2 New Features in OpenMP

In May 2008, the specifications of OpenMP 3.0 were released. This was the first update since the 2.5 release and a major improvement. Many existing features were enhanced, support for C++ was substantially improved, and the concept of tasks was introduced.

Since then, OpenMP has continued to evolve. OpenMP 3.1 was released in July of 2011. This was an update release. Several constructs and features were enhanced, including tasking. The specifications were clarified where needed, but no significant new functionality was added.

New functionality was implemented in OpenMP 4.0. When it was released in July of 2013, OpenMP made a huge step forward. For the first time in the history of OpenMP, support for cc-NUMA architectures became available. Heterogeneous computing support was added also. This integrated very smoothly with the rest of the language and preserved the ease of use of OpenMP. Another noticeable addition was support for handling a failure in a parallel region, or task. In addition, tasking was expanded, vectorization through SIMD instructions was added and User-Defined Reductions (UDRs) were now supported as well.

A little over two years later, in November 2015, OpenMP 4.5 was released. It is another significant step forward with many enhancements to the existing functionality.

In this chapter, most of the features and enhancements, added since the release of OpenMP 2.5, are presented and discussed. The exceptions are tasking, thread affinity, SIMD, and heterogeneous architectures. These are major topics, covered in great detail in the chapters to follow.

2.1 Enhancements to Existing Constructs

In this section, the enhancements to the constructs supported in OpenMP 2.5 and earlier releases are introduced and discussed. Since this release, entirely new concepts and constructs have been introduced as well. An overview of these can be found in Section 2.4.

2.1.1 The Schedule Clause

The `schedule` clause has been extended to support additional functionality. The updated syntax is shown in Figure 2.1.

schedule *([modifier [, modifier]:] kind [, chunk_size])*

Figure 2.1: **Syntax of the extended schedule clause in C/C++ and Fortran** – In addition to the already supported types, the `auto` type for the workload distribution schedule has been added. Another new feature is the support for an optional modifier, or even two, to provide additional details in which way the chunks of work, if applicable, are distributed over the threads.

Modifier	Description
`monotonic`	Each thread executes its chunks in increasing logical iteration order. This is the order the iterations would be executed in if the loop was executed sequentially.
`nonmonotonic`	Chunks are assigned to threads in any order.
`simd`	The new chunk size is set to *chunk_size/simd_width*, with *simd_width* an implementation-defined value.

Figure 2.2: **The three modifiers supported on the schedule clause** – These modifiers specify how the chunks should be distributed over the threads.

As of OpenMP 3.0, in addition to the `static`, `dynamic`, and `guided` options for the workload distribution schedule for parallel loops, a fourth choice is supported. Through the `auto` keyword, the selection of the schedule is determined by the implementation.

In OpenMP 4.5, an optional keyword on the `schedule` clause was introduced to support one, or even two, *modifiers* to specify in which way the chunks should be distributed over the threads. This feature was introduced to resolve an ambiguity in the earlier specifications that could cause applications to make an incorrect assumption of how the work was distributed over the threads. With the introduction of the modifiers, this problem is no longer an issue.

The three new keywords to select the modifier, plus a brief description, are listed in Figure 2.2. If more details are needed, we recommend checking the specifications.

There are several restrictions when using these modifiers:

- The `nonmonotonic` modifier may be used only in combination with the `guided` or `dynamic` scheduling types.
- If the `static` schedule type, or the `ordered` clause, is specified without a modifier, the `monotonic` modifier is assumed. In other words, in these cases, `monotonic` is the default.

Important - With the next release of the OpenMP specifications, this default will change to `nonmonotonic`. Any application relying on the monotonic behavior in these cases should set this explicitly.

- The `monotonic` and `nonmonotonic` modifiers are mutually exclusive. This implies that, when applicable, either one of these may be used in combination with the `simd` modifier only.
- The `nonmonotonic` modifier may not be specified in combination with the `ordered` clause.

2.1.2 The If Clause

With combined constructs, it is not always clear to which part(s) the `if` clause applies. This is why the clause has been extended with an optional name, formally called the *directive-name-modifier*, to specify to which construct it applies.

For example, when using tasking in a combined construct, the following makes it clear the clause applies to the tasking part of the construct: `if(task: n < 10)`.

2.1.3 The Collapse Clause

A short loop has a downside in a serial program, because the "loop overhead" dominates the performance if there is not much work performed per loop iteration. In a parallel program, the overhead is even worse.

A short loop not only limits the number of threads that may be used, it also easily leads to a load imbalance if the loop trip count is not a multiple of the number of threads. Depending on the amount of work performed per iteration, the parallel overhead may dominate as well.

Consider the serial code fragment in Figure 2.3, where a linearized 2D mxn array `a` is initialized. Both loops can be parallelized, but unless nested parallelism is used, only one loop may be selected for parallelization. The inner loop is longest, but for performance reasons it is not a good idea to select it for parallelization, because the parallel loop overhead is incurred "m" times. If the outer loop is parallelized, only two threads may be used.

There does not seem to be an easy way out of this dilemma. The only solution is to merge or "collapse" the two loops into one single and longer loop. This new loop has the length of both loops combined and may be parallelized. This transformation comes at the price of some additional computations to reconstruct the original loop

```
1 int m = 2;
2 int n = 5;
3      ........
4 for (int i=0; i<m; i++)
5   for (int j=0; j<n; j++)
6      a[i*n+j] = i+j+1;
```

Figure 2.3: **Example of a nested loop with short loops** – Both loops can be parallelized, but neither of the choices results in efficient code or substantial speed ups.

```
1 for (int k=0; k<m*n; k++)
2 {
3    int i = (k/n) % m;
4    int j = k % n;
5    a[i*n+j] = i+j+1;
6 }
```

Figure 2.4: **An example of explicit loop collapsing** – At the cost of some additional book-keeping operations, the two short loops are combined into a single longer loop that can be parallelized and executed more efficiently.

iteration values from the single loop variable. The code fragment that implements this idea is shown in Figure 2.4.

This optimization is not only beneficial in a serial program. Both issues when parallelizing the original loop nest have been eliminated as well. In general, and as demonstrated in this example, loop collapsing provides a good way to deal with short nested parallel loops, but it does not make the code any easier to read and maintain.

This is why the `collapse` clause is useful. It is ideally suited in the event a perfectly nested loop has more than one parallelizable loop, but all of these loops are short. The syntax of the clause is given in Figure 2.5.

Figure 2.6 shows how to use the clause in this example. The compiler first generates code similar to what is shown in Figure 2.4. The resulting single loop is then parallelized.

To demonstrate how this works, we parallelized the outer loop in Figure 2.3 and executed both versions using five threads. The loops were instrumented to print diagnostic information. The results are listed in Figure 2.7.

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

Figure 2.5: **Syntax of the loop collapse clause in C/C++ and Fortran** – This clause combines multiple loops into a single loop. The length of the resulting parallel loop is the combined length of the loops affected. The parameter `n` is used to specify how many loops should be collapsed.

```
1 #pragma omp parallel for default(none) shared(a,m,n)\
2                          collapse(2)
3 for (int i=0; i<m; i++)
4   for (int j=0; j<n; j++)
5      a[i*n+j] = i+j+1;
```

Figure 2.6: **Example of the `collapse` clause** – The compiler generates a single loop of length mxn and parallelizes it.

The output lines marked "Outer loop" are from the original loop nest, where the outer loop was parallelized. In this case only two threads may be used, although we requested five threads. This limitation has been lifted for the version that uses the `collapse` clause. The output lines are marked "Collapse" and clearly all five threads are used now.

There are some things to remember when using the `collapse` clause:

- This clause is supported on the `worksharing` loop construct, the `simd` construct, and the `distribute` construct.
- The collapsed loops must be perfectly nested.
- The collapsed loops must form a rectangular iteration space.
- The bounds and stride of each loop must be invariant over all the loops.
- Only one `collapse` clause may be used per loop nest.
- The iterations of the collapsed loop are scheduled according to the `schedule` clause in effect.

```
Outer loop: (0,0) has been initialized to 1 by thread 0
Outer loop: (0,1) has been initialized to 2 by thread 0
Outer loop: (0,2) has been initialized to 3 by thread 0
Outer loop: (0,3) has been initialized to 4 by thread 0
Outer loop: (0,4) has been initialized to 5 by thread 0
Outer loop: (1,0) has been initialized to 2 by thread 1
Outer loop: (1,1) has been initialized to 3 by thread 1
Outer loop: (1,2) has been initialized to 4 by thread 1
Outer loop: (1,3) has been initialized to 5 by thread 1
Outer loop: (1,4) has been initialized to 6 by thread 1

Collapse:   (0,2) has been initialized to 3 by thread 1
Collapse:   (0,3) has been initialized to 4 by thread 1
Collapse:   (0,0) has been initialized to 1 by thread 0
Collapse:   (0,1) has been initialized to 2 by thread 0
Collapse:   (1,3) has been initialized to 5 by thread 4
Collapse:   (1,4) has been initialized to 6 by thread 4
Collapse:   (0,4) has been initialized to 5 by thread 2
Collapse:   (1,0) has been initialized to 2 by thread 2
Collapse:   (1,1) has been initialized to 3 by thread 3
Collapse:   (1,2) has been initialized to 4 by thread 3
```

Figure 2.7: **Output of the original code and the version with the collapse clause** – This example was executed using five threads, but in the first version only two threads may be used. The second version uses all five threads.

2.1.4 The Linear Clause

The `linear` clause is mentioned here only for the sake of completeness. It is part of the support for SIMD instructions and covered in detail in Section 5.2.4 on page 230.

2.1.5 The Critical Construct

The `critical` construct has been enhanced with an optional hint. The syntax is given in Figure 2.8.

If a hint is used, the construct *must* have a name and the expression for the hint must evaluate to the same value for all critical regions with the same name.

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

Figure 2.8: **Syntax of the extended critical construct in C/C++ and Fortran** – The hint is optional, but if used, the critical construct must have a name and the expression for the hint must evaluate to the same value for all critical regions with the same name.

The purpose and use of the hint is the same as described in Section 2.3.3 on page 67 and allows an implementation to tailor the code to the expected access pattern of the critical region.

The expression for the hint must be of type `omp_lock_hit_kind` and must evaluate to a scalar value that is a valid lock hint. The supported values are listed in Figure 2.20 on page 68. Without a hint, the effect is as if `hint(omp_lock_hint_none)` was specified.

Because locks and critical regions potentially limit scalability, choosing the correct value for the hint can make a noticeable difference and is certainly worth considering.

2.1.6 The Atomic Construct

In earlier versions of the specifications, the `atomic` construct applied to a single update statement only (see also Section 1.6.3 on page 32), but as of OpenMP 3.1 it has been refined to support several additional access and update patterns. It is now also possible to capture the value of the target variable before, or after an atomic update. The syntax of the extended construct is given in Figure 2.9.

Through the optional *atomic-clause*, the type of modification performed is specified. In absence of a clause, the `update` clause is implied. This is consistent with the earlier definition.

The names of the clauses allude to their respective functionality:

- **Read** - This is an atomic read. The input variable is read atomically and stored into the output variable. This is guaranteed regardless of the size of the variable.

#pragma omp atomic *[*`seq_cst`*[,]] atomic-clause [[,]*`seq_cst`*] new-line* *expr. statement* **#pragma omp atomic** *[*`seq_cst`*] new-line* *expr. statement* **#pragma omp atomic** *[*`seq_cst`*[,]]* `capture` *[[,]*`seq_cst`*] new-line* *structured block*
!$omp atomic *[*`seq_cst`*[,]]* `[read\|write\|update]` *[[,]*`seq_cst`*]* *capture, write, or update statement respectively* **!$omp end atomic** **!$omp atomic** *[*`seq_cst`*]* *update statement* **!$omp atomic** *[*`seq_cst`*[,]]* `capture` *[[,]*`seq_cst`*]* *update+capture, capture+update or capture+write statement* **!$omp end atomic**

Figure 2.9: **Syntax of the extended atomic construct in C/C++ and Fortran** – The type of atomic operation performed can be specified. If necessary, sequential consistency may be enforced by using the `seq_cst` keyword.

- **Write** - This is an atomic write. The output variable is written atomically. This is guaranteed regardless of the size of the variable.
- **Update** - The same variable is both read and written. As before, only the read/write operations are guaranteed to be atomic. If, for example, an expression is used, the evaluation of this need not be atomic. The consequence is that any possible side effect(s) of such an evaluation could lead to an erroneous result. An example is a call to a function that updates a shared variable. If no precautions are taken, a data race may occur. Task scheduling pointss are not allowed between the read and write of the variable. See also page 144 in Section 3.7 for more information on task scheduling points.
- **Capture** - In addition to an update, the value of the modified variable is saved *atomically* either before or after the update.

  There is a subtle, but important, distinction between the two forms of the `capture` clause in C/C++. The first form supports a single statement only, while the second form is somewhat more general and not only supports the atomic update, but also a (simple) statement to capture the value of the variable modified. The details for this are given in Figure 2.11.

Clause	Expr. STMT (C/C++)	Expr. STMT (Fortran)
`read`	`v = x;`	`v = x`
`write`	`x = expr;`	`x = expr`

`update`	`x++;`
	`++x;`
	`x--;`
	`--x;`
	`x binop = expr;`
	`x = x binop expr;`
	`x = expr binop x;`
		`x = x operator expr`
		`x = expr operator x`
		`x = intrinsic_func(x,expr_list)`
		`x = intrinsic_func(expr_list,x)`

Figure 2.10: **Clauses and expression statements supported on the atomic construct** – This table lists the various combinations of clauses and statements one can use with the `read`, `write`, and `update` clauses. The `capture` clause is covered separately.

In Figure 2.10, the expression statements in C/C++ and Fortran supported with the `read`, `write` and `update` clauses are given. In this Figure, *binop* is one of the following operators: `+`,`-`,`*`,`/`, `&`,`^`, `|`, `<<`, or `>>`.

In C++, `binop`, `binop =`, `++`, and `--` cannot be overloaded operators. In Fortran, the operator is `+`, `-`, `*`, `/`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`. The intrinsic function is one of `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`.

The fine-print in the specifications on the combined use of an operator and an expression is not always easy to digest. With the following simple observation in mind, it is however hopefully easier to use this construct correctly.

All one needs to remember is that the loads and stores are executed atomically and whatever is contributed to the target variable should have no side effects regarding this variable. Whether it is a function call or an expression, evaluating this should not have an impact on the value of the variable to be updated.

Either the standard precedence rules apply, or parentheses may be used to enforce that the expression is evaluated prior to the update. For example, a statement of the form `x = x binop expr` must be equivalent to `x = x binop (expr)`.

The following example illustrates this: `x = x + y*y`. Because the multiplication is guaranteed to be performed prior to the addition, there is no parentheses. If this is not the case, or when in doubt, parentheses may be used to eliminate any ambiguity. For example, in an update of this type, the parentheses enforce the precedence: `x = x + (z + y*y)`.

Clause	Expr. STMT (C/C++)	Structured Block (C/C++)
capture	v = x++;	{v = x; x *binop* = *expr*;}
	v = ++x;	{x *binop* = *expr*; v = x;}
	v = x--;	{v = x; x = x *binop* *expr*;}
	v = --x;	{v = x; x = *expr* *binop* x;}
	v = x *binop* = *expr*;	{x = x *binop* *expr*; v = x;}
	v = x = x *binop* *expr*;	{x = *expr* *binop* x; v = x;}
	v = x = *expr* *binop* x;	{v = x; x = *expr*;}
		{v = x; x++;}
		{v = x; ++x;}
		{++x; v = x;}
		{x++; v = x;}
		{v = x; x--;}
		{v = x; --x;}
		{--x; v = x;}
		{x--; v = x;}

Figure 2.11: **Expression statements and structured blocks supported on the capture clause in C/C++** – With this construct, the value of the target variable is saved into another variable, either before or after the update.

While the other three clauses are refinements on the `atomic` construct prior to the OpenMP 3.1 specifications, the `capture` clause *extends* the functionality. With this clause, the value of the target variable may be saved into another variable before, or after, the update.

Due to syntactical differences between C/C++ and Fortran, the overview for the `capture` clause is split. In Figure 2.11, the supported statements and structured blocks for C/C++ are given. In Figures 2.12 and 2.13, the Fortran case is covered.

Why was there a need for the `capture` clause? Prior to the introduction of this feature there was no way to *atomically* intercept the value of the target variable. This could easily lead to a data race. If one thread accesses this variable outside the atomic construct, while another thread performs the atomic update, the result is undefined.

The example in Figure 2.14 shows such a case and demonstrates how to use the `capture` clause to avoid this problem. In this program, function `update` adds a certain contribution to a shared variable. We want to preserve the previous value and use it as the return value of this function.

Clause	Update STMT (Fortran)

`capture`	`x = x` *`operator expr`*
	`x =` *`expr operator`* `x`
	`x =` *`intrinsic_function`*`(x,`*`expr_list`*`)`
	`x =` *`intrinsic_function`*`(`*`expr_list`*`,x)`

Figure 2.12: **The update statements supported on the capture clause in Fortran** – The intrinsic function is one of the following: `min`, `max`, `iand`, `ior`, or `ieor`.

Clause	Capture STMT (Fortran)	Write STMT (Fortran)
`capture`	v = x	x = *expr*

Figure 2.13: **The capture and write statement supported on the capture clause in Fortran** – These are the various possibilities to choose from when using this clause.

The main program calls this function from within a parallel region and sets the contribution to the thread ID plus 1. The addition of 1 makes the output in Figure 2.15 easier to follow. The final result should be $1+2+\ldots+NT+(NT+1) = NT*(NT+1)/2$ if `NT` threads are used.

The output in Figure 2.15 demonstrates how the value is carried over from one thread to another.

Note there is one issue here. Printing variable `new_value` within the parallel region introduces a data race, because the value as modified by one thread could have been modified by another thread before the print statement is executed. There is no such risk with the old value, because it is stored in a private variable and performs an atomic read of the new value.

There is one feature that has not been discussed so far: the `seq_cst` clause. This has mainly been introduced to provide the same semantics as the `memory_order_seq_cst` and `memory_order_relaxed` atomic operations in the C11 and C++11 standards [4, 5].

In OpenMP, adding the `seq_cst` clause to the `atomic` construct implies a `flush` operation without a list. This guarantees sequential consistency but comes at a price. As discussed in section 1.2.3 on page 11, this consistency model limits potentially significant compiler optimizations and most likely negatively impacts performance.

```
 1  int main(int argc, char *argv[])
 2  {
 3    int new_value = 0;
 4
 5    #pragma omp parallel shared(new_value)
 6    {
 7       int TID       = omp_get_thread_num();
 8       int old_value = update(&new_value, TID+1);
 9
10       printf("TID = %4d: old_value = %4d\n",TID,old_value);
11
12    } // End of parallel region
13
14    printf("Final value is %4d\n",new_value);
15
16    return(0);
17  }
18  int update (int *new_value, int contribution)
19  {
20     int return_value;
21
22     #pragma omp atomic capture
23     {
24        return_value = *new_value;
25        *new_value  += contribution;
26     } // End of atomic capture
27
28     return(return_value);
29  }
```

Figure 2.14: **Example of the capture clause on the atomic construct** – Prior to OpenMP 3.1 it was not possible to reliably save the old value before the update. The `capture` clause used at line 22 addresses this issue. Before variable `new_value` is updated at line 25, its value is stored into variable `return_value` (line 24). These are atomic operations, so no other thread may interfere. The final result of this program is `NT*(NT+1)/2` if `NT` threads are used.

```
TID =    0: old_value =    5
TID =    1: old_value =    0
TID =    2: old_value =    2
TID =    3: old_value =   11
TID =    4: old_value =    6
TID =    5: old_value =   15
TID =    6: old_value =   29
TID =    7: old_value =   21
Final value is   36
```

Figure 2.15: **Sorted output of the update function that uses the atomic capture** – This result was obtained using 8 threads. Thread 1 happens to be the first one to perform the update. It adds 2 to it, and this new value is then picked up by thread 2, which adds 3 to it. The value is then subsequently updated by threads $0, 4, 3, 5, 7$ and 6. The final result is indeed $8 * 9/2 = 36$.

Hence the recommendation to use this clause only if really needed and no suitable alternative is available.

Many aspects of the `atomic` construct have been covered here, but there are some things glossed over. We recommend checking the specifications for all details.

2.2 New Environment Variables

In OpenMP 2.5 there were only four environment variables. Since then, the list given in Figure 1.28 on page 34 has been significantly expanded. These new variables standardize the typical extensions that most compiler vendors have provided already and also support the new functionality introduced since OpenMP 2.5.

In addition to this, the already existing variables `OMP_NUM_THREADS` and `OMP_SCHEDULE` are extended to support additional functionality.

These two, plus all additional environment variables supported in OpenMP 4.5, are listed in Figure 2.16.

Below, these new environment variables are covered in more detail. As before, the names must be in uppercase, but the value(s) assigned are case-insensitive. In our examples, we use lowercase for the values.

Please keep in mind that, in all cases, the default is system-dependent. Also, as with any environment variable, the way to set these depends on the Operating System used and/or the specific (Unix) shell.

Function name	**Description**
OMP_DISPLAY_ENV	If set, displays the OpenMP version number and ICV values. In verbose mode, more information is shown.
OMP_NUM_THREADS	Set the number of threads for (nested) parallel regions.
OMP_SCHEDULE	Set the workload distribution schedule for parallel loops.
OMP_STACKSIZE	Set the size of the stack for the threads.
OMP_WAIT_POLICY	Specify the behavior of idle threads.
OMP_MAX_ACTIVE_LEVELS	Set the maximum number of nested active parallel regions.
OMP_THREAD_LIMIT	Set the maximum number of threads to be created in a contention group.
OMP_MAX_TASK_PRIORITY	Set the maximum value that can be used in the `priority` clause on the `task` construct.
OMP_PROC_BIND	Set and control thread affinity.
OMP_PLACES	Define where threads are allowed to execute.
OMP_CANCELLATION	Enable or disable cancellation.
OMP_DEFAULT_DEVICE	Set the default device number.

Figure 2.16: **The additional OpenMP 4.5 environment variables** – These are the environment variables changed or added since OpenMP 2.5.

- `OMP_DISPLAY_ENV` - This is a very useful variable to verify the settings and check the defaults. Each environment variable and macro is printed in the format `ENV_VAR_NAME = 'value'`. Optionally a line starts with `[device-id]`.

  The information is printed after the environment variables have been processed, but before any of the corresponding ICVs are modified. The information is enclosed between the strings `OPENMP DISPLAY ENVIRONMENT BEGIN` and `OPENMP DISPLAY ENVIRONMENT END`.

  There are three choices for this variable:

  - `false` - No information is displayed.
  - `true` - Print the OpenMP version number and the values of all environment variables. These are either the defaults, or the value(s) set by the user prior to program start up.

  - `verbose` - In addition to the standard environment variables, vendor-specific extensions may be printed as well. It is up to the implementor to select which environment variables to include.

Below is an example after setting `OMP_DISPLAY_ENV` to `true`. This is printed immediately after program start up:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP='201307'
  OMP_CANCELLATION='FALSE'
  OMP_DISPLAY_ENV='true'
  OMP_DYNAMIC='TRUE'
  OMP_MAX_ACTIVE_LEVELS='4'
  OMP_NESTED='FALSE'
  OMP_NUM_THREADS='16'
  OMP_PLACES='N/A'
  OMP_PROC_BIND='FALSE'
  OMP_SCHEDULE='static'
  OMP_STACKSIZE='8388608B'
  OMP_THREAD_LIMIT='1024'
  OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

One of the things to observe in this output is that the settings for `OMP_PLACES` are "N/A," which stands for "Not Applicable." This is because processor binding is apparently disabled (`OMP_PROC_BIND` has been set to `false`). This illustrates how useful this environment variable is to perform a sanity check on the settings and to ensure the values are as expected.

- `OMP_NUM_THREADS` - This is probably the most often used OpenMP environment variable, but before OpenMP 3.1 it was not working well in conjunction with nested parallelism.

  This has since then been addressed, and it now supports a set of values, which are set by a list of positive integer values. These values are used to define the number of threads at the corresponding nesting level of parallel regions, starting at level 1.

For example, setting `OMP_NUM_THREADS` to "`3,2`" implies that a top-level parallel region uses 3 threads. The team size of the next nested parallel region is 2.

If the nesting level is deeper than the number of entries in the list, the last value in the list is used to set the number of threads in subsequent nested parallel regions. In the above example, this means that 2 threads are used in every parallel region at nesting level 3 or more.

- `OMP_SCHEDULE` - As before, this variable is used to set the type of workload scheduling for parallel loops in the event the `runtime` option is used on the `schedule` clause.

  In addition to `static`, `dynamic`, and `guided`, as of OpenMP 3.0, the `auto` schedule type is supported as well. With this, the choice of the schedule is determined by the implementation. This new scheduling type does not support a chunk size.

- `OMP_STACKSIZE` - Each OpenMP thread needs a certain amount of storage space to store information, such as private variables. This part of memory is referred to as "the stack."

  An OpenMP runtime system sets aside a certain default amount of storage space for each thread. This is controlled by the `OMP_STACKSIZE` environment variable. The setting can be checked in the documentation or through environment variable `OMP_DISPLAY_ENV`.

  Unfortunately an application is likely to crash if enough (stack) space is not available. In some cases, this problem can be determined at compile time and the compiler may issue a warning, but more often than not, this is not possible and the problem only shows up at runtime.

  That is when this variable needs to be used to increase the thread stack size.

  The syntax supports a case-insensitive unit qualifier that is appended (possibly with whitespace in between) to the number: `B` for bytes, `K` for 1024 bytes, `M` for 1024 Kbytes and `G` for 1024 Mbytes.

  For example, `export OMP_STACKSIZE=2M` or `export OMP_STACKSIZE="2 m"` both set the thread stack size to 2 Mbyte using Bash shell syntax.

  There are two more things worth noting:

  - In the absence of a unit qualifier (`B`, `K`, `M` or `G`), the default unit is *Kbytes*, not bytes. This is easily overlooked, resulting in very high memory requirements that may cause the application to fail.
  - The application may require its own *main* stack size to be set if the OS does not provide sufficient stack space by default. This has no relation to OpenMP and is *not* affected by the `OMP_STACKSIZE` variable. Typically this stack is set through an OS-specific command (e.g. `limit` or `ulimit` on Unix systems) and is most likely specified in bytes, but one is advised to check the documentation when in doubt.

- `OMP_WAIT_POLICY` - In the optimal scenario, all threads are busy all the time doing useful work, but in practice there are often shorter or longer periods of inactivity for one or more threads.

  The question is what to do with such threads. Putting them to sleep saves processor cycles and other resources, but waking up such a thread is relatively costly. Simply keeping the processor busy doing nothing wastes cycles.

  Until the `OMP_WAIT_POLICY` variable was introduced in OpenMP 3.0, there was no portable solution to specify the desired behavior.

  There are currently two settings for `OMP_WAIT_POLICY` to specify the behavior of such waiting threads, also called *idle threads*:

  - `active` - Idle threads should mostly be kept active. This is good for the performance of the application, because an idle thread is ready to execute as soon as it is needed again.

    The word *mostly* is important here. It is not good for the system throughput to keep idle threads in a busy loop for an extended period of time. It only wastes cycles.

    This is why, by default, an implementation may decide to put an idle thread to sleep after a certain time out. Obviously this choice, and the length of the time out, is implementation-, or even system-dependent.

    By setting the idle mode to `active`, this is overruled and threads are ready to go any time they are needed.
  - `passive` - Idle threads should mostly do nothing and not use any processor cycles. This favors overall system throughput, but activating an idle thread may take longer compared to the active setting.

Remember that the setting is merely a suggestion. The implementation is free to ignore it.

If more explicit control is needed, we recommend checking the documentation of the specific runtime environment used. Several OpenMP implementations support more control than provided by this environment variable.

- `OMP_MAX_ACTIVE_LEVELS` - The `OMP_MAX_ACTIVE_LEVELS` variable is set to a non-negative integer that defines the upper limit on the number of active parallel regions that may be nested.

  If the value is negative, not an integer at all, or exceeds the limit supported by the implementation, the behavior is undefined.

- `OMP_THREAD_LIMIT` - The `OMP_THREAD_LIMIT` variable is set to a non-negative integer that defines the maximum number of threads to be used in a *contention group*, where a contention group consists of an initial thread and its descendant threads.

  Its main use is to prevent an application from generating too many threads. Especially with recursive types of algorithms, this can easily happen. As a result, the system may become unresponsive. This limit can be used to avoid such a situation.

  If the value is negative, not an integer at all, or exceeds the limit supported by the implementation, the behavior is undefined.

- `OMP_MAX_TASK_PRIORITY` - The `OMP_MAX_TASK_PRIORITY` variable is set to a non-negative integer that defines the maximum priority level to be used in the `priority` clause on the `task` construct.

  If this variable is not set explicitly, the default value is zero, and all priority settings in the application are effectively ignored.

- `OMP_PLACES` - The `OMP_PLACES` variable is extensively covered in Chapter 4, starting on page 151. This is why there is only a very brief discussion here.

  The `OMP_PLACES` variable defines the OpenMP place list to be used by the runtime environment.

  A single place is defined through an *unordered* set of comma separated non-negative numbers enclosed in curly braces ({ and }). For example {`8,0,4`} defines a place with three entries, and the order is irrelevant.

Places are the building blocks of the *place list*, which consists of an *ordered* list with one or more comma-separated places.

In other words, the order within a place definition does not matter, but the order in which the places are specified in the place list does matter.

The integer numbers are system-specific and may need to be adapted in case a different system is used. This is not very convenient and that is why an alternative to such lists is supported.

Three *abstract names* to cover several common scenarios are available. The names are `sockets`, `cores`, and `threads`. They cannot be combined with a list specification.

As with most OpenMP environment variables, if `OMP_PLACES` has not been set, a *system-dependent default place list* is used. It is important to note that this means there is *always* a place list.

- `OMP_PROC_BIND` - The `OMP_PROC_BIND` variable sets the thread affinity policy, or policies, to be used for parallel regions.

  The settings control if and how OpenMP threads are bound to computational resources, such as sockets, cores and/or hardware threads.

  This is achieved through a tight coupling between binding and *OpenMP places*, a new concept introduced in OpenMP 4.0 and briefly covered under the description of the `OMP_PLACES` environment variable.

  The `OMP_PROC_BIND` variable was already introduced in OpenMP 3.1, but could be set to only `true` or `false`. With the introduction of thread affinity support in OpenMP 4.0, the functionality has been greatly enhanced and provides very fine-grained control over thread placement.

  There are three policies regarding how to place the threads: `master`, `close`, and `spread`.

  In the case of nested parallel regions, multiple and potentially different placement types may be defined by using a comma-separated list. For example `OMP_PROC_BIND="spread,close"`.

  If the nesting level exceeds the number of policies in the list, the last policy is applied to the remaining levels.

  If binding is requested, but no place list is defined, a default place list is used.

Note that if `OMP_PROC_BIND` is not set, the default may be `false`. In this case, or if it is explicitly set to `false`, the settings for the place list are *ignored*.

Much more detailed coverage of binding and places can be found in Chapter 4, which is dedicated to thread affinity, binding, and the places concept.

- `OMP_CANCELLATION` - If `OMP_CANCELLATION` variable is set to `true`, cancellation is activated. The effects of the `cancel` construct and of cancellation points are enabled. If set to `false`, cancellation is disabled.

  If this variable is not set explicitly, the default value is `false`.

- `OMP_DEFAULT_DEVICE` - The `OMP_DEFAULT_DEVICE` variable may be used to define the default device number to be used in device constructs. The value must be a non-negative integer.

2.3 New Runtime Functions

Since OpenMP 2.5, quite a number of runtime functions have been added to the functions listed in Figure 1.29 on page 36. Existing functionality has been expanded and newly introduced features have been augmented by a set of functions specific to the feature.

As before, these functions may be used to query settings and also change them, but in the case of heterogeneous systems, they go a step further. The device memory functions may be used to manage memory on the target device(s).

A new element in the specifications is that some functions require an OpenMP-specific data type. They are defined in include file `omp.h` in C/C++. Fortran programs require either file `omp_lib.h` to be included or a module called `omp_lib` to be used.

In Fortran, these new data types are specified using the `KIND` type selector.

For example, function "`omp_proc_bind_t omp_get_proc_bind()`" in C/C++ is "`integer (kind=omp_proc_bind_kind) function omp_get_proc_bind()`" in Fortran.

In the remainder of this section, functions added since OpenMP 2.5 are listed and discussed. To avoid lengthy tables, the list has been split into three parts, each covered in its own section. The syntax given is for C/C++ only, but the usage in Fortran is very similar. The exception are the device memory functions. There are

Function name	Description
`omp_get_thread_limit`	Return the maximum number of threads in the contention group.
`omp_get_schedule`	Return the loop schedule (and chunk size) in case the `runtime` clause is used.
`omp_set_schedule`	Change the loop schedule (and chunk size) in case the `runtime` clause is used.
`omp_get_max_active_levels`	Return the maximum number of nested active parallel regions.
`omp_set_max_active_levels`	Change the maximum number of nested active parallel regions.
`omp_get_level`	Return the number of nested parallel regions enclosing the current task.
`omp_get_active_level`	Return the number of nested *active* parallel regions enclosing the current task.
`omp_get_ancestor_thread_num`	Return the thread number of the ancestor of the current thread.
`omp_get_team_size`	Return the size of the thread team to which the current thread or ancestor belongs.

Figure 2.17: **Runtime functions for thread management, scheduling, and nested parallelism** – These functions provide support for thread management, workload scheduling and nested parallelism. The names are the same in C/C++ and Fortran, but the usage follows the language syntax.

no native interfaces for Fortran. One may be able to call the C/C++ versions from Fortran, but there is no such guarantee.

2.3.1 Runtime Functions for Thread Management, Thread Scheduling, and Nested Parallelism

The first set of additional runtime functions consists of functions related to thread management, loop iteration scheduling, and nested parallelism. They are listed in Figure 2.17. These functions augment the constructs already supported in OpenMP 2.5. The next two sections cover the functions related to the new features introduced since then.

- `int omp_get_thread_limit()` - This function returns the maximum number of OpenMP threads that the application is allowed to create in the current contention group.

  While the number of hardware scheduling resources is fixed for a given system configuration, this upper limit on the number of threads is controlled by the OpenMP implementation. This function returns the limit. Environment variable `OMP_THREAD_LIMIT` may be used to change this limit.[1]

  The difference between this function and the older `omp_get_num_procs()` function is that the latter returns the number of "processors" available on the system. The notion of what a processor is exactly, can be somewhat ambiguous, so it is up to the OS to decide. It could be a core, or a hardware thread, for example.

- `void omp_get_schedule(omp_sched_t *kind, int *chunk_size)` - The first argument of this function returns a pointer of type `omp_sched_t` to identify the schedule used to assign loop iterations to threads. Where relevant, the second argument returns an additional qualifier, such as the chunk size.

  There are four pre-defined schedule types: `static`, `dynamic`, `guided`, and `auto`. The corresponding codes are 1, 2, 3, and 4, respectively. Note that the first value is 1, not 0. This may require care if used as an index into an array in C/C++.

  An implementation has the freedom to support additional scheduling types. If so, the definition of the second argument for that type is implementation-dependent and the reason we also refer to it as a "qualifier."

- `void omp_set_schedule(omp_sched_t kind, int chunk_size)` - Function to change the scheduling type and (optionally) the chunk size. The first argument is an integer identifier (or a specific reserved variable name) to specify the schedule to be used.

[1] There may also be an OS or installation-specific upper limit. Getting this changed depends on the specific environment and local situation.

The currently supported reserved variable names include the name of the schedule:

  - `omp_sched_static = 1`
  - `omp_sched_dynamic = 2`
  - `omp_sched_guided = 3`
  - `omp_sched_auto = 4`

If supported by the scheduling type, the second argument may be used to pass on the chunk size. Any value less than one implies the *default* chunk size is used.

This function may be used to adapt the loop schedule on a per-loop basis. In Figures 2.22 and 2.23 the usage of the functions to get and set the schedule is demonstrated.

Note that this function has an effect only in combination with the `runtime` clause.

- `int omp_get_max_active_levels()` - This function returns how deep parallel regions may be nested.
- `void omp_set_max_active_levels(int max_levels)` - Environment variable `OMP_MAX_ACTIVE_LEVELS` may be used to initially set the maximum level of parallel regions that are allowed to be nested. This function increases or decreases this limit at runtime to adapt to specific needs.
- `int omp_get_level()` - This function returns the number of nested parallel regions enclosing the current task, regardless of whether the region is active or not. Recall that *active* in an OpenMP context means that more than one thread executes the region.

  This function is used to determine at which level a (nested) parallel region is. The first-level region is assigned a level of 1, the one nested within this region has a level of 2, and so on.

  One of the uses of this functionality is to determine a unique thread ID at any level. Figure 2.26 on page 83 shows a source code example of how to do this.

- `int omp_get_active_level()` - This function provides the same functionality as the function `omp_get_level()` described above, but it is restricted to active parallel regions only.

  Both functions may be used to determine at which level a (nested) parallel region is. The first level region is assigned a level of 1, the one nested within this region has a level of 2, and so on.

- `int omp_get_ancestor_thread_num(int level)` - The thread ID of the ancestor thread is returned by this function. The function argument is the nesting level this applies to.

  There is one important thing to point out. This is the level of the *ancestor*, not the level of the thread calling this function. For example, if at level 3, the function argument should be 2, not 3, to get the thread ID of the ancestor.

  If this function is called with the nesting level of the caller, the regular thread ID is returned. This is the same value as returned by function `omp_get_thread_num()`.

- `int omp_get_team_size(int level)` - This function returns the number of threads in the team executing the parallel region at the nesting level specified by the argument.

  If this function is called with the nesting level of the caller, the number of threads in the caller's current team is returned. This is the same value as returned by function `omp_get_num_threads()`.

2.3.2 Runtime Functions for Tasking, Cancellation, and Thread Affinity

This section presents and discusses the functions related to tasking, thread cancellation, and thread affinity. These functions return a value, or setting, only. There are no functions to change a setting.

The functions are listed in Figure 2.18. Note that for the affinity-related functions, we do not use the word "processor" as the specifications do. Instead, we use "(hardware) resource number" or simply "resource" for short. This terminology is explained in detail in Section 4.5.1 in Chapter 4.

- `int omp_in_final()` - This function returns `true` if called from a task that is final. Otherwise it returns `false`.

Function name	Description
`omp_in_final`	Return `true` if executed in a final task region; otherwise return `false`
`omp_get_max_task_priority`	Return the maximum value that may be specified in the `priority` clause.
`omp_get_cancellation`	Return the status of cancellation.
`omp_get_proc_bind`	Return the thread affinity policy to be used for the *subsequent* parallel region that does not use a `proc_bind` clause.
`omp_get_num_places`	Return the number of places in the place list.
`omp_get_place_num_procs`	Return the number of resources associated with the argument, which must be a place number.
`omp_get_place_proc_ids`	Return the numerical identifiers of each resource associated with the first argument, which must be a place number.
`omp_get_place_num`	Return the place number to which the encountering thread is bound.
`omp_get_partition_num_places`	Return the number of places in the subpartition.
`omp_get_partition_place_nums`	Return the list of place numbers corresponding to the places in the subpartition.

Figure 2.18: **Runtime functions for tasking, cancellation, and thread affinity** – These functions provide support for tasking, cancellation, and thread affinity. The names are the same in C/C++ and Fortran, but the usage follows the language syntax.

- `int omp_get_max_task_priority()` - This function returns the maximum value that may be used on the `priority` clause supported on the `task` construct.
- `int omp_get_cancellation()` - This function returns `true` if cancellation is enabled. Otherwise it returns `false`.
- `omp_proc_bind_t omp_get_proc_bind()` - This function uses the setting for

environment variable `OMP_PROC_BIND` to return the policy for the *subsequent* parallel region.

The return value has data type `omp_proc_bind_t` and may take one of the following values:

- `omp_proc_bind_false = 0`
- `omp_proc_bind_true = 1`
- `omp_proc_bind_master = 2`
- `omp_proc_bind_close = 3`
- `omp_proc_bind_spread = 4`

This function does not return a meaningful result if the `proc_bind` clause is used on the subsequent parallel region, because this takes precedence over the policy set through `OMP_PROC_BIND`.

Figure 2.28 on page 85 shows an example of how to use this function.

More on the choices for these thread affinity settings may be found under the description of the `OMP_PROC_BIND` environment variable.

- `int omp_get_num_places()` - This function returns the number of places in the place list.
- `int omp_get_place_num_procs(int place_num)` - This function returns the number of hardware resources, for example, hardware threads or cores, in the place identified by `place_num`.

  A value of zero is returned if `place_num` is either less than zero or is equal to or larger than the value as returned by `omp_get_num_places()`.
- `void omp_get_place_proc_ids(int place_num, int *ids)` - This function returns a list in array `ids`. This list contains the (hardware) resource numbers associated with the place identified by `place_num`. This place identifier, as well as the resource numbers, are implementation-dependent numbers to identify a place and its contents. Refer to Section 4.5.1 on page 161 for the definition of a resource number.

  Be sure to allocate sufficient storage for array `ids`. At least `omp_get_place_num_procs(place_num)` locations must be available.

This function has no effect if `place_num` is less than zero, or is equal to, or larger than the value as returned by `omp_get_num_places()`.

- `int omp_get_place_num()` - If the encountering thread is bound to a place, the place number associated with this thread is returned and the value is in the interval $[0, \texttt{omp\_get\_num\_places()} - 1]$.

  If the thread is not bound to a place, a value of -1 is returned.

- `int omp_get_partition_num_places()` - This function returns the number of places in the subpartition of the encountering thread.

- `void omp_get_partition_place_nums(int *place_nums)` - This function returns the list of place numbers corresponding to the places in the subpartition of the encountering thread.

  Array `place_nums` must be sufficiently large to contain the number of integers returned by `omp_get_partition_num_places()`. Otherwise, the behavior is undefined.

2.3.3 Runtime Functions for Locking

Before diving deeper into this, an important warning first: the ownership of locks has changed. In most cases, this has no impact on the application, but we recommend reading Section 2.4.1 on page 86 for the details.

The set of locking functions has been extended to support a *hint* the user may provide. This hint is used to specify the expected behavior of the lock. This allows the implementation to select a locking mechanism tuned for this specific purpose.

The syntax of the new functions is given in Figure 2.19. The hint does not change the mutual exclusion semantics of the lock, and the implementation is free to ignore it. The application should also not rely on the hint in order to work correctly.

The choices for the hint are listed in Figure 2.20. As usual, such values are defined in header file `omp.h` for C/C++ and the Fortran `omp_lib.h` include file and/or module file `omp_lib`.

At first sight, the support for `omp_lock_hint_none` seems somewhat peculiar. This informs the implementation there is no specific preference.

```
void omp_init_lock_with_hint (omp_lock_t *t, omp_lock_hint_t hint);

void omp_init_nest_lock_with_hint (omp_lock_t *t, omp_lock_hint_t hint);
```

```
subroutine omp_init_lock_with_hint (svar, hint)
 integer (kind=omp_lock_kind) svar
 integer (kind=omp_lock_kind_hint) hint

subroutine omp_init_nest_lock_with_hint (svar, hint)
 integer (kind=omp_lock_kind) svar
 integer (kind=omp_lock_kind_hint) hint
```

Figure 2.19: **Syntax of the new locking functions in C/C++ and Fortran** – These functions support an optional hint. This gives the implementation the opportunity to select the most suitable solution, but it is also free to ignore the hint. Both single-level, as well as, nested locks are available.

```
typedef enum omp_lock_hint_t {
  omp_lock_hint_none = 0,
  omp_lock_hint_uncontended = 1,
  omp_lock_hint_contended = 2,
  omp_lock_hint_nonspeculative = 4,
  omp_lock_hint_speculative = 8
} omp_lock_hint_t;
```

```
integer (kind=omp_lock_hint_kind), &
  parameter::omp_lock_hint_none = 0
integer (kind=omp_lock_hint_kind), &
  parameter::omp_lock_hint_uncontended = 1
integer (kind=omp_lock_hint_kind), &
  parameter::omp_lock_hint_contended = 2
integer (kind=omp_lock_hint_kind), &
  parameter::omp_lock_hint_nonspeculative = 4
integer (kind=omp_lock_hint_kind), &
  parameter::omp_lock_hint_speculative = 8
```

Figure 2.20: **Definition of the hint data type in C/C++ and Fortran** – These are the various choices supported for the hint. An implementation may add to this list. For an explanation of these hints, refer to the main text.

Why would one use a function that supports a hint and then effectively say it doesn't matter? This choice is available to accommodate flexibility in an application. The same locking function may be called for various values of the hint, including the lack of a preference, leaving it up to the system to decide.

The next two choices are about *lock contention*. The lock is said to be "contended," if multiple threads want to access a lock at the same time. If this is the case, a more advanced mechanism to acquire the lock could be beneficial. For example, by using an exponential backoff algorithm in case a lock cannot be acquired right away.

The last two types are included for systems with support for *Transactional Memory*. When using a *speculative lock*, the assumption is that there is no contention and that the code within the locked region can be executed unconditionally. Only afterward is it verified that there was indeed no contention. If there had been contention, a rollback strategy is applied, if necessary, to guarantee the correct results.

This highly improves the efficiency of the locking mechanism if there is little or no contention, but it is not without a potential cost. The rollback part is relatively expensive and if this is executed too often, the performance could be worse than without the speculation.

If the user knows upfront that the lock is only lightly contended, using this hint may improve performance.

There are several restrictions when using hints:

- Hints may be combined through wither the `+` or `|` operators in C/C++ or the `+` operator in Fortran, but the behavior is implementation-defined and may even be ignored.
- Combining `omp_lock_hint_none` with any other hint is equivalent to specifying the other hint only.
- As one might expect, when combining conflicting settings, for example "contended" with "uncontended," the behavior is undefined.
- An implementation is free to support additional choices for the hint, but be aware that these may not be portable to other implementations.

2.3.4 Runtime Functions for Heterogeneous Systems

One of the main additions to OpenMP is the support for heterogeneous systems, typically consisting of a host system, plus one or more accelerators. In OpenMP these are referred to as "devices."

In this section, only the runtime functions are summarized. For extensive coverage of this topic, plus an introduction to the device-specific terminology, refer to Chapter 6 starting on page 253.

The set of runtime functions supporting devices are used to query the settings, as well to set the default device. OpenMP 4.5 adds functions to explicitly manage the memory between the host and the device(s).

The full list can be found in Figure 2.21. Below is a more extensive description of each function.

- `int omp_get_default_device()` - This function returns the value of the *default-device-var* ICV, which is the default device number. If it is called from within a `target` region, the result is undefined.

- `void omp_set_default_device(int device_num)` - This function assigns the value of the integer argument `device_num` to the *default-device-var* ICV. If it is called from within a `target` region, the behavior is undefined.

- `int omp_get_initial_device()` - This function returns the host device number. If the return value is in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$, then it may be used in a `device` clause and in all device memory functions. If the value is outside that range, then it may be only in device memory functions only. Calling this function from within a `target` region results in undefined behavior.

- `int omp_is_initial_device()` - This function returns `true` if the thread that calls it is executing on the host device. Otherwise, it returns `false`.

- `int omp_get_num_devices()` - This function returns the number of available devices. Calling it from within a `target` region results in undefined behavior. Whether or not the host device is included in the number of available devices is implementation-defined.

- `int omp_get_num_teams()` - This function returns the number of teams in the current teams region. See Section 6.5 starting at page 275 for more details

Function name	Description
`omp_get_default_device`	Return the default device number.
`omp_set_default_device`	Set the default device number.
`omp_get_initial_device`	Return the host device number.
`omp_is_initial_device`	Return `true` if the current task is executing on the host device; otherwise return `false`.
`omp_get_num_devices`	Return the number of devices.
`omp_get_num_teams`	Return the number of teams in the current `teams` region.
`omp_get_team_num`	Return the team number of the calling thread.
`omp_target_alloc`	Allocate a block of memory on a device.
`omp_target_free`	Free memory allocated on a device by the `omp_target_alloc` function.
`omp_target_is_present`	Check whether a host address is mapped to a device.
`omp_target_memcpy`	Copy memory between any combination of host and device pointers.
`omp_target_memcpy_rect`	The same as `omp_target_memcpy`, but for entire sub-volumes of multi-dimensional arrays.
`omp_target_associate_ptr`	Associate a device address with a host address.
`omp_target_disassociate_ptr`	Remove the association between a device address and a host address.

Figure 2.21: **Runtime functions for heterogeneous computing** – These functions provide support for heterogeneous computing, including memory management between the host and the device(s). The last seven functions, the "device memory functions," are not supported in Fortran.

on the `teams` construct. A value of 1 is returned if called from outside a teams region.

- `int omp_get_team_num()` - This function returns the calling thread's team number. The return value is an integer number in the interval $[0, \texttt{omp\_get\_num\_teams()} - 1]$. If it is called from outside a `teams` region, a value of zero is returned.

- `void* omp_target_alloc(size_t size, int device_num)` - This function returns the device address of a memory block of `size` bytes dynamically allocated in the address space of the device number specified by `device_num`. If the allocation fails, then it returns `NULL`. The device number should be either the value returned by `omp_get_initial_device()` or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$.

  Upon a successful allocation, the device address may be used in an `is_device_ptr` clause on a `target` construct or in other device memory functions. See Section 6.3.3 starting on page 265 for more details on using device addresses.

  There are several restrictions when using this function:

  - The behavior of the function is undefined if it is called from a `target` region.
  - Pointer arithmetic cannot be performed on a device address returned by this function.
  - Memory allocated by this function can be freed using the `omp_target_free()` function only.

- `void omp_target_free(void *device_ptr, int device_num)` - This function frees a memory block that was dynamically allocated in the address space of the device number specified by `device_num`. The value of the `device_ptr` should be a device address returned by an earlier call to the `omp_target_alloc()` function. The device number should be either the value returned by `omp_get_initial_device()` or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$. If `device_ptr` is `NULL`, the function has no effect. The behavior of the function is undefined if it is called from a `target` region.

- `int omp_target_is_present(void *ptr, int device_num)` - This function is used to test if a variable is present in a device data environment. The device number should be either the value returned by `omp_get_initial_device()` or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$.

  The function returns `true` if the host address stored in `ptr`, has a corresponding address in the device data environment of the device identified by `device_num`. Otherwise, `false` is returned. The effect of this function is undefined if called from within a `target` region.

- `int omp_target_memcpy(`

  ```
  void *dst,
  void *src,
  size_t length,
  size_t dst_offst,
  size_t src_offst,
  int dst_device_num,
  int src_device_num) -
  ```

  This function copies a contiguous memory block of `length` bytes from a source device to a destination device. The `dst_device_num` and `src_device_num` are device numbers that identify the source and destination devices. The device numbers should either be the value returned by `omp_get_initial_device()` or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$. The source device address is determined by adding the `src_offset` to the value of the `src` pointer variable. The destination device address is determined by adding the `dst_offset` to the value of `dst` pointer variable.

  The return value is zero upon successful completion of the copy and non-zero in the event of a failure. The `dst` and `src` pointers must be valid device pointers. The calculated source and destination addresses must be valid on the respective devices. The effect of this function is undefined if called from within a `target` region. This function contains a task scheduling point.

- `int omp_target_memcpy_rect(`

  ```
  void *dst,
  void *src,
  size_t element_size,
  int num_dims,
  const size_t *volume,
  const size_t *dst_offsets,
  const size_t *src_offsets,
  const size_t *dst_dimensions,
  const size_t *src_dimensions,
  int dst_device_num,
  int src_device_num) -
  ```

This function copies a rectangular sub-volume of memory from a multi-dimensional array in the address space of the source device to a multi-dimensional array in the address space of the destination device.

The `dst_device_num` and `src_device_num` are device numbers that identify the source and destination devices. The device numbers should either be the value returned by `omp_get_initial_device()` or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$.

The size of an element in the source and destination arrays is `element_size` bytes. The `volume`, `dst_offsets`, `src_offsets`, `src_dimensions`, and `dst_dimensions` are constant arrays of length `num_dims`.

The `volume` array specifies the length of each dimension for the sub-volume. The `src_dimensions` and `dst_dimensions` arrays specify the length of each dimension for the source and destination arrays, respectively. The sub-volume and the source and destination arrays are defined by their dimensions and the size of an element. The `src_offset` and `dst_offset` arrays specify a multi-dimension offset into the source and destination arrays, respectively.

The starting source device address is determined by the `src` device pointer variable and the `src_offset` array. The starting destination device address is determined by the `dst` device pointer variable and the `dst_offset` array.

The value of `num_dims` must be between 1 and an implementation-defined limit, which must be at least three. The `dst` and `src` pointers must be valid device pointers. The effect of this function is undefined if called from within a `target` region. This function returns zero if successful. Otherwise, it returns a non-zero value. If both `src` and `dst` are `NULL` pointers, the number of dimensions supported by the implementation for the specified device numbers is returned. This function contains a task scheduling point.

- `int omp_target_associate_ptr(`

  ```
  void *host_ptr,
  void *device_ptr,
  size_t size,
  size_t device_offset,
  int device_num) -
  ```

  This function associates a host memory block of `size` bytes with a corresponding memory block in the address space of a device. The device number,

specified in `device_num`, should either be the value returned by `omp_get_initial_device()`, or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$. The corresponding device address is determined by adding the `dst_offset` to the value of `dst` pointer variable.

Once associated, the host address range is mapped to the corresponding address range in the device data environment of the `device_num` device. The corresponding address range has an infinite reference count. See Section 6.3.1 starting on page 263 for more details on mapped variables. The value of `device_ptr` must be a valid device address. See Section 6.3.3 starting on page 265 for more details on device addresses.

This function returns zero if successful. Otherwise, a non-zero value is returned. Upon return from the function, the value of the memory in the corresponding address range is undefined. Once associated, the behavior when accessing the corresponding address range from the host device via an `is_device_ptr` clause or another device memory function is undefined until the association is removed by the `omp_target_disassociate()` function. A host address may correspond to only one device address on one device. Attempting to associate a host address to more than one device address results in a non-zero return value. The `omp_target_is_present()` function may be used to check if a host address is already associated with a device address. The effect of this function is undefined if called from within a `target` region.

- `int omp_target_disassociate_ptr(void ptr, int device_num)` - This function removes the association of a host memory block to a corresponding memory block in the address space of a device. The association would have been established by a previous call to the `omp_target_associate()` function. The device number should either be the value returned by `omp_get_initial_device()` or in the range $[0, \texttt{omp\_get\_num\_devices()} - 1]$.

  The host address is unmapped. The corresponding address range's reference count is set to zero and removed from the device data environment of the `device_num` device. See Section 6.3.1 starting on page 263 for more details on mapped variables. See Section 6.3.3 starting on page 265 for more details on device addresses.

The effect of this function is undefined if called from within a `target` region. If the host address in `ptr` does not have a corresponding device address associated with it by a previous call to `omp_target_associate_ptr()`, then the behavior of this function is undefined. Upon return from the function, the value of the memory in the corresponding address range is undefined.

2.3.5 Usage Examples of the New Runtime Functions

Now that all new runtime functions are covered, it is time to show some examples in what way these functions may be used.

The first example demonstrates how to query and change the loop schedule, as set through the `runtime` clause. In the next example, the nested parallelism code fragment listed in Figure 1.17 on page 25, is revisited. The relevant runtime functions are used to define a global thread ID. The third example illustrates how to get information on the thread affinity policy (or policies) that are in place.

Examples that use the device memory functions can be found in Section 6.12, starting on page 323. Examples using other heterogeneous runtime functions are shown in Section 6.5, starting on page 275.

Change the Loop Schedule Type at Runtime The `runtime` clause provides flexibility to select an appropriate schedule to distribute the work over the threads. There was one main issue with the original design, though. Recall that the schedule, as set by environment variable `OMP_SCHEDULE`, applied to *all* loops with this clause.

The new runtime functions provide greater flexibility and allow the application to customize the schedule to match the requirements on a per-loop basis. This is illustrated in the following example, which is an extended version of the code given in [2].

The function listed in Figure 2.22 uses the runtime function `omp_get_schedule()` to query the settings. At line 9, this function is called and the information is printed at lines $11-15$. In case a known value for the schedule type is returned, one of the four regular names is printed. Otherwise, "`implementation-defined`" is printed.

The runtime function also returns a value in the second argument, `schedule_modifier`. If the schedule type is `static`, `dynamic`, or `guided`, this variable contains the chunk size. It is undefined for schedule type `auto`, and implementation-dependent in case an implementation-specific schedule is used.

```
 1 void print_schedule_info(char *comment)
 2 {
 3   char *alt_schedule = "implementation-defined";
 4   char *schedules[4] = {"static","dynamic","guided","auto"};
 5
 6   omp_sched_t schedule_kind;
 7   int         schedule_modifier;
 8
 9   (void) omp_get_schedule(&schedule_kind, &schedule_modifier);
10
11   printf("Schedule details %s:\n", comment);
12   printf(" schedule kind  = %d\n", schedule_kind);
13   printf(" schedule type  = %s\n",
14    schedule_kind <= 4 ? schedules[schedule_kind-1]:alt_schedule);
15   printf(" schedule modifier = %d\n", schedule_modifier);
16 }
```

Figure 2.22: **A function to verify the loop schedule** – This function queries the current loop schedule and prints the information.

In Figure 2.23, this function is called twice. At line 1, the current settings for the schedule and chunk size are printed. These are applied to the parallel loop spanning lines $3 - 9$.

At line 11, the runtime function `omp_set_schedule()` is called. The first argument is the OpenMP variable `omp_sched_static`. This changes the schedule to `static`, regardless of the previous setting. Any value less than 1 for the second argument, enforces the default settings for the chunk size to be used. In this call, a value of zero is used to achieve this. The new settings are printed by the function call at line 13. This is what is used for the parallel loop at lines $15 - 21$.

The example code has been executed using 3 threads and a loop length of 7. The initial schedule is defined through environment variable `OMP_SCHEDULE` and set to `"dynamic,2"`. The full results are given in Figure 2.24.

The output confirms that the initial schedule type is `dynamic`, with a chunk size of 2. The assignment of iterations to threads for this first loop is just a snapshot. It can be expected that the results vary from run to run.

The results for the second loop demonstrate that the `static` schedule has indeed been used. Thread 0 executes the first 3 iterations. The remaining 4 iterations are

```
 1 (void) print_schedule_info("loop 1");
 2
 3 #pragma omp parallel for default(none) schedule(runtime)\
 4                         shared(n)
 5   for (int i=0; i<n; i++)
 6   {  printf("Iteration %d executed by thread %d\n",
 7           i, omp_get_thread_num());
 8   for (int j=0; j<i; j++) system("sleep 1");
 9 } // End of parallel for
10
11 (void) omp_set_schedule(omp_sched_static, 0);
12
13 (void) print_schedule_info("loop 2");
14
15 #pragma omp parallel for default(none) schedule(runtime)\
16                         shared(n)
17   for (int i=0; i<n; i++)
18   {  printf("Iteration %d executed by thread %d\n",
19           i, omp_get_thread_num());
20   for (int j=0; j<i; j++) system("sleep 1");
21 } // End of parallel for
```

Figure 2.23: **Example how to change the loop schedule** – This code fragment prints and changes the loop schedule.

distributed equally over the other 2 threads and are in consecutive order. These results are reproducible.

Nested Parallelism Revisited The new functions for nested parallelism provide a convenient way to query the environment. In this example, they are used to more easily construct a unique *global* thread ID.

Recall the example given in Figure 1.18 on page 26. Because the threads in each team start with thread ID 0 and are numbered consecutively, the value returned by function `omp_get_thread_num()` cannot be used to distinguish threads between different teams. Usually this is not an issue, but what if one would like to have a thread ID that is unique at a certain nesting level, or even across all nesting levels?

```
Schedule details loop 1:
 schedule type     = 2
 schedule type     = dynamic
 schedule modifier = 2
Iteration 0 executed by thread 2
Iteration 1 executed by thread 2
Iteration 2 executed by thread 0
Iteration 4 executed by thread 1
Iteration 6 executed by thread 2
Iteration 3 executed by thread 0
Iteration 5 executed by thread 1
Schedule details loop 2:
 schedule type     = 1
 schedule type     = static
 schedule modifier = 0
Iteration 5 executed by thread 2
Iteration 3 executed by thread 1
Iteration 0 executed by thread 0
Iteration 1 executed by thread 0
Iteration 2 executed by thread 0
Iteration 4 executed by thread 1
Iteration 6 executed by thread 2
```

Figure 2.24: **Example results from the program to query and set the loop schedule** – The loop length is set to 7, and 3 threads are used. Environment variable `OMP_SCHEDULE` is initialized to `"dynamic,2"`.

The following formulas may be used to construct a unique thread ID when the team size is *constant* at a specific nesting level. It can be different across nesting levels, though. Without this restriction the formulas do not apply and one must dynamically scan the tree representing the nesting structure in order to compute a unique thread ID. This is far beyond the scope of this example and is left as an exercise for the reader.

In the remainder, the thread ID at a specific nesting level is referred to as the *Level Thread ID*, or L_{TID} for short. This identifier is unique at each nesting level, but not across the levels. The thread ID that is unique across all nesting levels and thread teams is denoted by G_{TID} (*Global Thread ID*).

These are the formulas for L_{TID} and G_{TID}:

$$L_{TID} = T_{TID} + O_L(L) = T_{TID} + \sum_{k=1}^{L-1} \{AT_{L-k} \prod_{j=1}^{k} TS_{L-j+1}\} \tag{2.1}$$

$$G_{TID} = L_{TID} + O_G(L) = L_{TID} + \sum_{k=1}^{L-1} \prod_{j=1}^{k} TS_j \tag{2.2}$$

In formulas 2.1 and 2.2:

- L denotes the nesting level of the parallel region.
- T_{TID} is the team-level thread ID as returned by `omp_get_thread_num()`.
- $O_L(L)$ is the offset added to the team thread ID. Note that $O_L(1) = 0$, as one would expect.
- $O_G(L)$ is the global offset added to L_{TID}, the thread ID unique across one nesting level. Because $O_G(1) = 0$, the values for L_{TID} and G_{TID} are the same at level $L = 1$.
- TS_{level} is the value returned by `omp_get_team_size(level)`.
- AT_{level} is the value returned by `omp_get_ancestor_thread_num(level)`.

As an example, these formulas are applied to the nested parallelism example shown in Figure 1.18 on page 26. There, the team size for the second-level parallel region is 2 for each team. Because this is the second level, $L = 2$ and therefore $k = 1$. To compute the L_{TID} and G_{TID} values for the rightmost thread at this second level, these values are substituted in the formulas to get the following result:

$$L_{TID} = T_{TID} + AT_1 * TS_2 = 1 + 2 * 2 = 5$$

$$G_{TID} = 5 + TS_1 = 5 + 3 = 8$$

The C source function listed in Figure 2.25 returns the offset $O_L(L)$ at the current nesting level L as defined in Formula 2.1. This value can be added to the local thread ID T_{TID} to obtain the G_{TID} for the local thread.

```
 1 int level_offset(int *G_offset)
 2 {
 3   int team_size_prod;
 4   int global_prod;
 5
 6   int level_offset  = 0;
 7   int global_offset = 0;
 8   int cur_level     = omp_get_level();
 9
10   for (int k=1; k<cur_level; k++)
11   {
12     team_size_prod = 1;
13     global_prod    = 1;
14     for (int j=1; j<=k; j++)
15     {
16       team_size_prod *= omp_get_team_size(cur_level-j+1);
17       global_prod    *= omp_get_team_size(j);
18     }
19     team_size_prod *= omp_get_ancestor_thread_num(cur_level-k);
20
21     level_offset  += team_size_prod;
22     global_offset += global_prod;
23   } // End of loop over k
24
25   *G_offset = global_offset;
26
27   return(level_offset);
28 }
```

Figure 2.25: **Function to compute the offsets to the level and global thread ID** – This is a very straightforward implementation to compute the offsets that may be used to compute the L_{TID} and G_{TID} values as given in formulas 2.1 and 2.2. It is assumed that the team is constant within one nesting level. It may be different across nesting levels, though.

The function also computes the offset that can be added to the L_{TID} to obtain the unique G_{TID}. This offset is returned through the argument pointer `G_offset`.

The computations are deliberately implemented in a very straightforward way,

which may not be the most efficient way to do this, but hopefully makes things easier to understand.

At line 8, the current nesting level L is obtained. The for-loop spanning lines $10-23$ computes the sum of the various products. The products of the team sizes are computed at lines $12-18$. At line 19, this result is then multiplied with the appropriate thread ID of the ancestor thread. At lines $21-22$, the total sums for the level and global offsets are updated with their respective contributions.

With this function to compute the two offsets to the team thread ID, it is very easy to obtain the L_{TID} and G_{TID} values. The code given in Figure 1.17 on page 26 has been augmented to get these values. The relevant part of the new source code is shown in Figure 2.26.

A five column header is printed at lines $1-2$. The key lines here are $10-11$ and $22-23$. This is where the offsets are computed and added to variable `T_tid`, the team level thread ID, and `L_tid`. The three thread ID values, plus the current level and team size for the respective levels, are printed at lines $13-14$ and $25-26$.

Sample output is shown in Figure 2.27. At the first level, the values for the L_{TID} and G_{TID} are indeed the same. At the second level, there are $3x2 = 6$ threads. Although printed in arbitrary order, it is not difficult to see that each thread at the second level has a L_{TID} level ID in the $[0 \ldots 5]$ range. The $3 + 6 = 9$ global G_{TID} values are unique across both regions and in the $[0 \ldots 8]$ range.

Verify Thread Binding With the support for cc-NUMA, first introduced in OpenMP 4.0, one can now optimize data access for memory locality by informing the runtime system where threads should run. This is extensively covered in Chapter 4, starting on page 151. Here it is shown how to use some of the affinity-related runtime functions to verify that the settings are as intended. The same two nested parallel regions from Figure 2.26 are used, but this time the goal is to control and verify in which way the OpenMP threads are distributed across the system. In the example code in Figure 2.28, the runtime setting at each nesting level is verified. This includes the policy in place for the top-level master region.

At line 1, a variable of type `omp_proc_bind_t` is declared. It is used to store the return value of function `omp_get_proc_bind`. Character array `binding_choices` is declared and initialized at lines $2-3$. It is used to map the binding type to a descriptive name.

After printing a header at line 5, the binding information for the first-level parallel region is printed at lines $8-10$. Keep in mind that the value returned for the

```
 1 printf("%7s %7s %9s %7s %7s\n",
 2        "T_TID", "Level", "Team Size", "L_TID", "G_TID");
 3
 4 #pragma omp parallel num_threads(3)
 5 {
 6   int global_offset;
 7   int cur_level     = omp_get_level();
 8   int cur_team_size = omp_get_team_size(cur_level);
 9   int T_tid         = omp_get_thread_num();
10   int L_tid         = T_tid + level_offset(&global_offset);
11   int G_tid         = L_tid + global_offset;
12
13   printf("%7d %7d %9d %7d %7d\n", T_tid, cur_level,
14                                    cur_team_size, L_tid, G_tid);
15
16   #pragma omp parallel num_threads(2)
17   {
18     int global_offset;
19     int cur_level     = omp_get_level();
20     int cur_team_size = omp_get_team_size(cur_level);
21     int T_tid         = omp_get_thread_num();
22     int L_tid         = T_tid + level_offset(&global_offset);
23     int G_tid         = L_tid + global_offset;
24
25     printf("%7d %7d %9d %7d %7d\n", T_tid, cur_level,
26                                      cur_team_size, L_tid, G_tid);
27
28   }   // End of nesting level 2
29 }   // End of nesting level 1
```

Figure 2.26: **Nested parallelism example revisited** – The function shown in Figure 2.25 is used to compute and print the level thread ID L_{TID} and global ID G_{TID} at both parallel nesting levels. The team thread ID, nesting level, and team size are printed as well.

affinity policy is for the *next* nesting level. This means the value returned in array element `binding_choices[binding_setting]` is the affinity policy for the upcoming parallel region, *not* the current one.

```
T_TID   Level Team Size   L_TID   G_TID
    2       1         3       2       2
    1       1         3       1       1
    0       1         3       0       0
    0       2         2       0       3
    1       2         2       1       4
    0       2         2       4       7
    1       2         2       5       8
    0       2         2       2       5
    1       2         2       3       6
```

Figure 2.27: **Results for the revisited nested parallelism example** – The G_{TID} value at the first nesting level is indeed the same as the local L_{TID} value. At the second level, there are 6 threads and each thread has a unique ID in the $[0 \ldots 5]$ range.

This is repeated for the two parallel regions spanning lines $12-31$. In both cases, a `single` region is used to print the binding only once per parallel region. Because it is the same for all threads in the same team, printing this for all threads is not very meaningful.

Thread placement is achieved by setting environment variables `OMP_PLACES` and `OMP_PROC_BIND`. In practice, both must be set, but for the purpose of this example, only `OMP_PROC_BIND` must be defined. It is set to the following list: `"spread,close,master"`.[2]

The output is given in Figure 2.29 and confirms the outer level threads are spread across the system. The first printed line shows that execution is at level 0, and the next parallel region has the `spread` affinity policy. Likewise, the threads at each second-level parallel region are requested to be scheduled close to each other.

The last three lines are from the three innermost parallel regions and confirm that a subsequent parallel region has the `master` affinity policy. In this case, there is no such third-level region, though.

If there are deeper runtime nesting levels than specified through environment variable `OMP_PROC_BIND`, the remaining levels inherit the settings from the last level specified. For example, if this variable was set to `"spread"` only, the thread teams have affinity type "spread" at all nesting levels.

[2]For a definition of these keywords, refer to Section 4.6, starting on page 168.

```
 1 omp_proc_bind_t binding_setting;
 2 char            *binding_choices[5] = \
 3                 {"false", "true", "master", "close", "spread"};
 4
 5 printf("%13s %25s\n\n","Current Level","Binding Type/Level");
 6
 7 binding_setting = omp_get_proc_bind();
 8 printf("%7d %23s/%-3d\n",omp_get_level(),
 9                          binding_choices[binding_setting],
10                          omp_get_level()+1);
11
12 #pragma omp parallel num_threads(3)
13 {
14  #pragma omp single
15  { omp_proc_bind_t binding_setting = omp_get_proc_bind();
16    printf("%7d %23s/%-3d\n",omp_get_level(),
17                             binding_choices[binding_setting],
18                             omp_get_level()+1);
19  } // End of single region
20
21  #pragma omp parallel num_threads(2)
22  {
23     #pragma omp single
24     { omp_proc_bind_t binding_setting = omp_get_proc_bind();
25       printf("%7d %23s/%-3d\n",omp_get_level(),
26                                binding_choices[binding_setting],
27                                omp_get_level()+1);
28     } // End of single region
29
30  } // End of nesting level 2
31 } // End of nesting level 1
```

Figure 2.28: **Example to verify the thread affinity settings** – There is one master region with two nested parallel regions. At each level, the upcoming thread affinity setting, as defined by environment variable `OMP_PROC_BIND`, is verified.

```
Current Level        Binding Type/Level

      0                   spread/1
      1                    close/2
      2                   master/3
      2                   master/3
      2                   master/3
```

Figure 2.29: **Output from the code shown in Figure 2.28** – Prior to executing the program, environment variable `OMP_PROC_BIND` is set to `"spread,close,master"`. The output confirms these settings. Note that the "master" policy applies to a possible third-level parallel region, but this example has two levels only.

2.4 New Functionality

Compared to OpenMP 2.5, OpenMP 4.5 has significant new functionality. In this section, those new features that have a relatively small impact on the programming model and are often implemented as an extension to already existing functionality are presented and discussed.

In subsequent chapters, the new concepts and features that introduce entirely new concepts in OpenMP are covered extensively.

2.4.1 Changed Ownership of Locks

The ownership of locks has changed. Although this is not new functionality, it is important enough to mention. Whereas the ownership of a lock was tied to threads in OpenMP 2.5, as of OpenMP 3.0, locks are owned by task regions.

In many cases, this should not have any consequences, but in the example in Figure 2.30, there is a difference.

The lock variable `my_lock` is declared, initialized and acquired at lines $1-4$. This code is executed by the master thread and because it executes the `omp_set_lock()` function, it owns the lock.

Within the parallel region spanning lines $6-15$, the release of the lock at line 11 is guaranteed to be performed by the master thread again.

The task region where the lock is released is not same task region where it was acquired. Although this was allowed in OpenMP 2.5, this is no longer OpenMP compliant code as of version 3.0.

```
 1  omp_lock_t my_lock;
 2
 3  (void) omp_init_lock (&my_lock);
 4  (void) omp_set_lock  (&my_lock);
 5
 6  #pragma omp parallel
 7  {
 8     #pragma omp master
 9     {
10                 ......
11         (void) omp_unset_lock (&my_lock);
12                 ......
13     } // End of master region
14     ......
15  } // End of parallel region
16
17  (void) omp_destroy_lock (&my_lock);
```

Figure 2.30: **Non-conforming use of locks** – The master thread sets the lock and therefore owns it. In OpenMP 2.5 it was permitted to have the master thread release the lock within the parallel region. This is however no longer allowed, because these two actions occur in different task regions.

Although one may, of course, write code like this, a better and cleaner approach is shown in Figure 2.31. There are two different task regions again, and the lock is accessed in both, but this program is OpenMP-compliant. The reason is that the lock is acquired (line 10) and released (line 12) in the same task region.

2.4.2 Cancellation

There are two reasons for a parallel region to terminate prematurely. An error may have occurred, for example a memory allocation that fails, and the thread(s) can no longer continue execution. It doesn't always have to be a fatal situation however. There may be a good reason the threads no longer need to continue. An example is a parallel search. Once the target element has been found, all threads can stop searching.

Prior to OpenMP 4.0, it was impossible to end the execution of an individual parallel region or, in general, any OpenMP construct within a parallel region. This

```
 1  omp_lock_t my_lock;
 2
 3  (void) omp_init_lock (&my_lock);
 4
 5  #pragma omp parallel
 6  {
 7     #pragma omp master
 8     {
 9                 ......
10         (void) omp_set_lock (&my_lock);
11                 .....
12         (void) omp_unset_lock (&my_lock);
13     } // End of master region
14  } // End of parallel region
15
16  (void) omp_destroy_lock (&my_lock);
```

Figure 2.31: **Conforming use of locks** – This program is OpenMP-compliant, because the lock is initialized and released in the same task region.

implied that every OpenMP region either had to be executed to the very end, or not started at all.

Other than to abort program execution, the only alternative was for the application to detect a certain situation and if needed, take action after the parallel region finished. With the introduction of the *cancellation constructs* in OpenMP 4.0, an elegant way to terminate the execution of an OpenMP region is supported. Aside from gracefully handling certain types of errors, this feature is useful for specific parallel methods with a dynamic behavior, such as divide-and-conquer algorithms.

Cancellation is supported on the `parallel` construct, as well as the `sections` and `for` (`do` in Fortran) worksharing constructs. When using tasks, the `taskgroup` construct may be used for cancellation.

The syntax of the `cancel` and `cancellation point` constructs in C/C++ and Fortran is shown in Figure 2.32. Both constructs are stand-alone directives. They trigger an action if, or when, encountered at runtime. The idea is similar to exception handling, supported in some programming languages. A thread that wishes to terminate execution, needs to have the `cancel` directive in its execution path. This is most likely under control of an if-statement. The effect of the `cancel` direc-

#pragma omp cancel *construct-type-clause[[,]if-clause] new-line*
!$omp cancel *construct-type-clause[[,]if-clause]*

#pragma omp cancellation point *construct-type-clause new-line*
!$omp cancellation point *construct-type-clause*

Figure 2.32: **Syntax of the cancellation construct in C/C++ and Fortran** – The `cancel` directive requests the termination of the corresponding construct. For the other threads, this takes effect at the next cancellation point.

tive is to raise a flag to signal that cancellation has occurred. The `cancellation point` directive defines a point where the encountering thread checks the status of cancellation.

This is not the only place though. The request for cancellation is checked at the following *cancellation points*:

- A `cancel` region.
- A `cancellation point` region.
- A `barrier` region.
- An implicit barrier region.

Cancellation is controlled through environment variable `OMP_CANCELLATION`, described in Section 2.2. This variable needs to be explicitly set to `true` to enable cancellation. To reduce the runtime overhead, it is *disabled* by default. If a `cancel`, or `cancellation point`, directive is encountered, while this feature is not enabled, the constructs have no effect.

When a thread raises the cancellation flag, the event is not detected by another thread, until it has encountered one of the cancellation points listed above. As a consequence, cancellation may not take effect immediately.

This is illustrated in Figure 2.33, where time flows from top to bottom. A team of five threads executes an "enclosing construct." This is one of the constructs `parallel`, `for` (or `do`), `sections`, or `taskgroup` that supports cancellation. Events 1 to 6 are assumed to happen in the order depicted by their position on the timeline.

At event 1, thread three encounters a `cancellation point` directive. At this point, cancellation has not yet been requested. The `cancellation point` has no effect and this thread continues to execute the enclosing construct.

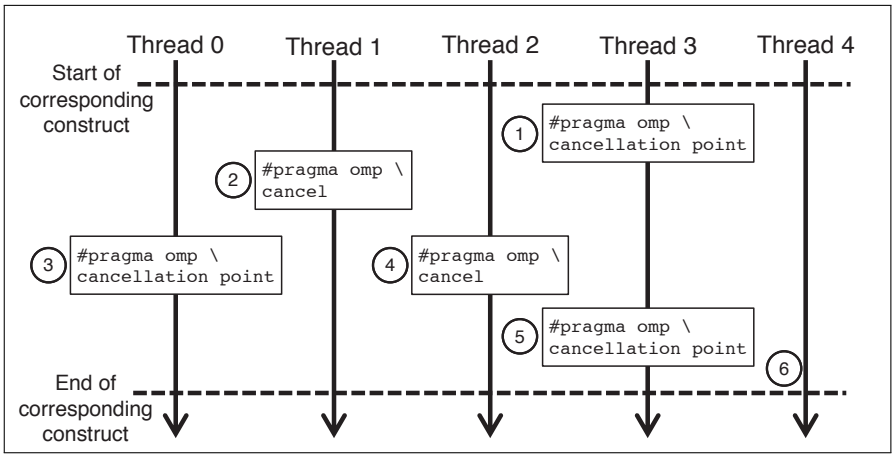

Figure 2.33: **Illustration of the cancellation feature** – In this diagram, time flows from top to bottom. Once cancellation is requested, threads only respond at the next cancellation point encountered at runtime.

Thread one encounters a `cancel` directive at event 2, and execution of the enclosing construct is cancelled. This raises a flag to indicate that cancellation has been requested. At the implied barrier or synchronization point at the end of the enclosing construct, thread one waits for all other threads to either complete execution of the enclosing construct, or be cancelled.

At event 3, thread zero encounters a `cancellation point` directive. Because the cancellation flag has already been set, it immediately cancels the execution of the enclosing construct. Again, the thread waits at the implied barrier, or synchronization point at the end of the enclosing construct, for all other threads, and then continues execution after the enclosing construct.[3]

Thread two encounters a `cancel` directive at event 4. Cancellation has already been requested and this thread immediately cancels execution of the enclosing construct. Thread three encounters a second `cancellation point` directive at event 5. This time (as opposed to event 1), cancellation is already requested and the thread cancels the execution of the enclosing construct.

[3]If the flag is set, the OpenMP runtime function `omp_get_cancellation()` returns the value `true`.

Thread four never encounters any cancellation points and completes execution of the construct.

To ensure timely cancellation, the user must insert `cancellation point` directives at suitable locations throughout the code, which is illustrated in the example in Figure 2.34.

Where exactly does a thread continue execution after the enclosing construct has been cancelled? This depends on the type of construct that is specified as a clause on the `cancel` directive. The clause can be any of these types:

- `parallel` - The thread cancels the current execution and the master thread continues after the implicit barrier at the end of the `parallel` construct. All tasks created in the parallel region are cancelled. In the case of a nested parallel region, cancellation only applies to the innermost parallel region.

- `for` (or `do` in Fortran), or `sections` - The thread cancels the current execution and jumps to the implicit barrier at the end of the worksharing construct. Task cancellation does not occur. If a `reduction` or `lastprivate` clause is specified on a worksharing construct and cancellation has occurred, the value of the corresponding variables are undefined.

  There are two restrictions. The worksharing loop construct (`for`, or `do`) may not contain an `ordered` construct. The `nowait` clause is not allowed on the worksharing constructs supported with cancellation.

- `taskgroup` - The task encountering the `cancel` directive sets the flag for the entire `taskgroup` region, but it continues execution to its end and then continues execution after the `task` region. The same holds for all tasks of the `taskgroup` that have already started execution. All other tasks belonging to the `taskgroup` are cancelled. If a task belongs to multiple (nested) `taskgroup` regions, and encounters a `cancellation point` construct, it checks if cancellation is requested in any `taskgroup` region it belongs to, and responds accordingly.

The use of the `cancel` and `cancellation point` directives is illustrated in Figure 2.34. This code fragment contains a nested parallel region. The outer parallel region spans lines $1 - 18$, and the inner parallel region spans lines $4 - 17$.

When a thread encounters the `cancel` construct at line 11, cancellation is requested for the innermost parallel region. If and when the condition at line 9

```
 1 #pragma omp parallel
 2 {
 3    // Do some work
 4    #pragma omp parallel
 5    {
 6      for (int i = 0; i < N; i++)
 7      {
 8        // Do some work
 9        if ( <some_condition> )
10        {
11           #pragma omp cancel parallel
12         }
13        // Do some more work
14        #pragma omp cancellation point
15        // Do even more work
16       }
17    } // End of inner parallel region
18 } // End of outer parallel region
```

Figure 2.34: **Cancellation of a parallel region** – Execution of the innermost parallel region is cancelled by the thread executing the cancel construct at line 11. Depending on where they are when this happens, the other threads either also cancel their work, or complete execution.

evaluates to `true`, the thread encountering the `cancel` construct terminates execution. At line 11, it is specified that the parallel region is affected. As a result, all remaining work is skipped and the thread waits in the implied barrier at the end of the inner parallel region (line 17). Those other threads that encounter the `cancellation point` at line 14, cancel the execution of the parallel region, and wait in the barrier. It a thread encounters line 14 before cancellation has been requested, the `cancellation point` has no effect. In case a thread is already past the cancellation point, it continues to execute and also waits in the barrier.

The execution of the outer parallel region is not affected by the cancellation. There is no support in OpenMP to request cancellation for an outer parallel region, or for both regions. If needed, the user needs to implement this manually.

Figure 2.35 illustrates the use of the `cancel` construct in a recursive algorithm employing tasks. Tasks are covered in great detail in Chapter 3, starting on page 103.

In this example, a binary tree, called `t`, is searched for the element `e`. Variable `present` is used to mark the termination of the search. Assuming it is initialized to `false`, it is changed to `true`, once the element has been found.

The search is parallelized using tasks that recursively process the left and right subtree of each element.[4]

The tasks are embedded in a `taskgroup` construct (line 7), because that is the granularity for cancellation with tasks. The `taskgroup` is embedded into a `master` construct to ensure that one thread only initiates the parallel search. Instead of `master`, a `single` construct has the same effect.

When the element is found (line 17), cancellation of the `taskgroup` is requested (line 19). Because the task executes to its end, the assignment to `present` may happen either after the `cancel` construct, or before. In this particular example, there is no need to use explicit `cancellation point` directives, because the generation of new tasks is terminated as soon as cancellation has been requested. All previously created, but not yet started, tasks are also terminated then.

If the element is not found, the initial value of variable `present` does not change. The algorithm terminates, because the recursion of `search` ends at leaf nodes, because then `t->left` and `t->right` are `NULL` and the `if` conditions (lines 22 and 27) evaluate to false.

2.4.3 User-Defined Reduction

Until OpenMP 4.0, there was no support to provide an application-specific reduction operation in case the standard reductions were not sufficient. Although one can always implement a reduction explicitly, this is not desirable. Through a feature called *User-Defined Reduction* (UDR) , a user has the option of defining a customized reduction operation and using it in a `reduction` clause to leverage the ease of use that OpenMP provides for such operations.

Recurrence calculations, such as the one shown in Figure 2.36, are common in scientific and engineering programs, but are also encountered in other disciplines. Recall from [2] that the `reduction` clause is provided to parallelize recurrence operations without the need for code modifications.

[4] A detailed example of a recursive algorithm parallelized with tasks can be found in Section 3.3, starting at line 118.

```
 1 void search_parallel(btree *t, element e, bool* present)
 2 {
 3   #pragma omp parallel
 4   {
 5     #pragma omp master
 6     {
 7       #pragma omp taskgroup
 8       {
 9         search(t, e, present);
10       }
11     } // End omp master
12   } // End omp parallel
13 }
14
15 void search(btree* t, element e, bool* present)
16 {
17   if (t->element == e)
18   {
19     #pragma omp cancel taskgroup
20     *present = true;
21   }
22   if (t->left)
23   {
24     #pragma omp task
25       {search(t->left, e, present);}
26   }
27   if (t->right)
28   {
29     #pragma omp task
30       {search(t->right, e, present);}
31   }
32 }
```

Figure 2.35: **Cancellation of a taskgroup in a recursive algorithm** – The binary tree is searched for element `e` and the parallel descent is cancelled if this element is found.

```
1 int sum = 0;
2 int *a, *b;  // Assume allocation and initialization occurred
3
4 #pragma omp parallel for default(none) firstprivate(n)\
5             shared(a,b) reduction(+:sum)
6 for (int i=0; i<n; i++)
7    sum += a[i] * b[i];
```

Figure 2.36: **Example of a recurrence calculation** – The `reduction` clause is used to parallelize the loop. Both the operation (`+`) and result variable (`sum`) are specified as part of the clause.

The reduction operator(s) and variable(s) are specified in the `reduction` clause. By definition, the result variable, like `sum` in this case, is shared in the enclosing OpenMP region.

This operation is implemented as follows: The thread performs the addition operation on a subset of the data. It uses a local, or private, variable to store the partial result. After completion of the operation, a thread updates the global result with its contribution. To avoid a data race, this update is protected through a lock.

The private copy of the reduction variable must be initialized with a value that is mathematically consistent with the reduction operation. The reduction operation must also be mathematically associative and commutative. This is because the order in which threads combine their value to construct the value for the shared result variable is non-deterministic.

A side effect of this is that floating-point results may vary (slightly) from run to run, and/or depend on the number of threads used. This is a numerical effect caused by round-off differences introduced when combining the result. It has nothing to do with the reduction operation as such, but could be a reason for numerical results to be somewhat different.

Figure 2.37 gives an overview of the pre-defined reduction operators supported in OpenMP. In this table, the second column lists the initialization value for the private instance of the reduction variable. This is the neutral element for the specific reduction operation. The third column is labeled "Combiner" and defines the arithmetic operation that needs to be performed to produce the result.

Variables `omp_priv`, `omp_in`, and `omp_out` used in this table are also needed in case of a UDR. This is discussed in more detail below.

Operator	Initializer	Combiner
+	omp_priv = 0	omp_out += omp_in
*	omp_priv = 1	omp_out *= omp_in
-	omp_priv = 0	omp_out += omp_in
&	omp_priv = ~0	omp_out &= omp_in
|	omp_priv = 0	omp_out |= omp_in
^	omp_priv = 0	omp_out ^= omp_in
&&	omp_priv = 1	omp_out = omp_in && omp_out
||	omp_priv = 0	omp_out = omp_in || omp_out
max	omp_priv = *Least(T)*	omp_out = omp_in > omp_out ? omp_in : omp_out
min	omp_priv = *Largest(T)*	omp_out = omp_in < omp_out ? omp_in : omp_out
+	omp_priv = 0	omp_out = omp_in + omp_out
*	omp_priv = 1	omp_out = omp_in * omp_out
-	omp_priv = 0	omp_out = omp_in + omp_out
.and.	omp_priv = .true.	omp_out = omp_in .and. omp_out
.or.	omp_priv = .false.	omp_out = omp_in .or. omp_out
.eqv.	omp_priv = .true.	omp_out = omp_in .eqv. omp_out
.neqv.	omp_priv = .false.	omp_out = omp_in .neqv. omp_out
max	omp_priv = *Least(T)*	omp_out = max(omp_in, omp_out)
min	omp_priv = *Largest(T)*	omp_out = min(omp_in, omp_out)
iand	omp_priv = not(0)	omp_out = iand(omp_in, omp_out)
ior	omp_priv = 0	omp_out = ior(omp_in, omp_out)
ieor	omp_priv = 0	omp_out = ieor(omp_in, omp_out)

Figure 2.37: **The pre-defined reduction operators supported by the reduction clause in C/C++ and Fortran** – The upper half of the table is for C/C++. The lower half of the table is for Fortran. This table lists the pre-defined reduction operators, along with their associated initializer and combiner expressions. `Least(T)` and `Largest(T)` are the least and largest representable values of the arithmetic type T.

While the `reduction` clause provides significant convenience to the user, certain applications have their own reduction operation that is not supported by OpenMP. Examples of that include the merging of lists or sets, the computation of a standard deviation over a set of elements, etc. It is always possible to explicitly code a reduction, but this adds to the complexity of the code, and it may not perform optimally.

```
 1 int *a; // Assume allocation and initialization occurred
 2 int result = INT_MAX;
 3
 4 #pragma omp declare reduction(my_abs_min : int :\
 5   omp_out = abs(omp_in) < omp_out ? abs(omp_in) : abs(omp_out))\
 6   initializer (omp_priv = INT_MAX)
 7
 8 #pragma omp parallel for default(none) firstprivate(n)\
 9             shared(a) reduction(my_abs_min : result)
10 for (int i=0; i<n; i++) {
11    if (abs(a[i]) < result) {
12       result = abs(a[i]);
13    }
14 }
```

Figure 2.38: **Example of a User-Defined Reduction** – The definition of a UDR used in combination with the `reduction` clause to compute the minimum absolute value of the array elements in parallel.

This is why the UDR was introduced in OpenMP 4.0. A UDR is defined through the `declare reduction` directive and consists of the following three components:

- Identifier - A name, or symbol, that identifies the reduction and the type(s) to which it applies.
- Combiner - A recipe to generate a result. It is expressed as an operation involving the two symbolic variables `omp_in` and `omp_out`, thus defining the actual reduction operation as applied on any given two arguments.
- Initializer - This is used to define the initial value for the private instances of a reduction variable.

The `declare reduction` directive must be used to define a UDR. Before introducing the formal syntax, the example in Figure 2.38 is presented and discussed. In this code, a UDR is defined to compute the minimum of the absolute values of an array.

At lines $4 - 6$, the `declare reduction` directive defines the UDR called `my_abs_min`. It returns a value of type `int`.

#pragma omp declare reduction*(reduction-identifier : typename-list : combiner) [initializer-clause] new-line*
!$omp declare reduction*(reduction-identifier : typename-list : combiner) [initializer-clause]*

Figure 2.39: **Syntax of the declare reduction directive in C/C++ and Fortran** – This directive defines a user-defined reduction. The various components are explained in the text. Figure 2.40 contains an explanation of the terminology.

Name	Description
reduction-identifier	A C, C++, or Fortran identifier or one of the pre-defined OpenMP reduction operators.
typename-list	The list with data type(s) to be used.
combiner	An expression in C/C++. In Fortran, it is an assignment statement, or subroutine name, followed by an argument list.
initializer-clause	This is of the form `initializer`*(initializer-expr)* with *initializer-expr* being one of the following: `omp_priv` = *initializer* (C/C++) `omp_priv` = *function-name(argument-list)* (C/C++) `omp_priv` = *expression* (Fortran) `omp_priv` = *subroutine-name(argument-list)* (Fortran)

Figure 2.40: **Terminology used with the declare reduction directive** – This table explains the various elements used to define a reduction.

The *combiner* is defined at line 5. Symbolic variable `omp_in` represents the variable used to store the partial result. This variable is private to each thread. Likewise, symbolic variable `omp_out` refers to the result of the combiner operation.

The expression at line 5 returns the absolute value of `omp_in` when it is less than the previous value of `omp_out`; otherwise, it returns the absolute value of the latter variable.

At line 6, another special variable, `omp_priv`, is set to the initial value each thread assigns to the local variable `omp_in`. In this case, it is `INT_MAX`, the largest integer value the system provides.

In this initialization expression, only special variables `omp_priv` and `omp_orig` are allowed. Although not used here, `omp_orig` represents the original value of the reduction variable. This variable is not allowed to be modified.

```
1 typedef std::vector< float > TVec;   // Assume some type here
2
3 #pragma omp declare reduction(+ : TVec : \
4         std::transform(omp_in.begin(),  omp_in.end(), \
5                        omp_out.begin(), omp_out.begin(), \
6                        std::plus< T >() ) ) \
7         initializer (omp_priv(omp_orig))
```

Figure 2.41: **Example of a User-Defined Reduction on C++ arrays** – The use of a UDR allows for the parallelization of the loop employing a C++ array type.

Now that the elements of the reduction operation are defined, it may be used in a regular `reduction` clause. This is shown at line 9, where the name `my_abs_min` is used in the clause.

The loop spanning lines $10 - 14$ does not need to be modified. At runtime, the reduction operation is executed the same way a pre-defined reduction is handled.

After this example, it is time to look at the more formal description. The syntax of the `declare reduction` directive is given in Figure 2.39. The terminology used is defined in Figure 2.40.

OpenMP defines the special variables `omp_orig`, `omp_priv`, `omp_in`, and `omp_out`. They are all of the same type and the only variables to be used when defining a UDR operation.

This might raise the question why the description of *typename-list* in Figure 2.40 shows that a list with multiple types is allowed. This is a convenience. It avoids the need to repeat the `declare reduction` directive for the data types to be supported. With a list, it is as if the definition is repeated for all the types in the list.

Neither the number of times the combiner is executed, nor the number of times and the order in which the initializer expression is evaluated, are implementation-defined. The results are undefined if a program makes any assumption about this.

Instead of the initializer expression, it is also possible to provide a function, or subroutine name in Fortran, with an argument list. In this case, one of the arguments must be the special variable `omp_priv`. For C++ programs, the `initializer` clause also accepts an expression that is not an assignment to `omp_priv`. This is demonstrated in Figure 2.41.

The code shown in Figure 2.41 implements a UDR called `+` to perform the reduction on arrays. The C++ data type `std::vector` represents the array, for which a

depend *(dependence-type: iteration-vector)*
depend *(dependence-type)*

Figure 2.42: **Syntax of the depend clause for the doacross loop in C/C++ and Fortran** – In the context of a `doacross` loop, this clause accepts two new keywords for the dependence-type. This can either be `sink`, followed by the iteration vector, or `source`, which does not have any further qualifiers. Refer to the main text for an explanation of these terms.

given type `T` is assumed. The code provides the `+` reduction operation for this data type. For the standard data types, the pre-defined reduction operation is used.

The combiner (lines $4-6$) makes use of the `std::transform` function. The initializer at line 7 specifies that every thread's private copy is constructed with the copy constructor, using the original reduction variable as the argument. The advantage is that by using this feature, the thread's private arrays in this example do not need to have a known size at compile time. Instead, the length is derived from the original list item.

2.4.4 The Doacross Loop

As of OpenMP 4.5, an additional type of parallel loop is supported. It is called the "doacross loop" [24].

A `doacross` loop has a loop-carried dependence. Such loops cannot be parallelized in a straightforward way, but through compiler transformations and/or additional synchronization, the code may be partially parallelized. It is important to note that these are *static* dependences. This technique cannot be used to handle runtime dependences.

In contrast with other newly introduced features, there is no specific construct to implement this functionality. Instead, it is supported through extensions on the `ordered` and `depend` clauses. The latter was introduced in OpenMP 4.0 for tasking. In OpenMP 4.5, the `depend` clause has been augmented to add functionality for tasks, but is also used to implement the `doacross` feature.

The two main new elements are called *sink* and *source*. The former takes an additional description, the *iteration vector*, to describe the dependences *relative* to the current loop variable. The syntax of this vector is $x_1\,[\pm d_1], x_2\,[\pm d_2], \ldots, x_n\,[\pm d_n]$. In this notation, x_i is the name of the loop variable of the i-th nested loop, d_i is

```
 1 #pragma omp parallel default(none) shared(n,a,b,c)
 2 {
 3   #pragma omp for ordered(1)
 4   for (int i=1; i<n; i++)
 5   {
 6      #pragma omp ordered depend(sink:i-1)
 7         a[i] = a[i-1] + b[i];
 8      #pragma omp ordered depend(source)
 9
10      c[i] = 2*a[i];
11
12    } // End of for-loop
13 } // End of parallel region
```

Figure 2.43: **Code fragment using a doacross loop** – The doacross loop is defined through the combination of the `ordered` and `depend` clauses.

a *constant* non-negative integer, and n is the value as specified in the `ordered(n)` clause on the loop directive.

Before we continue, it is probably best to show an example first. The parallel region shown in Figure 2.43 contains a `doacross` loop to parallelize the update of vector `a`.

The computation of `a[i]` at line 7 has a data dependence, but it can be partially parallelized. Another thing to note is that the computation of `c[i]` at line 10 depends on the recently computed value `a[i]`. Due to this dependence, this loop cannot be fully parallelized, but there is reduced parallelism that may be exploited by using the support for `doacross` loops. The `doacross` loop is within the parallel region and spans line $3 - 12$.

The `ordered` clause at line 3 should look familiar. Until OpenMP 4.5, this enforced serial execution of the ordered code block defined within the loop. The clause has been extended to support a number between parentheses to indicate the loop is a `doacross` loop. The number, which is 1 in this case, indicates the depth of the parallelism. In general, this number specifies the number of perfectly nested loops constituting the `doacross` loop. For example, if there are three loops with dependences, this number must be 3.

The `ordered` directives at line 6 and 8 are mandatory. They define the depen-

dence region. The `depend` clause at line 6 uses the one dimensional iteration vector to specify that there is a dependence of loop iteration `i` on `i-1` ($n = 1$, $x_1 = i$, the sign is negative, and $d_1 = 1$).

In general, *all* dependences in *all* dimensions must be specified on the `ordered` clause that has the `sink` keyword. If, for example, there are two loops and three dependences, there must be three `depend` clauses, and each clause must use an index vector with two components of the type $x_1 [\pm d_1], x_2 [\pm d_2]$, like `i-1,j+1`.

The `depend` clause with the `source` keyword at line 8 is mandatory and forces the runtime system to make the result `a[i]` available. This is to ensure the computation at line 10 uses the correct value for `a[i]`. If this clause is left omitted, the program deadlocks. The same is true if the dependences are not described correctly.

2.5 Concluding Remarks

In this chapter, a broad overview of where OpenMP stands today has been given. Many OpenMP 2.5 constructs have been enhanced, including the introduction of the `collapse` clause to better support nested loops, refinements on the `critical` construct, plus improved support for the `atomic` construct to support various types of updates.

But there is much more. Through the `doacross` construct, certain loops with dependences can now be parallelized. Another new feature is cancellation, to provide a more robust handling of errors during parallel execution. User-defined reductions allow the user to write application specific reductions.

In addition to all of this, major new functionality has been introduced. Tasking, affinity control, vectorization using SIMD instructions, and the support for heterogeneous architectures, all make OpenMP more suitable for a wider range of systems and applications. These four new features are so substantial that they deserve their own chapter. That is what the remainder of this book is about.