1 A Recap of OpenMP 2.5

In [2], OpenMP is covered at great length. At the time that [2] was published, the OpenMP 2.5 specifications were released and that is what that book focused on. Since then, four more specifications have been released. The enhancements and new features they introduced are the major topic of this book. For completeness, this chapter presents a brief recap of OpenMP 2.5. In many cases, the syntax description is informal and not every aspect is covered. For such details, refer to the specifications [20].

1.1 OpenMP Directives and Syntax

OpenMP is an Application Programming Interface (API) for shared memory parallelization, using C, C++, or Fortran. The API consists of compiler directives to specify and control parallelization, augmented with runtime functions, and environment variables. It is incumbent upon the user to identify the parallelism and insert the appropriate control structures in the program. These controls are called *directives*.

In C/C++, the directive is based on the `#pragma omp` construct. This is a regular language pragma and is ignored if it is not recognized by the compiler. This may happen for several reasons: 1) the compiler does not support OpenMP, although that is rare these days, 2) if a typo is made (e.g. `#pragma openmp`), or if the source is not compiled with an option to enable recognition of the directives.

In Fortran, the directive starts with either `!$omp` or `*$omp/c$omp` in old-style Fortran programs. To the compiler, this is a regular comment string, unless through the use of a specific option, the compiler is instructed to recognize the special `omp` keyword and generate the corresponding OpenMP code, but also here a typo gives unexpected results.

In C/C++ and Fortran, portability of the code is not violated when using the directives. The source compiles with, or without, OpenMP enabled. If the OpenMP-specific runtime functions are used, but no option to support OpenMP is specified, unresolved references to those functions are reported by the linker. Luckily, there is an easy way around this.

If a source is compiled with OpenMP enabled, the `_OPENMP` macro is guaranteed to be set in C/C++. It is set to the release date of the OpenMP version the compiler supports. Regardless of the value, this macro may be used in conjunction with an `#ifdef` construct to support the runtime functions.

For example, the code fragment below handles the use of the `omp_get_num_threads()` function and compiles both with and without OpenMP enabled. In the case of the former, the include file `omp.h` is used, exposing all OpenMP function definitions to the compiler. Outside an OpenMP environment, the function is defined to always return `1` for the number of threads.

```
1 #ifdef _OPENMP
2   #include <omp.h>
3 #else
4   #define omp_get_num_threads() 1
5 #endif
```

In Fortran, conditional compilation may be used to handle the use of runtime functions. With this mechanism, the character string `!$` (or `*$` and `c$`) is replaced by two spaces if the source is compiled with OpenMP enabled. Otherwise, it is considered to be a language comment and ignored. This allows for the compile-time activation of executable statements, as demonstrated below:

```
1 !$    use omp_lib
2          ....
3       nthreads = 1
4 !$    nthreads = omp_get_num_threads()
```

If not compiled for OpenMP, the first and third lines in the source snippet are ignored, because they are treated as language comments. If OpenMP is enabled, the `!$` string is replaced by two spaces. This results in the use of module `omp_lib` and the second assignment to variable `nthreads` is activated. The second assignment overrides the first one, and at runtime, the number of active threads is returned.[1]

An additional benefit of the directive-based approach is that compilers have full visibility of the (parallel) code and typically perform many optimizations, as well as issue warning messages. This is much more difficult to do with a programming model based upon function calls.

1.2 Creating a Parallel Program with OpenMP

The user must identify the code part(s) that may be executed in parallel. The parallelism is defined through the addition of the appropriate directives to the

[1] Optimizing compilers recognize that the first assignment is redundant and eliminate it.

#pragma omp parallel *[clause[[,] clause]...] new-line* *structured block*
!$omp parallel *[clause[[,] clause]...]* *structured block* **!$omp end parallel**

Figure 1.1: **Syntax of the parallel construct in C/C++ and Fortran** – The statements within the parallel region are executed by all threads. In C/C++, the parallel region implicitly ends at the end of the structured block.

source code. It is the responsibility of the user to identify which part(s) are selected to run in parallel and use the various constructs to ensure correct results.

The nature (private or shared), or "scoping," of the variables must also be specified, but everything else is handled by the compiler and OpenMP runtime system. This completely eliminates the book-keeping and handling of details found in many other parallel programming models.

In many cases, more control than simply the execution of a block of code in parallel is needed. Extensive additional functionality is available and easy to add to an application. Subsequent sections in this chapter describe the type of building blocks that are available to implement additional functionality.

1.2.1 The Parallel Region

A parallel program in OpenMP starts and ends with the *parallel region*. It is the cornerstone of OpenMP. This is where parallel execution takes place and multiple threads are put to work. The syntax in C/C++ and Fortran is given in Figure 1.1.

There is no limit on the number of parallel regions, but for performance reasons it is best to keep the number of regions to a minimum and make them as large as possible. In C/C++, it is good practice to use a comment string to mark the end of the parallel region. This is less of an issue in Fortran, because it has the mandatory terminating `!$omp end parallel` construct.

The thread that encounters the parallel region is called the *master thread*. It creates the additional threads and is in charge of the overall execution. The threads that are active within a parallel region are referred to as a *thread team*, or "team" for short. Multiple teams may be active simultaneously.

The specifications also distinguish between an *active* and *inactive* parallel region. In the case of the former, the team consists of more than the master thread. This

is in contrast with an inactive parallel region, which is executed only by the master thread.

The statements within the parallel region are executed by all threads, unless the optional `if` clause evaluates to `false`. In this case, only one thread executes the region. This implies that it is an inactive parallel region. Outside of the parallel region(s), the master thread executes the serial portions of the application.

All OpenMP directives are to be placed in the *dynamic* extent of a parallel region in order to have an effect. This means that they need not be visible from the parallel region, but must be in the execution path that starts at a parallel region.

An example is a function with OpenMP constructs that is called from within a parallel region. If the same function is called outside of a parallel region, the directives are ignored and only one thread executes the code.

The situation is somewhat different for the OpenMP runtime functions. These may be used anywhere in the program, but some return a meaningful result, only if executed from within a parallel region. Other functions *must* be executed outside of a parallel region. For example, the `omp_set_num_threads()` function is used to set the number of threads. It is illegal to use it within a parallel region, and the runtime behavior is undefined if this rule is violated.

Before discussing the major OpenMP constructs, the execution and memory model are presented. Every user must have a basic understanding of these topics.

1.2.2 The OpenMP Execution Model

The way an OpenMP program executes is defined by the execution model. The OpenMP model is illustrated in Figure 1.2.

An OpenMP program always starts in serial mode. The thread executing the serial parts of the application, that is, the code outside the parallel regions, is called the *master thread*. This thread runs throughout the lifetime of the program.

At some point during the execution, the additional threads are created by the runtime system. When and how this happens is implementation-defined, but the threads are guaranteed to be available when the parallel regions are encountered.[2]

The master thread is included in the total number of active threads in the parallel region. For example, in Figure 1.2, four additional threads are created when the first parallel region is encountered.

[2]This assumes nothing unusual has happened, causing one or more threads to be unavailable.

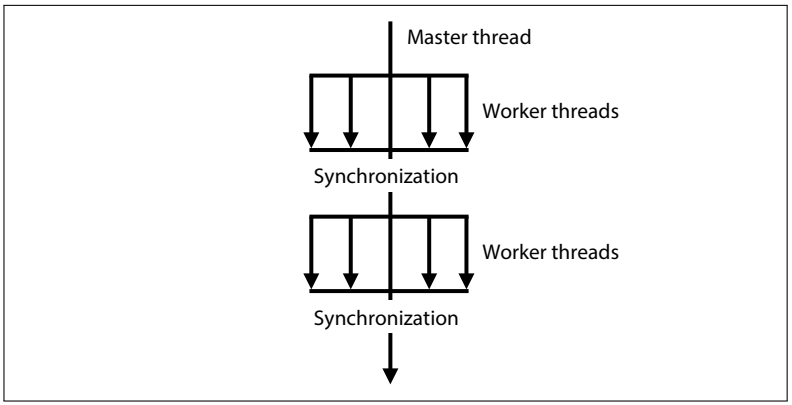

Figure 1.2: **The OpenMP execution model** – The master thread runs from start to finish. When a parallel region is encountered, the additional threads are engaged to perform the work that needs to be done. A barrier at the end of each parallel region synchronizes all threads. Execution continues only after the last thread has arrived at the barrier. After the parallel region exits, the master thread continues, until the next parallel region is encountered.

After this first thread-creation phase, multiple threads are active and within the parallel regions, the program executes in parallel. At the termination of a parallel region, the master thread continues until the next parallel region is encountered. This is called the *fork-join model*.

Until OpenMP 3.0, the user had no control over the behavior of the idle threads in-between the parallel regions, but the newly introduced environment variable `OMP_WAIT_POLICY` sets the behavior of idle threads. For more information on this variable, refer to Section 2.2 on page 53.

There are various ways to control the number of threads executing the parallel region. By default, this value is defined through environment variable `OMP_NUM_THREADS`. This sets the initial number of threads, but if not set by the user, a system-dependent default is used. Without any other action taken, it remains constant throughout the execution of the program.

If the number of threads needs to be more dynamic, the `omp_set_num_threads()` runtime function may be used to change the number of threads, while the application executes. This function affects the number of threads used in the next parallel region encountered, as well as all subsequent regions. This function

should only be executed *prior* to a parallel region. It is illegal to call it within a parallel region.

An alternative is to use the `num_threads(<nt>)` clause on the directive that defines the parallel region. The OpenMP runtime system uses the value of variable `nt` as the number of threads for the current parallel region, plus all subsequent parallel regions.

A useful feature is that the number of threads may increase, or decrease, on a per-parallel region basis. How this is implemented is transparent to the user and the responsibility of the runtime system. If the thread count shrinks, it is determined by the implementation how to handle the threads no longer needed. A common choice is to keep these threads around and avoid the relatively expensive setup cost of recreating threads in case the number increases again.

Thread synchronization occurs in the implied barrier at the end of the parallel region. The `nowait` clause is not supported on a parallel region. In other words, this barrier is always present and cannot be removed.

For a single-level parallel region this is a not a problem. It makes sense for all threads to finish their work before the master thread continues. With nested parallelism (see also section 1.5), this is not so obvious and the additional barriers may negatively impact performance.

1.2.3 The OpenMP Memory Model

Until now, a very important aspect of shared memory parallel programming has been glossed over: the memory model. It describes what types of memory there are and how threads interact with each of them. It also defines in what way the potentially different views on memory maintain consistency and, most importantly, when.

Until support for heterogeneous computing was introduced in OpenMP 4.0, there was a single address space only. This is still the case for the (host) system where the OpenMP program starts execution, but if accelerators are used, multiple address spaces may exist.

As illustrated in Figure 1.3, an OpenMP program has two different elementary types of memory: *private* and *shared*. To which memory type a variable belongs is defined by default rules, but may also be explicitly controlled through the appropriate clause on a construct.

In the case of the latter, the user must assign the appropriate memory type to variables used in the parallel region. Especially if the `default(none)` clause is

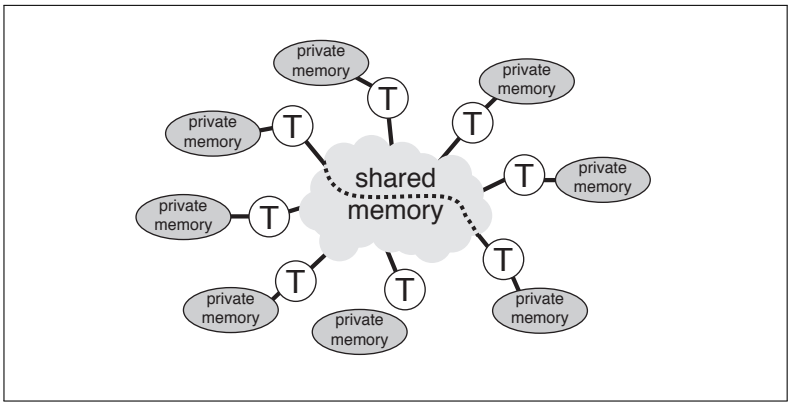

Figure 1.3: **The OpenMP memory model** – There are two different elementary types of memory: private and shared. Each thread has a private memory. A thread may access, or modify, variables in its own private memory, but not in another thread's private memory. In addition to this, there is a single shared memory. All threads have read and write access to variables stored there.

used, all variables need to be specified. Although more work upfront, it is strongly recommended not to rely on the default rules and explicitly label, or "scope," the variables.

The reason is that the default rules are clearly defined, but not always what one might expect. Knowing the subtleties is one option, but it is still easy to make a mistake. It is also not as difficult as it may seem at first sight to explicitly scope the variables.

There is one exception to this recommendation. In C/C++, variables declared within a code block are automatically made private. This is very convenient, and the next section has more information on this.

Private variables Each thread has unique access to its private memory and no other thread may interfere. There is never the risk of an access conflict with another thread. Although multiple threads may use the same name for a private variable, these variables are stored in different locations in memory. Modifications apply to the local copy only and a thread can never change the value of a private variable owned by another thread.

```
 1 x = 5;
 2 y = 20;
 3 #pragma omp parallel private(x) firstprivate(y)
 4 {
 5    x = 10;        // x is undefined on entry, but now set to 10
 6    int z = x + y; // y was pre-initialized to a value of 20
 7          ....
 8    y = 30;        // (first)private variables may be modified
 9          ....
10 } // End of parallel region
```

Figure 1.4: **Example of private variables** – Private variables are undefined on entry to (and exit from) the parallel region. The `firstprivate` clause at line 3 is used to pre-initialize variable `y` to 20 and use this value at line 6.

A loop variable is a simple example. If this variable is in private memory, as it must be, multiple threads may use the *same* variable name to execute a loop. It is impossible for one thread to modify the loop variable in the private memory of another thread.

If the same variable name is used outside, and within a parallel region, there is no risk of a conflict either. The reason is that the variable inside the region is treated as a different variable (even though it has the same name). For example, in the code fragment in Figure 1.4, variable `x` appears both outside, and inside the parallel region, but there is no conflict.

Although not yet available in OpenMP 2.5, more recent versions of the specifications describe many features in terms of tasks. In Chapter 3, tasking is discussed in great detail, but ahead of this, we need to introduce the concept of an *implicit task*. This is a task, generated by an implicit parallel region, or generated when a parallel construct is encountered during execution. In the context of a parallel region, or a worksharing construct, an "execution instance" is an implicit task. When a variable is private, each execution instance refers to its own instance of that variable. This topic also comes up with the support for SIMD instructions. See Section 5.2.1 for the definition of an execution instance with `simd` loops.

In OpenMP, the lifetime of a private variable is restricted. It only exists within a parallel region and by default, is undefined on entry to and exit from a construct. If needed, there are ways around this, but this must be explicitly handled through the appropriate constructs.

This is also illustrated in the example. Although variable `x` has been initialized to 5 by the master thread, its value is undefined within the parallel region. This is not a problem, because it is explicitly set to 10.

How about variable `y`? The value is needed within the parallel region, but it is not defined. The solution is to declare this variable to be `firstprivate`, instead of `private`. This not only guarantees that all threads have a private copy of this variable, but each copy is pre-initialized to the value `y` had prior to entering the parallel region.

This is also illustrated in the example. Upon executing the parallel region, variable `y` inherits the original value of 20. This is the value used at line 6. At line 8, it is reset to 30. This variable may actually have a different value within each thread, because it is a private variable.

The `firstprivate` clause is also supported on some constructs introduced after OpenMP 2.5. This is covered in the respective sections in the remainder of this book.

Variable `z` is declared within the parallel region. Per the rules of OpenMP, this makes it a private variable. This is a convenient feature and the exception to the recommendation not to rely on the default scoping rules.

In some cases, there is a need to use the value of a private variable after the parallel construct. For this situation, the worksharing `loop` and `sections` construct support the `lastprivate` clause.[3] This makes the value of such variable(s) accessible after the construct.

In a parallel application, it is not always clear what "last" means. For these constructs, this is defined such that the sequential semantics are preserved. For example, on a loop, it is the value as computed for the sequentially last iteration. This convenience comes at a (small) cost in performance, so use it with care.

There is one more thing worth mentioning here. What happens with the value of the original variable if it is also declared to be a private variable within the parallel region? The answer is that it depends on the version of OpenMP used. In OpenMP 2.5 the value is undefined upon exit of the parallel region. This was a

[3] This clause is also supported on `simd` loops, covered in Chapter 5.

highly undesirable limitation and has been lifted. As of OpenMP 3.0, the initial value of this variable is preserved.

Shared variables In addition to its private memory, each thread has access to the same single shared memory. There are two substantial differences with private memory.

In contrast with a private variable, only one instance of a shared variable exists. All threads "see" the same shared variable(s). In addition to this, each thread may read, as well as write, any shared variable. As long as the variable is within the dynamic extent, there is no constraint as to when this is allowed to occur. It is the responsibility of the user to correctly handle this situation

There is one case that requires extra attention and even warrants a specific directive: global, or static, variables. They are shared by default.

For example, consider the situation where a function uses global data as scratch space that is modified on each function call. If this function is called in parallel, a data race occurs, because multiple threads may modify the same memory address at the same time. A solution is to give each invocation of the function a private copy of the global data. In that way, even if multiple threads modify the data simultaneously, the updates are performed on different memory locations.

This is such a common scenario that, through the `threadprivate` directive, OpenMP provides elegant support for this situation. Each thread uses its private storage for the data that was originally shared. In case such privatized data must be pre-initialized, the `copyin` clause may be used.

The reason to be judicious when using shared variables, especially when they are modified, is performance-related. If multiple threads update the same cache line simultaneously and do this frequently, *false sharing* occurs and this has a negative impact on performance. In the worst case, this may lead to a substantial performance degradation. With private data, this is not an issue, so the recommendation is to use private data as much as possible. Shared variables are indispensable, but only use them if it is necessary to do so.

Memory Consistency Models The question is what happens when a modification has been made to a shared variable. The thread that made the change has immediate access to the new value, but what about the other threads? Intuitively, one might expect that all the threads see the same change at the same time as the thread that made the modification, but that is not the case.

Consider the following example. A thread initializes, or modifies, the value of two independent shared variables `x` and `y` in the order `x = ..., y = ....` The compiler may decide to interchange the statements and perform the assignment to `y` first. Unfortunately, such a valid optimization could be an issue if *another* thread assumes the value of `x` to be available *before* `y`. That is, the original order should not have been changed. The compiler has no way of knowing this though. It sees only the sequence of statements as executed by one thread and does not know how these statements may interact with other threads.[4]

When do the results of independent store instructions arrive in the main memory system and commit the values? It is not guaranteed that all the bytes of a variable are stored in one atomic transaction and actually may arrive piecemeal. Also, in what order do the variables arrive? Can it be guaranteed that they arrive in the order issued?

The fundamental question is how does one ensure that different threads see the same, consistent, value of a shared variable, and most importantly, when?

This is governed by the *memory consistency model*, or *consistency model* for short [13]. Such a model sets the rules regarding the handling of loads and stores. It defines *when* a thread is guaranteed to read the correct value of a variable after an update.

This is not a new issue. Over time, quite a number of consistency models have been defined, but for our purposes, only two cases are briefly covered. It is important to realize there is no right or wrong when it comes to the choice of a specific model. Performance and ease of use are the main differentiators and trade-offs.

Sequential Memory Consistency - This is defined by Lamport [16] and is intuitively easiest to understand. Each thread performs its loads and stores in the original sequential order, and stores are atomic. The memory operations may be thought of as a set, consisting of the loads and stores performed by this thread in the original sequence. Each of these sets is processed sequentially, but the order in which this happens is arbitrary.

In the code fragment in Figure 1.5, a classical example is shown. There are two threads and two shared variables `x` and `y`, both initialized to zero. Because of sequential consistency, either `{x,y} = {1,0}`, or `{x,y} = {0,1}`. This means only one thread executes the function call. Without sequential consistency, this is not

[4] At the hardware level, there is a potential issue too. Memory operations (among others) could be asynchronous because of caches, store buffers, buffer bypasses, and so on.

```
<Shared variables x and y are initialized to zero>

     Thread 0                Thread 1

     x = 1;                  y = 1;
     if ( y == 0 )           if ( x == 0 )
       function_call();        function_call();
```

Figure 1.5: **An example of the effect of sequential consistency** – Only one of the two threads executes the function call.

guaranteed. For example, if the compiler knows the function call has no side effects regarding `x` and `y`, it may decide to move the assignments after the if-statement. If that is the case, both threads call the function.

Unfortunately, excluding these kinds of optimizations and the imposed severe restrictions on the hardware regarding the handling of memory operations, reduces performance significantly. This is why many alternatives to sequential consistency have been proposed.

Relaxed Memory Consistency - This is not one single model, but a class of models, with a common goal to improve performance by relaxing the consistency requirements. The models differ in how far they go to relax the rules. The degrees of freedom are the order of the loads and stores, plus the atomicity.

Some examples of relaxed models are Total Store Order (TSO), Processor Consistency (PC), Relaxed Memory Ordering (RMO), and Release Consistency (RC). Covering these here is beyond the scope of this book. However, the details of the underlying memory consistency model need not be known in order to write a correct and efficient OpenMP program.

Memory Consistency and Cache Coherence - While a consistency model is needed for shared memory programming, cache coherence is not a necessary feature to implement OpenMP.

Cache coherence deals with the problem of *how* other cores see memory updates and does this by tracking changes in cache lines. A software implementation may do this, but hardware support is needed to make this efficient.[5] This is why hardware cache coherence has been implemented in shared memory systems for a long time.

[5]There are exceptions to this, but mostly in the area of special-purpose processors.

The confusion many users of OpenMP have is that, while cache coherence provides a single system image on a multi-core system, it does not prevent data races. There is no guarantee that a load instruction automatically returns the most recent value of a shared variable. This is defined by the consistency model and for performance reasons, most modern systems use a relaxed model.

The OpenMP View on Memory OpenMP assumes a relaxed memory consistency model, but does not require a specific implementation. Each thread is allowed to have its own *temporary view* of memory.[6]

As with any relaxed memory model, only at well-defined points, are threads guaranteed to have the same, consistent, view on the value of shared variables. In between such points, the temporary view may be different across the threads. OpenMP defines its own consistency points. At such a point, the temporary views are made consistent again. The question is how that impacts writing OpenMP code.

Read-only shared data is easy, because there are no concerns about consistency. Things get (much) more complicated when shared data is modified. Without precautions, any thread may modify shared data at any time. It is the responsibility of the user to ensure this is handled in the correct way.

As long as different threads write to a different memory location, for example, different elements of the same vector, there is no reason to worry. Problems arise if they simultaneously write to the *same* address in memory. Then, threads may step on each other and generate incorrect results. This is a bug in the code and is called a "data race."

A classical case is the accumulation of partial contributions into a shared variable. This occurs in reduction operations, for example. Each thread computes a contribution and adds it to the final solution, which is stored in a shared variable. This scenario is shown in the code fragment in Figure 1.6.

At line 1, shared variable `sum` is initialized to zero. This is outside the parallel region and performed by one thread only, the master thread. A local variable called `my_contribution` is declared at line 5 and per the rules of OpenMP, it is a private variable. Each thread performs its computations, and at line 7 the result is stored into variable `my_contribution`. In the next step, each thread reads the current

[6] The exception is the `seq_cst` clause on the `atomic` construct, which is covered in detail in Section 2.1.6, starting on page 47.

```
 1 sum = 0;
 2 #pragma omp parallel shared(sum)
 3 {
 4
 5   int my_contribution; // Contains the per-thread partial sum
 6     ....
 7   my_contribution = ....  // Thread computes a value
 8
 9   // Without the critical region this code has a data race
10
11   #pragma omp critical
12   {
13       sum += my_contribution;
14   } // End of critical region
15      ....
16 } // End of parallel region
```

Figure 1.6: **Example of the update of a shared variable** – To avoid multiple threads updating shared variable `sum` at the same time, a critical region is used.

value of variable `sum`, adds its contribution `my_contribution` to it, and stores the updated value of `sum`. This is the update statement at line 13, but without additional controls, this update has a data race on `sum` and the result is undefined.

The solution is to prevent another thread from reading `sum`, before the previous update has completed. OpenMP provides constructs like a critical section, or atomic update, to handle this situation. In Figure 1.6, the former is used to guard the update of variable `sum`. A thread is not allowed to enter a critical region, as long as another thread executes it. As a result, a thread cannot perform the update, while another thread is inside the critical region.

This is, however, not the end of the story. A relaxed memory consistency model is assumed, but what guarantee is there that a thread reads the correct value of `sum`, as it was recently updated by another thread? The answer is that, without additional measures, there is no such guarantee. Not even the initial value of zero is guaranteed to be available when the first thread performs the update. Unless this happens to be the master thread, but this cannot be counted upon.

This raises a very important question: *how does OpenMP ensure each thread reads the updated value?* This is where the `flush` construct comes in.

#pragma omp flush [*(flush-set)*] *new-line*
!$omp flush [*(flush-set)*]

Figure 1.7: **Syntax of the flush directive in C/C++ and Fortran** – This directive guarantees that all threads have the same consistent view of shared data.

The Flush Construct A key element of a relaxed consistency model is that the user needs to know when a shared variable may be read reliably.[7]

To enforce such global consistency, OpenMP provides the `flush` directive. After this directive, and before the next update of shared variables, all threads are guaranteed to have the same global view of all thread-visible variables. The syntax of the `flush` construct in C/C++ and Fortran is given in Figure 1.7.

The *flush-set* consists of a list with all the variables affected by the `flush` directive. Without this optional list, all thread-visible variables are affected. When a list is used, the flush-set consists of the variables in this list only. The recommendation is to *not* use the flush-set feature. It may be extremely difficult to employ correctly. If used incorrectly, optimizing compilers may apply valid code transformations, such that the `flush` does not have the intended effect.

Although compilers are not allowed to move operations on variables in the flush-set, beyond the `flush` directive, they are allowed to move flush constructs relative to each other in case the flush-sets are disjoint. For example, if the list contains variable `a` in one `flush` directive, and variable `b` in another one, both directives may be interchanged. This could result in incorrect values being read. To avoid this, both variables may be put in the same list, but this is also discouraged.

The `flush` directive affects only the thread executing it. The temporary view of other threads is not automatically updated. This implies that, in many cases, *all* threads must execute the same `flush` directive(s) to guarantee consistency.

The `flush` directive may be used explicitly in applications, but for convenience and as a safety net, a flush region is implied in the following situations:

- During a `barrier` region.
- At entry to, and exit from, the `parallel`, `critical`, `target`, and `target data` region.
- At exit from worksharing constructs, unless a `nowait` clause is present.

[7] The details depend on the model. Refer to Section 1.2.3 for an explanation of this.

- At entry to, and exit from, an `ordered` region, if a `threads` or `depend` clause is present, or if no clauses are present.
- Immediately before and after every task-scheduling point.
- During the following runtime functions: `omp_set_lock()` and `omp_unset_lock()`.
- During the following runtime functions, if the region causes the lock to be set or unset:
  - `omp_set_lock()`, `omp_unset_lock()`
  - `omp_test_lock()`, `omp_test_nest_lock()`
- At entry to, and exit from, the atomic operations (`read`, `write`, `update`, or `capture`), performed in a sequentially or non-sequentially consistent atomic region. In the case of the latter, the flush-set must contain the object updated in the construct.
- During a `cancel` or `cancellation point` region, in case cancellation has been enabled and activated.
- At entry to a `target update` region, whose corresponding construct has a `to` clause.
- At exit from a `target update` region, whose corresponding construct has a `from` clause.
- At entry to a `target enter data` region.
- At exit from a `target exit data` region.

The example in Figure 1.6 works correctly, because of the implied flush on the `critical` region. This ensures that the correct value is read before the update. The new value is synchronized, prior to exiting the region. In this case, the latter is not required for correctness, but it won't hurt either.

A flush region is *not* implied at the following locations:

- At entry to a `worksharing` region.
- At entry to, or exit from, a `master` region.

```
1 #pragma omp flush
2
3 while ( execution_state[i] != READ_FINISHED ) {
4
5    <wait for a short while>
6
7    #pragma omp flush
8
9 } // End of while-loop
```

Figure 1.8: **Example of the use of the flush directive** – The purpose of this while-loop is to wait for element `execution_stats[i]` to be modified.

The `master` construct is different from the `single` construct. Often, such a single thread region is used to initialize shared data. If the `master` construct is used for this, care needs to be taken that the other threads read the correct value(s). Without a `flush` region prior to accessing these variables, this is not guaranteed.

In many situations, the built-in flush operation avoids the need to explicitly use a flush directive. Even when the `nowait` clause is used, a barrier further down the execution path ensures a synchronized view on the shared data again.

This is why most OpenMP applications do not need to use an explicit `flush` construct, but in some cases, it is required for correct execution. An example of this is shown in Figure 1.8.

In this example, it is assumed that the master thread is in control of the value of variable `execution_state[i]`. The threads executing the loop, wait for the value to change. Without the two `flush` directives, there is no guarantee that changes in the value are propagated to the other threads. As a result, they might wait forever to actually see the change, so the program hangs.

The `flush` directive at line 1 is needed to ensure that the first time `execution_state[i]` is read, the correct value is returned at line 3. The second `flush` directive at line 7 guarantees this variable is updated, before it is read again at line 3. Although not shown here, the master thread must use a `flush` directive after setting and changing `execution_state[i]`.

A related, and commonly asked question is about pointers. If a pointer variable is part of a flush-set, *only* the pointer is synchronized, not the block of memory it points to.

The code fragment in Figure 1.8 is taken from the pipeline example presented and discussed in Section 1.3.2. The arrays used there implement dependences between activities. A much more robust and elegant solution, without the need for the `flush` directive, is provided by task dependences. This is discussed extensively in Chapter 3, where the same example is shown, but without explicit flush operations.

1.3 The Worksharing Constructs

A worksharing construct must be placed within a parallel region. Upon encountering a worksharing construct, the runtime system distributes the work to be performed over the threads active in the parallel region.

There are three worksharing constructs in C/C++/Fortran and a fourth one in Fortran. None of the worksharing constructs have a barrier on entry. There is one on exit, but it is omitted when using the `nowait` clause, which also means that there is no implied flush operation.

There are two important restrictions regarding worksharing constructs. First of all, either all threads in the team, or none at all, encounter the construct. As a consequence of this rule, threads are not allowed to skip a worksharing construct. The second rule is that the sequence of worksharing constructs and barriers encountered must be the same for all threads. If an OpenMP program does not comply with these rules, the behavior is undefined.

1.3.1 The Loop Construct

The loop construct provides for a straightforward way to assign the work associated with loop iterations to threads. The syntax for C/C++ and Fortran is shown in Figure 1.9.

The `schedule` clause is optional and may be used to explicitly specify the mapping of loop iterations onto threads. In OpenMP 2.5, there are three different scheduling types: `static`, `dynamic`, and `guided`. All three take an optional `chunk` parameter to control how many iterations are to be processed each time a thread gets assigned new work.

From a runtime overhead point of view, `static` is most efficient. The assignment of work to threads is pre-defined and very easy to determine at runtime. It works best with a regular workload, where each thread roughly gets assigned the same amount of work.

#pragma omp for *[clause[[,] clause]...] new-line* *for-loop(s)*
!$omp do *[clause[[,] clause]...]* *do-loop(s)* **[!$omp end do** *[nowait]***]**

Figure 1.9: **Syntax of the parallel loop construct in C/C++ and Fortran** – The loop iterations are distributed over the threads and executed in parallel. There are no curly braces in C/C++. The terminating `!$omp end do` on the Fortran directive is optional, but is recommended to clearly mark the end of the construct.

The other two types are more suitable in the case of a load imbalance, but they are more expensive in terms of overhead. The work assignment is handled at runtime and requires some locking. This is why these scheduling algorithms may not be the best choice if the imbalance is modest only.

A fourth keyword supported on this clause is `runtime`. As suggested by the name, the distribution is decided at runtime and controlled through the `OMP_SCHEDULE` environment variable. It is ideally suited to experiment with various policies and may even make this choice dependent upon the problem size and other characteristics.

If the schedule clause is not specified, the choice is made by the compiler and is therefore system-dependent.

OpenMP 3.0 and subsequent releases added more options and features to this construct. This includes an optional modifier on the `schedule` clause. For this and more details on the new features, refer to Section 2.1.1, starting on page 41.

1.3.2 The Sections Construct

Parallel sections provide for a very different structure to express and implement parallelism. Although generally applicable, this feature targets a higher level of parallelism. In Figure 1.10, the syntax in C/C++ and Fortran is shown for this directive.

Parallel sections are ideally suited to call different functions in parallel. In [2], it is explained how this is used to set up a pipeline to overlap I/O with computations. The assumption is that not all I/O needs to be completed before processing may start. In Figure 1.11, a code fragment based on this example is given. Three sections are used to implement the functionality needed.

```
#pragma omp sections [clause[[,] clause]...] new-line
{
    [#pragma omp section ]
        structured block
    [#pragma omp section
        structured block ]
    ...
}
```

```
!$omp sections [clause[[,] clause]...]
    [!$omp section ]
        structured block
    [!$omp section
        structured block ]
    ...
!$omp end sections [nowait]
```

Figure 1.10: **Syntax of the parallel sections construct in C/C++ and Fortran** – The number of sections both controls and limits the amount of parallelism. If there are "n" of these code blocks, at most "n" threads execute in parallel, but if nested parallelism is used, additional threads may be active.

The first section reads a block of data for each loop iteration. After completing the read for iteration `i`, a notification is posted. This is implemented in function `signal_read()`. Meanwhile, the second thread executes function `wait_read()`. It waits for a ready signal to be set. As soon as this is posted, the computations are performed. Once these have completed, another signal is set through function `signal_processed()`. The thread waiting in the third section picks this up, and starts the post-processing phase.

The threads communicate through flags set in memory and explicit flushes are needed to ensure threads see the modified values of the flags. This is illustrated in the code fragment shown earlier in Figure 1.8, which has been derived from the same pipeline example program.

Parallel sections are straightforward and easy to use, but they are most effective if there are only a relatively small number of sections. In cases where it is unknown upfront how many activities there are, or if the work performed is less regular, OpenMP tasks are more suitable. Tasks are covered extensively in Chapter 3, where this example is revisited. In Section 3.4, starting on page 128, it is shown that in

```
 1 #pragma omp parallel sections
 2 {
 3   #pragma omp section
 4   {
 5     for (int i=0; i<n; i++) {
 6        (void) read_input(i);
 7        (void) signal_read(i);
 8     }
 9   }
10   #pragma omp section
11   {
12     for (int i=0; i<n; i++) {
13        (void) wait_read(i);
14        (void) process_data(i);
15        (void) signal_processed(i);
16     }
17   }
18   #pragma omp section
19   {
20     for (int i=0; i<n; i++) {
21        (void) wait_processed(i);
22        (void) post_process_results(i);
23     }
24   }
25 } // End of parallel sections
```

Figure 1.11: **Example of the use of parallel sections** – A pipeline structure is set up to overlap I/O, computations, and post-processing.

this case, OpenMP tasks provide a more natural and elegant solution, because the arrays used for the signaling are not needed when using tasks. This eliminates the need for the explicit use of the `flush` construct.

1.3.3 The Single Construct

Sometimes there is a need within a parallel region to perform an activity by one thread only, such as when performing I/O for example.

#pragma omp single *[clause[[,] clause]...] new-line* *structured block*
!$omp single *[clause[[,] clause]...]* *structured block* **!$omp end single** *[nowait,[copyprivate]]*

Figure 1.12: **Syntax of the single construct in C/C++ and Fortran** – Only one thread executes the structured block. Unlike with the `master` construct, there is an implied barrier at the end.

The parallel region could be split into two separate regions and the master thread assigned to handle this single thread task, but this is not ideal for performance and requires more coding. The `single` construct is ideally suited for situations like this. It is not known which thread performs the work, which may vary from run to run. This is not an issue because there is no need to distinguish which thread does the work. The syntax in C/C++ and Fortran for this directive is given in Figure 1.12.

What is important, is that there is an implied barrier at the end of this construct. Regardless of which thread executes the code, all other threads wait for this thread. This is why the `nowait` clause may be useful, but be careful not to introduce a data race. If the `single` region is used to initialize data or to read from a file and the `nowait` clause is used, care needs to be taken that the other threads do not use this data before it has been initialized. There is also no implied `flush` construct. If the `nowait` clause is used, an explicit barrier, placed prior to the point where the data is used, guarantees the operations are completed and the data synchronized before execution continues.

This construct has a specific and unique clause called `copyprivate (list)`. This clause ensures all threads have a copy of the variables specified in the list. The values are copied into the private instance of the same variable. All threads wait in the single region until all copies have completed. This is why this clause is not allowed in combination with the `nowait` clause.

Often, the same may be achieved using shared variables. Each thread automatically has access to these, but this clause is more than a convenience. In the case of a recursion, it is more complicated to handle this explicitly, and the `copyprivate` clause provides a convenient solution.

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

Figure 1.13: **Syntax of the workshare construct in Fortran** – This construct is used to parallelize (blocks of) statements using array-syntax.

1.3.4 The Fortran Workshare Construct

The fourth and last worksharing construct is available in Fortran only. This is because it has been designed to parallelize array syntax, a language feature not supported in C/C++. The syntax is shown in Figure 1.13. The only clause supported is the `nowait` clause.

1.3.5 The Combined Worksharing Constructs

The combined constructs are shortcuts and useful in the event there is only a single worksharing construct in a parallel region. The semantics are identical to the version with one worksharing construct within a parallel region. The syntax in C/C++ and Fortran is given in Figures 1.14 and 1.15.

Clauses from both directives may be used, but only if there is no conflict.[8] The application is an illegal OpenMP program if the behavior depends on whether the clause was applied to either the parallel construct, or the worksharing construct.

The combined constructs are not only convenient and easier to read, there is also a (potential) performance difference. To start with, there is only one implied barrier at the end. The compiler may also generate more efficient code. A general parallel region provides significant flexibility, but this may cost some performance. The combined constructs are fairly simple from an OpenMP point of view and this might be exploited by compilers.

Although it may not seem meaningful to embed a single threaded code block in a parallel region, and there is indeed no combined construct for this, this combination actually does have an interpretation in the context of OpenMP tasking. This is explained in detail in Chapter 3.

[8] The `if` clause has been extended to support the name of the directive. See also Section 2.1.2.

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]
	  ...
	}

!$omp parallel
  !$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

Figure 1.14: **Syntax of the combined worksharing constructs in C/C++ and Fortran** – The combined constructs may have a performance advantage over the more general parallel region with only a single worksharing construct embedded. This is because a parallel region has some overhead the compiler may now eliminate. The Fortran combined workshare construct is listed in Figure 1.15.

Full version	Combined construct

!$omp parallel
  !$omp workshare
  structured block
  !$omp end workshare
!$omp end parallel
	!$omp parallel workshare
	  structured block
	!$omp end parallel workshare

Figure 1.15: **Syntax of the combined workshare construct in Fortran** – The combined construct may have a performance advantage.

#pragma omp master *new-line*   *structured block*
!$omp master   *structured block* **!$omp end master**

Figure 1.16: **Syntax of the master construct in C/C++ and Fortran** – The master thread is guaranteed to execute the structured block.

1.4 The Master Construct

Although not a worksharing construct, the `master` construct is very closely related to the `single` worksharing construct. The syntax in C/C++ and Fortran is shown in Figure 1.16.

As suggested by the name, the master thread is guaranteed to execute this block of code. It may be seen as a special case of the `single` construct, but there is one important difference. In contrast with the `single` construct, the `master` construct does *not* have an implied barrier on exit.

Depending on the operations performed in the master region, this may give rise to a data race. If the master thread initializes or reads shared data, and there is no barrier before the other threads use this data, a data race occurs. This construct does not have an implied `flush` construct either.

1.5 Nested Parallelism

For certain algorithms, it is natural to use *nested parallelism*. With this, each thread within a parallel region initiates a second parallel region. If needed, this repeats to the nesting level desired.

```
1 #pragma omp parallel num_threads(3)
2 {
3    <code in here is executed by 3 threads>
4    #pragma omp parallel num_threads(2)
5    {
6        <code executed by 3x2 = 6 threads>
7    } // End of second level parallel region
8 } // End of first level parallel region
```

Figure 1.17: **Code fragment for a two-level nested parallel region** – There are two parallel regions. Each region may have a different number of threads. In this example, they are set through the `num_threads` clause.

Nested parallelism is achieved by nesting the `parallel` constructs. At runtime, each thread that encounters the next parallel region creates a new parallel region. This happens as often as there are nested parallel region constructs.

In Figure 1.17, the code fragment for a two-level nested parallel region in C/C++ is shown. The dynamic behavior is illustrated in Figure 1.18.

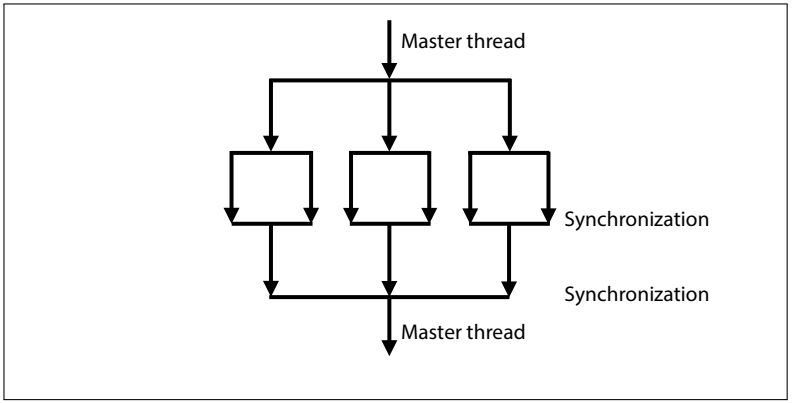

Figure 1.18: **An example of nested parallelism** – The master thread starts the program. Three threads are active in the first parallel region. Each one of these starts a separate parallel region. Because of the `num_threads(2)` clause on the second level parallel region, a total of 6 threads is active at this level.

In the example, three threads are active in the first parallel region. Because the `num_threads(2)` clause is used on the second parallel region, each of these threads creates a parallel region with two threads. At that point, a total of six threads are active.

This also demonstrates that thread teams do not need to be of the same size at every nesting level. A thread initiating another parallel region may set the number of threads through the `omp_set_num_threads()` function, or the `num_threads` clause. In case an `if` clause is used, and it evaluates to `false`, only one thread executes this parallel region.

At each nesting level, all parallel regions have their own master thread. As a consequence, the thread numbers are the same for all parallel regions at the same level. In this example, the second level threads in a team are numbered zero and one.

This is usually not a problem, but if a unique thread number is needed, the new runtime functions related to nested parallelism may be used to construct a thread ID that is unique across all nesting levels. These functions were introduced in OpenMP 3.0 and are discussed in Section 2.3.1 of Chapter 2. In Section 2.3.5, under "Nested Parallelism Revisited," it is shown how to use some of these functions to construct a unique thread ID across all nesting levels.

A common mistake made with nested parallelism is to try to nest *worksharing* constructs, which is not allowed. The nesting is at the level of the parallel region. It is valid to use the combined parallel worksharing constructs and nest them, because these are parallel regions after all. In Figure 1.19, this is illustrated with a parallel section that includes a parallel `for`-loop.

The three parallel sections execute simultaneously, assuming sufficient threads are available. If nested parallelism is enabled, the parallel for-loop at lines $8 - 13$ is executed by four threads and in total, six threads are active then.

Nested parallelism is an optional feature. An implementation is free to not support this, effectively ignoring the nested parallel regions. This feature may also be enabled, or disabled, either from within the application through the `omp_set_nested()` function, or by setting the `OMP_NESTED` environment variable. Keep in mind that the default setting for this variable is implementation-dependent. This means that nested parallelism may be turned off by default.

Nested parallelism definitely has its value, but it is rather rigid and the cost of the implied barrier that comes with each parallel region quickly adds up. In many cases, OpenMP tasks, covered extensively in Chapter 3, provide a much more convenient, flexible, and efficient mechanism.

```
 1 pragma omp parallel sections
 2 {
 3   #pragma omp section
 4   { printf("I am section 1\n"); }
 5   #pragma omp section
 6   {
 7      printf("I am section 2\n");
 8      #pragma omp parallel for shared(n) num_threads(4)
 9      for (int i=0; i<n; i++)
10      {
11          printf("Section 2:\tIteration = %d Thread ID = %d\n",
12                  i, omp_get_thread_num());
13      } // End of parallel for loop
14   }
15   #pragma omp section
16      { printf("I am section 3\n"); }
17
18 } // End of parallel sections
```

Figure 1.19: **Example of nested combined parallel worksharing constructs** – Three parallel sections are used. If sufficient threads are available, they are executed simultaneously. The second section includes another parallel region, consisting of a single `for`-loop. This loop is executed in parallel, resulting in nested parallelism. In this case the work in the loop is distributed over four threads.

1.6 Synchronization Constructs

In this section, several synchronization constructs are presented and discussed. They are used to control the behavior of the threads and it is quite likely an application needs at least one of them.

1.6.1 The Barrier Construct

The `barrier` construct forces all threads to wait until all of them have reached the barrier region in the program. Program execution continues once the last thread has arrived. The syntax in C/C++ and Fortran is given in Figure 1.20.

The barrier is used, for example, to avoid that a thread prematurely accesses data that is defined by another thread. This avoids data races and is illustrated in

#pragma omp barrier *new-line*
!$omp barrier

Figure 1.20: **Syntax of the barrier construct in C/C++ and Fortran** – All threads wait in the barrier until the last one has arrived.

```
 1 #pragma parallel shared(n,a,b)
 2 {
 3    #pragma omp for
 4    for (int i=0; i<n; i++)
 5        a[i] = i;
 6
 7    // The implied barrier on the for-loop above is needed there
 8
 9    #pragma omp for nowait
10    for (int i=0; i<n; i++)
11        b[i] += a[i];
12 } // End of parallel region
```

Figure 1.21: **Example to demonstrate the need for a barrier** – The implied barrier on the first loop is needed to avoid a data race on array element `a[i]`.

the code fragment in Figure 1.21. The implied barrier on the first parallel loop is needed to avoid a data race when reading array element `a[i]` at line 11. Without the implied barrier on the first for-loop, there is no guarantee that the value of `a[i]`, read in the second loop, has been updated.

The parallel region has an implied barrier upon exit. This is why the `nowait` clause at line 9 may be used to avoid two back-to-back barriers. Although the time spent in the second barrier should be short, there is still an additional cost that is best to avoid.

The implied flush operation that comes with the barrier is essential here. Without this, there is no guarantee that the correct value of `a[i]` is read in the second loop.

There is another possibility to fine-tune this code. The iteration space of both loops is identical, and the same elements of array `a` are accessed. Unfortunately in OpenMP 2.5, there is no guarantee that, even with static scheduling on loops with the same iteration space, the same thread operates on the same set of iterations across the loops. A data race may still occur if a `nowait` clause is used on the first loop.

```
1 #pragma parallel for shared(n,a,b)
2 for (int i=0; i<n; i++)
3 {
4     a[i] = i;
5     b[i] += a[i];
6 } // End of the parallel region
```

Figure 1.22: **Reworked example to eliminate a barrier** – By fusing (or merging) the two loops, one barrier is eliminated. On top of that, a combined worksharing construct is used.

This undesirable situation has been changed as of OpenMP 3.0. The `static` clause may now be used, because the mapping of iterations onto threads is preserved when using this scheduling type. This eliminates the need for the barrier, and a `nowait` clause may be used on the first loop. This is already more efficient, but in this case, another solution is preferred.

In addition to entirely avoid the barrier inside the parallel region, loop fusing creates more work per loop iteration, and cache line reuse improves. By using a combined worksharing construct, the execution time is reduced further. This approach is shown in Figure 1.22.

1.6.2 The Critical Construct

This is probably one of the most commonly used synchronization constructs, because it is ideally suited to avoid a data race when updating a shared variable.

A critical region is a block of code executed by all threads, but it is guaranteed that only one thread at a time may be active in the region. There is no implied

#pragma omp critical *[(name)] new-line* *structured block*
!$omp critical *[(name)]* *structured block* **!$omp end critical** *[(name)]*

Figure 1.23: **Syntax of the critical construct in C/C++ and Fortran** – The structured block is executed by all threads, but only one thread at a time executes the block. The construct may have an optional name.

```
 1 #pragma parallel
 2 {
 3              ...
 4    #pragma omp critical (c_region1)
 5    {
 6       sum1 += ...
 7    }
 8              ...
 9    #pragma omp critical (c_region2)
10    {
11       sum2 += ...
12    }
13                ...
14 } // End of the parallel region
```

Figure 1.24: **An example using named critical regions** – Different threads may be in either one of the two different critical regions at the same time.

barrier on exit from the region, but there is an implied flush construct on entry and exit. The syntax in C/C++ and Fortran is given in Figure 1.23.

The main use of a critical region is to update a shared variable, because with this construct, it can never happen that two or more threads simultaneously update the same variable. This prevents a data race when performing an update.

All critical regions without a name are considered to have the same internal name and are dynamically ordered. Critical regions with different names are allowed to execute in parallel. This allows for advanced synchronization, but care needs to be taken that there are no unwanted side effects when updating shared data. In Figure 1.24, an example with two different critical regions is given.

If multiple named critical regions are used, different threads may simultaneously execute different regions. If a lexically later region uses a variable that is updated in an earlier region, a data race is introduced.

This example produces correct results as long as variable `sum1` is not used in the second critical region. If so, both regions should have the same name or no name at all.

In OpenMP 4.5, an optional *hint* is supported. This may be used on the `critical` construct to allow the implementation to tune the underlying code to the specific

#pragma omp atomic *new-line* *statement*
!$omp atomic *statement*

Figure 1.25: **Syntax of the atomic construct in C/C++ and Fortran** – The statement is executed by all threads, but the loads and stores of the updated variable are atomic. There are constraints regarding the type of expression.

access pattern of the critical region. More details are found in Sections 2.1.5 and 2.3.3 on pages 46 and 67, respectively.

1.6.3 The Atomic Construct

The `atomic` construct has been in the specifications from the very first OpenMP 1.0 release. The name is inspired by particle physics and chosen to reflect that an atomic operation cannot be further broken down into sub-operations. It is uninterruptible. The syntax in C/C++ and Fortran is given in Figure 1.25.

The atomic construct can be used to guarantee mutually exclusive access to a specific memory location, represented through a variable. The access to this location is guaranteed to be atomic.

Conceptually, the atomic construct is somewhat similar to a critical region, but the level of atomicity is finer grained: only the load and store instructions are atomic. By restricting the functionality, compilers may generate more efficient code using low-level atomic instructions supported by many processors.

A simple, but typical, example is given in Figure 1.26. Variable `x` is incremented by `1`. While a thread is performing this operation, no other thread can access `x`. The load and store instructions involved in the update are guaranteed to be atomic. This is to avoid a thread reading variable `x` before the thread that currently executes the update has completed the write.

In C/C++, the `atomic` construct may be used together with an expression statement to initialize, or update, a single variable, possibly through a simple binary operation. The supported operations are: `+, *, -, /, &, ^, |, <<, >>`.

In Fortran, the statement must also take the form of an update to a variable. The operator used for the update may be only one of `+, *, -, /, .AND., .OR., .EQV., .NEQV.`, and the intrinsic procedure `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`.

```
1 #pragma omp parallel
2 {
3      ......
4    #pragma omp atomic
5        x += 1;
6      ......
7 } // End of parallel region
```

Figure 1.26: **Example of the atomic construct** – The `atomic` construct at line 4 guarantees that the load and store instructions related to the update of variable `x` at line 5 are atomic.

There are several 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. If the right-hand side includes a function call that is not thread-safe, the result is undefined.

Since OpenMP 3.1, this construct has been extended over the various releases and is quite comprehensive by now. Refer to Section 2.1.6 in Chapter 2 for an overview and more details.

1.6.4 The Ordered Construct

The ordered construct is applicable to parallel loops only. The construct must be placed within the parallel loop. The syntax in C/C++ and Fortran is given in Figure 1.27.

In addition to this construct, the `ordered` clause must be specified on the loop directive as well. The ordered region within the loop body is guaranteed to be executed in the order of the loop iterations, essentially serializing and ordering execution.

Valid use might be to enforce output to be printed in the original sequential order, or to debug a particular block of code. Because parallel processing changes the order of computations, floating-point round-off behavior could be different compared to the serial code. To verify whether this is the case, the `ordered` construct may be used. This construct may also be used to isolate a data race by narrowing down the region where this may occur. This construct is expensive and therefore should be used only if no suitable alternative is available.

#pragma omp ordered *new-line* *structured block*
!$omp ordered *structured block* **!$omp end ordered**

Figure 1.27: **Syntax of the OpenMP 2.5 ordered construct in C/C++ and Fortran** – This construct is placed within a parallel loop. The structured block is executed in the sequential order of the loop iterations. As of OpenMP 4.5, this construct has been enhanced to support additional features.

Function name	Description
`OMP_NUM_THREADS`	Set the number of threads used.
`OMP_SCHEDULE`	Set the runtime scheduling type and chunk size.
`OMP_DYNAMIC`	Enable/disable dynamic thread adjustment.
`OMP_NESTED`	Enable/disable support for nested parallelism.

Figure 1.28: **The OpenMP 2.5 environment variables** – These are the four variables one may set prior to program start up.

As of the OpenMP 4.5 specifications, the `ordered` construct has been enhanced to support `doacross` and `SIMD` loops. For details refer, to Sections 2.4.4 and 5.2.7, respectively.

1.7 The OpenMP 2.5 Environment Variables

In OpenMP 2.5, four environment variables are available to initialize the corresponding runtime settings. They are listed in Figure 1.28. If not set by the user, an implementation-dependent default is used. The exception is nested parallelism. It is turned off by default and may be activated by setting the `OMP_NESTED` variable to `true`.

As a general rule, a runtime function with the same functionality as an environment variable takes precedence. For example, by using the `omp_set_num_threads()` function, the initial value set by `OMP_NUM_THREADS` is overruled.

Since OpenMP 2.5, many new environment variables have been added. These are introduced and discussed in Section 2.2, starting on page 53. As explained there, variables `OMP_NUM_THREADS` and `OMP_SCHEDULE` are augmented to better support nested parallelism and additional workload scheduling features, respectively.

1.8 The OpenMP 2.5 Runtime Functions

Runtime functions may be used to query settings and also override initial values, either set by default or specified through environment variables defined prior to program start-up. Some of the runtime functions may also be used to modify settings while the application executes.

An example is the number of threads used to execute a parallel region. The initial value is implementation-dependent, but through the `OMP_NUM_THREADS` environment variable, this number may be set explicitly before the program starts. During program execution, function `omp_set_num_threads()` may be used to increase, or decrease, the number of threads to be used in the next parallel region(s).

When using the runtime functions in C/C++, file `omp.h` must be included. Fortran programs require either file `omp_lib.h` to be included or a module called `omp_lib` to be used. Either one of these, or both, are guaranteed to be available in Fortran.

Here, and in Section 2.3, the syntax is given for C/C++ only, but the usage in Fortran is very similar. Functions returning, or using, `true` or `false` as a function argument in C/C++, require the `LOGICAL` data type in Fortran. Many C/C++ functions are functions in Fortran as well, but in some cases they are subroutines instead. Refer to the specifications for details.

In Figure 1.29, the functions related to the execution environment are listed. Only the names and a short description are given. Most of these functions are straightforward to use, but there are a few things worth pointing out:

- When the `omp_get_num_threads()` function is used outside of a (nested) parallel region, a value of 1 is returned.
- The value returned by `omp_get_thread_num()` is relative to the parallel region that the thread executing this function is part of. As a result, it always starts at zero upon the execution of a new parallel region or when nested parallelism is used.
- The `omp_get_max_threads()` function returns the number of threads used in the next parallel region (unless the `num_threads` clause is used). This, for example, allows for the allocation of storage based on the number of threads in the parallel region.

Function name	Description
`omp_set_num_threads`	Set the number of threads.
`omp_get_num_threads`	Number of threads in the current team.
`omp_get_max_threads`	Number of threads in the next parallel region.
`omp_get_num_procs`	Number of processors available to the program.
`omp_get_thread_num`	Thread number within the parallel region.
`omp_in_parallel`	Check if within a parallel region.
`omp_get_dynamic`	Check if thread adjustment is enabled.
`omp_set_dynamic`	Enable, or disable, thread adjustment.
`omp_get_nested`	Check if nested parallelism is enabled.
`omp_set_nested`	Enable, or disable, nested parallelism.

Figure 1.29: **The OpenMP 2.5 runtime functions** – These are the functions to query or change settings related to threads, processors/cores, and the parallel execution environment. The names are the same in C/C++ and Fortran, but the usage follows the language syntax.

Function name	Description
`omp_get_wtime`	Absolute wall-clock time in seconds.
`omp_get_wtick`	The number of seconds between successive clock ticks.

Figure 1.30: **The two OpenMP 2.5 timing functions** – The first function is used to time specific portions of the program. The second function returns the clock resolution. It is used to determine what kind of time interval may be measured accurately.

In addition to this functionality, there are two functions to support measuring the execution time. The rationale to include these functions is to provide an accurate, as well as portable, way to measure the execution time of specific parts of the program. Although the specifications do not require this, the expectation is that the OpenMP implementation provides the most accurate timer available on a specific platform.

The two timer functions are listed in Figure 1.30 and are quite straightforward to use. The one thing to remember is that `omp_get_wtime()` returns the number of wall-clock or "elapsed" seconds. In other words, it returns an absolute value. A meaningful timing value is therefore obtained by taking the difference between two calls. This is illustrated in Figure 1.31.

The third set of runtime functions is related to locks. An overview of the locking functions is given in Figure 1.32. Locks provide a way to avoid that a block of code is executed by multiple threads at the same time. Conceptually this is very simple.

```
1 t_start = omp_get_wtime();
2    <code block to be timed>
3 t_wall_clock = omp_get_wtime() - t_start;
```

Figure 1.31: **An example how to use function omp_get_wtime()** – The value returned in variable `t_wall_clock` is the wall-clock time of the code block enclosed between the two calls to this timer function. To obtain accurate measurements, it is highly recommended for this time to significantly exceed the value returned by the `omp_get_wtick()` function.

A thread tests for the lock to be available. If this is not the case, the application needs to handle this. In most cases, it waits for the lock to become available but may also perform some other work and check again later.

Once the lock is owned by the thread, no other thread may access it. This allows the thread owning the lock to perform its work, without the risk that other threads execute the same block of code at the same time. When the thread has finished, it needs to release the lock in order for another thread to gain access to it. Failure to do so results in a deadlock.

Locking functionality is by no means unique to OpenMP, but by including it in the specifications, the user has the guarantee that it is available and a natural part of the OpenMP environment.

Many more runtime functions have been added since OpenMP 2.5. These may be found in Section 2.3, starting on page 60.

1.9 Internal Control Variables in OpenMP

Internal to the implementation, OpenMP uses a set of *Internal Control Variables* (*ICVs*) to access and control runtime settings. An example of this is the number of threads used in the next parallel region. The ICVs are an implementation detail, but must be mentioned, because in the specifications, some behavior is explained in terms of ICVs.

The names of the ICVs are given in the specifications, for example `nthreads-var`, but these are suggestions only. An implementation is free to deviate from this and by design there is no impact on the user, because an OpenMP application never uses ICVs directly. Instead, the interaction is through environment variables and runtime functions.

Function name	Description
`omp_init_lock`	Initialize a lock variable.
`omp_init_nest_lock`	Similar, but for a nestable lock.
`omp_set_lock`	Blocking request to acquire the lock.
`omp_set_nest_lock`	Similar, but for a nestable lock.
`omp_unset_lock`	Release the lock.
`omp_unset_nest_lock`	Similar, but for a nestable lock.
`omp_destroy_lock`	Change the state of the lock to be unitialized.
`omp_destroy_nest_lock`	Similar, but for a nestable lock.
`omp_test_lock`	Non-blocking request to acquire the lock.
`omp_test_nest_lock`	Similar, but for a nestable lock.

Figure 1.32: **The OpenMP 2.5 locking functions** – These functions define, acquire, and release a lock. A nestable lock may be set multiple times by the same thread.

Initial values are often system-dependent and pre-defined. They may, however, be set by the user through OpenMP environment variables. The corresponding ICV is initialized to this value by the implementation.

The number of threads is again a good example. The default is system-specific and the implementation assigns an initial value to the corresponding ICV. The value set for environment variable `OMP_NUM_THREADS` overrides this ICV upon program start up.

At runtime, OpenMP runtime functions are available to query the value of a specific ICV (for example, `omp_get_num_threads()`) and if needed, may be used to change the value. For example, function `omp_set_num_threads()` changes the number of threads at the user level. The implementation handles this by changing the value of the corresponding ICV.

1.10 Concluding Remarks

This chapter has provided a brief summary of OpenMP 2.5 and has set the foundation for the remainder of this book. At the time this release of OpenMP came out, it was a compact, yet very powerful, parallel programming model for the SMP systems of those days.

Since then, shared memory computer systems have evolved to contain hundreds of cores with multiple threads per core. Most of these systems use a cc-NUMA architecture and optionally support hardware accelerators.

Through the subsequent four releases of the specifications that came out after OpenMP 2.5, these very complex and extremely powerful systems are fully supported in OpenMP. In the remaining chapters, all of the enhancements and new features added are covered in great detail.