possibly with clauses. The `workshare` construct is available in Fortran only. It is used to parallelize Fortran array statements.

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work. However, the programmer can suppress this by using the `nowait` clause (see Section 4.5 for more details).

4.4.1 Loop Construct

The *loop construct* causes the iterations of the loop immediately following it to be executed in parallel. At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features. Its syntax is shown in Figure 4.7 for C/C++ and in Figure 4.8 for Fortran.

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

Figure 4.7: **Syntax of the loop construct in C/C++** – Note the lack of curly braces. These are implied with the construct.

```
!$omp do [clause[[,] clause]...]
    do-loop
[!$omp end do [nowait]]
```

Figure 4.8: **Syntax of the loop construct in Fortran** – The terminating `!$omp end do` directive is optional, but we recommend using it to clearly mark the end of the construct.

In C and C++ programs, the use of this construct is limited to those kinds of loops where the number of iterations can be counted; that is, the loop must have an integer counter variable whose value is incremented (or decremented) by a fixed amount at each iteration until some specified upper (or lower) bound is reached. In particular, this restriction excludes loops that process the items in a list.

The loop header must have the general form shown in Figure 4.9, where *init-expr* stands for the initialization of the loop counter `var` via an integer expression, `b` is

for (*init-expr* ; *var relop b* ; *incr-expr*)

Figure 4.9: **Format of C/C++ loop** – The OpenMP loop construct may be applied only to this kind of loop nest in C/C++ programs.

also an integer expression, and *relop* is one of the following: `<, <=, >, >=`. The *incr-expr* is a statement that increments or decrements `var` by an integer amount using a standard operator (++, –, +=, -=). Alternatively, it may take a form such as *var = var + incr*. Many examples of this kind of loop are presented here and in the following chapters.

We illustrate this construct in Figure 4.10, where we use a parallel directive to define a parallel region and then share its work among threads via the `for` work-sharing directive: the `#pragma omp for` directive states that iterations of the loop following it will be distributed. Within the loop, we again use the OpenMP function `omp_get_thread_num()`, this time to obtain and print the number of the executing thread in each iteration. Note that we have added clauses to the parallel construct that state which data in the region is shared and which is private. Although not strictly needed since this is enforced by the compiler, loop variable `i` is explicitly declared to be a private variable, which means that each thread will have its own copy of `i`. Unless the programmer takes special action (see `lastprivate` in Section 4.5.3), its value is also undefined after the loop has finished. Variable `n` is made shared. We discuss shared and private data in Sections 4.5.1 and 4.5.2.

```
#pragma omp parallel shared(n) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)
      printf("Thread %d executes loop iteration %d\n",
             omp_get_thread_num(),i);
} /*-- End of parallel region --*/
```

Figure 4.10: **Example of a work-sharing loop** – Each thread executes a subset of the total iteration space $i = 0, \ldots, n-1$.

In Figure 4.11, we also give output produced when we executed the code of Figure 4.10 using four threads. Given that this is a parallel program, we should not expect the results to be printed in a deterministic order. Indeed, one can easily see that the order in which the `printf` statements are executed is not sorted with respect to the thread number. Note that threads 1, 2, and 3 execute two loop

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Figure 4.11: **Output from the example shown in Figure 4.10** – The example is executed for $n = 9$ and uses four threads.

iterations each. Since the total number of iterations is 9 and since four threads are used, one thread has to execute the additional iteration. In this case it turns out to be thread 0, the so-called master thread, which has done so.

The implementer must decide how to select a thread to execute the remaining iteration(s), and the choice may even change between various releases of the same compiler. In fact, if the programmer does not say how to map the iterations to threads, the compiler must decide what strategy should be used for this. Potentially, it could even choose a different mapping strategy for different loops in the same application. Another of the clauses, the `schedule` clause (see Section 4.5.7 on page 79), is the means by which the programmer is able to influence this mapping. Our second example in Figure 4.12 contains two work-shared loops, or parallel loops. The second loop uses values of `a` that are defined in the first loop. As mentioned above, the compiler does not necessarily map iterations of the second loop in the same way as it does for the first loop. But since there is an implied barrier at the end of a parallel loop, we can be certain that all of the values of `a` have been created by the time we begin to use them.

The clauses supported by the loop construct are listed in Figure 4.13.

4.4.2 The Sections Construct

The *sections construct* is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which will be executed by one of the threads. It consists of two directives: first, `#pragma omp sections` in C/C++ (and `!$omp sections` in Fortran) to indicate the start of the construct (along with a termination directive in Fortran), and second, the `#pragma omp section` directive in C/C++ and `!$omp section` in For-

```
#pragma omp parallel shared(n,a,b) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)
      a[i] = i;

   #pragma omp for
   for (i=0; i<n; i++)
      b[i] = 2 * a[i];
} /*-- End of parallel region --*/
```

Figure 4.12: **Two work-sharing loops in one parallel region** – One can not assume that the distribution of iterations to threads is identical for both loops but the implied barrier ensures that results are available when needed.

private*(list)*	
firstprivate*(list)*	
lastprivate*(list)*	
reduction*(operator:list)*	(C/C++)
reduction({*operator* | *intrinsic_procedure_name*}*:list)*	(Fortran)
ordered	
schedule *(kind[,chunk_size)]*	
nowait	

Figure 4.13: **Clauses supported by the loop construct** – They are described in Section 4.5 and Section 4.8.

tran, respectively, to mark each distinct section. Each section must be a structured block of code that is independent of the other sections. At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once. If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks. If there are fewer code blocks than threads, the remaining threads will be idle. Note that the assignment of code blocks to threads is implementation-dependent.

The syntax of this construct in C/C++ is given in Figure 4.14. The syntax for Fortran is shown in Figure 4.15.

Although the `sections` construct is a general mechanism that can be used to get threads to perform different tasks independently, its most common use is probably to execute function or subroutine calls in parallel. We give an example of this kind of usage in Figure 4.16. This code fragment contains one `sections` construct,

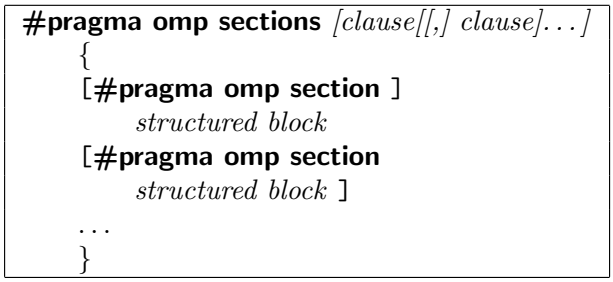

Figure 4.14: **Syntax of the sections construct in C/C++** – The number of sections controls, and limits, the amount of parallelism. If there are "n" of these code blocks, at most "n" threads can execute in parallel.

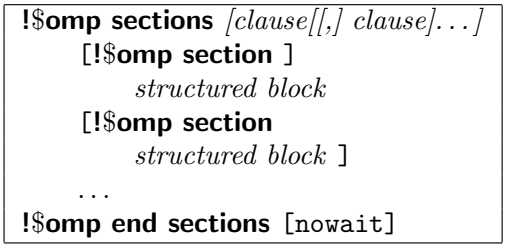

Figure 4.15: **Syntax of the sections construct in Fortran** – The number of sections controls, and limits, the amount of parallelism. If there are "n" of these code blocks, at most "n" threads can execute in parallel.

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
        (void) funcA();

      #pragma omp section
        (void) funcB();
   } /*-- End of sections block --*/

} /*-- End of parallel region --*/
```

Figure 4.16: **Example of parallel sections** – If two or more threads are available, one thread invokes `funcA()` and another thread calls `funcB()`. Any other threads are idle.

comprising two sections. The immediate observation is that this limits the parallelism to two threads. If two or more threads are available, function calls `funcA` and `funcB` are executed in parallel. If only one thread is available, both calls to `funcA` and `funcB` are executed, but in sequential order. Note that one cannot make any assumption on the specific order in which section blocks are executed. Even if these calls are executed sequentially, for example, because the directive is not in an active parallel region, `funcB` may be called before `funcA`.

The functions are very simple. They merely print the thread number of the calling thread. Figure 4.17 lists `funcA`. The source of `funcB` is similar. The output of this program when executed by two threads is given in Figure 4.18.

```
void funcA()
{
   printf("In funcA: this section is executed by thread %d\n",
        omp_get_thread_num());
}
```

Figure 4.17: **Source of funcA** – This function prints the thread number of the thread executing the function call.

```
In funcA: this section is executed by thread 0
In funcB: this section is executed by thread 1
```

Figure 4.18: **Output from the example given in Figure 4.16** – The code is executed by using two threads.

Depending on the type of work performed in the various code blocks and the number of threads used, this construct might lead to a *load-balancing* problem. This occurs when threads have different amounts of work to do and thus take different amounts of time to complete. A result of load imbalance is that some threads may wait a long time at the next barrier in the program, which means that the hardware resources are not being efficiently exploited. It may sometimes be possible to eliminate the barrier at the end of this construct (see Section 4.5.6), but that does not overcome the fundamental problem of a load imbalance *within* the sections construct. If, for example, there are five equal-sized code blocks and only four threads are available, one thread has to do more work.[4] If a lot of computation

[4]Which thread does so depends on the mapping strategy. The most common way to distribute

is involved, other strategies may need to be considered (see, e.g., Section 4.9.1 and Chapter 6).

The clauses supported by the `sections` construct are listed in Figure 4.19.

private*(list)*	
firstprivate*(list)*	
lastprivate*(list)*	
reduction*(operator:list)*	(C/C++)
reduction*({operator | intrinsic_procedure_name}:list)*	(Fortran)
nowait	

Figure 4.19: **Clauses supported by the sections construct** – These clauses are described in Section 4.5 and Section 4.8.

4.4.3 The Single Construct

The *single construct* is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only. It does not state which thread should execute the code block; indeed, the thread chosen could vary from one run to another. It can also differ for different `single` constructs within one application. This is not a limitation, however, as this construct should really be used when we do not care which thread executes this part of the application, as long as the work gets done by exactly one thread. The other threads wait at a barrier until the thread executing the single code block has completed.

The syntax of this construct in C/C++ is given in Figure 4.20. The syntax for Fortran is shown in Figure 4.21.

#pragma omp single *[clause[[,] clause]...]*	*structured block*

Figure 4.20: **Syntax of the single construct in C/C++** – Only one thread executes the structured block.

The code fragment in Figure 4.22 demonstrates the use of the single construct to initialize a shared variable.[5]

code blocks among threads is a round-robin scheme, where the work is distributed nearly evenly in the order of thread number.

[5]The curly braces are not really needed here, as there is one executable statement only; it has been put in to indicate that the code block can be much more complex and contain any number of statements.

!$omp single *[clause[[,] clause]...]*
structured block
!$omp end single [nowait,[copyprivate]]

Figure 4.21: **Syntax of the single construct in Fortran** – Only one thread executes the structured block.

```
#pragma omp parallel shared(a,b) private(i)
{
   #pragma omp single
   {
      a = 10;
      printf("Single construct executed by thread %d\n",
             omp_get_thread_num());
   }
   /* A barrier is automatically inserted here */

   #pragma omp for
   for (i=0; i<n; i++)
       b[i] = a;

} /*-- End of parallel region --*/

   printf("After the parallel region:\n");
   for (i=0; i<n; i++)
       printf("b[%d] = %d\n",i,b[i]);
```

Figure 4.22: **Example of the single construct** – Only one thread initializes the shared variable `a`.

The intention is clear. One thread initializes the shared variable `a`. This variable is then used to initialize vector `b` in the parallelized `for`-loop. Several points are worth noting here. On theoretical grounds one might think the single construct can be omitted in this case. After all, every thread would write the same value of 10 to the same variable `a`. However, this approach raises a hardware issue. Depending on the data type, the processor details, and the compiler behavior, the write to memory might be translated into a sequence of store instructions, each store writing a subset of the variable. For example, a variable 8 bytes long might be written to memory through 2 store instructions of 4 bytes each. Since a write

operation is not guaranteed to be atomic,[6] multiple threads could do this at the same time, potentially resulting in an arbitrary combination of bytes in memory. This issue is also related to the memory consistency model covered in Section 7.3.1.

Moreover, multiple stores to the same memory address are bad for performance. This and related performance matters are discussed in Chapter 5.

The other point worth noting is that, in this case, a barrier is essential before the `#pragma omp for` loop. Without such a barrier, some threads would begin to assign values to elements of `b` before `a` has been assigned a value, a particularly nasty kind of bug.[7] Luckily there is an implicit barrier at the end of the `single` construct. The output of this program is given in Figure 4.23. It shows that in this particular run, thread 3 initialized variable `a`. This is nondeterministic, however, and may change from run to run.

```
Single construct executed by thread 3
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
```

Figure 4.23: **Output from the example in Figure 4.22** – The value of variable n is set to 9, and four threads are used.

The clauses supported by the single construct are listed in Figure 4.24.

A similar construct, `master` (see Section 4.6.6), guarantees that a code block is executed by the master thread. It does not have an implied barrier.

4.4.4 Workshare Construct

The *workshare construct* is supported in Fortran only, where it serves to enable the parallel execution of code written using Fortran 90 array syntax. The statements

[6] Loosely said, if an operation is atomic no other thread can perform the same operation while the current thread executes it.

[7] Chapter 7 covers these kinds of problems in depth.

private*(list)*
firstprivate*(list)*
copyprivate*(list)*
nowait

Figure 4.24: **Clauses supported by the single construct** – These clauses are described in Section 4.5 and Section 4.8 on page 100. Note that in Fortran the `copyprivate` clause (as well as the `nowait` clause) is specified on the `!$omp end single` part of the construct.

in this construct are divided into units of work. These units are then executed in parallel in a manner that respects the semantics of Fortran array operations. The definition of "unit of work" depends on the construct. For example, if the `workshare` directive is applied to an array assignment statement, the assignment of each element is a unit of work. We refer the interested reader to the OpenMP standard for additional definitions of this term.

The syntax is shown in Figure 4.25.

!$omp workshare
structured block
!$omp end workshare `[nowait]`

Figure 4.25: **Syntax of the workshare construct in Fortran** – This construct is used to parallelize (blocks of) statements using array-syntax.

The structured block enclosed by this construct must consist of one or more of the following.

- Fortran array assignments and scalar assignments
- Fortran `FORALL` statements and constructs
- Fortran `WHERE` statements and constructs
- OpenMP `atomic`, `critical`, and `parallel` constructs

The code fragment in Figure 4.26 demonstrates how one can use the `workshare` construct to parallelize array assignment statements. Here, we get multiple threads to update three arrays `a`, `b`, and `c`. In this case, the OpenMP specification states that each assignment to an array element is a unit of work.

Two important rules govern this construct. We quote from the standard (Section 2.5.4):

```
!$OMP PARALLEL SHARED(n,a,b,c)
!$OMP WORKSHARE
        b(1:n) = b(1:n) + 1
        c(1:n) = c(1:n) + 2
        a(1:n) = b(1:n) + c(1:n)
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

Figure 4.26: **Example of the workshare construct** – These array operations are parallelized. There is no control over the assignment of array updates to the threads.

- It is unspecified how the units of work are assigned to the threads executing a `workshare` region.
- An implementation of the `workshare` construct must insert any synchronization that is required to maintain standard Fortran semantics.

In our example the latter rule implies that the OpenMP compiler must generate code such that the updates of `b` and `c` have completed before `a` is computed. In Chapter 8, we give an idea of how the compiler translates `workshare` directives.

Other than `nowait` there are no clauses for this construct.

4.4.5 Combined Parallel Work-Sharing Constructs

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work-sharing region includes all the code in the parallel region. The semantics of the shortcut directives are identical to explicitly specifying the `parallel` construct immediately followed by the work-sharing construct.

For example, the sequence in Figure 4.27 is is equivalent to the shortcut in Figure 4.28.

In Figure 4.29 we give an overview of the combined constructs available in C/C++. The overview for Fortran is shown in Figure 4.30. Note that for readability the clauses have been omitted.

The combined parallel work-sharing constructs allow certain clauses that are supported by both the `parallel` construct and the `workshare` construct. If the behavior of the code depends on where the clause is specified, it is an illegal OpenMP program, and therefore the behavior is undefined.

```
#pragma omp parallel
{
  #pragma omp for
  for (.....)
}
```

Figure 4.27: **A single work-sharing loop in a parallel region** – For cases like this OpenMP provides a shortcut.

```
#pragma omp parallel for
  for (.....)
```

Figure 4.28: **The combined work-sharing loop construct** – This variant is easier to read and may be slightly more efficient.

Full version	Combined construct

#pragma omp parallel
{
#pragma omp for
`for-loop`
}
	#pragma omp parallel for
	`for-loop`

#pragma omp parallel
{
#pragma omp sections
{
[**#pragma omp section**]
structured block
[**#pragma omp section**
structured block]
...
}
}
	#pragma omp parallel sections
	{
	[**#pragma omp section**]
	structured block
	[**#pragma omp section**
	structured block]
	...
	}

Figure 4.29: **Syntax of the combined constructs in C/C++** – The combined constructs may have a performance advantage over the more general `parallel` region with just one work-sharing construct embedded.

The main advantage of using these combined constructs is readability, but there can also be a performance advantage. When the combined construct is used, a compiler knows what to expect and may be able to generate slightly more efficient

Full version	Combined construct
!$omp parallel **!$omp do** `do-loop` **[!$omp end do]** **!$omp end parallel**	**!$omp parallel do** `do-loop` **!$omp end parallel do**
!$omp parallel **!$omp sections** **[!$omp section]** *structured block* **[!$omp section** *structured block* **]** ... **!$omp end sections** **!$omp end parallel**	**!$omp parallel sections** **[!$omp section]** *structured block* **[!$omp section** *structured block* **]** ... **!$omp end parallel sections**
!$omp parallel **!$omp workshare** *structured block* **!$omp end workshare** **!$omp end parallel**	**!$omp parallel workshare** *structured block* **!$omp end parallel workshare**

Figure 4.30: **Syntax of the combined constructs in Fortran** – The combined constructs may have a performance advantage over the more general `parallel` region with just one work-sharing construct embedded.

code. For example, it will not insert more than one barrier at the end of the region.

4.5 Clauses to Control Parallel and Work-Sharing Constructs

The OpenMP directives introduced above support a number of *clauses*, optional additions that provide a simple and powerful way to control the behavior of the construct they apply to. Indeed, some of these clauses are all but indispensable in practice. They include syntax needed to specify which variables are shared and which are private in the code associated with a construct, according to the OpenMP memory model introduced in Section 2.3.3 in Chapter 2. We have already indicated which clauses can be used with these constructs and have informally introduced a few of them. Let us now zoom in on them. We focus on practical aspects: what type of functionality is provided, and what are common ways to use them? For

other details, including rules and restrictions associated with specific clauses, we refer the reader to the OpenMP standard.

In this section, we introduce the most widely used clauses. In Section 4.8 we introduce the remaining ones. Since the clauses are processed before entering the construct they are associated with, they are evaluated in this "external" context, and any variables that appear in them must be defined there. Several clauses can be used with a given directive. The order in which they are given has no bearing on their evaluation: in fact, since the evaluation order is considered to be arbitrary, the programmer should be careful not to make any assumptions about it.

4.5.1 Shared Clause

The `shared` clause is used to specify which data will be shared among the threads executing the region it is associated with. Simply stated, there is one unique instance of these variables, and each thread can freely read or modify the values. The syntax for this clause is `shared(`*`list`*`)`. All items in the list are data objects that will be shared among the threads in the team.

```
#pragma omp parallel for shared(a)
  for (i=0; i<n; i++)
   {
       a[i] += i;
   } /*-- End of parallel for --*/
```

Figure 4.31: **Example of the shared clause** – All threads can read from and write to vector `a`.

The code fragment in Figure 4.31 illustrates the use of this clause. In this simple example, vector `a` is declared to be shared. This implies that all threads are able to read and write elements of `a`. Within the parallel loop, each thread will access the pre-existing values of those elements `a[i]` of `a` that it is responsible for updating and will compute their new values. After the parallel region is finished, all the new values for elements of `a` will be in main memory, where the master thread can access them.

An important implication of the shared attribute is that multiple threads might attempt to simultaneously update the same memory location or that one thread might try to read from a location that another thread is updating. Special care has to be taken to ensure that neither of these situations occurs and that accesses to shared data are ordered as required by the algorithm. OpenMP places the

responsibility for doing so on the user and provides several constructs that may help. They are discussed in Section 4.6.1 and Section 4.6.3. Another construct ensures that new values of shared data are available to all threads immediately, which might not otherwise be the case; it is described in Section 4.9.2.

4.5.2 Private Clause

What about the loop iteration variable `i` in the example in the previous section? Will it be shared? As we pointed out in Section 4.4.1 on page 58, the answer to that is a firm "no." Since the loop iterations are distributed over the threads in the team, each thread must be given a unique and local copy of the loop variable `i` so that it can safely modify the value. Otherwise, a change made to `i` by one thread would affect the value of `i` in another thread's memory, thereby making it impossible for the thread to keep track of its own set of iterations.

There may well be other data objects in a parallel region or work-sharing construct for which threads should be given their own copies. The `private` clause comes to our rescue here. The syntax is `private(`*`list`*`)`. Each variable in the list is replicated so that each thread in the team of threads has exclusive access to a local copy of this variable. Changes made to the data by one thread are not visible to other threads. This is exactly what is needed for `i` in the previous example.

By default, OpenMP gives the iteration variable of a parallel loop the `private` data-sharing attribute. In general, however, we recommend that the programmer not rely on the OpenMP default rules for data-sharing attributes. We will specify data-sharing attributes explicitly.[8]

```
#pragma omp parallel for private(i,a)
  for (i=0; i<n; i++)
   {
       a = i+1;
       printf("Thread %d has a value of a = %d for i = %d\n",
              omp_get_thread_num(),a,i);
   } /*-- End of parallel for --*/
```

Figure 4.32: **Example of the private clause** – Each thread has a local copy of variables `i` and `a`.

[8] We do make some exceptions: variables declared locally within a structured block or a routine that is invoked from within a parallel region are private by default.

A simple example of the use of the `private` clause is shown in Figure 4.32. Both the loop iteration variable `i` and the variable `a` are declared to be private variables here. If variable `a` had been specified in a shared clause, multiple threads would attempt to update the *same* variable with different values in an uncontrolled manner. The final value would thus depend on which thread happened to last update `a`. (This bug is a data race condition.) Therefore, the usage of `a` requires us to specify it to be a private variable, ensuring that each thread has its own copy.

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
```

Figure 4.33: **Output from the example shown in Figure 4.32** – The results are for $n = 5$, using three threads to execute the code.

Figure 4.33 shows the output of this program. As can be seen, threads 0 and 1 each execute two iterations of the loop, producing a different value for `a` each time. Thread 2 computes one value for `a`. Since each thread has its own local copy, there is no interference between them, and the results are what we should expect.

We note that the values of private data are *undefined* upon entry to and exit from the specific construct. The value of any variable with the same name as the private variable in the enclosing region is also undefined after the construct has terminated, even if the corresponding variable was defined prior to the region. Since this point may be unintuitive, care must be taken to check that the code respects this.

4.5.3 Lastprivate Clause

The example given in Section 4.5.2 works fine, but what if the value of `a` is needed after the loop? We have just stated that the values of data specified in the `private` clause can no longer be accessed after the corresponding region terminates. OpenMP offers a workaround if such a value is needed. The `lastprivate` clause addresses this situation; it is supported on the work-sharing loop and `sections` constructs.

The syntax is `lastprivate(`*`list`*`)`. It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution. In a parallel program, however, we must explain what "last" means. In the case of its use with a work-shared loop, the object will have the value from the iteration of

the loop that would be last in a sequential execution. If the `lastprivate` clause is used on a `sections` construct, the object gets assigned the value that it has at the end of the lexically last sections construct.

```
#pragma omp parallel for private(i) lastprivate(a)
  for (i=0; i<n; i++)
   {
       a = i+1;
       printf("Thread %d has a value of a = %d for i = %d\n",
              omp_get_thread_num(),a,i);
   } /*-- End of parallel for --*/

  printf("Value of a after parallel for: a = %d\n",a);
```

Figure 4.34: **Example of the lastprivate clause** – This clause makes the sequentially last value of variable `a` accessible outside the parallel loop.

In Figure 4.34 we give a slightly modified version of the example code from the previous section. Variable `a` now has the `lastprivate` data-sharing attribute, and there is a print statement after the parallel region so that we can check on the value `a` has at that point. The output is given in Figure 4.35. According to our definition of "last," the value of variable `a` after the parallel region should correspond to that computed when `i = n-1`. That is exactly what we get.

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
Value of a after parallel for: a = 5
```

Figure 4.35: **Output from the example shown in Figure 4.34** – Variable n is set to 5, and three threads are used. The last value of variable `a` corresponds to the value for $i = 4$, as expected.

In fact, all this clause really does is provide some extra convenience, since the same functionality can be implemented by using an additional shared variable and some simple logic. We do not particularly recommend doing so, but we demonstrate how this can be accomplished in the code fragment in Figure 4.36. The additional variable `a_shared` has been made shared, allowing us to access it outside the parallel

loop. All that needs to be done is to keep track of the last iteration and then copy the value of `a` into `a_shared`.

```
#pragma omp parallel for private(i) private(a) shared(a_shared)
  for (i=0; i<n; i++)
   {
       a = i+1;
       printf("Thread %d has a value of a = %d for i = %d\n",
              omp_get_thread_num(),a,i);
       if ( i == n-1 ) a_shared = a;
   } /*-- End of parallel for --*/
```

Figure 4.36: **Alternative code for the example in Figure 4.34** – This code shows another way to get the behavior of the `lastprivate` clause. However, we recommend use of the clause, not something like this.

A performance penalty is likely to be associated with the use of `lastprivate`, because the OpenMP library needs to keep track of which thread executes the last iteration. For a static workload distribution scheme this is relatively easy to do, but for a dynamic scheme this is more costly. More on the performance aspects of this clause can be found in Chapter 5.

4.5.4 Firstprivate Clause

Recall that private data is also undefined on entry to the construct where it is specified. This could be a problem if we need to to pre-initialize private variables with values that are available prior to the region in which they will be used. OpenMP provides the `firstprivate` construct to help out in such cases. Variables that are declared to be "firstprivate" are private variables, but they are pre-initialized with the value of the variable with the same name before the construct. The initialization is carried out by the initial thread prior to the execution of the construct. The `firstprivate` clause is supported on the `parallel` construct, plus the work-sharing `loop`, `sections`, and `single` constructs. The syntax is `firstprivate(`*`list`*`)`.

Now assume that each thread in a parallel region needs access to a thread-specific section of a vector but access starts at a certain (nonzero) offset. Figure 4.37 shows one way to implement this idea. The initial value of `indx` is initialized to the required offset from the first element of `a`. The length of each thread's section of the array is given by `n`. In the parallel region, the OpenMP function `omp_get_thread_num()` is used to store the thread number in variable `TID`. The

```
for(i=0; i<vlen; i++) a[i] = -i-1;

indx = 4;
#pragma omp parallel default(none) firstprivate(indx) \
                     private(i,TID) shared(n,a)
   {
      TID = omp_get_thread_num();

      indx += n*TID;
      for(i=indx; i<indx+n; i++)
         a[i] = TID + 1;
   } /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<vlen; i++)
    printf("a[%d] = %d\n",i,a[i]);
```

Figure 4.37: **Example using the firstprivate clause** – Each thread has a pre-initialized copy of variable `indx`. This variable is still private, so threads can update it individually.

start index into the thread-specific section is then given by `indx += n*TID` which uses the initial value of `indx` to account for the offset. For demonstration purposes, vector `a` is initialized with negative values. A part of this vector will be filled with positive values when the parallel region is executed, to make it easy to see which values have been modified.

We have executed this program for `indx = 4` using three threads and with `n = 2`. The output is given in Figure 4.38. It can be seen that the first four elements of `a` are not modified in the parallel region (as should be the case). Each thread has initialized two elements with a thread-specific value.

This example can actually be implemented more easily by using a shared variable, `offset` say, that contains the initial offset into vector a. We can then make `indx` a private variable. This is shown in the code fragment in Figure 4.39.

In general, *read-only* variables can be passed in as `shared` variables instead of `firstprivate`. This approach also saves the time incurred by runtime initialization. Note that on *cc-NUMA* systems, however, `firstprivate` might be the preferable option for dealing with read-only variables. OpenMP typically offers multiple ways to solve a given problem. This is a mixed blessing, however, as the performance implications of the different solutions may not be clearly visible.

```
After the parallel region:
a[0] = -1
a[1] = -2
a[2] = -3
a[3] = -4
a[4] = 1
a[5] = 1
a[6] = 2
a[7] = 2
a[8] = 3
a[9] = 3
```

Figure 4.38: **Output from the program shown in Figure 4.37** – The initial offset into the vector is set to $indx = 4$. Variable $n = 2$ and three threads are used. Therefore the total length of the vector is given by $vlen = 4 * 2 * 3 = 10$. The first $indx = 4$ values of vector `a` are not initialized.

```
#pragma omp parallel default(none) private(i,TID,indx) \
                shared(n,offset,a)
   {
      TID = omp_get_thread_num();

      indx = offset + n*TID;
      for(i=indx; i<indx+n; i++)
         a[i] = TID + 1;
   } /*-- End of parallel region --*/
```

Figure 4.39: **Alternative to the source shown in Figure 4.37** – If variable `indx` is not updated any further, this simpler and more elegant solution is preferred.

4.5.5 Default Clause

The `default` clause is used to give variables a default data-sharing attribute. Its usage is straightforward. For example, `default(shared)` assigns the shared attribute to all variables referenced in the construct. The `default(private)` clause, which is not supported in C/C++, makes all variables private by default. It is applicable to the `parallel` construct only. The syntax in C/C++ is given by `default (none | shared)`. In Fortran, the syntax is `default (none | shared | private)`.

This clause is most often used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed:

`#pragma omp for default(shared) private(a,b,c)`, for example, declares all variables to be shared, with the exception of `a`, `b`, and `c`.

If `default(none)` is specified instead, the programmer is forced to specify a data-sharing attribute for each variable in the construct. Although variables with a predetermined data-sharing attribute need not be listed in one of the clauses, we strongly recommend that the attribute be explicitly specified for *all* variables in the construct. In the remainder of this chapter, `default(none)` is used in the examples.

4.5.6 Nowait Clause

The `nowait` clause allows the programmer to fine-tune a program's performance. When we introduced the work-sharing constructs, we mentioned that there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP; in other words, if it is added to a construct, the barrier at the end of the associated construct will be suppressed. When threads reach the end of the construct, they will immediately proceed to perform other work. Note, however, that the barrier at the end of a parallel region cannot be suppressed.

Usage is straightforward. Once a parallel program runs correctly, one can try to identify places where a barrier is not needed and insert the `nowait` clause. The code fragment shown in Figure 4.40 demonstrates its use in C code. When a thread is finished with the work associated with the parallelized `for` loop, it continues and no longer waits for the other threads to finish as well.

```
#pragma omp for nowait
  for (i=0; i<n; i++)
   {
      ............
   }
```

Figure 4.40: **Example of the nowait clause in C/C++** – The clause ensures that there is no barrier at the end of the loop.

In Fortran the clause needs to be added to the `end` part of the construct, as demonstrated in Figure 4.41.

Some care is required when inserting this clause because its incorrect usage can introduce bugs. For example, in Figure 4.12 we showed a parallel region with two parallel loops, where one loop produced values that were used in the subsequent one. If we were to apply a nowait to the first loop in this code, threads might attempt

```
!$OMP DO

   ............

!$OMP END DO NOWAIT
```

Figure 4.41: **Example of usage of the nowait clause in Fortran** – In contrast with the syntax for C/C++, this clause is placed on the construct at the end of the loop.

to use values that have not been created. In this particular case, the application programmer might reason that the thread that creates a given value `a[i]` will also be the one that uses it. Then, the barrier could be omitted. There is, however, no guarantee that this is the case. Since we have not told the compiler how to distribute iterations to threads, we cannot be sure that the thread executing loop iteration `i` in the first loop will also execute loop iteration `i` in the second parallel loop. If the code depends on a specific distribution scheme, it is best to specify it explicitly.[9] In the next section, we show how to do so.

4.5.7 Schedule Clause

The `schedule` clause is supported on the loop construct only. It is used to control the manner in which loop iterations are distributed over the threads, which can have a major impact on the performance of a program. The syntax is `schedule(`*`kind [,chunk_size]`*`)`.

The schedule clause specifies how the iterations of the loop are assigned to the threads in the team. The granularity of this workload distribution is a *chunk*, a contiguous, nonempty subset of the iteration space. Note that the `chunk_size` parameter need not be a constant; any loop invariant integer expression with a positive value is allowed.

In Figure 4.42 on page 80 the four different schedule kinds defined in the standard are listed, together with a short description of their behavior. The most straightforward schedule is `static`. It also has the least overhead and is the default on many OpenMP compilers, to be used in the absence of an explicit `schedule` clause. As mentioned in Section 4.4.1, one can *not* assume this, however. Both the `dynamic` and `guided` schedules are useful for handling poorly balanced and unpredictable workloads. The difference between them is that with the `guided` schedule, the size

[9] This is expected to change in OpenMP 3.0. Under certain conditions the assignment of iteration numbers to threads is preserved across work-sharing loops.

Schedule kind	Description
static	Iterations are divided into chunks of size *chunk_size*. The chunks are assigned to the threads statically in a round-robin manner, in the order of the thread number. The last chunk to be assigned may have a smaller number of iterations. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size. Each thread is assigned at most one chunk.
dynamic	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the *chunk_size* parameter), then requests another chunk until there are no more chunks to work on. The last chunk may have fewer iterations than *chunk_size*. When no *chunk_size* is specified, it defaults to 1.
guided	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the *chunk_size* parameter), then requests another chunk until there are no more chunks to work on. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1. For a *chunk_size* of "k" ($k > 1$), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (with a possible exception for the last chunk to be assigned, which may have fewer than k iterations). When no *chunk_size* is specified, it defaults to 1.
runtime	If this schedule is selected, the decision regarding scheduling kind is made at run time. The schedule and (optional) chunk size are set through the `OMP_SCHEDULE` environment variable.

Figure 4.42: **Schedule kinds supported on the schedule clause** – The `static` schedule works best for regular workloads. For a more dynamic work allocation scheme the `dynamic` or `guided` schedules may be more suitable.

of the chunk (of iterations) decreases over time. The rationale behind this scheme is that initially larger chunks are desirable because they reduce the overhead. Load

balancing is often more of an issue toward the end of computation. The system then uses relatively small chunks to fill in the gaps in the schedule.

All three workload distribution algorithms support an optional *chunk_size* parameter. As shown in Figure 4.42, the interpretation of this parameter depends on the schedule chosen. For example, a *chunk_size* bigger than 1 on the `static` schedule may give rise to a round-robin allocation scheme in which each thread executes the iterations in a sequence of chunks whose size is given by *chunk_size*. It is not always easy to select the appropriate schedule and value for *chunk_size* up front. The choice may depend (among other things) not only on the code in the loop but also on the specific problem size and the number of threads used. Therefore, the `runtime` clause is convenient. Instead of making a compile time decision, the OpenMP `OMP_SCHEDULE` environment variable can be used to choose the schedule and (optional) *chunk_size* at run time (see Section 4.7).

Figure 4.43 shows an example of the use of the `schedule` clause. The outer loop has been parallelized with the loop construct. The workload in the inner loop depends on the value of the outer loop iteration variable `i`. Therefore, the workload is not balanced, and the static schedule is probably not the best choice.

```
#pragma omp parallel for default(none) schedule(runtime) \
                     private(i,j) shared(n)
  for (i=0; i<n; i++)
   {
      printf("Iteration %d executed by thread %d\n",
          i, omp_get_thread_num());
      for (j=0; j<i; j++)
          system("sleep 1");
   } /*-- End of parallel for --*/
```

Figure 4.43: **Example of the schedule clause** – The runtime variant of this clause is used. The `OMP_SCHEDULE` environment variable is used to specify the schedule that should be used when executing this loop.

In order to illustrate the various workload policies, the program listed in Figure 4.43 was executed on four threads, using a value of 9 for `n`. The results are listed in Table 4.1. The first column contains the value of the outer loop iteration variable `i`. The remaining columns contain the thread number (labeled "TID") for the various workload schedules and chunk sizes selected.

One sees, for example, that iteration `i = 0` was always executed by thread 0, regardless of the schedule. Iteration `i = 2`, however, was executed by thread 0

Table 4.1: **Example of various workload distribution policies** – The behavior for the `dynamic` and `guided` scheduling policies is nondeterministic. A subsequent run with the same program may give different results.

Iteration	TID static	TID static,2	TID dynamic	TID dynamic,2	TID guided	TID guided,2
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	1	3	3	3	3
3	1	1	2	3	2	3
4	1	2	1	2	1	2
5	2	2	0	2	0	2
6	2	3	3	1	3	1
7	3	3	2	1	2	1
8	3	0	1	0	1	0

with the default chunk size on the `static` schedule; but thread 1 executed this iteration with a chunk size of 2. Apparently, this iteration has been executed by thread 3 for both the `dynamic` and the `guided` schedule, regardless of the chunk size.

To illustrate this further, we executed the above loop for `n = 200` using four threads. The results are shown in Figure 4.44. Three scheduling algorithms—`static`, `dynamic,7`, and `guided,7`—have been combined into a single chart. The horizontal axis represents the value of the loop iteration variable `i` in the range `0,...,199`. The vertical axis gives the thread number of the thread that executed the particular iteration.

The first set of four horizontal lines shows the results for the `static` scheme. As expected, thread 0 executes the first 50 iterations, thread 1 works on the next 50 iterations, and so forth. The second set of four horizontal lines gives the results for the `dynamic,7` workload schedule. There are striking differences between this and the `static` case. Threads process chunks of 7 iterations at the time, since a chunk size of 7 was specified. Another difference is that threads no longer work on contiguous sets of iterations. For example, the first set of iterations executed by thread 0 is `i = 0,...,6`, whereas thread 1 processes `i = 21,...,27`, thread 2 handles `i = 7,...,13` and thread 3 executes `i = 14,...,20`. Thread 0 then continues with `i = 28,...,34` and so on.

The results for the `guided,7` schedule clearly demonstrate that the initial chunk sizes are larger than those toward the end. Although there is no notion of time

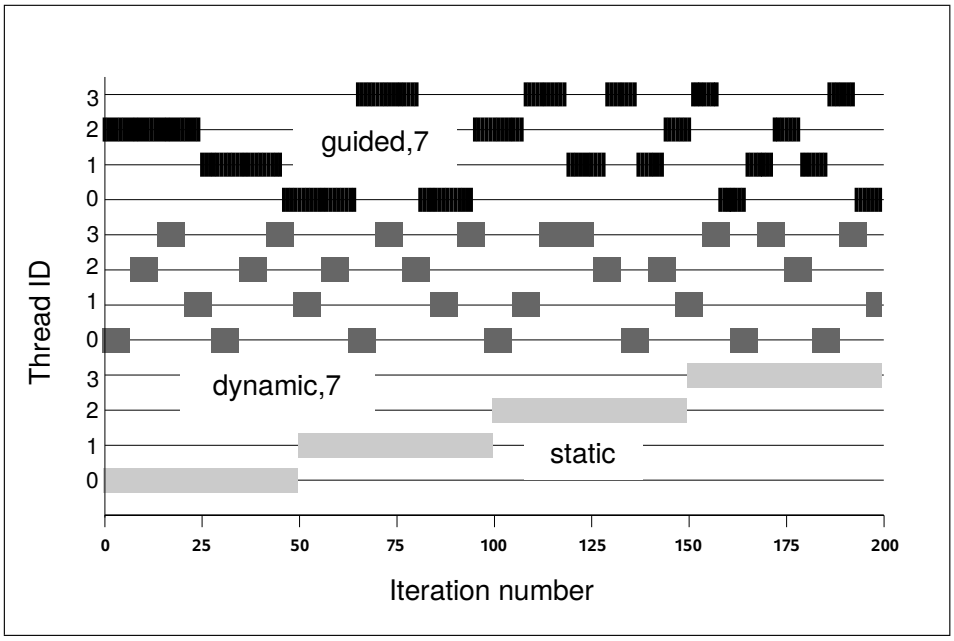

Figure 4.44: **Graphical illustration of the schedule clause** – The mapping of iterations onto four threads for three different scheduling algorithms for a loop of length $n = 200$ is shown. Clearly, both the `dynamic` and the `guided` policy give rise to a much more dynamic workload allocation scheme.

in Figure 4.44, thread 2 was probably the first one to start. With a total of 25 iterations, it gets the largest chunk of iterations. Thread 1 has the next 21 iterations to work on. Thread 0 gets 19 iterations, whereas thread 1 works on the next 16 iterations.

We emphasize that, other than for the `static` schedule, the allocation is nondeterministic and depends on a number of factors including the load of the system. We note, too, that programs that depend on which thread executes a particular iteration are nonconforming. The `static` schedule is most efficient from a performance point of view, since `dynamic` and `guided` have higher overheads. The size of the penalty for using them depends on the OpenMP implementation.

4.6 OpenMP Synchronization Constructs

In this section, we introduce OpenMP constructs that help to organize accesses to shared data by multiple threads. An algorithm may require us to orchestrate

the actions of multiple threads to ensure that updates to a shared variable occur in a certain order, or it may simply need to ensure that two threads do not simultaneously attempt to write a shared object. The features discussed here can be used when the implicit barrier provided with work-sharing constructs does not suffice to specify the required interactions or would be inefficient. Together with the work-sharing constructs, they constitute a powerful set of features that suffice to parallelize a large number of applications.

4.6.1 Barrier Construct

A barrier is a point in the execution of a program where threads wait for each other: no thread in the team of threads it applies to may proceed beyond a barrier until all threads in the team have reached that point. We have already seen that many OpenMP constructs imply a barrier. That is, the compiler automatically inserts a barrier at the end of the construct, so that all threads wait there until all of the work associated with the construct has been completed. Thus, it is often unnecessary for the programmer to explicitly add a barrier to a code. If one is needed, however, OpenMP provides a construct that makes this possible. The syntax in C/C++ is given in Figure 4.45. The Fortran syntax is shown in Figure 4.46.

```
#pragma omp barrier
```

Figure 4.45: **Syntax of the barrier construct in C/C++** – This construct binds to the innermost enclosing parallel region.

```
!$omp barrier
```

Figure 4.46: **Syntax of the barrier construct in Fortran** – This construct binds to the innermost enclosing parallel region.

Two important restrictions apply to the `barrier` construct:

- Each barrier *must* be encountered by all threads in a team, or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

Without these restrictions, one could write programs where some threads wait forever (or until somebody kills the process) for other threads to reach a barrier. C/C++ imposes an additional restriction regarding the placement of a barrier

construct within the application: The `barrier` construct may only be placed in the program at a position where ignoring or deleting it would result in a program with correct syntax.

The code fragment in Figure 4.47 illustrates the behavior of the barrier construct. To ensure that some threads in the team executing the parallel region take longer than others to reach the barrier, we get half the threads to execute the `sleep 3` command, causing them to idle for three seconds. We then get each thread to print out its the thread number (stored in variable `TID`), a comment string, and the time of day in the format `hh:mm:ss`. The barrier is then reached. After the barrier, each thread will resume execution and again print out this information. (We do not show the source code of the function called `print_time` that was used to realize the output.)

```
#pragma omp parallel private(TID)
  {
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads()/2 ) system("sleep 3");
    (void) print_time(TID,"before");

    #pragma omp barrier

    (void) print_time(TID,"after ");
  } /*-- End of parallel region --*/
```

Figure 4.47: **Example usage of the barrier construct** – A thread waits at the barrier until the last thread in the team arrives. To demonstrate this behavior, we have made sure that some threads take longer than others to reach this point.

In Figure 4.48, the output of this program is shown for a run using four threads. Threads 2 and 3 arrive at the barrier 3 seconds before threads 0 and 1, because the latter two were delayed by the system call. The subsequent time stamps show that all threads continue execution once the last two have reached the barrier.

The most common use for a barrier is to avoid a data race condition. Inserting a barrier between the writes to and reads from a shared variable guarantees that the accesses are appropriately ordered, for example, that a write is completed before another thread might want to read the data.

```
Thread 2 before barrier at 01:12:05
Thread 3 before barrier at 01:12:05
Thread 1 before barrier at 01:12:08
Thread 0 before barrier at 01:12:08
Thread 1 after  barrier at 01:12:08
Thread 3 after  barrier at 01:12:08
Thread 2 after  barrier at 01:12:08
Thread 0 after  barrier at 01:12:08
```

Figure 4.48: **Output from the example in Figure 4.47** – Four threads are used. Note that threads 2 and 3 wait for three seconds in the barrier.

4.6.2 Ordered Construct

Another synchronization construct, the ordered construct, allows one to execute a structured block within a parallel loop in sequential order. This is sometimes used, for instance, to enforce an ordering on the printing of data computed by different threads. It may also be used to help determine whether there are any data races in the associated code. The syntax of the `ordered` construct in C/C++ is shown in Figure 4.49. The Fortran syntax is given in Figure 4.50.

```
#pragma omp ordered
        structured block
```

Figure 4.49: **Syntax of the ordered construct in C/C++** – This construct is placed within a parallel loop. The structured block is executed in the sequential order of the loop iterations.

```
!$omp ordered
        structured block
!$omp end ordered
```

Figure 4.50: **Syntax of the ordered construct in Fortran** – This construct is placed within a parallel loop. The structured block is executed in the sequential order of the loop iterations.

An `ordered` construct ensures that the code within the associated structured block is executed in sequential order. The code outside this block runs in parallel. When the thread executing the first iteration of the loop encounters the construct, it enters the region without waiting. When a thread executing any subsequent

iteration encounters the construct, it waits until each of the previous iterations in the sequence has completed execution of the region.

An `ordered` *clause* has to be added to the parallel region in which this construct appears; it informs the compiler that the construct occurs. We defer an example of the usage of this feature to our discussion of the clause in Section 4.8.3. The `ordered` construct itself does not support any clauses.

4.6.3 Critical Construct

The `critical` construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously. The associated code is referred to as a critical region, or a *critical section*.

An optional *name* can be given to a critical construct. In contrast to the rules governing other language features, this name is *global* and therefore should be unique. Otherwise the behavior of the application is undefined.

When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name. In other words, there is never a risk that multiple threads will execute the code contained in the same critical region at the same time.

The syntax of the critical construct in C/C++ is given in Figure 4.51. The Fortran syntax is shown in Figure 4.52.

#pragma omp critical *[(name)]*
structured block

Figure 4.51: **Syntax of the critical construct in C/C++** – The structured block is executed by all threads, but only one at a time executes the block. Optionally, the construct can have a name.

!$omp critical *[(name)]*
structured block
!$omp end critical *[(name)]*

Figure 4.52: **Syntax of the critical construct in Fortran** – The structured block is executed by all threads, but only one at a time executes the block. Optionally, the construct can have a name.

To illustrate this construct, consider the code fragment in Figure 4.53. The `for`-loop sums up the elements of vector `a`. This operation can be readily parallelized. One approach is to let each thread independently add up a subset of the elements

of the vector. The result is stored in a private variable. When all threads are done, they add up their private contributions to get the total `sum`.

```
sum = 0;
for (i=0; i<n; i++)
    sum += a[i];
```

Figure 4.53: **Loop implementing a summation** – This operation can be parallelized with some help from the compiler.

Figure 4.54 gives pseudo-code showing how two threads might collaborate to form the sum if `n` is even. Variable `sumLocal` has to be made private to the thread. Otherwise the statement `sumLocal += a[i]` would cause a data race condition, since both threads will try to update the same variable at the same time.

```
/*-- Executed by thread 0 --*/    /*-- Executed by thread 1 --*/

sumLocal = 0;                      sumLocal = 0;
for (i=0; i<n/2; i++)          for (i=n/2-1; i<n; i++)
    sumLocal += a[i];                  sumLocal += a[i];

sum += sumLocal;
                                 sum += sumLocal;
```

Figure 4.54: **Pseudo parallel code for the summation** – This indicates how the operations might be split up among two threads. There is no control over accesses to `sum`, however, so that there is a data race condition.

However, we still have to deal with the updates to `sum`. Without special measures, this also causes a data race condition. We do not need to enforce a certain ordering of accesses here, but we must ensure that only one update may take place at a time. This is precisely what the critical construct guarantees.

The corresponding OpenMP code fragment is shown in Figure 4.55. We have inserted a named critical region ("update_sum") and put a print statement into the critical region. It prints the thread number (stored in variable `TID`), the value of the partial sum that the thread has calculated (stored in variable `sumLocal`), and the value of `sum` so far.

We point out that this example is shown only to illustrate the workings of the critical construct. The explicit reduction algorithm given here is very naive and

should not be applied as written. For extensive coverage of explicit reduction algorithms we refer to [123]. OpenMP also provides the `reduction` clause to have the compiler handle these kind of cases.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
  {
     TID = omp_get_thread_num();
     sumLocal = 0;
     #pragma omp for
       for (i=0; i<n; i++)
          sumLocal += a[i];
     #pragma omp critical (update_sum)
      {
       sum += sumLocal;
       printf("TID=%d: sumLocal=%d sum = %d\n",TID,sumLocal,sum);
      }
  } /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Figure 4.55: **Explicit implementation of a reduction operation** – The critical region is needed to avoid a data race condition when updating variable `sum`. Note that the code is shown only for illustration purposes. OpenMP provides a `reduction` clause to make it even easier to implement a reduction operation. This should be preferred.

Within the parallel region, each thread initializes `sumLocal`. The iterations of the `#pragma omp for` loop are distributed over the threads. This process results in each thread computing a partial sum, stored in `sumLocal`. When the threads are finished with their part of the `for`-loop, they enter the critical region. By definition, only one thread at a time updates `sum`.

The output of this program is given in Figure 4.56. One can clearly see that each thread computes its partial sum and then adds this value to `sum`. Apparently, thread 0 has entered the critical region first, and thread 1 is the last one to enter.

This functionality is required in many situations. One simple example is the need to avoid garbled output when multiple threads print messages, as shown in the code snippet in Figure 4.57.

Another common situation where this construct is useful is when minima and maxima are formed. The code fragment in Figure 4.58 is similar to code in the WUPWISE program used in lattice gauge theory (quantum chromodynamics) and contained in the SPEC OpenMP benchmark suite [16]. It uses a critical region to

```
TID=0: sumLocal=36 sum = 36
TID=2: sumLocal=164 sum = 200
TID=1: sumLocal=100 sum = 300
Value of sum after parallel region: 300
```

Figure 4.56: **Output from the program shown in Figure 4.55** – Three threads are used and variable $n = 25$.

```
#pragma omp parallel shared(n) private(TID)
   {
      TID = omp_get_thread_num();
      #pragma omp critical (print_tid)
      {
        printf("I am thread %d\n",TID);
      }
   } /*-- End of parallel region --*/
```

Figure 4.57: **Avoiding garbled output** – A critical region helps to avoid intermingled output when multiple threads print from within a parallel region.

ensure that when one thread performs a comparison of the shared `Scale` with its local `LScale` to find out which value is smaller, no other thread can interfere with this sequence of operations. Note that the order in which threads carry out this work is not important here, so that a critical construct is just what is needed.

4.6.4 Atomic Construct

The `atomic` construct, which also enables multiple threads to update shared data without interference, can be an efficient alternative to the critical region. In contrast to other constructs, it is applied only to the (single) assignment statement that immediately follows it; this statement must have a certain form in order for the construct to be valid, and thus its range of applicability is strictly limited. The syntax is shown in Figures 4.59 and 4.60.

The `atomic` construct enables efficient updating of shared variables by multiple threads on hardware platforms which support *atomic* operations. The reason it is applied to just one assignment statement is that it protects updates to an individual memory location, the one on the left-hand side of the assignment. If the hardware supports instructions that read from a memory location, modify the value, and write back to the location all in one action, then `atomic` instructs the compiler to

```
#pragma omp parallel private(ix, LScale, lssq, Temp) \
                     shared(Scale, ssq, x)
  {
  #pragma omp for
      for(ix = 1, ix<N, ix++)
        {
        LScale = ....;
      }
  #pragma omp critical
      {
        if(Scale < LScale){
           ssq = (Scale/LScale) *ssq + lssq;
           Scale = LScale;
        }else
           ssq = ssq + (LScale / Scale) * Lssq
       } /* End of critical region --*/
  } /*-- End of parallel region --*/
```

Figure 4.58: **Critical region usage to determine minimum value** – The critical region is needed to avoid a data race condition when comparing the value of the private variable `LSCALE` with the shared variable `Scale` and when updating it and `ssq`. The execution order does not matter in the case.

#pragma omp atomic
statement

Figure 4.59: **Syntax of the atomic construct in C/C++** – The statement is executed by all threads, but only one thread at a time executes the statement.

!$omp atomic
statement

Figure 4.60: **Syntax of the atomic construct in Fortran** – The statement is executed by all threads, but only one thread at a time executes the statement.

use such an operation. If a thread is atomically updating a value, then no other thread may do so simultaneously. This restriction applies to all threads that execute a program, not just the threads in the same team. To ensure this, however, the programmer must mark *all* potentially simultaneous updates to a memory location by this directive. A simple example is shown is Figure 4.61, where multiple threads update a counter.

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
         ic = ic + 1;
    }
printf("counter = %d\n", ic);
```

Figure 4.61: **Example for the use of** `atomic` – The `atomic` construct ensures that no updates are lost when multiple threads are updating a counter value.

The `atomic` construct may only be used together with an expression statement in C/C++, which essentially means that it applies a simple, binary operation such as an increment or decrement to the value on the left-hand side. The supported operations are: `+, *, -, /, &, ^, |, <<, >>`. In Fortran, the statement must also take the form of an update to the value on the left-hand side, which may not be an array, via an expression or an intrinsic procedure. The operator may be one of `+, *, -, /, .AND., .OR., .EQV., .NEQV.`, and the intrinsic procedure may be one of `MAX, MIN, IAND, IOR, IEOR`. There are a number of restrictions on the form that the expression may take; for example, it must not involve the variable on the left-hand side of the assignment statement. We refer the reader to the OpenMP standard for full details.

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
        ic = ic + bigfunc();
    }
printf("counter = %d\n", ic);
```

Figure 4.62: **Another use of** `atomic` – The `atomic` construct does not prevent multiple threads from executing the function `bigfunc` at the same time.

In our slightly revised example shown in Figure 4.62, the `atomic` construct does

not protect the execution of function `bigfunc`. It is only the update to the memory location of the variable `ic` that will occur atomically. If the application developer does not intend to permit the threads to execute `bigfunc` at the same time, then the `critical` construct must be used instead.

4.6.5 Locks

In addition to the synchronization features introduced above, the OpenMP API provides a set of low-level, general-purpose locking runtime library routines, similar in function to the use of semaphores. These routines provide greater flexibility for synchronization than does the use of `critical` sections or `atomic` constructs. The general syntax of the locking library routines is shown in Figures 4.63 and 4.64.

```
void omp_func_lock (omp_lock_t *lck)
```

Figure 4.63: **General syntax of locking routines in C/C++** – For a specific routine, *func* expresses its functionality; *func* may assume the values `init, destroy, set, unset, test`. The values for nested locks are `init_nest, destroy_nest, set_nest, unset_nest, test_nest`.

```
subroutine omp_func_lock (svar)
integer (kind=omp_lock_kind) svar
```

Figure 4.64: **General syntax of locking routines in Fortran** – For a specific routine, *func* expresses its functionality; *func* may assume the values `init, destroy, set, unset, test`.The values for nested locks are `init_nest, destroy_nest, set_nest, unset_nest, test_nest`.

The routines operate on special-purpose *lock variables*, which should be accessed via the locking routines only. There are two types of locks: *simple locks*, which may not be locked if already in a locked state, and *nestable locks*, which may be locked multiple times by the same thread. Simple lock variables are declared with the special type `omp_lock_t` in C/C++ and are integer variables of `kind = omp_lock_kind` in Fortran. Nestable lock variables are declared with the special type `omp_nest_lock_t` in C/C++ and are integer variables of `kind = omp_nest_lock_kind` in Fortran. In C, lock routines need an argument that is a pointer to a lock variable of the appropriate type. The general procedure to use locks is as follows:

1. Define the (simple or nested) lock variables.

2. Initialize the lock via a call to `omp_init_lock`.
3. Set the lock using `omp_set_lock` or `omp_test_lock`. The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution.
4. Unset a lock after the work is done via a call to `omp_unset_lock`.
5. Remove the lock association via a call to `omp_destroy_lock`.

A simple example is shown in Figure 4.65.

```
      ...
      CALL OMP_INIT_LOCK (LCK)
      ...
C$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
      ...
  100 CONTINUE
      IF (.NOT. OMP_TEST_LOCK(LCK)) THEN
        CALL WORK2 ()
        GO TO 100
      ENDIF
      CALL WORK(ID)
      CALL OMP_UNSET_LOCK(LCK)
C$OMP END PARALLEL
      CALL OMP_DESTROY_LOCK(LCK)
```

Figure 4.65: **Example of lock usage** – The example demonstrates how asynchronous thread execution can be achieved by using explicit locking

Note that special care is needed when the programmer synchronizes the actions of threads using these routines. If these routines are used improperly, a number of programming errors are possible. In particular, a code may deadlock. We discuss parallel programming pitfalls and problems separately in Chapter 7.

4.6.6 Master Construct

The `master` construct defines a block of code that is guaranteed to be executed by the master thread only. It is thus similar to the `single` construct (covered in Section 4.4.3). The `master` construct is technically not a work-sharing construct, however, and it does not have an implied barrier on entry or exit. The syntax in C/C++ is given in Figure 4.66, and the syntax in Fortran is given in Figure 4.67.

The lack of a barrier may lead to problems. If the `master` construct is used to initialize data, for example, care needs to be taken that this initialization is completed *before* the other threads in the team use the data. The typical solution is either to rely on an implied barrier further down the execution stream or to use an explicit *barrier* construct (see Section 4.6.1).

```
#pragma omp master
        structured block
```

Figure 4.66: **Syntax of the master construct in C/C++** – Note that there is *no* implied barrier on entry to, or exit from, this construct.

```
!$omp master
      structured block
!$omp end master
```

Figure 4.67: **Syntax of the master construct in Fortran** – Note that there is *no* implied barrier on entry to, or exit from, this construct.

Figure 4.68 shows a code fragment that uses the `master` construct. It is similar to the example in Section 4.4.3. The two differences are that the initialization of variable `a` is now guaranteed to be performed by the master thread *and* the `#pragma omp barrier` needs to be inserted for correctness.

In this simple case, there is no particular reason to choose this rather than the `single` construct. In a more realistic piece of code, there may be additional computation after the `master` construct and before the first use of the data initialized by the master thread. In such a situation, or whenever the barrier is not required, this construct may be preferable.

The output of this program shows that thread 0, the master thread, has performed the initialization of variable `a`. In contrast to the `single` construct, where it is not known which thread will execute the code, this behavior is deterministic.

4.7 Interaction with the Execution Environment

The OpenMP standard provides several means with which the programmer can interact with the execution environment, either to obtain information from it or to influence the execution of a program. If a program relies on some property of the environment, for example, expects that a certain minimum number of threads will

```
#pragma omp parallel shared(a,b) private(i)
 {
    #pragma omp master
     {
       a = 10;
       printf("Master construct is executed by thread %d\n",
              omp_get_thread_num());
     }

    #pragma omp barrier

    #pragma omp for
     for (i=0; i<n; i++)
          b[i] = a;

} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
     printf("b[%d] = %d\n",i,b[i]);
```

Figure 4.68: **Example of the master construct** – This is similar to the example shown in Figure 4.22. The difference is that the master thread is guaranteed to initialize variable `a`. Note the use of a barrier to ensure availability of data.

```
Master construct is executed by thread 0
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
```

Figure 4.69: **Output from the example in Figure 4.68** – This clearly demonstrates that the master thread has performed the initialization.

execute a parallel region, then the programmer must test for its satisfaction explicitly. Before we discuss these features, we need to explain just how the environment can be manipulated.

The OpenMP standard defines *internal control variables*. These are variables controlled by the OpenMP implementation that govern the behavior of a program at run time in important ways. They cannot be accessed or modified directly at the application level; however, they can be queried and modified through OpenMP functions and environment variables. The following internal control variables are defined.

- *nthreads-var* – stores the number of threads requested for the execution of future parallel regions.
- *dyn-var* – controls whether dynamic adjustment of the number of threads to be used for future parallel regions is enabled
- *nest-var* – controls whether nested parallelism is enabled for future parallel regions
- *run-sched-var* – stores scheduling information to be used for loop regions using the `runtime` schedule clause
- *def-sched-var* – stores implementation-defined default scheduling information for loop regions

Here, we introduce the library functions and environment variables that can be used to access or modify the values of these variables and hence influence the program's execution. The four environment variables defined by the standard may be set prior to program execution. The library routines can also be used to give values to control variables; they override values set via environment variables. In order to be able to use them, a C/C++ program should include the `omp.h` header file. A Fortran program should either include the `omp_lib.h` header file or `omp_lib` module, depending on which of them is provided by the implementation.

Once a team of threads is formed to execute a parallel region, the number of threads in it will not be changed. However, the number of threads to be used to execute future parallel regions can be specified in several ways:

- At the command line, the `OMP_NUM_THREADS` environment variable may be set. The value specified will be used to initialize the *nthreads-var* control variable. Its syntax is `OMP_NUM_THREADS(`*`integer`*`)`, where the integer must be positive.

- During program execution, the number of threads to be used to execute a parallel region may be set or modified via the `omp_set_num_threads` library routine. Its syntax is `omp_set_num_threads(`*`scalar-integer-expression`*`)`, where the evaluation of the expression must result in a positive integer.

- Finally, it is possible to use the `num_threads` clause together with a `parallel` construct to specify how many threads should be in the team executing that specific parallel region. If this is given, it *temporarily* overrides both of the previous constructs. It is discussed and illustrated in Section 4.8.2.

If the parallel region is conditionally executed and the condition does not hold, or if it is a nested region and nesting is not available, then none of these will have an effect: the region will be sequentially executed. During program execution, the number of threads available for executing future parallel regions can be retrieved via the `omp_get_max_threads()` routine, which returns the largest number of threads available for the next parallel region.

One can control the value of *dyn-var* to permit (or disallow) the system to dynamically adjust the number of threads that will be used to execute future parallel regions. This is typically used to optimize the use of system resources for throughput. There are two ways to do so:

- The environment variable `OMP_DYNAMIC` can be specified prior to execution to initialize this value to either *true*, in which case this feature is enabled, or to *false*, in which case the implementation may not adjust the number of threads to use for executing parallel regions. Its syntax is `OMP_DYNAMIC(`*`flag`*`)`, where *flag* has the value *true* or *false*.

- The routine `omp_set_dynamic` adjusts the value of *dyn-var* at run time. It will influence the behavior of parallel regions for which the thread that executes it is the master thread. `omp_set_dynamic(`*`scalar-integer-expression`*`)` is the C/C++ syntax; `omp_set_dynamic(`*`logical-expression`*`)` is the Fortran syntax. In both cases, if the argument of this procedure evaluates to *true*, then dynamic adjustment is enabled. Otherwise, it is disabled.

Routine `omp_get_dynamic` can be used to retrieve the current setting at run time. It returns *true* if the dynamic adjustment of the number of threads is enabled; otherwise *false* is returned. The result is an integer value in C/C++ and a logical value in Fortran.

If the implementation provides nested parallelism, then its availability to execute a given code can be controlled by assigning a value to the *nest-var* variable. If the

implementation does not provide this feature, modifications to the *nest-var* variable have no effect.

- This variable can be set to either *true* or *false* prior to execution by giving the OMP_NESTED environment variable the corresponding value. Note that the standard specifies that it is initialized to *false* by default.

- As with the previous cases, a runtime library routine enables the programmer to adjust the setting of *nest-var* at run time, possibly overriding the value of the environment variable. It is `omp_set_nested`, and it applies to the thread that executes it; in other words, if this thread encounters a parallel construct, then that region will become active so long as the implementation can support such nesting. The syntax in C/C++ is as follows:

  `omp_set_nested(`*`scalar-integer-expression`*`).`

  The corresponding Fortran is `omp_set_nested(`*`logical-expression`*`)`. In both cases, if the argument of this procedure evaluates to *true*, then nesting of parallel regions is enabled; otherwise, it is disabled.

The `omp_get_nested` routine, whose result is an integer value in C/C++ and a logical value in Fortran, returns the current setting of the *nest-var* variable for the thread that calls it: *true* if nesting is enabled for that thread and otherwise *false*.

The `OMP_SCHEDULE` environment variable enables the programmer to set *def-sched-var* and thereby customize the default schedule to be applied to parallel loops in a program. Its value, which is otherwise implementation-defined, will be used to determine the assignment of loop iterations to threads for all parallel loops whose schedule type is specified to be `runtime`. The value of this variable takes the form *type [,chunk]*, where *type* is one of `static`, `dynamic` or `guided`. The optional parameter *chunk* is a positive integer that specifies the *chunk_size*.

The OpenMP standard includes several other user-level library routines, some of which we have already seen:

- The `omp_get_num_threads` library routine enables the programmer to retrieve the number of threads in the current team. The value it returns has integer data type. This value may be used in the programmer's code, for example, to choose an algorithm from several variants.

- `omp_get_thread_num` returns the number of the calling thread as an integer value. We have seen its use in many examples throughout this chapter, primarily to assign different tasks to different threads explicitly.