

3 Writing a First OpenMP Program

In this chapter, we give a first impression of what it means to use OpenMP to parallelize an application. We familiarize the reader with the OpenMP syntax and give a few basic rules. We also explain how OpenMP can be used to parallelize an existing application in a manner that preserves the original sequential version.

3.1 Introduction

Using OpenMP to parallelize an application is not hard. The impact of OpenMP parallelization is frequently localized, in the sense that modifications to the original source program are often needed in just a few places. Moreover, one usually does not need to rewrite significant portions of the code in order to achieve good parallel performance. In general, the effort of parallelizing a program with OpenMP goes mainly into identifying the parallelism, and not in reprogramming the code to implement that parallelism.

Another benefit of this API is that one can organize the program source in such a way that the original sequential version is preserved. This is a major advantage if one or more validation runs of the parallel code should produce a wrong answer. As a temporary workaround, one can easily have the compiler generate a sequential version of the suspect parts of the application, while debugging the OpenMP code. One can also restructure the sequential code for parallelization first, test this version (without running it in parallel) and then implement the parallelism through the control structures provided by OpenMP. This two-phase approach enables a smoother migration from a sequential to a parallel program than is possible with an “all or nothing” paradigm such as MPI.

For C and C++ programs, pragmas are provided by the OpenMP API to control parallelism. In OpenMP these are called *directives*. They always start with `#pragma omp`, followed by a specific keyword that identifies the directive, with possibly one or more so-called clauses, each separated by a comma. These clauses are used to further specify the behavior or to further control parallel execution. The directive must not be on the same line as the code surrounding it. The general form of a directive is given in Figure 3.1. The standard continuation symbol (backslash, `\`) for pragmas can be used in directives. This helps to improve readability by breaking up long pragma sequences into smaller blocks, something we highly recommend for readability and maintenance. White space (spaces and tab characters) can be inserted before and after the `#` and should be used between the words. Note that OpenMP directives in C/C++ are *case-sensitive*.

```
#pragma omp directive-name [clause[[,] clause]...] new-line
```

Figure 3.1: **General form of an OpenMP directive for C/C++ programs** – The directive-name is a specific keyword, for example `parallel`, that defines and controls the action(s) taken. The clauses can be used to further specify the behavior.

In Fortran all OpenMP directives are special comments that must begin with a directive *sentinel*. The format of the sentinel differs between fixed- and free-form source files. In fixed-source format Fortran there are three choices, but the sentinel *must* start in column one and appear as a single word with no intervening characters. Fortran fixed-source form line length, white space, continuation, and column rules apply to the directive line. The sentinels for fixed-source format are listed in Figure 3.2.

In free-source format only one sentinel is supported. It can appear in any column as long as it is preceded only by white space. It must also appear as a single word with no intervening character. Fortran free-format line length, white space, and continuation rules apply. Figure 3.3 gives the syntax for free-format source.

```
!$omp directive-name [clause[[,] clause]...] new-line
c$omp directive-name [clause[[,] clause]...] new-line
*$omp directive-name [clause[[,] clause]...] new-line
```

Figure 3.2: **OpenMP directive syntax for fixed-source format in Fortran** – The directive-name is a specific keyword, for example `parallel`, that defines and controls the action(s) taken. The clauses can be used to further specify the behavior. With fixed-format syntax the sentinel *must* start in column one.

```
!$omp directive-name [clause[[,] clause]...] new-line
```

Figure 3.3: **OpenMP directive syntax for free-source format in Fortran** – The directive-name is a specific keyword, for example `parallel`, that defines and controls the action(s) taken. The clauses can be used to further specify the behavior. With free-format syntax the sentinel can appear in any column.

With Fortran, initial directive lines must have a space after the sentinel. Continued directive lines must have an ampersand (&) as the last nonblank character on the line, prior to any comment placed inside the directive. Note that the sentinel has to be repeated, as shown in the example in Figure 3.4.

The need in Fortran to repeat the sentinel on continuation lines distinguishes Fortran from C/C++. A classical beginner's mistake in Fortran is to forget to

!\$OMP	PARALLEL PRIVATE(...)	&
!\$OMP	SHARED(...)	

Figure 3.4: **Example of continuation syntax in free-format Fortran** – Note that the sentinel is repeated on the second line.

start the continued line with the required sentinel, resulting in a syntax error at compile time. Another difference from C/C++ syntax is that OpenMP Fortran directives are *case-insensitive*.

We mostly use the `!$omp` sentinel throughout this book.¹ This has the advantage that it is supported with both types of Fortran source text formatting.

A word of caution is needed regarding directives. If a syntax error occurs in their specification (e.g., if a keyword is misspelled in C/C++ or the directive does not start in the first column in a fixed-format Fortran program), an OpenMP compiler ignores the directive and may not issue a warning either. This situation can give rise to some surprising effects at run time. Unfortunately, all we can do here is advise the programmer to double check the syntax of all directives and to read the compiler documentation to see whether there is an option for sending warnings on potential parallelization problems.

3.2 Matrix Times Vector Operation

We now show the use of OpenMP to parallelize a simple operation that realizes a basic, but important, problem: multiplying an $m \times n$ matrix B with a vector c of length n , storing the result into a vector a of length m : $a_{mx1} = B_{mxn} * c_{nx1}$. This example was chosen because it is straightforward and is likely to be familiar to most readers, but at the same time it allows us to demonstrate key features of OpenMP. Moreover, it has been used in OpenMP versions of some important application codes [139].

Our example codes also illustrate one of the comments made above: one can take any of the parallelized source codes shown in this chapter, or in the rest of this book for that matter, compile it with an OpenMP compiler, and link the new object instead of the sequential one. At that point one has a parallel program. Typically the same compiler can be used for the sequential and parallel program versions, but with the appropriate option selected to get the compiler to recognize OpenMP features.

¹We may deviate from this rule when source fragments are taken directly from applications.

3.2.1 C and Fortran Implementations of the Problem

The most straightforward serial implementation of the matrix-vector multiply is to calculate the result vector a by computing the dot product of the rows of the matrix B and vector c as shown in Formula (3.1).

$$a_i = \sum_{j=1}^n B_{i,j} * c_j \quad i = 1, \dots, m \quad (3.1)$$

In the remainder of this chapter we refer to the algorithm based on Formula (3.1) as the “row variant.”

3.2.2 A Sequential Implementation of the Matrix Times Vector Operation

In Figures 3.5 and 3.6 on pages 39 and 40, respectively, we present the complete Fortran and C source code for a program that calls a matrix times vector routine with the name `mxv`. For ease of reference, source line numbers are shown in the program listings.

The main program code is similar for both languages. In the setup part, the user is prompted for the matrix dimensions m and n . Next, memory is allocated that will store the matrix B plus vectors a and c , and the data is initialized. Then, the `mxv` routine is invoked. After the computations have finished, the memory is released, and the program terminates. The source code for the C version is shown in Figure 3.7; the corresponding Fortran version is shown in Figure 3.8. Note that in Figure 3.7 the *restrict* keyword is used in lines 1–2 to notify the C compiler that pointers a , b , and c are restricted and hence occupy disjoint regions in memory.² This gives a compiler more possibilities to optimize the code. Another point worth mentioning is that, for performance reasons in the C version, array `b` is declared and used as a linear array, rather than a two-dimensional matrix. We explain this action in Section 5.2 of Chapter 5.

The source code for the Fortran implementation also closely follows the mathematical description of this operation. In this case, however, one does not need to specify that `a`, `b`, and `c` are restricted because Fortran does not permit `a` to overlap in memory with `b` or `c` unless this is specifically declared to be the case.

²This is a feature of the C99 standard. If it is not supported by a compiler, the keyword can be omitted, but performance may not be optimal. If the compiler provides a flag to indicate the pointers are restricted, it should then be used.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void mxv(int m, int n, double * restrict a,
5          double * restrict b, double * restrict c);
6
7  int main(int argc, char *argv[])
8  {
9      double *a,*b,*c;
10     int i, j, m, n;
11
12     printf("Please give m and n: ");
13     scanf("%d %d",&m,&n);
14
15     if ( (a=(double *)malloc(m*sizeof(double))) == NULL )
16         perror("memory allocation for a");
17     if ( (b=(double *)malloc(m*n*sizeof(double))) == NULL )
18         perror("memory allocation for b");
19     if ( (c=(double *)malloc(n*sizeof(double))) == NULL )
20         perror("memory allocation for c");
21
22     printf("Initializing matrix B and vector c\n");
23     for (j=0; j<n; j++)
24         c[j] = 2.0;
25     for (i=0; i<m; i++)
26         for (j=0; j<n; j++)
27             b[i*n+j] = i;
28
29     printf("Executing mxv function for m = %d n = %d\n",m,n);
30     (void) mxv(m, n, a, b, c);
31
32     free(a);free(b);free(c);
33     return(0);
34 }

```

Figure 3.5: **Main program in C** – Driver program for the `mxv` routine. The user is prompted for the matrix dimensions `m` and `n`. Memory is allocated and initialized prior to the call to `mxv`. It is released again after the call.

```

1  program main
2  interface
3      subroutine mxv(m, n, a, b, c)
4          integer(kind=4), intent(in)    :: m, n
5          real    (kind=8), intent(in)    :: b(1:m,1:n), c(1:n)
6          real    (kind=8), intent(inout):: a(1:m)
7      end subroutine mxv
8  end interface
9  real(kind=8), allocatable:: a(:), b(:,,:), c(:)
10 integer(kind=4)          :: m ,n, i, memstat
11
12 print *, 'Please give m and n: '; read(*,*) m, n
13
14 allocate ( a(1:m), stat=memstat )
15 if ( memstat /= 0 ) stop 'Error in memory allocation for a'
16 allocate ( b(1:m,1:n), stat=memstat )
17 if ( memstat /= 0 ) stop 'Error in memory allocation for b'
18 allocate ( c(1:n), stat=memstat )
19 if ( memstat /= 0 ) stop 'Error in memory allocation for c'
20
21 print *, 'Initializing matrix B and vector c'
22 c(1:n) = 1.0
23 do i = 1, m
24     b(i,1:n) = i
25 end do
26
27 print *, 'Executing mxv routine for m = ',m,' n = ',n
28 call mxv(m, n, a, b, c)
29
30 if ( allocated(a) ) deallocate(a,stat=memstat)
31 if ( allocated(b) ) deallocate(b,stat=memstat)
32 if ( allocated(c) ) deallocate(c,stat=memstat)
33 stop
34 end program main

```

Figure 3.6: **Main program in Fortran** – Driver program for the `mxv` routine. The user is prompted for the matrix dimensions `m` and `n`. Memory is allocated and initialized prior to the call to `mxv`. It is released after the call.

```

1 void mxv(int m, int n, double * restrict a,
2         double * restrict b, double * restrict c)
3 {
4     int i, j;
5
6     for (i=0; i<m; i++)
7     {
8         a[i] = 0.0;
9         for (j=0; j<n; j++)
10            a[i] += b[i*n+j]*c[j];
11     }
12 }

```

Figure 3.7: **Sequential implementation of the matrix times vector product in C** – This source implements the row variant of the problem. The loop at lines 9–10 computes the dotproduct of row i of matrix b with vector c . The result is stored in element i of vector a . The dotproduct is computed for all rows of the matrix, implemented through the `for`-loop starting at line 6 and ending at line 11.

3.3 Using OpenMP to Parallelize the Matrix Times Vector Product

We now develop the first OpenMP implementation of our problem, using major OpenMP control structures to do so. Here, we describe them briefly. Details of these and other OpenMP constructs are given in Chapter 4.

The row variant of our problem has a high level of parallelism. The dotproduct implemented in Formula (3.1) on page 38 computes a value a_i for each element of vector a by multiplying the corresponding elements of row i of matrix B with vector c . This computation is illustrated in Figure 3.9. Since no two dotproducts compute the same element of the result vector and since the order in which the values for the elements a_i for $i = 1, \dots, m$ are calculated does not affect correctness of the answer, these computations can be carried out independently. In other words, this problem can be parallelized over the index value i .

In terms of our implementation this means that we are able to parallelize the outer loop with iteration variable `i` in both the C and Fortran versions. We give the corresponding listings of the OpenMP source code for `mxv` in Figure 3.10 on page 44 for the C version and in Figure 3.11 on page 45 for the Fortran implementation.

In both program versions, we have inserted a `parallel` directive at lines 9–10 to define a *parallel* region. Three so-called clauses, `default`, `shared`, and `private`,

```

1      subroutine mxv(m, n, a, b, c)
2
3      implicit none
4      integer(kind=4):: m , n
5      real    (kind=8):: a(1:m), b(1:m,1:n), c(1:n)
6
7      integer(kind=4):: i, j
8
9      do i = 1, m
10         a(i) = 0.0
11         do j = 1, n
12             a(i) = a(i) + b(i,j)*c(j)
13         end do
14     end do
15
16     return
17 end subroutine mxv

```

Figure 3.8: **Sequential implementation of the matrix times vector product in Fortran** – This source implements the row variant of the problem. The loop at lines 11–13 computes the dotproduct of row i of matrix b with vector c . The result is stored in element i of vector a . The dotproduct is computed for all rows of the matrix, implemented by the `do` loop starting at line 9 and ending at line 14.

have been added. The meaning of these will be explained shortly. To improve readability, we use the continuation feature to break the directive into two pieces.

In the C version, the start of the parallel region is marked by the `#pragma omp parallel for` directive at line 9 and comprises the block of statements that ends at line 16. We have added a comment string to indicate the end of the parallel region, to help avoid the programming error that would arise if the curly braces that define the extent of the parallel region were incorrectly nested. In this particular case no ambiguity exists, but in general we have found this strategy to be helpful.

In the Fortran source code, the parallel region also starts at line 9, where the `!$omp parallel do` directive has been inserted, and ends at line 17 with the `!$omp end parallel do` directive. Fortran programs require an explicit `!$omp end parallel do` directive, since the language does not provide the equivalent of C’s curly braces to define a block of statements. Although this directive is not actually required in this specific situation, we use it here to clearly mark the end of the parallel region.

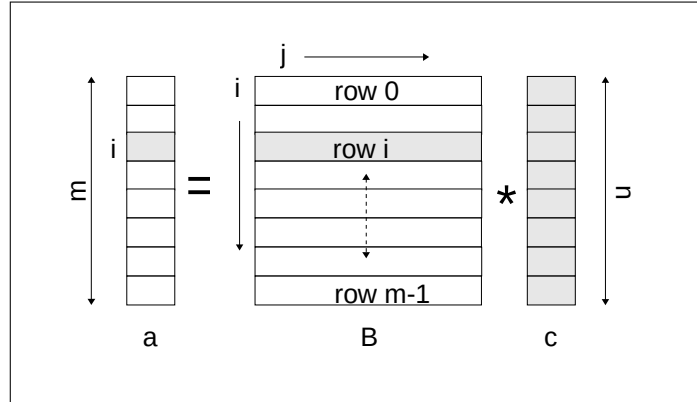


Figure 3.9: **Graphical representation of the row variant of the matrix times vector operation** – Element a_i is obtained by computing the dot product of row i of matrix B with vector c .

The directive is somewhat special. Since just one loop needs to be parallelized, we may use a single directive (the `#pragma omp parallel for` in C and the `!$omp parallel do` construct in Fortran) both to create a parallel region and to specify that the iterations of the loop should be distributed among the executing threads. This is an example of a *combined* parallel work-sharing construct (see Section 4.4.5 in the next chapter).

Data in an OpenMP program either is shared by the threads in a team, in which case all member threads will be able to access the same shared variable, or is private. In the latter case, each thread has its own copy of the data object, and hence the variable may have different values for different threads. The additional information given in the clauses for the directive in our example code specifies what data is shared between threads in the parallel region and which variables are private. Thus, each thread will access the same variable m , but each will have its own distinct variable i .

OpenMP provides built-in data-sharing attribute rules that could be relied on, but we prefer to use the `default(none)` clause instead. This informs the OpenMP compiler that we take it upon ourselves to specify the data-sharing attributes. This approach forces us to explicitly specify, for each variable, whether it is shared by multiple threads, and therefore implies more work on the part of the programmer. The reward, however, is twofold. First, we must now carefully think about the usage of variables, and this action helps us to avoid mistakes. Unless one understands the default data-sharing attribute rules very well, it is safer to be explicit

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void mxv(int m, int n, double * restrict a,
5          double * restrict b, double * restrict c)
6  {
7      int i, j;
8
9      #pragma omp parallel for default(none) \
10         shared(m,n,a,b,c) private(i,j)
11      for (i=0; i<m; i++)
12      {
13          a[i] = 0.0;
14          for (j=0; j<n; j++)
15              a[i] += b[i*n+j]*c[j];
16      } /*-- End of omp parallel for --*/
17  }

```

Figure 3.10: **OpenMP implementation of the matrix times vector product in C** – A single pragma (directive) is sufficient to parallelize the outer loop. We have also opted here to specify explicitly, for each variable, whether it is shared by all threads (**shared**) or whether each thread has a private copy. A comment string is used to clearly mark the end of the parallel region.

about them. Novice OpenMP programmers in particular may easily fall into the trap of making incorrect assumptions about the default rules. The second advantage of explicitly specifying the data-sharing attributes is more subtle. For good performance, one really would like to use private variables as much as possible. By carefully considering which variables have to be shared and which ones could be made private, we are sometimes able to create a faster parallel performance. This topic is discussed in Chapter 5.

Because we have used the **default(none)** clause, we have to decide on the nature of the use of variables in our parallel region. This task may seem harder than it actually is. First, all threads have to be able to access **m**, **n**, **a**, **b**, and **c**. Therefore, they need to be declared as **shared**. If the reason is not immediately clear, consider the alternative. If these variables were to be declared **private**, they would be uninitialized (by one of the rules of OpenMP). But then we would not know how many iterations of the loop to perform. Similarly, array **b** and vector **c** would also not be defined, and the computations could not produce meaningful results.

```

1      subroutine mxv(m, n, a, b, c)
2
3      implicit none
4
5      integer(kind=4):: m , n
6      real    (kind=8):: a(1:m), b(1:m,1:n), c(1:n)
7      integer      :: i, j
8
9      !$OMP PARALLEL DO DEFAULT(NONE)      &
10     !$OMP SHARED(m,n,a,b,c) PRIVATE(i,j)
11     do i = 1, m
12         a(i) = 0.0
13         do j = 1, n
14             a(i) = a(i) + b(i,j)*c(j)
15         end do
16     end do
17     !$OMP END PARALLEL DO
18     return
19     end subroutine mxv

```

Figure 3.11: **OpenMP implementation of the matrix times vector product in Fortran** – One OpenMP directive is sufficient to parallelize the outer loop. We have opted here to specify explicitly, for each variable, whether it is shared by all threads (**shared**) or whether each thread has a private copy. We have used the **!\$OMP END PARALLEL DO** directive to mark the end of the parallel region.

Moreover, if made private, the output vector **a** would not be accessible outside of the parallel region (another OpenMP rule).³ Therefore, this vector has to be shared as well. Loop counter variables **i** and **j** need to be declared **private**, however. Each thread has to have access to a local, unique copy of these variables. Otherwise, a thread would be able to modify the loop counter of another thread, thereby giving rise to incorrect and unpredictable runtime behavior.

We can now focus on how our code will be executed. Code outside a parallel region will be executed sequentially by a single thread. Only the code that is within a parallel region will be executed in parallel, by a *team of threads*. Typically, threads other than the initial thread are created the first time a parallel region is encountered. They may be reused for subsequent execution of the same parallel region or a different parallel region. More details can be found in Chapter 8.

³See also Sections 4.5.3 and 4.5.4 on how to initialize and save private variables.

The programmer assigns a portion of the work in the parallel region to each thread in the team by using one or more worksharing directives. In our case, we have specified that the loop immediately following the `#pragma omp parallel for` or `!$omp parallel do` construct (with loop variable *i*) should be distributed among threads. This means that different threads will execute different iterations of this loop, and each loop iteration will be performed exactly once. For example, thread 0 might work on iteration `i=5`, and iteration `i=10` could be assigned to thread 1. If there are more iterations than threads available, multiple iterations are assigned to the threads in the team. Distributing loop iterations among threads is one of the main *work-sharing* constructs OpenMP offers. By not specifying details of how the work should be distributed, here we have left it up to the implementation to decide exactly which thread should carry out which loop iterations. However, OpenMP also provides a `schedule` clause to allow the user to prescribe the way in which the loop iterations should be distributed. Section 4.5.7 goes into more detail on this.

As each thread in a team completes its portion of the work in a parallel region, it encounters a *barrier*. Here, the threads wait until the last one arrives, indicating that the work inside the parallel region has been completed. Subsequently, only the master thread (the thread that worked on the code prior to the parallel region) proceeds with the execution of statements outside the parallel region. More details on barriers can be found in Section 4.6.1. What happens with the other threads at the end of the parallel region is implementation dependent. All the OpenMP standard says is that the threads are forked at the start of the parallel region and joined at the end. For example, the threads that are no longer needed might be destroyed and threads recreated when another parallel region is encountered. Alternatively, they could be parked for later use or kept around in a busy-wait loop to be available when the program executes the same or another parallel region. It is worthwhile to check the documentation to find out how this situation is handled by an implementation and to discover how to change the behavior (if one is allowed to do so).

The C and Fortran sources listed above also demonstrate the “localized” impact an OpenMP parallelization often has. All it takes to get a parallel version is to compile the source with the appropriate OpenMP compiler option and use this object file to pass on to the linker instead of the original sequential object file.

3.4 Keeping Sequential and Parallel Programs as a Single Source Code

The parallelized version of `mxv` in Section 3.3 is fully functional. By compiling with the appropriate option, the OpenMP directives are translated into the appropriate multithreaded code, resulting in parallel execution at run time.

One of the powerful features of OpenMP is that one can write a parallel program, while preserving the (original) sequential source. In a way, the sequential version is “built-in.” If one does not compile using the OpenMP option (flag), or uses a compiler that does not support OpenMP, the directives are simply ignored, and a sequential executable is generated. However, OpenMP also provides runtime functions that return information from the execution environment. In order to ensure that the program will still compile and execute correctly in sequential mode in their presence, special care needs to be taken when using them. For example, let’s say one wishes to use the `omp_get_thread_num()` function that returns the thread number. If the application is compiled without OpenMP translation, the result will be an unresolved reference at link time. A workaround would be to write one’s own dummy function, but a better solution exists.

Specifically, the OpenMP runtime functions in both C/C++ and Fortran can be placed under control of an `#ifdef _OPENMP`, so that they will be translated only if OpenMP compilation has been invoked. The standard requires that this macro be set to `yyyyymm`, with `yyyy` being the year and `mm` the month when the specific standard for the OpenMP version was released. For example, for OpenMP 2.5, `_OPENMP` is set to 200505.

An example of the use of this macro in a C/C++ program is given in Figure 3.12. Here, the file `omp.h` will be included only if `_OPENMP` is defined. This header file is guaranteed to be available if an OpenMP-compliant compiler is used. It includes the interfaces of the OpenMP runtime functions. To be able to use function `omp_get_thread_num()`, we set its value to zero in sequential mode. This is also the thread number of the initial thread in an OpenMP code.

Figure 3.13 shows an example of similar functionality in Fortran. Just as in the C version, we check whether the `_OPENMP` macro is defined. If so, we use the `omp_lib` module, which serves the same purpose as the include file `omp.h` in C/C++. Unfortunately, the OpenMP standard does not require this module to be present. One must therefore check the compiler documentation to find out whether this module, or a Fortran `include` file named `omp_lib.h`, or both are provided. Our Fortran code also sets the thread number (stored in variable `TID`) to zero if `_OPENMP`

```

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif
.....
int TID = omp_get_thread_num();

```

Figure 3.12: **Example of conditional compilation in C** – This mechanism allows us to maintain a single source for sequential and parallel code even if we use OpenMP runtime functions.

has not been defined. Otherwise the value returned by the `omp_get_thread_num()` function call is assigned to `TID`.

```

#ifdef _OPENMP
    use omp_lib
#endif
.....
integer:: TID
.....
#ifdef _OPENMP
    TID = omp_get_thread_num()
#else
    TID = 0
#endif

```

Figure 3.13: **Example of conditional compilation in Fortran** – This mechanism allows us to maintain a single source for sequential and parallel code even if we use OpenMP runtime functions. There is an alternative solution in Fortran.

!\$	*\$	c\$
-----	-----	-----

Figure 3.14: **Conditional compilation sentinels in fixed-format Fortran** – At compile time an OpenMP compiler replaces the sentinel by two spaces.

An alternative solution in Fortran uses the *conditional compilation sentinel*. This special sentinel is recognized by the OpenMP compiler and at compile time is replaced by two spaces; when compiled without OpenMP translation, it is simply a comment line and will be discarded. The conditional compilation sentinels recognized in fixed-format source files are listed in Figure 3.14. A line with a conditional

compilation sentinel in fixed-format Fortran source will be left unchanged unless it satisfies the following criteria:

- The sentinel *must* start in column 1 and appear as a single word with no intervening space.
- After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and white space and numbers in columns 1 through 5.
- After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5.

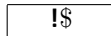


Figure 3.15: **Conditional compilation sentinel in free-format Fortran** – At compile time an OpenMP compiler replaces the sentinel by two spaces.

Free-format source has only one conditional compilation sentinel. The syntax is shown in Figure 3.15. The sentinel will be replaced by two spaces, and thus enable conditional compilation, only if the following four criteria are met:

- The sentinel can appear in any column but must be preceded by white space only.
- The sentinel must appear as a single word with no intervening white space.
- Initial lines must have a space after the sentinel.
- Continued lines must have an ampersand (&) as the last nonblank character on the line, prior to any comment appearing on the conditionally compiled line; continued lines can have an ampersand after the sentinel, with optional white space before and after the ampersand

The conditional compilation mechanism is quite useful. However, one has to be careful not to make a mistake with the syntax because, otherwise, the sentinel will be treated as a comment. Unfortunately, a program might have genuine comments that begin with these characters. If one uses the same style for general documentation of the program (for example `!$ This is my comment`), many syntax errors are going to be generated by an OpenMP compiler.

```
integer:: TID
.....
TID = 0
!$  TID = omp_get_thread_num()
```

Figure 3.16: **Example of the use of the conditional compilation sentinel** – At compile time an OpenMP compiler will replace the !\$ sentinel by two spaces, changing the comment line into an executable statement.

The most straightforward (and common) use of the conditional compilation sentinel in Fortran is shown in Figure 3.16. In this example, the program variable `TID` contains the thread number. It is initialized to zero, to be consistent with the thread number of the master thread. If, however, the source has been compiled with OpenMP enabled, the !\$ sentinel will be replaced by two spaces. The effect is that the “comment” line becomes the executable statement `TID = omp_get_thread_num()`, assigning the runtime thread number to variable `TID`.

The combination of directive-based syntax and conditional compilation enables one to write an OpenMP program that preserves the sequential version of the application and that can be translated into either sequential or parallel code.

3.5 Wrap-Up

In this chapter, we have introduced the basic idea of OpenMP via a simple example program that has enabled us to demonstrate the use of several of the most common features of the API. We have explained how OpenMP directives are written.

It is often easy to write an OpenMP program where the sequential version of the application is “built-in.” A sequential compiler simply ignores the OpenMP directives, because it does not recognize them. By checking whether the `_OPENMP` symbol has been defined or by using Fortran conditional compilation, one can make a compile-time substitution for the runtime functions to avoid unresolved references at link time. This feature can also be useful in case of a regression. If the numerical results are incorrect, for example, one can simply not compile the suspect source parts under OpenMP. The sequential versions of these sources will then be used as a (temporary) workaround.

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

¹All 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

²We 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:

³This 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 (<i>scalar-expression</i>)	(C/C++)
if (<i>scalar-logical-expression</i>)	(Fortran)
num_threads (<i>integer-expression</i>)	(C/C++)
num_threads (<i>scalar-integer-expression</i>)	(Fortran)
private (<i>list</i>)	
firstprivate (<i>list</i>)	
shared (<i>list</i>)	
default (none shared)	(C/C++)
default (none shared private)	(Fortran)
copyin (<i>list</i>)	
reduction (<i>operator:list</i>)	(C/C++)
reduction (<i>{ operator intrinsic_procedure_name } : list</i>)	(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 $i = 0, \dots, n - 1$.

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 (<i>list</i>)	
firstprivate (<i>list</i>)	
lastprivate (<i>list</i>)	
reduction (<i>operator:list</i>)	(C/C++)
reduction (<i>{operator intrinsic_procedure_name}:list</i>)	(Fortran)
ordered	
schedule (<i>kind[,chunk_size]</i>)	
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,

```

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

```

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.

```

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

```

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

⁴Which 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 (<i>list</i>)	
firstprivate (<i>list</i>)	
lastprivate (<i>list</i>)	
reduction (<i>operator:list</i>)	(C/C++)
reduction (<i>{ operator intrinsic_procedure_name } : list</i>)	(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 [<i>clause</i> [[,] <i>clause</i>]. . .]	<i>structured block</i>
--	-------------------------

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.

⁵The 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

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

⁷Chapter 7 covers these kinds of problems in depth.

private (<i>list</i>) firstprivate (<i>list</i>) copyprivate (<i>list</i>) 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 <i>structured block</i> !\$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 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
<pre>#pragma omp parallel { #pragma omp for for-loop }</pre>	<pre>#pragma omp parallel for for-loop</pre>
<pre>#pragma omp parallel { #pragma omp sections { [#pragma omp section] structured block [#pragma omp section structured block] ... } }</pre>	<pre>#pragma omp parallel sections { [#pragma omp section] structured block [#pragma omp section structured block] ... }</pre>

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] <i>structured block</i> [\$omp section <i>structured block</i>] ... !\$omp end sections !\$omp end parallel	!\$omp parallel sections [\$omp section] <i>structured block</i> [\$omp section <i>structured block</i>] ... !\$omp end parallel sections
!\$omp parallel !\$omp workshare <i>structured block</i> !\$omp end workshare !\$omp end parallel	!\$omp parallel workshare <i>structured block</i> !\$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`.

⁸We 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 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

⁹This 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 <i>chunk_size</i> . 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 <i>chunk_size</i> 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 <i>chunk_size</i> parameter), then requests another chunk until there are no more chunks to work on. The last chunk may have fewer iterations than <i>chunk_size</i> . When no <i>chunk_size</i> 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 <i>chunk_size</i> parameter), then requests another chunk until there are no more chunks to work on. For a <i>chunk_size</i> 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 <i>chunk_size</i> of “ <i>k</i> ” ($k > 1$), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than <i>k</i> iterations (with a possible exception for the last chunk to be assigned, which may have fewer than <i>k</i> iterations). When no <i>chunk_size</i> 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 <code>OMP_SCHEDULE</code> 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 $0, \dots, 199$. 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 $i = 0, \dots, 6$, whereas thread 1 processes $i = 21, \dots, 27$, thread 2 handles $i = 7, \dots, 13$ and thread 3 executes $i = 14, \dots, 20$. Thread 0 then continues with $i = 28, \dots, 34$ 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

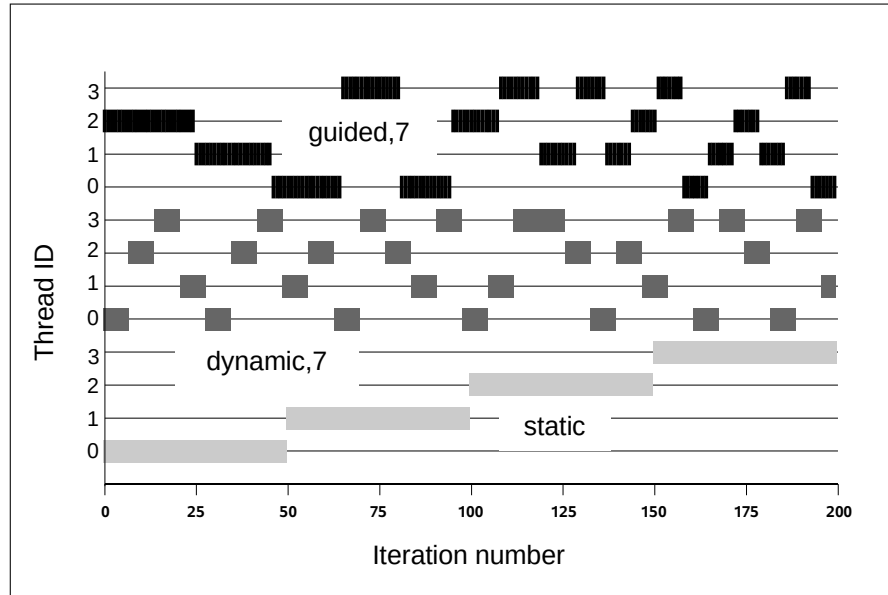


Figure 4.44: **Graphical illustration of the schedule clause** – The mapping of iterations onto four threads for three different scheduling algorithms for a loop of length $n = 200$ is shown. Clearly, both the `dynamic` and the `guided` policy give rise to a much more dynamic workload allocation scheme.

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 non-deterministic 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.

<pre>#pragma omp ordered <i>structured block</i></pre>
--

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.

<pre>!\$omp ordered <i>structured block</i> !\$omp end ordered</pre>
--

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.

<pre>#pragma omp critical [(name)] structured block</pre>

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.

<pre>!\$omp critical [(name)] structured block !\$omp end critical [(name)]</pre>

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 in 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.

- `omp_get_num_procs` returns, as an integer, the total number of processors available to the program at the instant in which it is called. The number will not depend on which thread calls the routine, since it is a global value.
- `omp_in_parallel` returns *true* if it is called from within an active parallel region (see Section 4.3). Otherwise, it returns *false*. The result value is of type integer in C/C++ and logical in Fortran.

The runtime library also includes routines for implementing locks and portable timers in an OpenMP program. The lock routines are described in Section 4.6.5.

4.8 More OpenMP Clauses

We introduced the most commonly used clauses in Section 4.5. In this section, we introduce the remaining ones. We remind the reader that no assumptions may be made about the order in which the clauses are evaluated. Except for the *if*, *num_threads* and *default* clauses, they may occur multiple times on a given construct, with distinct arguments. We give important rules for the use of clauses here. We refer the reader to the OpenMP standard for other rules and restrictions associated with specific clauses. The syntax for clauses is similar in Fortran and C/C++. The two exceptions are the **copyprivate** (see Section 4.8.6) and **nowait** (Section 4.5.6) clauses.

4.8.1 If Clause

The **if** clause is supported on the **parallel** construct only, where it is used to specify conditional execution. Since some overheads are inevitably incurred with the creation and termination of a parallel region, it is sometimes necessary to test whether there is enough work in the region to warrant its parallelization. The main purpose of this clause is to enable such a test to be specified. The syntax of the clause is **if**(*scalar-logical-expression*). If the logical expression evaluates to *true*, which means it is of type integer and has a non-zero value in C/C++, the parallel region will be executed by a team of threads. If it evaluates to *false*, the region is executed by a single thread only.

An example is shown in Figure 4.70. It uses the **if** clause to check whether the value of variable `n` exceeds 5. If so, the parallel region is executed by the number of threads available. Otherwise, one thread executes the region: in other words, it is then an *inactive* parallel region. Two OpenMP runtime functions are used. The function `omp_get_num_threads()` returns the number of threads

```
#pragma omp parallel if (n > 5) default(none) \
    private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
            omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*--- End of parallel region ---*/
```

Figure 4.70: **Example of the if clause** – The parallel region is executed by more than one thread only if $n > 5$.

in the current team. As seen before, the thread number is returned by function `omp_get_thread_num()`. The value is stored in variable `TID`. A `#pragma omp single` pragma is used (see also Section 4.4.3) as we want to avoid executing the first two print statements multiple times. Example output for $n = 5$ and $n = 10$ is given in Figure 4.71.

```
Value of n = 5
Number of threads in parallel region: 1
Print statement executed by thread 0
Value of n = 10
Number of threads in parallel region: 4
Print statement executed by thread 0
Print statement executed by thread 3
Print statement executed by thread 2
Print statement executed by thread 1
```

Figure 4.71: **Output from the program listed in Figure 4.70** – Four threads are used, but when $n = 5$, only one thread executes the parallel region. For $n = 10$ all four threads are active, because the condition under the if clause now evaluates to true.

4.8.2 Num_threads Clause

The `num_threads` clause is supported on the `parallel` construct only and can be used to specify how many threads should be in the team executing the parallel region (cf. Section 4.7). The syntax is `num_threads(scalar-integer-expression)`. Any expression that evaluates to an integer value can be used.

Figure 4.72 shows a simple example demonstrating the use of the `num_threads` and `if` clauses. To demonstrate the priority rules listed in Section 4.7, we insert a call to the OpenMP runtime function `omp_set_num_threads`, setting the number of threads to four. We will override it via the clauses.

```
(void) omp_set_num_threads(4);
#pragma omp parallel if (n > 5) num_threads(n) default(none)\
    private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
            omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```

Figure 4.72: **Example of the `num_threads` clause** – This clause is used on the parallel region to control the number of threads used.

This program has been executed for `n = 5` and `n = 10`. The output is shown in Figure 4.72. For `n = 5`, the `if` clause evaluates to false. As a result, the parallel region is executed by one thread only. If `n` is set to 10, however, the `if` clause is true and consequently the number of threads is set to 10 by the `num_threads(n)` clause. In neither of these two cases were four threads used, because of the higher priority of the `if` and `num_threads` clauses on the `#pragma omp parallel` construct.

4.8.3 Ordered Clause

The `ordered` clause is rather special: it does not take any arguments and is supported on the loop construct only. It has to be given if the `ordered` construct (see

```
Value of n = 5
Number of threads in parallel region: 1
Print statement executed by thread 0
Value of n = 10
Number of threads in parallel region: 10
Print statement executed by thread 0
Print statement executed by thread 4
Print statement executed by thread 3
Print statement executed by thread 5
Print statement executed by thread 6
Print statement executed by thread 7
Print statement executed by thread 8
Print statement executed by thread 9
Print statement executed by thread 2
Print statement executed by thread 1
```

Figure 4.73: **Output of the program given in Figure 4.72** – For $n = 5$ the if clause evaluates to false and only one thread executes the parallel region. If $n = 10$, however, the if clause is true, and then the `num_threads` clause causes 10 threads to be used.

Section 4.6.2 on page 86) is used in a parallel region, since its purpose is to inform the compiler of the presence of this construct.

An example of the usage of this clause and the associated construct is shown in the code fragment in Figure 4.74. Note that the `schedule(runtime)` clause is used (see also Section 4.5.7) to control the workload distribution at run time. The ordered *clause* informs the compiler of the ordered *construct* in the `#pragma omp parallel for` loop, which is used here on a print statement to ensure that the elements `a[i]` will be printed in the order $i = 0, 1, 2, \dots, n-1$. The updates of the elements `a[i]` of array `a` can and might be processed in any order.

In Figure 4.75 the output obtained using four threads and $n = 9$ is shown. Environment variable `OMP_SCHEDULE` is set to `guided` to contrast the dynamic workload distribution for the `#pragma omp for` loop with the ordered section within the loop. One clearly sees that the second `printf` statement (the one within the ordered construct) is printed in sequential order, in contrast to the first `printf` statement.

We note that the `ordered` clause and construct come with a performance penalty (see also Section 5.4.2). The OpenMP implementation needs to perform additional book-keeping tasks to keep track of the order in which threads should execute the

```

#pragma omp parallel for default(none) ordered schedule(runtime) \
    private(i,TID) shared(n,a,b)
for (i=0; i<n; i++)
{
    TID = omp_get_thread_num();

    printf("Thread %d updates a[%d]\n",TID,i);

    a[i] += i;

    #pragma omp ordered
    {printf("Thread %d prints value of a[%d] = %d\n",TID,i,a[i]);}

} /*-- End of parallel for --*/

```

Figure 4.74: **Example of the ordered clause** – Regardless of which thread executes which loop iteration, the output from the second `printf` statement is always printed in sequential order.

```

Thread 0 updates a[3]
Thread 2 updates a[0]
Thread 2 prints value of a[0] = 0
Thread 3 updates a[2]
Thread 2 updates a[4]
Thread 1 updates a[1]
Thread 1 prints value of a[1] = 2
Thread 3 prints value of a[2] = 4
Thread 0 prints value of a[3] = 6
Thread 2 prints value of a[4] = 8
Thread 2 updates a[8]
Thread 0 updates a[7]
Thread 3 updates a[6]
Thread 1 updates a[5]
Thread 1 prints value of a[5] = 10
Thread 3 prints value of a[6] = 12
Thread 0 prints value of a[7] = 14
Thread 2 prints value of a[8] = 16

```

Figure 4.75: **Output from the program listed in Figure 4.74** – Note that the lines with “prints value of” come out in the original sequential loop order.

corresponding region. Moreover, if threads finish out of order, there may be an additional performance penalty because some threads might have to wait.

4.8.4 Reduction Clause

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

Figure 4.76: **Summation of vector elements** – This operation can be parallelized with the `reduction` clause.

In Section 4.6.3 on page 87, we used a critical construct to parallelize the summation operation shown in Figure 4.76. There is a much easier way to implement this, however. OpenMP provides the `reduction` clause for specifying some forms of recurrence calculations (involving mathematically associative and commutative operators) so that they can be performed in parallel without code modification. The programmer must identify the operations and the variables that will hold the result values: the rest of the work can then be left to the compiler. The results will be shared and it is not necessary to specify the corresponding variables explicitly as “shared.” In general, we recommend using this clause rather than implementing a reduction operation manually. The syntax of the reduction clause in C/C++ is given by `reduction(operator:list)`. In Fortran, certain intrinsic functions are also supported. The syntax is as follows:

```
reduction({operator | intrinsic_procedure_name}:list).
```

The type of the result variable must be valid for the reduction operator (or intrinsic in Fortran).

We now show how easily the example given in Section 4.6.3 on page 87 can be implemented using the `reduction` clause. In Figure 4.77, this clause has been used to specify that `sum` will hold the result of a reduction, identified via the `+` operator. Based on this, an OpenMP compiler will generate code that is roughly equivalent to our example in Section 4.6.3, but it may be able to do so more efficiently. For example, the final summation could be computed through a binary tree, which scales better than a naive summation. Output of this program from a run using three threads is given in Figure 4.78.

Reductions are common in scientific and engineering programs, where they may be used to test for convergence or to compute statistical data, among other things. Figure 4.79 shows an excerpt from a molecular dynamics simulation. The code

```
#pragma omp parallel for default(none) shared(n,a) \
    reduction(+:sum)
    for (i=0; i<n; i++)
        sum += a[i];
/*-- End of parallel reduction --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Figure 4.77: **Example of the reduction clause** – This clause gets the OpenMP compiler to generate code that performs the summation in parallel. This is generally to be preferred over a manual implementation.

Value of sum after parallel region: 300

Figure 4.78: **Output of the example shown in Figure 4.77** – Three threads are used. The other values and settings are also the same as for the example output given in Figure 4.56 on page 90.

collects the forces acting on each of the particles as a result of the proximity of other particles and their motion and uses it to modify their position and velocity. The fragment we show includes two reduction operations to gather the potential and kinetic energy.

We note that, depending on the operator or intrinsic used, the initial value of the shared reduction variable (like `sum` in our example) may be *updated*, not overwritten. In the example above, if the initial value of `sum` is, for example, 10 prior to the reduction operation, the final value is given by $sum = 10 + \sum_{i=0}^{n-1} a[i]$. In other words, for this operator the original value is updated with the new contribution, not overwritten.

The order in which thread-specific values are combined is *unspecified*. Therefore, where floating-point data are concerned, there may be numerical differences between the results of a sequential and parallel run, or even of two parallel runs using the same number of threads. This is a result of the limitation in precision with which computers represent floating-point numbers: results may vary slightly, depending on the order in which operations are performed. It is not a cause for concern if the values are all of roughly the same magnitude. The OpenMP standard is explicit about this point: “There is no guarantee that bit-identical results will be obtained or that side effects (such as floating-point exceptions) will be identical” (see Section 2.8.3.6 of the 2.5 standard). It is good to keep this in mind when using the reduction clause.


```

! The force computation for each particle is performed in parallel
!$omp parallel do
!$omp& default(shared)
!$omp& private(i,j,k,rij,d)
!$omp& reduction(+ : pot, kin)
  do i=1,nparticles
    ! compute potential energy and forces
    f(1:nd,i) = 0.0
    do j=1,nparticles
      if (i .ne. j) then
        call distance(nd,box,pos(1,i),pos(1,j),rij,d)
        ! result is saved in variable d
        pot = pot + 0.5*v(d)
        do k=1,nd
          f(k,i) = f(k,i) - rij(k)*dv(d)/d
        enddo
      endif
    enddo
    ! compute kinetic energy
    kin = kin + dotprod(nd,vel(1,i),vel(1,i))
  enddo
!$omp end parallel do
  kin = kin*0.5*mass

  return
end

```

Figure 4.79: **Piece of a molecular dynamics simulation** – Each thread computes displacement and velocity information for a subset of the particles. As it is doing so, it contributes to the summation of potential and kinetic energy.

The operators supported (plus the intrinsic functions available in Fortran for this clause) are given in the first column of Figures 4.80 (C/C++) and 4.81 (Fortran). Each operator has a specific initial value associated with it, listed in the second column. This is the initial value of each *local* copy of the reduction variable.

OpenMP defines which type of statements are applicable to the **reduction** clause. In Figure 4.82 all the reduction statements supported in C/C++ are listed. The statements and intrinsic functions available in Fortran are given in Figure 4.83. In Fortran, the *array reduction* is also supported, which permits the reduction

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

Figure 4.80: **Operators and initial values supported on the reduction clause in C/C++** – The initialization value is the value of the local copy of the reduction variable. This value is operator, data type, and language dependent.

Operator	Initialization value
+	0
*	1
-	0
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.
.neqv.	.false.
Intrinsic	Initialization value
max	Smallest negative machine representable number in the reduction variable type
min	Largest negative machine representable number in the reduction variable type
iand	All bits on
ior	0
ieor	0

Figure 4.81: **Operators, intrinsic functions, and initial values supported on the reduction clause in Fortran** – The initialization value is the value of the local copy of the reduction variable. This value is operator, data type, and language dependent.

“variable” to be an entire array; see the example in Figure 4.84, where array **a** is updated in parallel in a manner that is similar to the scalar case. Each thread computes a *partial* update by calculating $a(1:n) = a(1:n) + b(1:n,j)*c(j)$ for

specific values of j , storing the result in a private array. This partial solution is then added to the global solution, the (shared) result array a . The details of the implementation depend on the specific OpenMP compiler.

$x = x \text{ } op \text{ } expr$
$x \text{ } binop = expr$
$x = expr \text{ } op \text{ } x$ (except for subtraction)
$x++$
$++x$
$x--$
$--x$

Figure 4.82: **Typical reduction statements in C/C++** – Here, $expr$ has scalar type and does not reference x , op is not an overloaded operator, but one of $+$, $*$, $-$, $\&$, $\&\&$, or $||$, and $binop$ is not an overloaded operator, but one of $+$, $*$, $-$, $\&$, \wedge , or $|$.

$x = x \text{ } op \text{ } expr$
$x = expr \text{ } op \text{ } x$ (except for subtraction)

$x = \textit{intrinsic}(x, \textit{expr_list})$
$x = \textit{intrinsic}(\textit{expr_list}, x)$

Figure 4.83: **Typical reduction and intrinsic statements in Fortran** – Here, op is one of the operators from the list $+$, $*$, $-$, $\&$, $\&\&$, $||$, \wedge , or $|$. The expression does not involve x , the reduction op is the last operation performed on the right-hand side, and $expr_list$ is a comma-separated list of expressions not involving x . The $\textit{intrinsic}$ function is one from the list given in Figure 4.81.

```
!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(j) SHARED(n,b,c) &
!$OMP      REDUCTION(+:a)
  do j = 1, n
    a(1:n) = a(1:n) + b(1:n,j)*c(j)
  end do
!$OMP END PARALLEL DO
```

Figure 4.84: **Example of an array reduction** – This type of reduction operation is supported in Fortran only.

We note that there are some further restrictions on both the variables and the operators that may be used. In C/C++ the following restrictions apply:

- Aggregate types (including arrays), pointer types, and reference types are not supported.
- A reduction variable must not be `const`-qualified.
- The operator specified on the clause can not be overloaded with respect to the variables that appear in the clause.

In Fortran there are some restrictions as well:

- A variable that appears in the clause must be definable.
- A list item must be a named variable of intrinsic type.
- Fortran pointers, Cray pointers, assumed-size array and allocatable arrays are not supported.

4.8.5 Copyin Clause

The `copyin` clause provides a means to copy the value of the master thread's threadprivate variable(s) to the corresponding threadprivate variables of the other threads. As explained in Section 4.9.3, these are global variables that are made private to each thread: each thread has its own set of these variables. Just as with regular private data, the initial values are undefined. The `copyin` clause can be used to change this situation. The copy is carried out after the team of threads is formed and prior to the start of execution of the `parallel` region, so that it enables a straightforward initialization of this kind of data object.

The clause is supported on the `parallel` directive and the combined parallel work-sharing directives. The syntax is `copyin(list)`. Several restrictions apply. We refer to the standard for the details.

4.8.6 Copyprivate Clause

The `copyprivate` clause is supported on the `single` directive only. It provides a mechanism for broadcasting the value of a private variable from one thread to the other threads in the team. The typical use for this clause is to have one thread read or initialize private data that is subsequently used by the other threads as well.

After the single construct has ended, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads. Since the barrier is essential in this case, the standard prohibits use of this clause in combination with the `nowait` clause.

The syntax is of this clause is: `copyprivate (list)`. In Fortran this clause is added to the `end` part of the construct. With C/C++, `copyprivate` is a regular clause, specified on the `single` construct.

4.9 Advanced OpenMP Constructs

We have covered the most common features of the API. These are sufficient to parallelize the majority of applications. Here, we complete our overview of the OpenMP API by covering a few remaining, specialized constructs. These are considered special-purpose because the need to use them strongly depends on the application. For example, certain recursive algorithms can take advantage of nested parallelism in a natural way, but many applications do not need this feature.

4.9.1 Nested Parallelism

If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team and becomes the master of that new team. This is generally referred to in OpenMP as “nested parallelism.”

In contrast to the other features of the API, an implementation is free to not provide nested parallelism. In this case, parallel constructs that are nested within other parallel constructs will be ignored and the corresponding parallel region serialized (executed by a single thread only): it is thus inactive. Increasingly, OpenMP implementations support this feature. We note that frequent starting and stopping of parallel regions may introduce a non-trivial performance penalty.

Some care is needed when using nested parallelism. For example, if the function `omp_get_thread_num()` is called from within a nested parallel region, it still returns a number in the range of zero up to one less than the number of threads of the *current* team. In other words, the thread number may no longer be unique. This situation is demonstrated in the code fragment in Figure 4.85, where `omp_get_nested()` is used to test whether nested parallelism is available. The `num_threads` clause is used (see Section 4.8.2 on page 102) to specify that the second level parallel region should be executed by two threads.

The output obtained when using three threads to execute the first, “outer,” parallel region is given in Figure 4.86. We have used indentation to see that a message comes from the inner parallel region, but it is no longer possible to distinguish messages from the individual threads that execute this region.

The code fragment from Figure 4.87 shows one way to address the problem for this specific case. Here, variable `TID` is used to store the number of the thread at

```

printf("Nested parallelism is %s\n",
      omp_get_nested() ? "supported" : "not supported");
#pragma omp parallel
{
    printf("Thread %d executes the outer parallel region\n",
          omp_get_thread_num());

    #pragma omp parallel num_threads(2)
    {
        printf("  Thread %d executes inner parallel region\n",
              omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/

```

Figure 4.85: **Example of nested parallelism** – Two parallel regions are nested. The second parallel region is executed by two threads.

```

Nested parallelism is supported
Thread 0 executes the outer parallel region
  Thread 0 executes the inner parallel region
  Thread 1 executes the inner parallel region
Thread 2 executes the outer parallel region
  Thread 0 executes the inner parallel region
Thread 1 executes the outer parallel region
  Thread 0 executes the inner parallel region
  Thread 1 executes the inner parallel region
  Thread 1 executes the inner parallel region

```

Figure 4.86: **Output from the source listed in Figure 4.85** – Three threads are used. The values returned by the OpenMP function `omp_get_thread_num()` do not reflect the nesting level. It is not possible to use the thread number to uniquely identify a thread.

the outer level. This variable is then passed on to the inner level parallel region by means of the `firstprivate` clause. Thus, each thread has a local copy of variable `TID` that is *initialized* with the value it had prior to entering the inner parallel region. This is exactly what is needed.

The output from the code fragment in Figure 4.87 is shown in Figure 4.88. As before, three threads execute the outer parallel region. One can now determine which inner level thread has executed the `printf` statement within the inner parallel

```

printf("Nested parallelism is %s\n",
      omp_get_nested() ? "supported" : "not supported");
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();

    printf("Thread %d executes the outer parallel region\n",TID);

    #pragma omp parallel num_threads(2) firstprivate(TID)
    {
        printf("TID %d: Thread %d executes inner parallel region\n",
              TID,omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/

```

Figure 4.87: **Modified version of nested parallelism example** – A thread at the first parallel level stores the thread number and passes it on to the second level.

```

Nested parallelism is supported
Thread 0 executes the outer parallel region
TID 0: Thread 0 executes inner parallel region
Thread 1 executes the outer parallel region
TID 1: Thread 0 executes inner parallel region
Thread 2 executes the outer parallel region
TID 2: Thread 0 executes inner parallel region
TID 2: Thread 1 executes inner parallel region
TID 0: Thread 1 executes inner parallel region
TID 1: Thread 1 executes inner parallel region

```

Figure 4.88: **Output from the source listed in Figure 4.87** – At least one can now distinguish at what nesting level the message is printed.

region.

Where nested parallelism is concerned, it is not always obvious what region a specific construct relates to. For the details of *binding* rules we refer to the standard.

4.9.2 Flush Directive

We have seen that the OpenMP memory model distinguishes between shared data, which is accessible and visible to all threads, and private data, which is local to an individual thread. We also explained in Chapter 2 that, where the sharing of values is concerned, things are more complex than they appear to be on the surface. This is because, on most modern computers, processors have their own “local,” very high speed memory, the registers and cache (see Fig. 1.1). If a thread updates shared data, the new values will first be saved in a register and then stored back to the local cache. The updates are thus not necessarily immediately visible to other threads, since threads executing on other processors do not have access to either of these memories. On a cache-coherent machine, the modification to cache is broadcast to other processors to make them aware of changes, but the details of how and when this is performed depends on the platform.

OpenMP protects its users from needing to know how a given computer handles this data *consistency* problem. The OpenMP standard specifies that all modifications are written back to main memory and are thus available to all threads, at *synchronization points* in the program. Between these synchronization points, threads are permitted to have new values for shared variables stored in their local memory rather than in the global shared memory. As a result, each thread executing an OpenMP code potentially has its own *temporary view* of the values of shared data. This approach, called a *relaxed consistency model*, makes it easier for the system to offer good program performance.

But sometimes this is not enough. Sometimes updated values of shared values must become visible to other threads in-between synchronization points. The OpenMP API provides the `flush` directive to make this possible.

The purpose of the `flush` directive is to make a thread’s temporary view of shared data consistent with the values in memory. The syntax of the directive in C/C++ is given in Figure 4.89.

#pragma omp flush [(list)]

Figure 4.89: **Syntax of the flush directive in C/C++** – This enforces shared data to be consistent. Its usage is not always straightforward.

The syntax in Fortran is shown in Figure 4.90.

The `flush` operation applies to all variables specified in the list. If no list is provided, it applies to all thread-visible shared data. If the `flush` operation is invoked by a thread that has updated the variables, their new values will be flushed

!\$omp flush [(list)]

Figure 4.90: **Syntax of the flush directive in Fortran** – This enforces shared data to be consistent. Its usage is not always straightforward.

to memory and therefore be accessible to all other threads. If the construct is invoked by a thread that has not updated a value, it will ensure that any local copies of the data are replaced by the latest value from main memory. Some care is required with its use. First, this does *not* synchronize the actions of different threads: rather, it forces the executing thread to make its shared data values consistent with shared memory. Second, since the compiler reorders operations to enhance program performance, one cannot assume that the flush operation will remain exactly in the position, relative to other operations, in which it was placed by the programmer. What can be guaranteed is that it will not change its position relative to any operations involving the flushed variables. Implicit flush operations with no list occur at the following locations.

- All explicit and implicit barriers (e.g., at the end of a parallel region or work-sharing construct)
- Entry to and exit from **critical** regions
- Entry to and exit from lock routines

This design is to help avoid errors that would probably otherwise occur frequently. For example, if flush was not implied on entry to and exit from locks, the code in Figure 4.91 would not ensure that the updated value of **count** is available to threads other than the one that has performed the operation. Indeed, versions of the OpenMP standard prior to 2.5 required an explicit **flush(count)** before and after the update. The implied flushing of all shared variables was introduced to ensure correctness of the code; it may, however carry performance penalties.

The following example demonstrates how to employ **flush** to set up pipelined thread execution. We consider the NAS Parallel Benchmark LU from the NPB version 3.2.1. This is a simulated computational fluid dynamics application that uses a symmetric successive overrelaxation method to solve a seven-band block-diagonal system resulting from finite-difference discretization of the 3D compressible Navier-Stokes equations. The OpenMP parallelization of the code is described in [90] and is summarized below. All of the loops involved carry dependences that prevent straightforward parallelization. A code snippet is shown in Figure 4.92.

```

!$omp parallel shared (lck, count)
...
call omp_set_lock (lck)
count = count + 1
call omp_unset_lock (lck)
...
!$omp end parallel

```

Figure 4.91: **A lock implies a flush** – The example uses locks to increment a shared counter. Since a call to a lock routine implies flushing of all shared variables, it ensures that all threads have a consistent view of the value of variable `count`.

```

do k=2,nz
  do j = 2, ny
    do i = 2, nx
      v(i,j,k) = v(i,j,k) + v(i-1,j,k)
                  + v(i,j-1,k) + v(i,j,k-1)
      ...
    end do
  end do
end do

```

Figure 4.92: **Code snippet from a time consuming loop in the LU code from the NAS Parallel Benchmarks** – Dependences in all three dimensions prevent straightforward parallelization of the loop.

A certain amount of parallelism can be exploited by setting up pipelined thread execution. The basic idea is to enclose the outer loop in a parallel region but share the work on the next inner level. For our example, an `!$omp parallel do` directive is placed on the `k`-loop, and the `!$omp do` work-sharing directive is placed on the `j`-loop. On entry to the parallel region, the team of threads is created and starts executing. Since the work in the `k`-loop is not parceled out, all threads will execute all iterations of it.

For a given iteration of the `k`-loop, each thread will work on its chunk of iterations of the `j`-loop. But because of data dependences, the threads cannot all work on the same iteration of the `k`-loop at the same time. Each thread needs data that will be updated by another thread, with the exception of the thread that receives the first chunk of iterations. If we explicitly schedule this code so that thread 0 receives the first chunk, thread 1 the second, and so forth, then thread 0 can start to work

```

!$OMP PARALLEL PRIVATE(k, iam)

    iam = OMP_GET_THREAD_NUM()
    isync(iam) = 0 ! Initialize synchronization array

! Wait for neighbor thread to finish

!$OMP BARRIER
    do k = 2, nz
        if (iam .gt. 0) then
            do while(isync(iam-1) .eq. 0)
!$OMP FLUSH(isync)
            end do
            isync(iam-1) = 0
!$OMP FLUSH(isync,v)
            end if

!$OMP DO SCHEDULE(STATIC, nchunk)
            do j = 2, ny; do i = 2, nx
                v(i,j,k) = v(i,j,k) + v(i-1,j,k) + ....
            end do; end do
!$OMP END DO NOWAIT

! Signal the availability of data to neighbor thread

            if (iam .lt. nt) then
!$OMP FLUSH(isync,v)
                do while (isync(iam) .eq. 1)
!$OMP FLUSH(isync)
                end do
                isync (iam) = 1
!$OMP FLUSH(isync)
            end if
        end do

!$OMP END PARALLEL

```

Figure 4.93: **One-dimensional pipelined thread execution in the NAS Parallel Benchmark LU** – The `flush` directive is used several times here. Note that `ny` is assumed to be a multiple of the number of threads `nt`.

immediately on its first chunk of data in the *j* direction. Once thread 0 finishes, thread 1 can start on its chunk of the *j*-loop for iteration *k*=2 and, in the meantime, thread 0 moves on to work on iteration *k*=3. Eventually, all threads will be working on their chunk of data in the *j* dimension, but on different iterations of the *k*-loop. Implementing this kind of pipelined thread execution is a more challenging problem for the programmer because it requires synchronization of individual threads, rather than global barrier synchronization. A thread has to wait for the availability of the data it needs before it can start on a new chunk, and it must signal the availability of updated data to the thread that is waiting for that data. The `flush` directive can be used for this purpose, as shown in Figure 4.93.

The code invokes OpenMP runtime library routines `omp_get_thread_num`, to obtain the current thread identifier, and `omp_get_num_threads` for the total number of threads. The shared array `isync` is used to indicate the availability of data from neighboring threads. Static scheduling has to be specified for this technique. In addition, loop lengths are assumed to be a multiple of the number of threads, thereby eliminating unpredictable behavior introduced by compiler-specific treatment of end cases. Thread 0 can start processing right away. All other threads have to wait until the values they need are available. To accomplish this, we place the `flush` directive inside two subsequent `while`-loops. The first `flush` ensures that the array `isync` is read from memory, rather than using a value stored locally in a register or cache. The second `flush` ensures that the updated value of `isync` is visible to other threads and that array `v` is read from memory after the `while`-loop has exited.

After processing its chunk of the *j*-loop, a thread needs to signal the availability of the data to its successor thread. To this end we use two `flush` directives, one of which is placed in a `while`-loop. The first `flush` ensures that the updated values of array `v` are made visible to the successor thread before the synchronization takes place. The second `flush` ensures that the synchronization array `isync` is made visible after it has been updated.

4.9.3 Threadprivate Directive

We have seen clauses for declaring data in parallel and work-sharing regions to be shared or private. However, we have not discussed how to deal with global data (e.g., static in C and common blocks in Fortran). By default, global data is shared, which is often appropriate. But in some situations we may need, or would prefer to have, private data that persists throughout the computation. This is where

the `threadprivate` directive comes in handy.¹⁰ The effect of the `threadprivate` directive is that the named global-lifetime objects are replicated, so that each thread has its own copy. Put simply, each thread gets a private or “local” copy of the specified global variables (and common blocks in case of Fortran). There is also a convenient mechanism for initializing this data if required. See the description of the `copyin` clause in Section 4.8.5 for details.

The syntax of the `threadprivate` directive in C/C++ is shown in Figure 4.94. The Fortran syntax is given in Figure 4.95.

#pragma omp threadprivate (*list*)

Figure 4.94: **Syntax of the `threadprivate` directive in C/C++** – The *list* consists of a comma separated list of file-scope, namespace-scope, or static block scope variables that have incomplete types. The `copyin` clause can be used to initialize the data in the `threadprivate` copies of the list item(s).

!\$omp threadprivate (*list*)

Figure 4.95: **Syntax of the `threadprivate` directive in Fortran** – The *list* consists of a comma-separated list of named variables and named common blocks. Common block names must appear between slashes, for example, `!$omp threadprivate (/mycommonblock/)`. The `copyin` clause can be used to initialize the data in the `threadprivate` copies of the list item(s).

Among the various types of variables that may be specified in the `threadprivate` directive are pointer variables in C/C++ and Fortran and allocatables in Fortran. By default, the `threadprivate` copies are not allocated or defined. The programmer must take care of this task in the parallel region. The example in Figure 4.96 demonstrates this point.

In order to exploit this directive, a program must adhere to a number of rules and restrictions. For it to make sense for global data to persist, and thus for data created within one parallel region to be available in the *next* parallel region, the regions need to be executed by the “same” threads. In the context of OpenMP, this means that the parallel regions must be executed by the same number of threads. Then, each of the threads will continue to work on one of the sets of data previously produced. If all of the conditions below hold, and if a `threadprivate` object is referenced in two consecutive (at run time) parallel regions, then threads with the same thread

¹⁰Technically this is a directive, not a construct.

number in their respective regions reference the same copy of that variable.¹¹ We refer to the OpenMP standard (Section 2.8.2) for more details on this directive.

- Neither parallel region is nested inside another parallel region.
- The number of threads used to execute both parallel regions is the same.
- The value of the *dyn-var* internal control variable is false at entry to the first parallel region and remains false until entry to the second parallel region.
- The value of the *nthreads-var* internal control variable is the same at entry to both parallel regions and has not been modified between these points.

```

1 int *pglobal;
2
3 int main()
4 {
5     .....
6
7     for (i=0; i<n; i++)
8     {
9         if ((pglobal=(int *) malloc(length[i]*sizeof(int))) != NULL) {
10
11             for (j=sum=0; j<length[i]; j++) pglobal[j] = j+1;
12             sum = calculate_sum(length[i]);
13             printf("Value of sum for i = %d is %d\n",i,sum);
14             free(pglobal);
15
16         } else {
17             printf("Fatal error in malloc - length[%d] = %d\n",
18                 i,length[i]);
19         }
20     }
21
22     .....

```

Figure 4.96: **Program fragment** – This program uses a global pointer `pglobal` to allocate, initialize, and release memory.

¹¹Section 4.7 on page 95 defines and explains these control variables.

The need for and usage of the `threadprivate` directive is illustrated by a somewhat elaborate example in Figure 4.96, where the fragment of a sequential program is listed. At line 1 a global pointer `pglobal` to an `int` is defined. The main `for`-loop spans lines 7 through 20. In this loop, storage is allocated at line 9. Pointer `pglobal` is used to point to this block of memory, which is initialized in the `for`-loop (line 11). At line 12 function `calculate_sum` is called; it sums up the elements of `pglobal`. At line 13 the checksum called `sum` is printed. At line 14 the memory block is released again through the call to the `free` function. If the memory allocation should fail, the code block under the `else` branch is executed. Function `calculate_sum` is given in Figure 4.97. It simply adds up all the elements pointed to by `global`, using parameter `length` as an argument to the function.

```
1  extern int *pglobal;
2
3  int calculate_sum(int length)
4  {
5      int sum = 0;
6
7      for (int j=0; j<length; j++)
8          sum += pglobal[j];
9
10     return(sum);
11 }
```

Figure 4.97: **Source of function `calculate_sum`** – This function sums up the elements of a vector pointed to by the global pointer `pglobal`.

The main loop over `i` at line 7 in Figure 4.96 can be easily parallelized with a `#pragma omp parallel for` combined work-sharing construct. However, this requires care when using pointer `pglobal` in particular. By default, `pglobal` is a shared variable. This creates various (related) problems. In the parallelized loop over `i`, multiple threads update `pglobal` simultaneously, creating a data race condition. If we do not find a way to overcome this, we already have a fatal error of course. But on top of that, the size of the memory block depends on `i` and is therefore thread-dependent. As a result, some threads access memory outside of the area that has been allocated, another fatal error. The third problem encountered is that one thread may release memory (through the call to `free`) while another thread or multiple threads still need to access this portion of memory. This results in undetermined runtime behavior, leading to a wrong answer (because of the data

race condition) or a segmentation violation caused by the out-of-bounds access or premature release of memory. Luckily, the `threadprivate` directive helps out. The OpenMP version of the code fragment using this is shown in Figure 4.98.

```

1 int *pglobal;
2
3 #pragma omp threadprivate(pglobal)
4
5 int main()
6 {
7     .....
8
9 #pragma omp parallel for shared(n,length,check) \
10     private(TID,i,j,sum)
11 for (i=0; i<n; i++)
12 {
13     TID = omp_get_thread_num();
14
15     if ((pglobal=(int *) malloc(length[i]*sizeof(int))) != NULL) {
16
17         for (j=sum=0; j<length[i]; j++) pglobal[j] = j+1;
18         sum = calculate_sum(length[i]);
19         printf("TID %d: value of sum for i = %d is %d\n",
20             TID,i,sum);
21         free(pglobal);
22
23     } else {
24         printf("TID %d: fatal error in malloc - length[%d] = %d\n",
25             TID,i,length[i]);
26     }
27 } /*-- End of parallel for --*/
28
29     .....

```

Figure 4.98: **OpenMP version of the program fragment** – The `threadprivate` directive is used to give each thread a private copy of the global pointer `pglobal`. This is needed for the parallel loop to be correct.

The source code changes needed to parallelize this loop are minimal. A `#pragma omp threadprivate` directive is used at line 3 to give each thread a local copy of our pointer `pglobal`. At line 9 the `#pragma omp parallel for` directive is inserted

to parallelize the main loop over `i`. For diagnostic purposes the thread number is stored in variable `TID` at line 13. This identifier is used in the print statements. The output is given in Figure 4.99.

```
TID 0: value of sum for i = 0 is 55
TID 0: value of sum for i = 1 is 210
TID 2: value of sum for i = 4 is 1275
TID 1: value of sum for i = 2 is 465
TID 1: value of sum for i = 3 is 820
```

Figure 4.99: **Output of the program listed in Figure 4.98** – Variable `n` is set to 5, and three threads are used.

Sometimes a private variable, such as a pointer with the `private` data-sharing attribute, can be used to achieve the same result. In the example here, this is not possible without modifying the source of function `calculate_sum` and all the places in the source program where it is called. Since this requires more work, and most likely additional testing, the `threadprivate` directive is more convenient to use.

4.10 Wrap-Up

In the early sections of this chapter, we introduced some terminology and then presented and discussed a basic set of OpenMP constructs, directives, and clauses. This set is more than sufficient to parallelize many different applications. We next introduced synchronization constructs and explained how to influence and exchange information with the execution environment. We then showed some slightly less common clauses and, finally, some advanced features.

OpenMP is straightforward to use. The programmer's time is often spent thinking about where the parallelism is in the application. Once this has been identified, implementing it using the features provided by OpenMP is often straightforward. Challenges may arise if an algorithm implies tricky synchronization or if additional performance is needed.

What helps when parallelizing an application is to have a clean sequential version to start with. In particular, the control and data flow through the program should be straightforward. Use of global data that is modified should be minimized to reduce the chance of introducing a data race condition. Something else that helps when parallelizing loops is to avoid a bulky loop body, which makes the specification of data-sharing attributes tedious and error prone. If the loop body performs a substantial amount of work, one should push it into a function. All variables local

to the function are private by default, often dramatically reducing the data-sharing list. This is not only more pleasing to the eye but also easier to maintain. The use of block scoping in C can also help in this respect.