CHAPTER

OpenMP 10

Overview

OpenMP is a mature, standard API for writing shared memory parallel applications in C, C++, and FORTRAN. When SMP architectures with shared memory nodes emerged in the computing market in the mid-1990s, all leading software and hardware vendors quickly agreed on a standard aimed at easing the task of developing multithreaded applications. At that time, native multithreading libraries—widely used in system programming—lacked high-level, easy-to-use interfaces for thread management and synchronization. The advent of OpenMP encouraged application programmers to quickly take advantage of the availability of a few CPUs to boost the performance of their sequential applications. Indeed, the OpenMP strategy is based on directives inserted in sequential codes, guiding the compiler in producing multithreaded code. Clearly, starting from a sequential code and quickly boosting its performance by adding a few directives is indeed a very attractive option, and it played a major role in driving the adoption of shared memory application programming in the early days. The examples in previous chapters adopted instead a parallel focus from the start, and organized the application code accordingly: OpenMP is indeed a very flexible programming environment, also well adapted to this programming style.

The include file for the OpenMP API is `omp.h`, which consists of:

- *Directives* used to manage and synchronize threads.
- *Environment variables* used to set the main configuration options of OpenMP.
- *Runtime routines*: There are three kinds:
  - Lock routines that provide fine control of mutual exclusion (already discussed in Chapter 6).
  - Execution environment routines that provide useful information and allow the programmer to set up configuration options of an OpenMP program at runtime.
  - Portable timing routines.

The core of the OpenMP API is the management of an *implicit thread pool*, present in any OpenMP code. By implicit we mean that the thread pool is already created by the runtime system when the `main()` function starts. All the user has to do is activate the worker threads with the `parallel` directive, as shown by several examples in the previous chapters. OpenMP has many other features required to implement sophisticated parallel applications (nested parallel jobs, dynamically generated tasks, etc.). In addition, OpenMP proposes *work-sharing* directives that perform an automatic parallelization of loops. This

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00010-1

OpenMP feature is used mainly in a microtasking programming style, where the code remains most of the time in sequential mode, and individual loops are parallelized as they are encountered.

Given the substantial length of this chapter, it is probably useful to start by explaining how its content is organized:

- Overview of the basic OpenMP execution model, and the programming interfaces for configuring OpenMP:
  - Section 10.1 extends the discussion of the basic OpenMP execution model, initiated in previous chapters.
  - Section 10.2 discusses the various ways of configuring OpenMP to adapt the execution environment to the requirements of specific applications.
- OpenMP directives for thread management and synchronization:
  - Section 10.3 describes the OpenMP directives for managing a team of worker threads and sharing the computational workload among them, as well as the accompanying clauses that adapt the directive operation to a specific parallel context.
  - Section 10.4 deals with the OpenMP synchronization directives.
  - Section 10.5 proposes several examples of thread management, work-sharing, and thread synchronization. All these examples refer to the basic OpenMP programming model dealing with regular parallel contexts, where the work-sharing patterns are easily controlled by the programmer. More complex examples along the same lines are found in the three chapters (Chapters 13–15) devoted to specific applications.
- Extended, task-centric, OpenMP execution model. In-depth discussion of the task API, including new OpenMP 4.0 features:
  - Section 10.6 describes the task API—introduced in OpenMP 3.0, 3.1, and further extended in OpenMP 4.0—as an extended execution model tailored to deal with irregular or recursive parallel patterns. The way in which the task API is articulated with the basic programming model is carefully discussed.
  - Section 10.7 proposes several examples of irregular or recursive problems where the task API plays a central role, including all the new task API features incorporate by OpenMP 4.0.
  - Section 10.8 discusses a few best practices to keep in mind when using the task API.
- Further OpenMP 4.0 features:
  - Section 10.9 introduces a new OpenMP 4.0 feature: the cancellation of parallel regions or task groups. An example simulating a database search illustrates the relevance of this feature.
  - Section 10.10 describes the OpenMP 4.0 new programming interfaces for heterogeneous computing: offloading code blocks to accelerators. The basic ideas are exposed, but at the time of the preparation of this manuscript this feature was not fully implemented in the available compilers. Examples will be in due time incorporated to the accompanying software.
  - Section 10.11 deals with the new OpenMP 4.0 programming interfaces implementing thread affinity.
  - Section 10.12 presents the new OpenMP 4.0 programming interfaces implementing vectorization, as a way of enhancing single-core performance. Again, at the time of the

preparation of the manuscript this feature was not fully implemented in the available compilers. But vectorization has already been available for some time in the Intel compilers, and examples of vectorization are proposed in the three chapters (Chapters 13–15) dealing with specific applications.

Some of the sections quoted above—in particular Sections 10.2 and 10.3—deal with a substantial amount of material. We have tried very hard to make them adaptable both to a first reading, providing a global overview of the subject, and to subsequent readings searching for specific information.

OpenMP RESOURCES

Given the OpenMP status—a widely adopted programming standard—there are naturally a huge amount of bibliographic references, in particular articles and tutorials on the Internet that can be easily consulted. We make below a few comments on this subject:

- The classical OpenMP reference is the book by Barbara Chapman, Gabrielle Yost, and Ruud van der Paas, *"Using OpenMP"* [26], which, given its high pedagogical standards, remains a very useful tool for programmers.
- The official OpenMP Architecture Review Board (ARB) site [27] has a wealth of information, in particular:
  - The official OpenMP Application Programming Interface [28]
  - The official OpenMP4.0 examples document [29]
  - Pointers to articles and tutorials, at the /resources link
- Among the tutorials, I particularly appreciate the SC14 tutorial "Advanced OpenMP: Performance and OpenMP4.0 Features," by B.R. de Supisnki et al. [30]. The material proposed in this tutorial provides additional insight and examples on many of the subjects discussed in this chapter. A useful OpenMP SC14 video can be found in [31].

10.1 BASIC EXECUTION MODEL

The basic OpenMP execution model, in its simplest form, was already introduced in Chapter 3. OpenMP implements a *fork-join* execution model in which at some point of its lifetime a thread meets a *parallel directive* that activates a team of N threads to perform some parallel treatment (how the team size N is fixed is discussed later on). This model is a *thread-centric* model, in the sense that, as shown in Figure 10.1(a), each active thread in the team has a well-defined task assignment when a parallel region is entered. Barring a very mild exception to this statement that will be discussed later on, it is fair to say that the programmer knows at any time what every thread in the worker team is doing. This corresponds to the traditional view implemented in the basic multithreading libraries, in which a thread is identified with the task it runs.

OpenMP creates a unique task pool for the whole application, but this thread pool is dynamic in the sense that the number of worker threads in the OpenMP pool *is not fixed; it can be enlarged dynamically*. If more parallelism is needed, more threads are added to the pool. Indeed, OpenMP supports nested parallel regions. As indicated in Figure 10.1(b), a thread in a parallel team can in

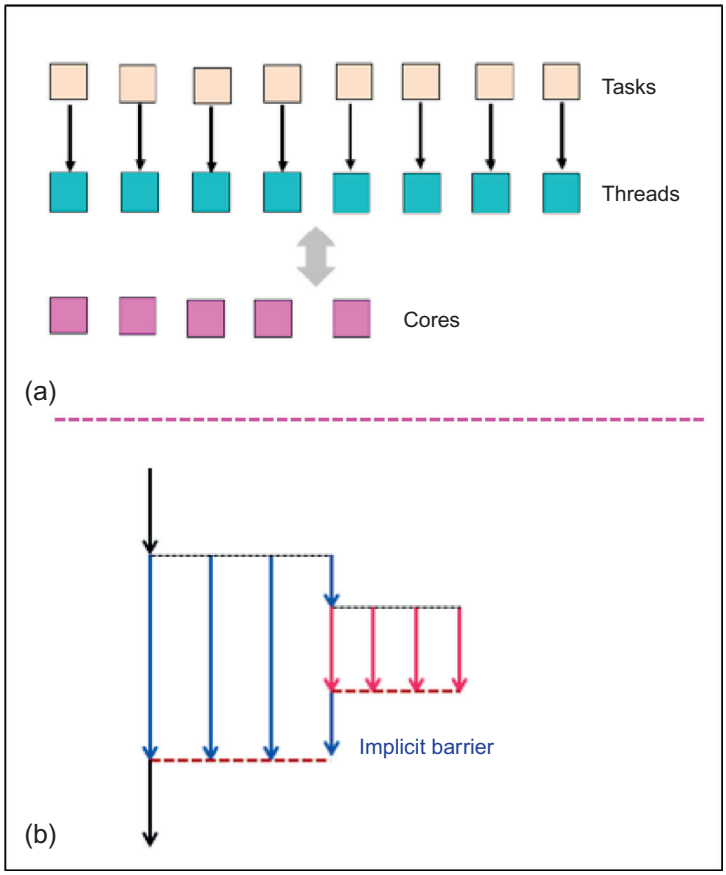

FIGURE 10.1

OpenMP execution model.

turn create a new parallel region and become the master thread of the new worker team. When this happens, OpenMP incorporates the new threads required to run it in such a way that each active implicit task is mapped to a thread. The encountering thread can fix the number of threads required by setting the number of threads for the next parallel regions. Then the execution of the nested parallel region proceeds as follows:

- The current task being executed by the black thread is suspended.
- (N-1) new threads are activated. Whether the new threads are created on demand or simply activated from a dock of previously created sleeping threads is an implementation-dependent issue.
- Including the initial thread triggering the nested parallel region, *which becomes the master thread of the new team*, the N worker threads start immediate execution of a set of *implicit tasks* that have been prepared by the programmer, in the code block that follows the parallel directive.

- Threads in the team have ranks in [0, N-1] that can be obtained by an OpenMP library function call. Rank 0 corresponds to the master thread. As usual, the thread rank (which is called the *thread number* in OpenMP) can be used to select the particular job to be done by the thread, using conditional statements.
- When the nested parallel region completes its treatment, the master thread resumes the execution of the interrupted task.

In spite of the fact that one task in the external parallel region was suspended when the inner parallel region was encountered, the mapping of implicit tasks to threads is very precise. We know at any time what every thread in the team is doing. Do not be confused by the *implicit* qualification given above to the tasks executed at the start of a parallel region; this simply means they are completely defined when the parallel region is encountered.

OpenMP 3.0 introduced a new task directive, used to dynamically incorporate an additional workload in an already running parallel region, with a fixed numbers of worker threads. Tasks created in this way are called *explicit tasks*. There is no conceptual difference between explicit and implicit tasks; they are in all cases units of sequential computation assigned to the worker threads. But the addition of explicit tasks is a major extension of the basic execution model that requires careful discussion, because in this case potential parallelism (more concurrent tasks) is incorporated in the application without increasing the number of threads in the pool. Section 10.6 discusses at length the way this new, extended execution model is interfaced with the basic historical OpenMP execution model described above.

Notice that there is no notion of "job submission" in OpenMP. The master thread that encounters a parallel region suspends whatever it is doing and joins the new worker team to contribute to the execution of the new predefined tasks. Rather than a "job submission," there is a temporary transition to a new parallel context. At the end of the parallel region, an implicit barrier synchronization operates, and then the master thread resumes the execution of the initially suspended task.

Notice also that the thread number (the rank of a thread) depends on the context. The thread that creates the nested parallel region in Figure 10.1(b) is not the master thread, so its thread number is initially different from 0. However, inside the nested parallel region, this thread becomes the master thread of the new inner team and acquires the thread number 0. This will be shown explicitly in one of the examples in Section 10.5.

The basic, traditional OpenMP execution model is largely sufficient for regular problems in which programmers can easily grasp the way of distributing the parallel workload among the worker threads. However, this execution model is not optimal for addressing irregular, unstructured, or recursive problems for reasons that will be discussed in Section 10.6. The task extension started in the 3.0 release and vigorously continued in the last 4.0 release is motivated by this fact.

10.2 CONFIGURING OpenMP

OpenMP is a very flexible programming environment. It relies on a set of internal control variables (ICVs) to configure the behavior of a program. ICVs are *conceptual variables*. Indeed, the OpenMP standard fixes the way OpenMP behaves, not the way it is implemented. It establishes that OpenMP

must behave *as if* its behavior was determined by the values of a set of configuration variables with given names. But this does not necessarily mean that they exist as such in each specific implementation of the standard.

As programmers, we are concerned by the logical behavior of OpenMP, not by its implementation. ICVs must therefore be used as they are described in the standard. ICVs store information such as the number of threads to activate in future parallel regions, whether nested parallelism are enabled or not, the size of the threads stack, and so on. Their values can be queried with runtime library functions, and set with environment variables, runtime library functions, and sometimes with directive clauses.

Different ICVs can have different scope:

- *Global scope*: they are fixed once and for all and apply to the whole program.
- *Device scope*: OpenMP 4.0 introduces the possibility of deploying an application on heterogeneous platforms mixing different kinds of computing architectures. The *device* concept is introduced to distinguish standard CPUs, co-processors, or GPU accelerators. It is obvious that teams of worker threads running on different devices should be allowed to have different attributes. Therefore, many previously global ICVs now have device scope: there can be one copy per device. In addition, a new ICV called default_device_var has been introduced, to identify the device on which the current code is executed by default.
- *Task scope*: Finally, most ICVs—like the number of threads to be activated the future parallel sections—have a local *task scope*. This means that each OpenMP task keeps its own value of these ICVs. When new tasks are created at parallel and task directives, they inherit the ICV values of the parent task. When a task modifies some ICVs, the modification is valid during its lifetime, but ancestor values are not modified. In this way, dynamic modifications of the OpenMP operational environment have a hierarchical structure: they do not propagate upward in the hierarchy of nested parallel regions or recursive task creations.

Initial values of ICVs can be set with environment variables. These initial values can in some cases be modified at runtime with library function calls or with directive clauses. In order to offer an overall view of the configuration options available in OpenMP, we list in Table 10.1 all the available ICVs, including the new ones introduced by OpenMP 4.0). This table lists the scope of each ICV, the name of the associated environment variable, and the capability offered to the programmer for retrieving its value or for modifying it at runtime. The get qualifier in the function call column means there is a library function of the form omp_get_xxxxx() that retrieves its value. The set qualifier means there is a library function of the form omp_set_xxxxx() that can be called to override previous values.

There are three types of actions performed by IVCs: controlling some global aspects of the program execution, controlling the behavior of parallel regions, and controlling the way the directive for automatic parallelization of loops shares the loop workload across the different worker threads. A qualitative discussion of their role is proposed next. *The OpenMP Reference document has a very precise description of every environment variable, as well as the library accessory functions* [28].

Controlling program execution

- *stack-size-var*: Controls the stack size for the OpenMP threads. Note that there is no way of knowing or modifying the implementation defined default stack size at runtime. In the case of stack overflow error, the program must be run with a hopefully bigger stack size by setting OMP_STACKSIZE to an integer equal to the requested size in bytes.

Table 10.1 Internal Control Variables

ICV	Scope	Environment	fct Calls
dyn-var	Task	OMP_DYNAMIC	Get, set
nest-var	Task	OMP_NESTED	Get, set
nthreads-var	Task	OMP_NUM_THREADS	Get, set
run-sched-var	Device	OMP_SCHEDULE	Get, set
def-sched-var	Device	None	None
stacksize-var	Device	OMP_STACKSIZE	None
wait-policy-var	Device	OMP_WAIT_POLICY	None
thread-limit-var	Task	OMP_THREAD_LIMIT	Get
max-active-var	Device	OMP_MAX_ACTIVE_LEVELS	Get, set
active-levels-var	Task	None	Get
bind-var	Task	OMP_PROC_BIND	Get
place-partition-var	Task	OMP_PLACES	None
cancel-var	Global	OMP_CANCELLATION	Get
default-device-var	Task	OMP_DEFAULT_DEVICE	Get, set

Note: the last four ICVs listed above are specific to OpenMP 4.0

- *wait-policy-var*: Controls the preferred behavior of waiting threads: spin or idle wait. The default value is implementation dependent. Possible values for OMP_WAIT_POLICY are active for spin wait and passive for idle wait.
- *thread-limit-var*: Defines the maximum number of threads that can be activated in an OpenMP program. Its initial default value is implementation dependent. The value of OMP_THREAD_LIMIT must be equal to a positive integer. The behavior of the program is implementation dependent if this value is bigger than the number of threads the implementation can support.
- *cancel-var*: Enables the cancellation of parallel regions or other parallel constructs. The values of OMP_CANCELLATION must be true (cancellation enabled) or false (cancellation disabled). Cancellation is disabled by default. A full discussion of this issue is given in Section 10.9.
- *bind-var*: Controls the binding of threads to places (cores). When binding is requested, the runtime system is asked to bind each thread to a specific core. Values of OMP_PROC_BIND can be false (binding not requested), true (binding requested), or a string specifying the binding policy. All these details are discussed in Section 10.11 on thread affinity.
- *place_partition_var*: OpenMP 4.0 introduced a way of describing the places (cores, sockets) where the threads can be allocated and run. This ICV describes the places available for allocating threads, in order to implement thread affinity. Section 10.11 offers a detailed discussion of the different ways of setting the associated OMP_PLACES environment variable.
- *default-device-var*: Defines the default target device, where code is executed by default. Its value is set by the OMP_DEFAULT_DEVICE environment variable, and modified by a call to the omp_set_default_device() function at runtime.

Controlling parallel regions

- *nthreads-var*: Fixes the number of threads requested for the next encountered parallel region. Its default initial value is implementation dependent. An initial value is specified by setting OMP_NUM_THREADS. Values can be retrieved at runtime by calling omp_get_num_threads(), or modified by calling omp_set_num_threads(int n).
- *nest-var*: Controls whether nested parallelism is enabled for encountered parallel regions. Its values can be false (0) in which case OpenMP ignores the nested parallel directive and produces sequential code, or true (1). Its default initial value is implementation dependent. Values assigned to OMP_NESTED are true of false. Values retrieved by omp_get_nested()—or passed to omp_set_nested()—are 0 or 1.
- *max-active-levels-var*: Controls how deep one can go in nesting parallel regions. The initial default value depends on the implementation.
- *active-levels-var*: The only role of this variable is to retrieve, at runtime, by a call to the omp_get_active_level() function, an integer that informs how deep the executing code is placed in the nested parallel sections hierarchy.
- *dyn-var*: Controls whether the dynamic adjustment of the number of threads is enabled for encountered parallel regions. The comments that follow clarify the relevance of this ICV.

Note that, in many implementations, nested parallel regions are by default disabled, so the first thing to do in a program with nested parallelism is to call omp_set_nested(1) to enable this functionality. Otherwise, OpenMP will ignore nested parallel regions: only the encountering (master) thread is allocated to the nested parallel region, and this is the only thread that executes the related tasks. In other words, we end up with sequential code.

Setting the number of threads for the next parallel section when nested parallelism is enabled is also very helpful. When encountering a nested parallel regions, OpenMP adds the number of threads needed to comply with the new request. The number of threads in the pool is in this way adapted to the application requirements, as long as it does not exceed the maximum number of threads authorized to participate in the OpenMP program, determined by the thread-limit ICV. Or, as long as it is authorized by the nesting level control, which we discuss next.

Setting the global ICV, called max-active-levels-var, is also important in programs with substantial nested parallelism. If the level of nested parallelism goes beyond this limit, OpenMP again ignores the parallel directives and generates sequential code. The existence of this limit is to a large extent required by OpenMP execution model we just described. Having lots of threads in a parallel region, each one encountering a nested parallel region, each one encountering in turn a nested parallel region, and so on, may produce an explosive growth of the number of threads in the applications, leading to an overwhelming over-subscription of CPU and memory resources (remember, each time a new thread is created a new stack memory buffer is allocated). This ICV is therefore a safe-net preventing this from happening.

Next, we come to the meaning of the *dynamic adjustment of threads* ICV. As the number of threads in the pool increases, it may happen that, when the number of threads requested for the next parallel region is added to the current team, the maximum number of threads in the application is exceeded. If this ICV is enabled, OpenMP allocates fewer threads than requested, to fit the thread limit. This is the dynamical adjustment of the number of threads. If this ICV is disabled, OpenMP ignores the parallel directive and generates sequential code. In most implementations, the default thread limit is enormous

(several millions for the Linux GNU compiler). Therefore, unless one is dealing with a very peculiar application, this ICV can safely be maintained at its default value (false).

Controlling the automatic parallelization of loops

OpenMP has a work-sharing parallel for directive that acts on the for loop that follows by sharing the execution of the loop iterations across the worker threads in the parallel region. There are several ways of scheduling the loop workload among the worker threads, which are selected by the value attributed to the *run-sched-var* ICV. The *def-sched-var* ICV determines the scheduling policy to be used by default, in the absence of an explicit choice. These issues are reviewed in more detail in Section 10.3.4, when discussing the work-sharing directives.

10.3 THREAD MANAGEMENT AND WORK-SHARING DIRECTIVES

OpenMP is today a very rich programming environment. The parallel behavior of an application is created and structured with directives. There are three kinds of basic, traditional directives: thread management and work sharing—directly related to the activation and scheduling of parallel jobs—and directives that synchronize the worker threads activity. OpenMP 4.0 has incorporated a number of additional directives for task synchronizations, vectorization, cancellation of parallel constructs, and code offloading to external devices. We will first concentrate on the basic directives used to activate and structure a parallel treatment, including their OpenMP 4.0 extensions. New features introduced by OpenMP 4.0 are discussed at the end of the chapter.

In C-C++, the OpenMP directives take the following form:

```
...
#pragma omp directive (optional clauses)
   {
   // code block (present in some directives)
   }
...
```

LISTING 10.1

General form of an OpenMP directive

Directives apply to the code block that follows. Note, however, that most synchronization directives—like the barrier or taskwait directives—induce a local synchronization action, and do not act on a code block. The optional clauses—that apply mainly to thread management and work-sharing directives—provide the compiler with additional information about the way the required parallel treatment should be organized. These optional clauses are of course needed to make the programmer intentions clear. Otherwise, they can be ignored.

10.3.1 PARALLEL DIRECTIVE

The parallel directive, which creates a team of N worker threads, is the fundamental parallel construct in OpenMP. Any other management, work-sharing, or synchronization directive operates inside the parallel context so established. The parallel directive together with the code block that follows will be referred to as a *parallel construct*.

The parallel directive operates in the way described in detail in Section 10.1, summarized as follows:

#pragma omp parallel

- Constructs a parallel region with N worker threads.
- Drives a single program, multiple data (SPMD)-like job: the N threads execute the same task function.
- There are therefore as many identical implicit tasks as threads in the team.
- The common task function is constructed by the compiler from the code block that follows.
- There is an implicit barrier at the end of the code block.
- Several clauses are accepted.

There are two types of clauses:

1. *Functional clauses* that define some details of the operation of the directive.
2. *Data-sharing attributes clauses* that are required to clarify the scope of data items, and help the compiler construct the task function to be executed by the worker threads.

The parallel code block contains the input needed by the compiler to construct the task function to be executed by each worker thread. OpenMP is a very flexible programming environment, supporting a large variety of programming styles as long as the programmer's intentions are clear to the compiler. The parallel code block can be a set of explicit statements, including possible calls to auxiliary functions, taken directly from the sequential version of the code. In this case, the directive has to be completed with clauses—called data-sharing clauses—that make it very clear to the compiler the scope of the different variables in the code, and enables the construction of an unambiguous task function.

```
...
int n, m;
float a, b;
#pragma omp parallel (optional clauses)
   {
   // the variables n, m, a, b are referenced in this
   // code block
   }

...
```

LISTING 10.2

General structure of a parallel directive code block

Remember that one of the initial objectives of OpenMP enables the parallelization of sequential codes by inserting directives. Therefore, the scope of data items when the transition from sequential to parallel execution (and vice versa) takes place, often needs to be clarified. Variables declared as global from the start are shared by all threads no matter how many threads and parallel regions are introduced in the code. But what happens to the variables that are local to the thread encountering the parallel region, like the variables `n, m, a, b` in Listing 10.2 above? Should they become shared by all the threads in the new parallel region, or should they be made private, each thread in the team having its own local copy? And, if each thread has its own copy, how should their initial values be captured? These are the issues resolved by the data-sharing clauses. The detailed description of the data-sharing clauses is

postponed to Section 10.3.6, because they are common to other management or work-sharing directives not yet introduced.

Functional parallel clauses:

Here is the list of parallel directive clauses not related to data-sharing attributes:

- if(*scalar-expression*): This clause is evaluated before the creation of the parallel region. If the scalar expression evaluates to true the parallel region is created. If it evaluates to false, the directive is ignored and the compiler produces sequential code.
- num_threads(*integer-expression*): Again, this clause is evaluated before the parallel region is created. The number of threads activated when a parallel directive is encountered is fixed by the OMP_NUM_THREADS environment variable, or by a call to the omp_set_num_threads(N) library function—which overrides the environment variable value if it exists. If this clause is used, it overrides both the environment variable and the function call values if they exist.
- proc_bind(master | close | spread): Specifies the mapping of OpenMP threads to places (cores), within the places listed in the *place-partition-var* ICV. This is a new OpenMP 4.0 feature, discussed in Section 10.11 on thread affinity.

10.3.2 MASTER AND SINGLE DIRECTIVES

The master and single directives—to be used inside a parallel construct—encapsulate code to be executed by only one worker thread. Considered as synchronization directives by OpenMP, they are very useful in structuring OpenMP code. They are introduced when, in the middle of a parallel region, some specific computational activity must be performed by only one thread, like increasing a global counter, opening a file, or initializing a global variable. Here is a description of their mode of operation, shown in Figure 10.2:

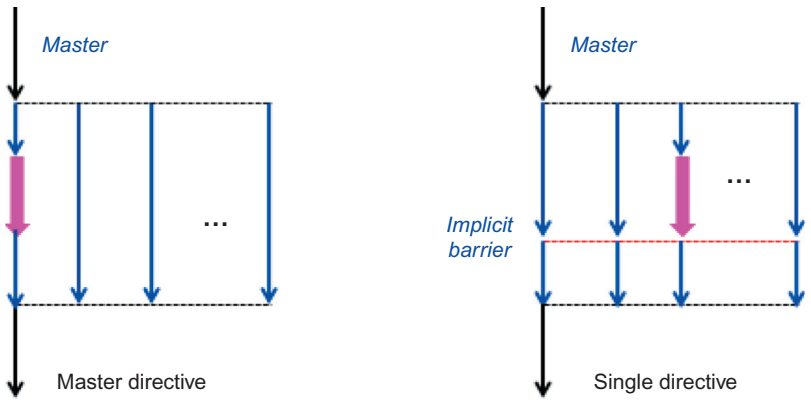

FIGURE 10.2

Operation of the master and single directives.

#pragma omp master
- *To be used inside the a parallel construct.*
- The code block that follows is only executed by the master thread.
- There is no implicit barrier at the end of the master code block.
- The remaining worker do not wait for the termination of the code block.
- No clauses required.

#pragma omp single
- *To be used inside a parallel construct.*
- The code block that follows is only executed by one of the threads of the team.
- There is an implicit barrier at the end of the single code block.
- The remaining worker threads wait for the termination of the code block.
- The implicit barrier can be overridden with the nowait clause.

The listing below shows a program where the main() thread creates a parallel section, and master and a single directives are encountered inside the parallel code block.

- When the master directive is encountered, the code block that follows is executed only by the master thread, and there is no implicit barrier at the end. If the other threads need to wait for the master thread, an explicit barrier directive is required.
- When the single directive is encountered, the code block that follows is executed by only one thread, but now there is an implicit barrier at the end that forces all other worker threads to wait for the completion of the single code block. This barrier can be disabled with the nowait clause in the directive.
- Waiting or not waiting for the sequential code block depends on the context. If all one needs is to have one of the threads of the team write a report message to stdout, for example, then the remaining threads do not need to wait. If, instead, a file is opened and the remaining threads will access it, then the implicit or explicit barrier synchronization is needed.

```
int main()
  {
  ...
  #pragma omp parallel [clause] ... [clause]
      {
      // –  implicit tasks code block – ––
      // all threads are active here
      ...
      #pragma omp master
         {
         // executed by master
         }
      // all threads active again
      ...
      #pragma omp single (nowait?)
         {
         // executed by single thread
         ..
         } // implicit barrier here
```

```
        // all threads active again
        ...
        }
    // task encountering the parallel directive resumes here
    ...
    }
```

LISTING 10.3

Parallel region encountering master and single directives

Many of the examples that follow later in the chapter show the relevance of these directives. The single directive will be repeatedly used in Section 10.7, dealing with the task API examples, in order to implement a parallel environment in which a team of N worker threads is created by a parallel directive, but only one thread is initially activated to perform a task. The remaining threads are activated when further explicit tasks are created and submitted for execution by this initial implicit task. This pattern is shown in the listing below:

```
int main()
  {
  ...
  #pragma omp parallel [clause] ... [clause]
       {
       #pragma omp single
          {
          // this task will create further tasks, which in
          // turn may create further tasks...
          }
       }
  ...
  }
```

10.3.3 SECTIONS AND SECTION WORK-SHARING DIRECTIVES

It was previously stated that in the basic OpenMP execution model—being a task-centric environment—there is a one-to-one mapping between implicit tasks and thread, *barring one minor exception*. These directives implement the exception we referred to. They are used, once a parallel region has been established, to establish a work-sharing pattern different from the default one (one unique task function, with all workers executing the same task). Inserted as subdirectives inside a parallel construct, they specify one-by-one a set of different individual tasks to be executed by the worker threads.

#pragma omp sections

- *To be used inside a parallel construct.*
- The code block that follows provides a precise list of individual tasks for execution.
- Each task is defined by the *section* directive that follows.
- The number of tasks may be different from the number of threads.
- This mechanism can be seen as the submission of a number of individual tasks to a pool of N worker threads.
- Several clauses are accepted.

- *#pragma omp section*
  - To be used in the code block of a sections directive.
  - The code block that follows is an individual task to be executed by a thread in the team.
  - No clauses required.

The sections directive corresponds therefore to a context in which N different tasks are submitted for execution to the pool. If this is all that happens in the enclosing parallel region, the sections directive can be merged with the enclosing parallel directive, as shown in Listing 10.4. Tasks are defined by N successive section directives in the code block, also shown in the listing below. The number of worker threads in the team is chosen in the usual way *and is in general different from the number of submitted tasks*.

```
...
...
#pragma omp parallel sections (optional clauses)
   {
   #pragma omp section
      {  task1(); }
   #pragma omp section
      {  task2(); }
   ...
   #pragma omp section
      {  taskN(); }
   }                 // implicit barrier here
...
```

LISTING 10.4

General form of the parallel sections directive

OpenMP always activates the requested number of threads Nth. If N>Nth, some tasks will be postponed, as in any ordinary thread pool. If, instead, N<Nth, some threads will do nothing initially. But they can be activated later on to execute new tasks dynamically created with the task directive, discussed in Section 10.6. Explicit examples will be proposed later on. Finally, there is always an implicit barrier at the end of the parallel sections code block, so that the master thread exits the parallel section when all submitted tasks have been executed.

As in the previous cases, the code blocks following the section directive that define each individual task can consist of a task function like in the listing above, or a list of instructions and arbitrary function calls. In this case, *keep in mind that the clauses used to guide the compiler are global to all tasks* and must be introduced at the top level of the parallel sections directive, since the inner section directives do not admit clauses. In other words: data-sharing instructions provided to the compiler must be common to all tasks. Obviously, when lots of tasks doing very different things need to be run, this programming style can become error prone and confusing. For this particular construct, it seems more adequate to encapsulate the different task functions, avoiding clauses as much as possible. This is what is done in the examples that follow later on.

10.3.4 FOR WORK SHARING DIRECTIVE

This directive requests the automatic parallelization of the (nested, in general) loops that follows. The work distribution across threads is performed by OpenMP. This directive is well adapted to a microtasking programming style, where the code remains mainly in sequential mode and loops are parallelized as they are encountered. When the parallel construct is created with the only intention of parallelizing a loop, and contains no other statement, the parallel for shortcut directive can be used before the loop.

#pragma omp for

- The work performed by the for loops that follows is distributed among the worker threads.
- One specific schedule clause provides a fine control on the way the loop range is split into chunks and distributed across threads.
- Default is equal size chunks, one chunk per worker thread in the team.
- There is an implicit barrier at the end of the loop region unless the nowait clause is specified.

Besides the data-sharing clauses discussed in the next section, this directive admits a few functional directives:

- schedule specifies how iterations of the associated loops are grouped into contiguous nonempty subsets, called chunks, and how these chunks are distributed among the worker threads.
  - schedule(static, *chunk-size*): Iterations are divided into chunks of size *chunk-size*, and chunks are assigned to the worker threads in a round-robin fashion in the order of the thread number.
  - schedule(dynamic, *chunk-size*): Chunks are assigned to worker threads as they request them. Each thread executes a chunk and then requests another one, until no chunks are left.
  - schedule(guided, *chunk-size*): Similar to dynamic, with a minor difference in the way chunks are prepared. See the OpenMP documentation [28].
  - schedule(auto): The scheduling decision is relegated to the compiler and runtime system.
  - schedule(runtime): The scheduling decision is deferred until runtime, and the schedule policy and chunk size are taken from the *run-sched-var* ICV.
- collapse(n) indicates to the compiler that *loop fusion* must be performed. The n nested for loops that follow the directive are merged in a larger, unique iteration space, which is next distributed among the worker threads following the policy selected by the schedule clause. The order of iterations in the merged iteration space is determined by the natural order implied by the sequential execution of the nested loops.

An example of the operation of the different schedule strategies is proposed in Section 10.6.1, when comparing the performance of the parallel for construct against the task construct in the context of an irregular problem. Examples of the schedule and collapse clauses are found in the applications discussed in Chapters 13–15.

10.3.5 TASK DIRECTIVE

Once the number of active threads in a parallel region has been launched and the parallel job starts execution, OpenMP can depart from the initial *one task per thread* mode of operation, and implement

some way of queuing tasks and scheduling their execution. We have already seen that the number of tasks defined under a parallel sections directive may be bigger than the number of active worker threads in the team. On the other hand, the task directive enables the *on-the-fly creation of new, explicit tasks inside a parallel region*, to be executed by the current worker team. Even in an SPMD context where there are initially as many tasks as active threads, the number of tasks can therefore be dynamically increased by the task directive.

The task directive, introduced by OpenMP 3.0, constitutes a major evolution of OpenMP programming model. We postpone to Section 10.6 the detailed discussion of the task programming interfaces, as well as the motivations for their adoption in the OpenMP programming model.

10.3.6 DATA-SHARING ATTRIBUTES CLAUSES

In the simple examples proposed in previous chapters, a parallel focus was adopted from the start, by declaring all variables with the correct scope (global or local), and explicitly writing the task function that encapsulates the parallel block code. In this case, data-sharing clauses were not needed, because the task function was already unambiguously constructed. In OpenMP, the signature of the task function is open, as long as the compiler understands the programmer intentions: it is possible to pass data values, and even recover return values from it, as the examples that follow will show.

Table 10.2 lists all optional accompanying clauses that come with OpenMP thread management directives that require them: parallel, parallel sections, for, task, and single, as well as the new OpenMP 4.0 directives simd and teams. The clauses listed in blue are the so-called data-sharing attribute directives, needed to explain the compiler the role and fate of different data items when the task code is not totally encapsulated in a task function. The examples that follow show how they operate.

The necessity of the data-sharing attribute clauses can be easily understood. Variables declared outside main are unambiguously shared, and seen by all threads. Variables declared *inside* a code block are unambiguously local variables. The point is how to interpret variables that are referenced both inside and outside a parallel code block, in the enclosing environment corresponding to the task that triggers the parallel region. Should they be shared by all parallel tasks, or should each task have its own local copy? In addition, when entering a parallel region, we may want local variables to be initialized

Table 10.2 Directive Clauses

	Parallel	Sections	Task	Single	for	simd	Teams
private	X	X	X	X	X	X	X
shared	X		X				X
default	X		X				X
firstprivate	X	X	X	X	X		X
lastprivate		X			X	X	
copyin	X						
copyprivate				X			
reduction	X	X			X	X	X
nowait		X		X	X		

Notes: The first five clauses listed above are data sharing attributes. Notice also that "sind" and "teams" are new OpenMP 4.0 constructs.

with their previous immediate value in the enclosing environment, and when exiting the parallel region we may want to recover the last value they received inside some specific parallel construct (sections or loop regions). Then, the following clauses are available:

- *private* (list of variables): Declares that the variables in the list are private to each task. They will be implemented as local variables to each task function.
- *shared* (list of variables): Declares that the variables in the list are shared by all tasks. They will behave as global variables for the duration of the construct. But this does not means they are reallocated as global variables: *as explained below, this is a mechanism for passing data by reference* from the parent task encountering the parallel construct to the tasks executed by the worker threads.
- *default* (shared/none): Determines the default data-sharing attribute of variables referenced in a parallel or task construct.
- *firstprivate* (list of variables): Declares that the variables in the list are private to each task, and initializes their value inside the tasks with the value of the original data item when the parallel construct is encountered. *This is a mechanism for passing data by value* from the parent task encountering the parallel construct to the tasks executed by the worker threads.
- *lastprivate* (list of variables): Declares that the variables in the list are private, and causes the corresponding original list items to be updated *after the end of the* sections *or* for *constructs* with its value in the lexically last section construct, or its value in the sequentially last iteration of the associated loop. This directive is only relevant for work-sharing constructs. See the OpenMP documentation for further details.
- *copyin* (list of variables): Copies the values of the master thread's private variables in the list, to the corresponding private variables of all the other members of the team. This copy is done after the team is formed, but prior to the start of execution of the parallel construct.
- *copyprivate* (list of variables): This clause only applies to the single directive. It provides a mechanism for broadcasting a result from the thread executing the single implicit task to the other implicit tasks in the parallel region. This occurs after the execution of the block associated to the single construct, but before any of the threads in the team have left the implicit barrier. In this way, all the remaining tasks capture a result computed by the single implicit task.
- *reduction* (reduction-symbol: list): The *reduction-symbol* corresponds to an associative operation. Each variable in the list is treated as firstprivate: local instances are initialized with its value before the parallel construct. On exit from the construct, its original value is replaced by the result of the reduction operation applied to all local instances. We have often used reductions based on the addition operation to collect partial results produced by the worker threads.

Default Scope

Default scope for variables not referenced in data-sharing clauses. For the parallel directive, they are shared by default in C-C++. For the task directive, they are firstprivate by default in all programming languages.

Comments on data-sharing clauses

Consider first the shared clause: as stated above variables tagged as shared *behave as global variables* for the duration of the construct. It looks as if OpenMP was promoting a private local variable in the

parent task to a global status. In fact, this is not really needed, because shared does not necessarily means *global*: shared only means that all worker threads have been granted access the original local variable sitting in the parent thread stack. For this to happen, it is sufficient that the parent thread publishes its local data by passing to other threads a pointer to it. Therefore, when a shared variable occurs in a task code block, the compiler is really implementing a pointer or a reference to the parent task local data. This explains a statement often found in the literature: local data items in the parent task tagged as firstprivate are captured by value, and those tagged shared are captured by reference.

Consider next the reduction clause. Listing 10.5 shows how it operates. Let us assume that partial results computed by the worker threads need to be accumulated. This was often done this in the past by asking the worker threads to accumulate their partial results in a mutex-protected global variable. OpenMP can perform the required reduction directly, as shown.

```
...
int n, m;
float a, b;

a = 0.0;   // initialize a
#pragma omp parallel reduction(+:a)
   {
   ...
   // each implicit task computes its value of a
   ...
   }

// Here, a is the sum of all the partial results computed
// by the implicit tasks
...
```

LISTING 10.5

Reduction operation

- Before the parallel section, a is a local variable in the main function or in the encountering thread. This variable is initialized before the parallel region is entered.
- The a variable declared in the reduction clause is treated as firstprivate: each worker thread gets an initialized copy.
- Before exiting the parallel section, the values of these private copies are collected together by the reduction operation indicated in the clause, and the result is affected by the initial local main variable a.
- After the parallel section, a contains the result of the reduction operation performed by the worker threads.

OpenMP has a reduction operation implemented for the most relevant arithmetic and logical associative operations. OpenMP 4.0 has introduced the possibility of incorporating user-defined reductions. For more details on this feature, look at the declare reduction clause in the OpenMP documentation [28].

10.4 SYNCHRONIZATION DIRECTIVES

We already covered the master and single directives, considered by OpenMP as synchronization directives, which is not unreasonable: they establish a privileged role for one of the threads in the team, and in addition the single directive adds a barrier synchronization (see below) at the end of its code block.

OpenMP proposes two directives for mutual exclusion:

#pragma omp critical name

- Implements mutual exclusion in the code block that follows.
- Critical sections can be named.
- Critical sections with the same name share the same mutex.
- All unnamed critical sections share also a unique *no name* mutex.

#pragma omp atomic

- The instruction that follows must be executed atomically.
- Mainly used to atomically update a variable.
- See Chapter 5 for a detailed description of this directive.

The critical directive is discussed below, in order to show how it relates to the mutual exclusion interfaces based on library function calls, discussed in Chapter 6.

#pragma omp barrier

- Establishes a barrier synchronization point.
- The barrier engages all the worker threads in the current parallel section.

This is probably the most widely used synchronization directive in OpenMP. It is important to keep always in mind that *this directive synchronizes all the threads of the worker team*. Remember that the barrier interfaces proposed by the basic and vath libraries require an initialization specifying the number of threads expected at the synchronization point, which may be different from the number of worker threads in the pool. This is not the case in OpenMP. We will come back to this point when discussing in Section 10.6 the three-task synchronization directives that follow. Their relation to the barrier synchronization mechanism is original and subtle, and requires careful explanation.

#pragma omp taskwait

- To be used inside a task code block.
- Implements task synchronization.
- The current task waits for completion of all its direct descendant child tasks.

#pragma omp taskyield

- To be used inside a task code block.
- Establishes a user-defined task synchronization point.
- This means that, at this point, the current task can be suspended to allow the executing thread service another task.
- This directive helps the runtime system to optimize task scheduling.

#pragma omp taskgroup (OpenMP 4.0)

- To be used inside a task code block.
- New parallel construct, implementing task synchronization.

- Directive is followed by a code block.
- The current task waits at the end of the code block for completion of all its descendant tasks: (children, children of children, etc.).

Task synchronizations are discussed in full detail in Section 10.6, devoted to the task API. Finally, there are two miscellaneous directives. The first one is useful, for example, when all the worker threads are asynchronously writing to stdout, and we want to order the output. The second one re-establishes memory consistency in specific contexts.

#pragma omp ordered

- Orders the execution of code block inside a loop region.
- The ordering is established according to the loop iterations.
- The thread executing the first iterations enters the ordered block without waiting.
- Threads executing subsequent iterations wait until all ordered blocks of previous iterations are completed.

#pragma omp flush(list)

- Restored memory consistency for all the variables in the list.
- Very low level directive, to be used with extreme care.
- There are restrictions on its placement. See the OpenMP documentation.

There is indeed in OpenMP a rather limited number of high-level synchronization directives. OpenMP does not offer any other high-level synchronization tools of the kind discussed in Chapter 9. This is quite understandable, given the initial motivation for OpenMP: the directive-based approach to the parallelization of existing sequential codes. When directives are inserted in a sequential code to make it multithreaded, it is not possible to pinpoint individual threads to establish synchronization patterns among them. The best we can do is a collective synchronization (barrier directive) or isolate individual threads (master and single directives). However, this synchronization context is sufficient to cope with any concurrency pattern in the traditional OpenMP context not involving explicit tasks. In any case, we have already insisted on the fact that the supplementary synchronization utilities discussed in Chapter 9 can also be used in an OpenMP environment.

10.4.1 CRITICAL DIRECTIVE

All the OpenMP run-time library functions managing mutual exclusion were presented in Chapter 6. This API is as rich and complete as those proposed in other programming environments, including ordinary locks, recursive locks, and the *testlock* functions, which try to lock a mutex and return if the mutex is not immediately available. This section examines the alternative, directive-based tools for mutual exclusion.

The purpose of the critical directive is to implement mutual exclusion in a code block. Does this mean we can forget about the lock routines API? The answer is clearly no if a refined implementation of mutual exclusion involving recursive locks or the testlock functions is required. The critical directive is at best a substitute to ordinary mutex locks. Understanding its relation with mutex locks is important, in order to make the correct strategic decisions in developing applications.

Critical sections can be named

The critical directive can be followed by an arbitrary name. *This name names a specific hidden mutex*. All critical directives with the same name in the code, lock the same hidden mutex. This means there is contention among them. Therefore, giving different names to different critical sections is a way of reducing the contention among different blocks of code for which mutual exclusion is not required, by locking different hidden mutexes.

The critical directive can also be used *as such*, with no name. In this case, OpenMP allocates a mutex to the *no name* critical construct, and again, it is always the same mutex that will be locked, so that there is contention among all the *no name* critical constructs. Programmers must keep these observations in mind in order to avoid introducing unnecessary mutual exclusion contention in a code.

Named critical sections versus mutex locking

Since a named critical section is equivalent to locking a specific mutex, can we conclude that named critical sections can replace mutex locking in all cases? The answer is no, as the following example shows. Imagine that, in a C++ class implementing some service, thread safety requires exclusive access by client code to some internal private variables in the class. Two options are available:

- A mutex is explicitly declared as a class member and used to guard the access to the critical variables, members of the class.
- Rather than declaring a class member mutex, member functions access the critical variables inside *a named critical section*.

This looks the same, but it is not. Imagine that the client code instantiates 15 objects of this class. In the first case, we end up with 15 different mutexes, and there is no contention among them. Threads cannot access simultaneously *the same object*, but they can freely access simultaneously different objects, which is what we want. In the second case, there is a unique mutex—linked to the critical section name—that protects all 15 objects, which creates unnecessary contention, and this is most likely not what we want.

10.5 EXAMPLES OF PARALLEL AND WORK-SHARING CONSTRUCTS

This section develops eight simple examples, with the purpose of illustrating the thread management and work-sharing concepts discussed so far. The first three examples reconsider the Monte-Carlo computation of π discussed in Chapter 3, illustrating the OpenMP flexibility in adapting to different programming styles for parallel sections, including the parallel for directive. Next, two examples illustrate the usage of the master and single directives inside a parallel section, as well as the generation of nested parallel regions. The values of some ICVs are checked, and the way the different implicit tasks are mapped to threads is exhibited. Finally, three simple examples deal with the parallel sections construct.

10.5.1 DIFFERENT STYLES OF PARALLEL CONSTRUCTS

In the simple examples proposed in previous chapters, a parallel focus was adopted from the start. All variables were declared with the correct scope (global or local), a task function encapsulating the

parallel block was explicitly written, and reductions were computed by accumulating partial results in mutex-protected global variables. Data-sharing attribute clauses were not needed in this context, because OpenMP was already provided with an unambiguously constructed task function.

An OpenMP version of the Monte-Carlo computation of π using this programming style was discussed in Section 4.1. It is instructive to see how the same problem can be handled by using directives to directly modify the original sequential code, listed below:

```
// Initial sequential code:
// ......................
int main(int argc, char **argv)
   {
   long nsamples;
   Rand R(999);
   double x, y, pi;
   unsigned long count;

   // get nsamples from command line
   count = 0;
   for(size_t n=0; n<nsamples; n++)
      {
      x = R.draw();
      y = R.draw();
      if((x*x+y*y) <= 1.0 ) count++;
      }
   pi = 4.0 * (double)(count) / nsamples;
   cout << "\n Value of PI = " << pi << endl;
   }
```

LISTING 10.6

Sequential Monte-Carlo computation of π

Remember that objects of the class Rand generate uniformly distributed random doubles in the interval [0, 1]. The code listed above generate nsamples of uniformly distributed points inside a box of side 1, and counts how many of them are inside the unit circle. The ration count/nsamples determines the value of π.

Adding directives to the listing above

In constructing the parallel version, the purpose is to dispose of several threads sharing the work of producing samples and counting how many of them are accepted. Two threads are activated, each one examining nsamples/2 samples. The listing below shows the modified, parallel code. The only modification needed to implement the parallel version is the introduction of the parallel directive with a few relevant clauses, and in particular the reduction clause. Another important modification—using local random number generators—is needed in this case to make the code thread-safe, as discussed in Chapter 4. All this can be accomplished by adding a few OpenMP statements and directives to the sequential code.

```
int main(int argc, char **argv)
   {
   long nsamples;
   double pi;
   unsigned long count;

   // get nsamples from command line
   count = 0;
   omp_set_num_threads(2);

   #pragma omp parallel firstprivate(nsamples) \
                        reduction(+:count)
      {
      // task function ......................
      double x, y;
      int rank = omp_get_thread_num()+1;
      Rand R(999*rank);
      for(size_t n=0; n<nsamples/2; n++)
         {
         x = R.draw();
         y = R.draw();
         if((x*x+y*y) <= 1.0 ) count++;
         }
      // end of task function ................
      }
   pi = 4.0 * (double)(count) / nsamples;
   cout << "\n Value of PI = " << pi << endl;
   }
```

LISTING 10.7

Parallel Monte-Carlo computation of π

The modifications introduced to the sequential code are the following:

- The number of threads in the next parallel section is fixed by a call to omp_set_num_threads(2) just before the parallel directive.
- The scratch variables x, y are declared in the body of the task function, and they are unambiguously local variables in each thread.
- Consider next nsamples, a local variable initialized by main(). By declaring it firstprivate, each thread gets a local copy with the value the local main variable had on entry to the parallel section. Note that nsamples could have been declared initially with global scope, in which case this clause is not needed.
- Finally, there is the reduction clause that performs the reduction operation needed to accumulate the acceptance count computed by each worker thread.

This example shows how clauses serve to transfer information to tasks. The firstprivate clause transfers local variable values from the encountering master thread to all the other workers. The reduction

clause collects local results from the worker threads and transfers the result of the reduction operation to the master thread that resumes the task in execution when the parallel region was encountered.

Example 1: CalcPi1.C

To compile, run make calcpi1. The number of threads is 2. The number of Monte-Carlo events is read from the command line (default is 10000000).

A version using parallel for

The next example is the construction of a parallel version using the parallel for work-sharing directive. The main computational work in this code takes place after all in a for loop. This will be done, again, by adding directives to the sequential code, but the structure of the code is different from the previous example. The purpose in that case was to help the compiler to construct a thread function, and we were able to refer to the rank of the executing thread, and to introduce local random number generators. Moreover, the work distribution among threads was done by hand: each one of the two threads was managing nsamples/2 samples.

Nothing like that is possible when the parallel work sharing is totally implicit. The parallel for directive acts only on the for loop that follows. This means that it is not possible to define local random number generators with thread-specific initialization, and we have to rely on a global function Rand(), with thread safety being ensured by using thread local storage, as discussed in Chapter 4. Here is the listing of the main() function.

```
int main(int argc, char **argv)
   {
   long nsamples;
   double pi;
   double x, y;
   unsigned long count;

   // get nsamples from command line
   count = 0;
   omp_set_num_threads(2);
   #pragma omp parallel for firstprivate(nsamples) private(x, y) \
                        reduction(+:count)
    for(size_t n=0; n<nsamples; n++)
       {
       x = Rand();
       y = Rand();
       if((x*x+y*y) <= 1.0 ) count++;
       }

   pi = 4.0 * (double)(count) / nsamples;
   cout << "\n Value of PI = " << pi << endl;
   }
```

LISTING 10.8

Parallel Monte-Carlo computation of π

The reduction operated on the count variable works as in the previous example. The code uses the thread-safe Rand() generator function discussed in Chapter 4, in which a thread-specific initialization is implemented.

Example 2: CalcPi2.C

To compile, run make calcpi2. Uses the for work-sharing directive. The number of threads is 2. The number of Monte-Carlo events is read from the command line (default is 10000000).

Yet another version of the Monte-Carlo code

Leaving aside the previous version using the parallel for directive, we have seen so far two different versions of the Monte-Carlo computation of π: a fully encapsulated task function with no directives in Chapter 3, and the version above constructed by adding directives to the sequential code. To show the OpenMP flexibility, yet another version is proposed next in which the task function is only partially encapsulated in an auxiliary function.

```
unsigned long Get Acceptance(long NS)
   {
   double x, y;
   unsigned long my count;

   int rank = omp_get_thread_num()+1;
   Rand R(999*rank);
   for(size_t n=0; n<NS/2; n++)
      {
      x = R.draw();
      y = R.draw();
      if((x*x+y*y) <= 1.0 ) my count++;
      }
   return my count;
   }

int main(int argc, char **argv)
   {
   long nsamples;
   double pi;
   unsigned long count;

   // get nsamples from command line
   count = 0;
   omp_set_num_threads(2);

   #pragma omp parallel firstprivate(nsamples) \
                        reduction(+:count)
```

Continued

```
    { count = Get Acceptance(nsamples); }

  pi = 4.0 * (double)(count) / nsamples;
  cout << "\n Value of PI = " << pi << endl;
  }
```

LISTING 10.9

Yet another parallel Monte-Carlo computation of π

The computation is now organized in a different way:

- The auxiliary function Get Acceptance, *executed by each thread*, receives as argument the number of samples and returns the computed acceptance. This auxiliary function uses internally a local random number generator with thread-dependent initialization: the thread rank is used to select the initial seed of the generator (it is increased by one unit to avoid the value 0 for the seed).
- The parallel code block just calls this function, receives the return value in count, and performs the reduction of count as in the previous examples.
- The variable nsamples must again be declared firstprivate. Indeed, each task function needs its value in order to pass it to the auxiliary function. Again, this clause could be avoided if nsamples was declared with global scope.

Example 3: CalcPi3.C

To compile, run make calcpi3. The number of threads is 2. The number of Monte-Carlo events is read from the command line (default is 10000000).

10.5.2 CHECKING ICVs AND TRACKING THREAD ACTIVITY

Checking ICVs

The example that follows demonstrates the usage of the single directive inside a parallel section, checks some global ICV values, and shows how the ICVs can be modified in nested parallel regions. The program creates first a parallel section with two threads, and then each one of these threads creates in turn a nested parallel region of three threads, as shown in Figure 10.3.

When main() starts, some default values of global ICVs are checked, and in particular the maximum number of threads accepted in the application. Then, a parallel region of two threads is created (region 1 in Figure 10.3). Each one of the two threads in region 1 creates in turn a three-thread parallel region (region 2 in Figure 10.3). A barrier is placed after the nested parallel regions, so that the two initial master threads are synchronized when they enter region 3 in the external parallel region. The purpose of the program is to check and print some ICV values in all the parallel regions in the program: the two nested parallel construct—in region 2—and the external parallel construct—in region 3. However, our intention is to print one message per team, not one message per thread, and for this reason the single directive is used. Here is a sketch of the code organization, starting from the first (external) parallel region encountered:

```
int main(int argc, char **argv)
   {
   ..
   omp_set_num_threads(2);
```

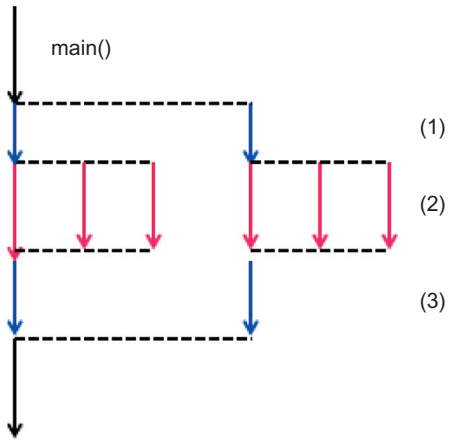

FIGURE 10.3

Nested parallel sections implemented in Examples 4 and 5.

```
#pragma omp parallel          // external, with 2 threads
   {
   ...                        // executing region 1
   omp_set_num_threads(3);
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   #pragma omp parallel       // entering region 2 in Fig 10.3
       {
       #pragma omp single
          {
          // checks of ICVs and messages
          }
       }                      // exiting region 2
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   #pragma omp barrier        // synchronizes the 2 threads when
   ...                        // entering region 3
   #pragma omp single
       {
       // checks of ICVs and messages
       }
   }                          // end of external
}
```

LISTING 10.10

Layout of the code related to Figure 10.3

This code is a useful exercise to understand how the single directive works. Its role in the listing above is to have only one thread in each team checking IVCs and writing to stdout. The other threads in each team are idle. The role of the barrier after the nested parallel regions is to synchronize the two external threads and make sure the region (3) message is not printed before they both emerged from the nested parallel region.

Example 4: Icv1.C

To compile, run `make icv1`. The number of threads is hardwired to the configuration shown in Figure 10.3.

Tracking the mapping of tasks to threads

This example implements exactly the same scenario as before, but now our intention is to check in detail how the eight tasks (the two initial external tasks and the six nested tasks) are distributed among the total number of threads allocated to this program, which is six. In this example, every single task in the external and nested parallel regions prints message indicating:

- The OpenMP thread number (rank) of the executing thread.
- The `pthread_t` identity of the executing thread, as returned by a call to the `pthread_self()` function.

The point here is that the OpenMP thread number depends on the context. When the second thread in the first parallel region (thread number 1) enters its nested parallel region, it becomes master and takes the thread number 0. After exiting the nested region, it recovers the thread number 1. On the other hand, the `pthread_t` identity of a thread is an absolute identifier controlled by the system, with no reference to OpenMP. Following the output of this program we can follow what each one of the six different threads is doing.

How can we output a `pthread_t` value? We know that, strictly speaking, this is an opaque data item with unknown structure. However, it turns out that, in Linux, the C++ instruction `cout << pthread_self()` works correctly because the `pthread_t` data type returned by `pthread_self()` is in fact implemented as a long integer.

Example 5: Icv2.C

To compile, run `make icv2`. The number of threads is hardwired to the configuration shown in Figure 10.3.

10.5.3 PARALLEL SECTION EXAMPLES

Behavior of the barrier directive

Contrarily to the `barrier` function calls in the Pthreads or vath libraries—in which the barrier utility is always initialized with the number of threads participating in the synchronization—the `barrier` directive in OpenMP does not requires any initialization: it adapts automatically to the number of worker threads in the parallel region where the directive is used. This is perfectly consistent with the parallel region execution model: all worker threads executing the same task function. There are therefore N worker threads and N barrier calls.

One may, however, ask what happens in an OpenMP `parallel sections` environment involving `Nth` worker threads, in which there are `N` implicit tasks with an internal barrier directive, submitted for execution. The barrier directive appears then in a number of implicit tasks different from the number of worker threads. How does the code behave? The situation is ambiguous because OpenMP only sees independent tasks since the notion of "parallel job" is absent. If `N>Nth`, should the execution context be interpreted as one big job involving all the tasks, or as consecutive jobs involving a lower number of tasks each?

The example developed in the source file Sections.C—listed below, some irrelevant details have been left aside—submits six identical tasks with an internal barrier directive, and *before* and *after* messages on both sides of the barrier. The number of threads is read from the command line, with a default of six (the same as the number of tasks).

```
void Barrier_Task()
    {
    int rank = omp_get_thread_num();
    ...
    std::cout << "\n Thread  " << rank <<  " before barrier "
                << std::endl;
    #pragma omp barrier
    std::cout << "\n Thread  " << rank <<  " after barrier "
                << std::endl;
    }

int main(int argc, char **argv)
    {
    int nth;
    // read nth from command line
    ...
    #pragma omp parallel sections thunderheads(nth)
       {
       #pragma omp section { Barrier_Task();}  //
       #pragma omp section { Barrier_Task();}
       #pragma omp section { Barrier_Task();}
       #pragma omp section { Barrier_Task();}
       #pragma omp section { Barrier_Task();}
       #pragma omp section { Barrier_Task();}
       }
    return 0;
    }
```

LISTING 10.11

Barrier behavior, Sections.C

When the code is executed with the default values, or with more threads than tasks, the output corresponds to our expectations: all the *before* messages precede the after messages. With fewer threads than tasks, *the code does not deadlock*, but I suspect that the resulting behavior may be implementation dependent. In the case of the GNU compiler with, for example, three threads, OpenMP adapts the barrier behavior to the number of threads. Three tasks are executed first, correctly synchronized at the barrier point, and three other tasks are executed next, again correctly synchronized at the barrier point. We obtain three *before* messages followed by three *after* messages, and then the same pattern repeats when the three threads execute the remaining three tasks. Try running four threads, and see what happens.

Example 6: Sections1.C

To compile, run make sect1. The number of tasks is 6. The number of threads is read from the command line (the default is 6).

Dispatching a I/O operation

The code used to test Boolean locks in Chapter 9, where one extra thread is dispatched to perform a lengthy I/O operation, is re-examined. The I/O task simulates the operation by putting the executing thread to wait for 5 s. The other task checks every second if the I/O operation is finished. The synchronization of both threads is implemented with a Boolean lock.

The source code is in file Sections2.C. Here is the listing of the main() function:

```
int main(int argc, char **argv)
    {
    int status;
    B = new BLock(false);

    #pragma omp parallel sections num_threads(2)
       {
       #pragma omp section
          MainTask();
       #pragma omp section
          TaskIO();
       }
    delete B;
    return 0;
    }
```

LISTING 10.12

Main() function for Example 6

The main thread initializes the global Boolean lock to false, and launches the parallel job. There is no more OpenMP in this code. The two task functions: TaskIO() that waits for 5 s before toggling the Boolean lock to true, and MainTask() that checks every second if the toggling has taken place, are the same functions we used in the previous versions of this code.

Example 7: Sections2.C

To compile, run make sect2. The number of threads is hardwired to 2.

Data transfers among tasks

Finally, the example proposed in Chapter 9 to test the Synch<T> is also re-examined. Remember that this utility synchronizes the writing of a type T data item by one thread with the reading of the new value by other threads, and that the number of expected readers is fixed when the writer thread posts the new value.

This example constructs a three-thread parallel region, in which there is one writer thread that passes the value of a double data item to two reader threads, which is done three consecutive times. Reader threads print the received values, so there will be altogether six reader messages.

This is a pure OpenMP example. The code uses a global OSynch<double> object to synchronize the threads. This class is the pure OpenMP version of the Synch<T> class introduced in Chapter 9, and the public interfaces are the same. The writer and readers task functions are therefore practically the same functions used before. The driver code in the main function is very simple:

```
int main(int argc, char **argv)
    {
    #pragma omp parallel sections num_threads(3)
       {
       #pragma omp section
           WriteTaskFunction();
       #pragma omp section
           ReadTaskFunction();
       #pragma omp section
           ReadTaskFunction();
       }
    }
```

LISTING 10.13

Partial listing of Sections3.C

Example 8: Sections3.C

To compile, run make sect3. The number of threads is hardwired to 3.

10.6 TASK API

Prior to the introduction of the task directive in OpenMP 3.0, the thread-centric OpenMP execution model was based on the fact that the only tasks that could be executed in a parallel region were the implicit tasks initially declared when the parallel region was encountered. The basic principle was a default one-to-one mapping of implicit tasks to threads, with the exception of the single construct—that activates only one thread—and the parallel sections construct that enable the execution of a number of implicit tasks different from the number of threads in the team. However, in all cases, the tasks executed in the parallel construct were there from the start. The introduction of explicit tasks triggered *inside* a parallel region, for reasons that are discussed next, corresponds to a significant extension of the original execution model.

10.6.1 MOTIVATIONS FOR A TASK-CENTRIC EXECUTION MODEL

When parallel regions are nested to incorporate a finer structure into the parallel treatment, OpenMP adds the additional threads required to cope with the enhanced parallel activity, maintaining a kind of

one-to-one mapping between implicit tasks and threads. Nested parallel regions enable programmers to cope with, for example, irregular or highly recursive patterns. But this approach is not totally satisfactory.

The reasons are simple to understand. The fact that nested parallel regions keep adding the additional worker threads that are needed to run the new, nested implicit tasks may substantially increase the number of over-committed threads running on the fixed number of cores allocated to the application. As discussed in Chapter 2, this is not by itself a problem: the operating system will distribute time slices of the available CPU resources to give all threads a chance to run. However, preempting and rescheduling threads is a *system* activity that, if excessive, may negatively impact the parallel performance of the application. Moreover, every time a new thread is created, a new stack memory buffer is attributed to the new thread: excessive thread creation may lead to exhaustion of memory resources.

This is probably the reason why OpenMP has always allowed programmers to control the level of nesting in an application. Beyond the level L of nesting set by the `set_nested(L)` library call, nested parallel regions are ignored and replaced by sequential code.

The different ways of scheduling a large number of tasks on a fixed number of threads has attracted substantial attention from computer scientists. The problem is complex, because, if the main target is obtaining the maximum possible performance, there are several parameters to consider, like memory usage, cache memory re-usage optimization, and a few others. There is a clear consensus that over-committing a large number of tasks on a *fixed* number of threads—equal, ideally, to the number of cores allocated to the application—is much more efficient than over-committing a large number of threads on a fixed number of cores. One important reason is that task rescheduling is much more efficient and can be implemented in user space. This execution model is shown in Figure 10.4. Cilk [3] was the first programming environment to implement this approach, based on a *task stealing scheduling strategy* that will be described in Chapter 15, when discussing TBB. Cilk strongly influenced the OpenMP tasking extension started in the 3.0 release, as well as the TBB multithreading environment.

The `task` directive incorporates a task-centric scheduling strategy into the original OpenMP execution model, which is extended, not abandoned. This directive enables the inclusion of new explicit

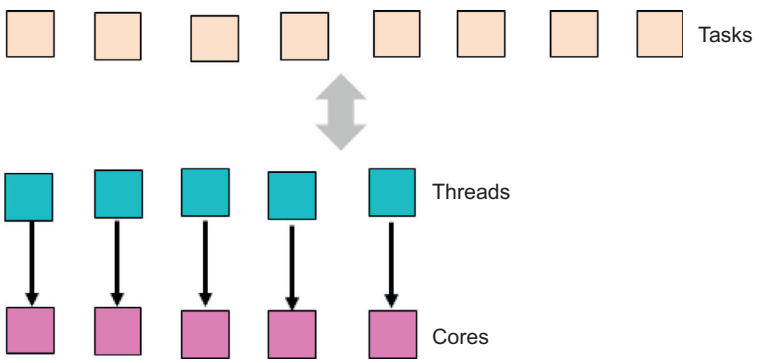

FIGURE 10.4

Task-centric execution model.

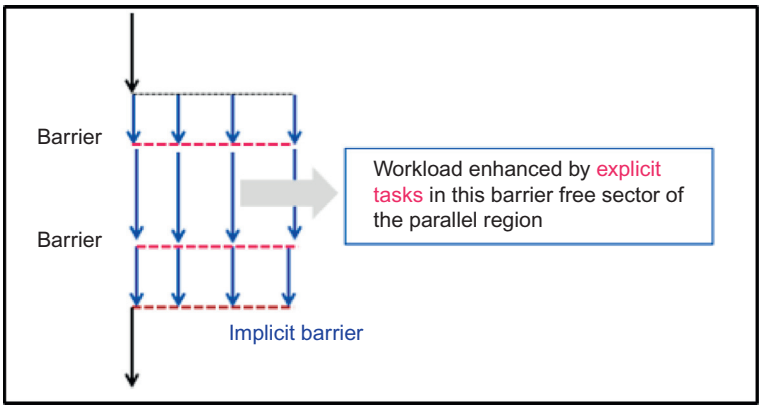

FIGURE 10.5

Task-centric implementation in OpenMP.

tasks inside a running parallel region, with a fixed number of threads. The number of worker threads is not changed, but the number of tasks submitted for execution is allowed to grow dynamically. Tasks are therefore executed by the threads of the team, and their data environment is constructed at creation time. Once created, task execution may be delayed or suspended. These features add enormous flexibility to the management of irregular problems or recursive parallel patterns.

Figure 10.5 shows the way the task-centric scheduling strategy is incorporated into the OpenMP execution model. *In a barrier-free sector inside a running parallel region*, the computational workload can be extended with additional explicit tasks. The initial implicit tasks can spawn new tasks, which in turn can spawn new tasks, and so on and so forth. The important point to keep in mind is that all the parallel activity (initial implicit tasks plus additional explicit tasks) will be completed by the time the worker threads reach the following (implicit or explicit) barrier. In other words, explicit tasks have a lifetime limited by the next barrier in the enclosing parallel region.

This fact is easy to understand. Barriers synchronize *threads*, not *tasks*. Barriers only know that at some point they have to release the threads reaching the synchronization point. Explicit tasks before the barrier, which only increase the workload of the worker threads, are totally unknown to the barrier. The task scheduler will make sure that threads reach the barrier only after having exhausted all the workload provided by the implicit and explicit tasks.

10.6.2 EVENT SYNCHRONIZATIONS AND EXPLICIT TASKS

An important observation is in order at this point: care must be exercised when synchronizing explicit tasks generated in a parallel region. It was often stated before that Chapter 9 synchronization utilities constructed with basic libraries tools could be used in a OpenMP environment, and several examples were presented in previous chapters. However, this is true only in parallel regions without explicit tasks, where the thread-centric execution model applies. As stated before, synchronization tools synchronize threads, not tasks. In the absence of a precise mapping of tasks to threads, event synchronization tools are unsafe, as shown below.

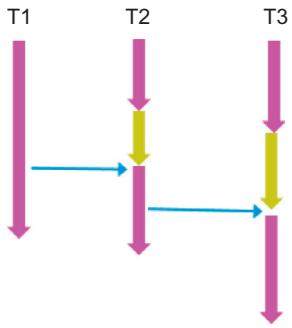

FIGURE 10.6

Parallel context that may deadlock on two threads.

To summarize, the synchronization rules in a task-centric region are the following:

- *Mutual exclusion* is safe, and critical sections, mutex locking or atomic operations coexist peacefully with the unfair synchronization policies for tasks.
- *Event synchronizations* like spin or idle wait barriers, Boolean locks, or any other low- or high-level event synchronization tools are not safe, and cannot be used to synchronize explicit tasks.

Figure 10.6 shows a simple parallel context that may or may not deadlock when tasks are over-committed, according to the way they are scheduled. Task T3 performs a spin or idle wait, and is released by task T2. Task T2, in turn, performs also a spin or idle wait and is released by T1, before releasing T2. When the tasks are run by two threads, they execute correctly if T1 and T2 are executed first. However, if T2 and T3 are executed first, the code deadlocks because the two threads executing T2 and T3 are blocked, and there is no thread available to execute T1 and release them.

However, the question can be asked, what are we supposed to do if, nevertheless, we need to synchronize tasks? The answer is that tasks must be synchronized by the task scheduler itself. This is accomplished by introducing hierarchical relations among tasks that guide the way the scheduler executes them. TBB introduced from the start a powerful environment that implements this strategy, to be discussed in detail in Chapter 16. OpenMP 4.0 introduced a `depend` clause that establishes hierarchical relations among sibling tasks. Section 10.7.5 shows in detail how to implement a barrier for tasks by using this clause.

10.6.3 TASK DIRECTIVE AND CLAUSES

We call an *explicit task* a task created by the `task` directive. An explicit task will always be a child of an initial implicit task, or the child of another explicit task previously created by the same directive. Therefore, the `task` directive only appears inside a task function code. Here is the form of the directive:

```
int main()
  {
  ...
  #pragma omp parallel
```

```
        {
        ...
        #pragma omp task [clause] ... [clause]
            {
            // -  task code block - --
            }
        }
    }
```

LISTING 10.14

General form of the task directive

#pragma omp task

- *To be used inside a task region.*
- The code block that follows is a new task submitted to the pool.
- This new task joins the ongoing parallel job in the current parallel region.
- The task will ultimately be executed by a thread in the team where it is created. No additional threads are added to the worker team.
- Several clauses are accepted.

A task created in this way will be scheduled according to the worker threads' availability. It may start immediate execution, it may be delayed, and when scheduled it may be suspended and resumed later if needed. We will see later on that task suspension is critical to avoid deadlocks when tasks are over-committed in a thread pool. Here are the clauses associated with this directive:

- Four basic data-sharing clauses are listed in Table 10.2: private, shared, default, and firstprivate.
- *if(expression)*: If the Boolean expression evaluates to false, the encountering thread does not submit the task to the pool. Instead, it suspends the execution of its current task and starts immediate execution of the new task. Execution of the suspended task is resumed later. In this case, the new task creation reduces for all practical purposes to a simple function call. The purpose of this clause is to put some limits on the creation of too many explicit asynchronous tasks.
- *untied*: Tasks—implicit in a parallel region or explicitly created by the task directive—are tied to a thread by default. This means the thread that starts executing them keeps the ownership of the task for all its lifetime. If for some reason the task is suspended, its execution will be resumed by the same thread. Declaring a task untied means the initial executing thread does not keep ownership, and that any other thread can resume execution. This gives the runtime system more flexibility for rescheduling suspended tasks and may provide better performance. However, allowing suspended tasks to migrate across threads may introduce other potential pitfalls that programmers should be aware of. This issue is discussed in Section 10.8.
- *final (Boolean expression)*: If the Boolean expression is true, this clause declares the task as final. A final task forces all its children to become *included* tasks. An *included* task is a task whose execution is sequentially included in the current task region. This means the task is undeterred, and immediately executed by the encountering threads.
- *mergeable*: Declares that the task can be merged into an ordinary function call.
- *depend (dependence-type: list)*: Establishes dependencies among sibling tasks, that is, among tasks that have been directly spawned by a common ancestor. The dependencies so established are used to order the way in which the tasks are scheduled: a task will only be executed after all tasks it

depends are terminated. The way this directive operates is discussed in one of the examples that follow.

Note the difference between the if() and final() clauses. When the Boolean expression is true, the if() clause replaces the task itself by a function call, and the final() clause forces the encountering thread to execute the task immediately instead of submitting it to a ready pool for later scheduling. Note also that OpenMP is the only programming environment that allows migration of suspended tasks. This is not the case in TBB and in the NPool utility discussed in the following chapters.

The *depend* clause is a new OpenMP 4.0 feature. Introducing hierarchical dependencies among tasks is a very powerful evolution. Explicit hierarchical dependencies among tasks are a major feature of the Intel TBB environment. Task dependencies are needed to implement *synchronization patterns among tasks*, not among threads. A forthcoming example shows in detail how the depend clause can be used to implement a barrier synchronization among an arbitrary number of tasks running in a parallel section with a fixed number of threads.

10.6.4 TASK SYNCHRONIZATION

The next question that needs to be asked is: once new explicit tasks are dynamically created, how can the parent tasks that launched them know they are finished? OpenMP has a powerful and clever synchronization model for explicit tasks, as shown in Figure 10.7. It can be summarized as follows:

- *barrier synchronization*: All possible explicit tasks created in a parallel section (children, children of children, etc.) terminate at the next barrier directive (implicit or explicit).
- *taskwait synchronization*: A task can wait for termination of its *direct children tasks* at a taskwait directive in its task function code. Note that children of children are not catched by this directive.

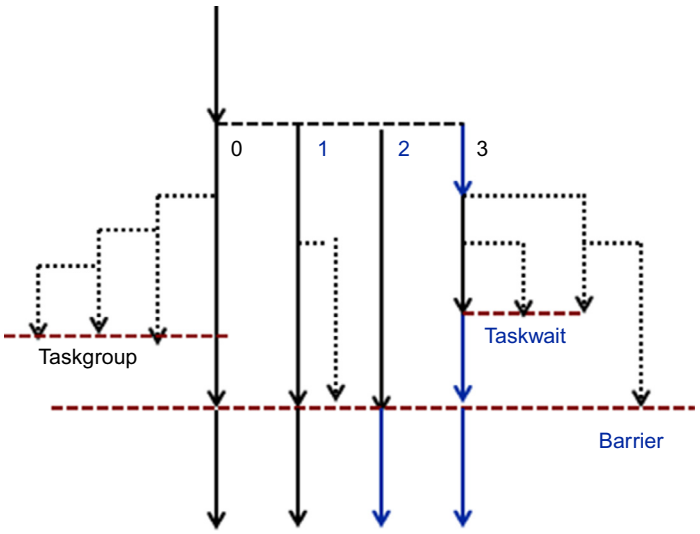

FIGURE 10.7

Task synchronizations at barrier, taskwait, and taskgroup directives.

- *taskgroup synchronization, in OpenMP 4.0*: At the end of a code block associated to a taskgroup directive inserted in its task function code, a task waits for termination of *all, direct, or indirect descendants* generated inside the code block.

The four blue tasks in Figure 10.7 are the initial implicit tasks associated with the parallel region. The explicit tasks created with the task directive are marked in red. The figure shows several recursive patterns of explicit task generation. For some of the explicit tasks, their terminations are synchronized with their ancestors via the taskwait or taskgroup directives Other explicit tasks terminate directly. However, in all cases, the scope of the explicit tasks is bounded by the next barrier directive. The initial implicit tasks are not released from the barrier synchronization point until all the explicit tasks generated before the barrier are terminated.

The two other thread pool environments to be discussed in the next two chapters—TBB and NPool—are also task-centric environments proposing also taskwait and taskgroup utilities with features very similar to the ones discussed above for OpenMP.

10.6.5 OpenMP 4.0 TASKGROUP PARALLEL CONSTRUCT

The taskgroup directive was introduced above as a new task synchronization feature in OpenMP 4.0. In fact, it is much more than that: it is a new parallel construct that enhances the expressive power of OpenMP, for applications requiring dynamical task generation. We may think of a traditional parallel region as a construct encapsulating a parallel job in a thread-centric environment. In the same way, it is possible to think of the taskgroup construct as a construct encapsulating a parallel job executed by explicit tasks.

The taskgroup construct enables implementing a programming style in which a task launches a complex recursive parallel treatment, continues to do something else and, when the time comes, waits for the job termination. This style of code organization is shown in the listing below. The significant added value provided by the taskgroup construct will be demonstrated later on, first by looking at a divide and conquer algorithm used before for the computation of the area under a curve, and then discussing an unbalanced, unstructured, recursive algorithm that cannot possibly be parallelized without the taskgroup construct.

```
int main()
  {
  ...
  #pragma omp parallel
      {
      #pragma omp single
         {
         ...
         #pragma omp taskgroup
            {
            // - - - - - - - - - - -
            // implicit task continues here
```

Continued

```
            ...
            // task spawns as many explicit tasks as
            // needed, which in turn can spawn more
            // tasks, and so on
            ...
            // implicit task continues here, doing other things
            ..
            // The implicit task waits, before exiting the taskgroup
            // code block for the termination of all its direct or
            // indirect descendants
            }
         // The implicit task continues here
         ...
         }     // end of single task
      }        // end of parallel region
...
}
```

LISTING 10.15

General structure of a taskgroup construct

In the listing above, the main thread opens a parallel region with the intention of activating a set of worker threads. Inside the parallel region, there is a unique implicit task executed, defined inside the single code block. This task can be considered as the continuation of the execution stream suspended when the parallel region was encountered. This unique implicit task continues execution, and at some point decides to launch a parallel job. To do so, it opens a taskgroup code block, spawns all the tasks of the parallel job (which in turn can spawn other tasks), and continues the execution of its own workload. When the task reaches the end of the code block, it waits for the termination of all its direct or indirect descendants, that is, the termination of the submitted job. Then, it continues execution, using the outcome of the job. The task terminates at the end of the single code block, and at the end of the parallel region the worker threads are decommissioned and the initial execution stream resumed.

10.7 TASK EXAMPLES

The examples that are developed in this section have been designed to illustrate the operation of OpenMP tasks.

10.7.1 PARALLEL OPERATION ON THE ELEMENTS OF A CONTAINER

The purpose of this example is to demonstrate the efficiency of tasks in handling a strongly unbalanced parallel pattern. Unbalanced means the amount of work performed by each task is highly erratic, so it is difficult if not impossible, with the standard methods, to achieve a balanced workload distribution among the worker threads.

A map operation is performed on all the elements `V[n]` of a vector of doubles of size N. A map operation changes the initial value of the vector element by another value:

```
for(int n=0; n<N; n++) V[n] = f(V[n]);
```

and the task function $f(x)$ is such that the amount of computation it generates is highly irregular as we move along the container. It can be described as follows:

- Each container element is first initialized with a private uniform random value in [0, 1], by calling a global random number generator.
- The task acting on a given container element repeatedly calls a uniform random generator until it gets a value that is close to the initial value within a given tolerance, for example, 0.0001, called precision in the code.
- The initial container element value is replaced by the new close value so obtained.
- Obviously, increasing (or decreasing) the precision increases (or decreases) the average computational work done on container elements.
- In our example, we take a vector size N = 100000, and the precision is decreased by four orders of magnitude from an initial value of 0.0000001, as we move along the container. This produces the workload pattern shown in Figure 10.8.

Using OpenMP tasks to do this operation is not really mandatory. The operation described above is a map operation, and it is sufficient to write a single loop and share the loop operation among several threads. However, a straightforward application of the parallel for directive is inefficient in this unbalanced problem. With N threads, this directive splits the loop iteration space in N chunks, one for each thread. It is, however, obvious that the first chunk workload is much bigger than the last one, and the code performance will be very far from exhibiting a speedup equal to the number of threads. The situation can be improved by using the schedule clause to reduce the chunk size. Nevertheless, we will show that in this case the task scheduling strategy implements a better load balance for the operation of the underlying thread pool, which outperforms the parallel for construct.

The complete listing is in source file Foreach1.C. Let us first look at the global variables in the code, and to the task function executed by each task:

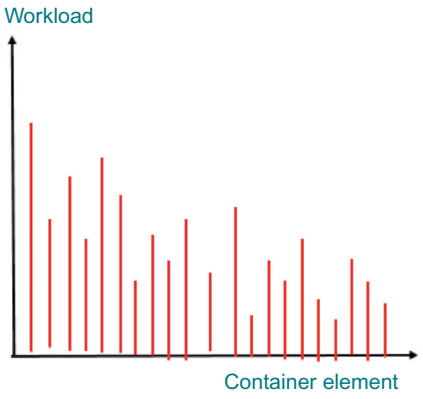

FIGURE 10.8

Computational workload for vector elements.

```
// Global variables
// – – – – – – – – –
int  nTh;
int  N;
double p_initial, p_final;
std::vector<double> V;
SafeCounter SC;

double precision(int n)
   {
   double a = (p_final – p_initial)/N;
   return (a*n+p_initial);
   }

void Replace(int n)
   {
   Rand R(999 * SC.Next());
   double x;
   double eps = precision(n);
   double d = V[n];
   do
      {
      x = R.draw();
      }while( fabs(x–d)>eps );
   V[n] = x;
   }
```

LISTING 10.16

Task function in this example

The target of the map operation is the vector V, initialized by main() in the way described above. The task function uses internally a random number generator to compare its output with the initial value of the target element. Following the best practices established in Chapter 4, each task owns its own, local, random number generator. However, we also want each task to dispose of a distinct and personalized random suite. For this reason, the initial seed of each local generator is chosen in a task-specific way. Tasks get a unique, integer rank from a global SafeCounter object. This class has been designed to increase and return an integer counter at each call of its Next() member function, in a thread-safe way.

The second point to be observed is that the precision decreases as we move along the container: it interpolates between an initial precision p_initial at the container head and a final precision p_final at the container tail. True enough, the computational cost of each task is random, but if the precision was constant along the container we would expect the computational cost of each subrange computation in a parallel for loop to be roughly the same. Decreasing the precision along the container adds load imbalance to the problem.

Let us next consider the main() function, listed below. After array initialization and a sequential computation, two parallel computations are performed, the first using a parallel for construct and the second by launching N explicit tasks.

```
int main(int argc, char **argv)
   {
   int n;
   CpuTimer T;
   Rand rd(777);

   // Set parameter values
   // Initialize vector target
   // Perform sequential computation

   omp_set_nested(1);
   omp_set_num_threads(2);

   // parallel for computation
   //  - - - - - - - - - - - - -
   T.Start();
   #pragma omp parallel for schedule(static, N/24)
      for(n=0; n<N; ++n) Replace(n);
   T.Stop();
   T.Report();

   // task computation
   //  - - - - - - - - -
   T.Start();
   #pragma omp parallel
      {
      #pragma omp single
         {
         n = 0;
         while(n < N)
            {
            #pragma omp task untied
              { Replace(n); }
            n++;
            }
         }
      #pragma omp barrier
      }
   T.Stop();
   T.Report();
   // print V[0], V[N/2], V[N-1]
   }
```

LISTING 10.17

Main() function in the present example

First, look at the parallel for computation. With the default scheduling for the work distribution—as many chunks as threads, four in this case—the parallel computation is totally unbalanced. Smaller chunks and different scheduling strategies can be used to better distribute the workload among threads: chunks of sizes N/24 and N/48, and the static and dynamic scheduling options, as discussed below.

Consider the task computation. Note that, because of the single directive, *there is only one initial implicit task executed by one of the worker threads*, whose role is to generate all the remaining explicit tasks that operate on the container. As the extra tasks are generated, all the worker threads in the team participate in their execution as soon as they are available. This, as we shall see, is confirmed by the observed speedup of the program execution. After the sequential and parallel computations, the code prints some selected container elements to verify that they have indeed been modified.

This example also shows the OpenMP flexibility in the definition of task functions. Let us take a closer look at the code in order to understand the scope of the different variables, as well as how and why the correct information is conveyed to the ultimate explicit tasks. Here is the scenario implemented by OpenMP:

- The loop variable n is initially a local variable to main().
- As the parallel section and then the single directive are entered, n is by default private. Therefore, inside the single code block where the loop over n operates, it is just a local variable to the task generating thread.
- As the task code block is entered, *local variables in the parent task are treated as firstprivate by default*, so a new variable n is created *and initialized with the parent task value*.
- Finally, inside each explicit task, this variable is passed as argument to the Replace() function. This is the reason why the loop index n is correctly transferred to the explicit tasks.

Example 9: Foreach1.C

To compile, run make feach1. The number of threads is hardwired to 4. The vector size is N = 1000000, the default initial precision is 0.0000001, and the default final precision is 1000 times smaller. The initial and final precisions can be overridden from the command line. Look at the code source.

The performance measurements for the configuration described in the box above are given in Table 10.3. These results speak for themselves. It was known from the start that the parallel for

Table 10.3 Four Worker Threads: Execution Times in Seconds

Performance of Foreach1.C			
Algorithm	**Wall**	**User**	**System**
sequential	9.25	9.25	0
for: auto	7.37	9.42	0
for: static, N/24	5.68	9.42	0
for: static, N/48	5.14	9.55	0
for: dynamic, N/24	5.02	9.51	0
for: dynamic, N/48	4.25	9.72	0
task	2.54	10.13	0.02

computation with automatic scheduling would strongly suffer from load unbalance. Using chunks of sizes N/24 an N/48 improve the performance, but we are far from obtaining the expected speedup for four threads. The task computation is flawless, in spite of the fact that we have generated one million tasks! The speedup is close to 4, with small system overhead and very limited user overhead (total user time of 10.13 s instead of the sequential user time of 9.25 s).

10.7.2 TRAVERSING A DIRECTED ACYCLIC GRAPH

The example proposed in this section deals with a parallel context that cannot be parallelized with a traditional thread-centric approach. The parallel context is unstructured (there is no predefined pattern establishing the order in which tasks are scheduled) and unbalanced (the amount of computational work performed by each task is unpredictable). The parallel context is also recursive, and the critical role played by the taskgroup construct will be shown.

This section incorporates excerpts of the Intel Threading Building Blocks documentation. The example discussed here is an OpenMP version of the parallel do preorder_traversal example available in the TBB release, in directory //examples//parallel_do. Most of the source files are directly taken from this example, with Intel's permission.

Description of the parallel context

We will consider a collection of Cell objects, instances of a class defined in the header file Cell.h. Cell objects have internally, among other members, a data item updated by a member function, if and when the input data required by the operation is available.

In a directed acyclic graph, cell objects are hierarchically organized as shown in Figure 10.9. *This hierarchical organization is not known at compile time, it is randomly initialized at runtime*. Cells may have 0, 1, or 2 ancestors. Ancestors provide the input data for the cell update, and this operation depends naturally on the number of ancestors in a way to be described next. It is clear that a cell will

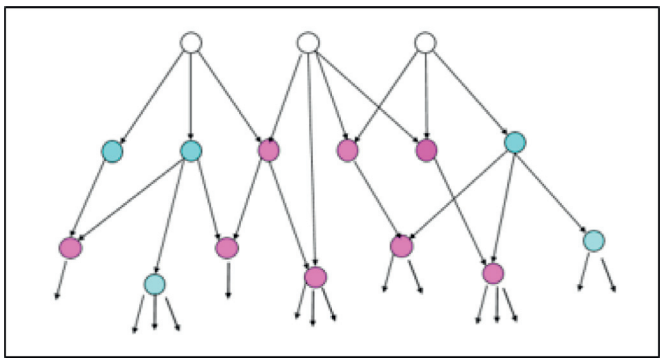

FIGURE 10.9

Directed acyclic graph. White, gray and black cells have 0, 1 or 2 ancestors, respectively.

be updated only when all its ancestors are updated. The purpose of the parallel code is to update the complete graph.

The only modification introduced in the TBB example is in the nature of the data items contained in the cells and their update operations, which has absolutely no impact on the real issue—parallel update of the graph—addressed by the example. In the original TBB code, cells contain 24×24 matrices, and update operations are matrix operations involving the matrices in ancestor cells. The only objective is to provide some relevant work to do to the parallel tasks. In our case, we adopted our traditional trick for providing unbalanced work for a task: cells contain a double target value, and update operations repeatedly call a random number generator until they get a value, close within a given precision, to a target to be specified below. The task workload is not uniform, and the amount of work can be increased by increasing the precision.

Updating the cells

The update operation itself is not critical for the discussion that follows. The real issue here is a sophisticated task synchronization pattern to implement a parallel graph traversal. The purpose of the update operation is to introduce some reasonable amount of computation in each task, in order to test the parallel efficiency of the parallel algorithm. This is the only place in which the original Intel code is modified. In the original TBB code, cells contain 24×24 matrices, and update operations are matrix operations involving the matrices in ancestor cells. Here, we adopted our traditional trick for providing unbalanced work for a task: cells contain a double target value, and update operations repeatedly call a random number generator until they get a value, close within a given precision to a target specified below. The task workload is not uniform, and the amount of work can be increased by increasing the precision.

- Each task in the parallel code will dispose of a private random number generator in [0, 1].
- Cells with no ancestors are initialized by a simple call to the generator.
- Cells with one ancestor: let a be the value of the ancestor data item. Tasks keep calling the generator until a number close to (1-a) is obtained, with a given precision ϵ.
- Cells with two ancestors: let a1 and a2 be the values of the ancestor data items. Tasks keep calling the generator until a number close to (a1+a2)/2 is obtained, with a given precision ϵ.
- The precision ϵ can be used to control the average computational workload of the tasks.

Cell and graph classes

The source file Graph.h contains the original Intel code with the definition of the Cell and Graph classes. File VGraph.h, used in this example, incorporates the minor modifications related to the replacement of matrix data items by random doubles. The declaration of the Cell class is given below:

```
class Cell
    {
    public:

    double  value;                  // cell data
    int ref_count;                  // number of ancestors not yet ready
    int n_ancestors;                // number of ancestors

    Cell*   input[2];               // ancestors of this cell
```

```
    std::vector<Cell*> successor;  // set of successors

    void update();                 // cell update function
    };
```

LISTING 10.18

Cell class

Some comments on Cell class members:

- ref_count is an integer that counts initially the number of ancestors. This integer is decreased every time an ancestor is updated. When it reaches zero, the Cell object knows that its update() function can be called.
- ancestors also registers the number of ancestors. It used by update() to select the update operation to be performed. Indeed, ref_count cannot be used for this purpose, because this integer value is decreased during the computation, as explained above.
- input[2] is an array that store pointers to ancestor cells.
- successor is a dynamical vector of pointers to successor cells. As we will see below, the number of successors—that can take any value—is determined when the graph is constructed.

Constructing the directed acyclic graph

Figure 10.10 shows the construction of the directed acyclic graph, which proceeds as follows:

- The constructor Graph G creates a graph. It defines internally an empty STL vector of cells.
- The member function G.create_random_dag(N) starts by allocating an STL vector of cells of size N. Then, successive vector elements are visited. For each vector element, the number of ancestors is randomly selected. If there are ancestors, they are again randomly chosen among the preceding vector elements. Then, the cell is initialized (ref_count, input pointers) and, in the ancestor cells, a pointer to the current cell is added to the successors vector. At this point, the graph is constructed. It follows from this procedure that the number of ancestors is limited to 2, but the number of successors is arbitrary.
- G.get_root_set(std::vector<Cell*>& root_set) is another useful function. It receives as argument a reference to an empty STL vector of cell pointers. It fills this vector with the addresses of the root cells, namely, cells that have no ancestors. Obviously, cells with no ancestors are the starting point of the graph traversal algorithm.

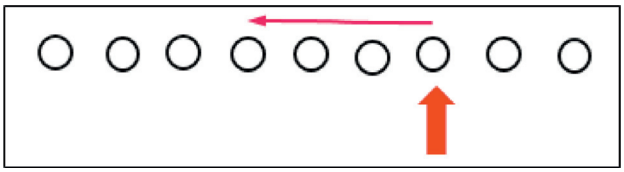

FIGURE 10.10

Constructing a directed acyclic graph.

Graph traversal code

The parallel code for this example is in source file PreorderOmp.C. The listing below shows the function that used to generate the tasks that perform the cells updates, as well as a high-level function that drives the whole graph traversal.

```
int  nTh;          //
Graph G;

void  UpdateCell(Cell *C)
   {
   C->update();

   #pragma omp atomic write    // restore reference count
   C->ref_count = C->ancestors;

   // Visit successors, and decrease their ref_count.
   // If ref_count reaches 0, launch an UpdateCell task
   // - - - - - - - - - - - - - - - - - - - - - - - - - - -
   for(size_t k=0; k<C->successor.size(); ++k)
       {
       Cell *successor = C->successor[k];
       #pragma omp atomic update           // atomic update
       - -(successor->ref_count);

       if( 0 == (successor->ref_count) )
          {
          #pragma omp task
             { UpdateCell(successor); }
          }
       }
   }

// Auxiliary function that does the job
// - - - - - - - - - - - - - - - - - - -
void ParallelPreorderTraversal(std::vector<Cell*>& root_set)
   {
   std::vector<Cell*>::iterator pos;
   #pragma omp taskgroup
      {
      for(pos=root_set.begin(); pos!=root_set.end(); pos++)
         {
         Cell *ptr = *pos;
         #pragma omp task
             { UpdateCell(ptr);}
         }
      }
   }
```

LISTING 10.19

PreorderOmp.C (partial listing)

Let us first examine the UpdateCell(Cell *C) function, which is the function executed by the parallel tasks that updates C. This function is called when the ref_count of C reaches zero. In the first place, the data update is performed. Then, the initial value of ref_count, equal to the number of ancestors, is restored, to return to the initial state in case the graph is traversed again.

Next, this function visits all the successor cells and atomically decreases their reference count, to inform them that one ancestor has been updated. If the successor ref_count reaches zero, a new OpenMP task is launched recursively to update it. The whole graph traversal is triggered by explicitly calling UpdateCell() on the root cells, with no ancestor. The remaining cells will be updated recursively. The fundamental task synchronization mechanism in this problem is the atomic update of the successors ref_count.

The ParallelPreorderTraversal() function that drives the whole graph traversal procedure takes as argument a reference to an STL vector containing the pointers to the root cells. All this function does is execute a for loop updating the root cells. However, root cells are approximately one-third of the total number of cells, and all the others are updated recursively as explained above. In order to make sure the function returns when the whole traversal is completed, the for loop must be inserted in a taskgroup code block. On exit from the code block, all the recursive tasks are terminated.

```
int  nTh;          //
Graph G;
...
int main(int argc, char **argv)
   {
   int nTh, nCells, nSwaps;
   CpuTimer T;
   ...
   ...                           // initializations
   G.create_random_dag(nNodes);  // set the acyclic graph
   std::vector<Cell*> root_set;
   G.get_root_set(root_set);

   omp_set_num_threads(nTh);     // do the traversal
   T.Start();
   // - - - - - - - - - - - - - - - - - - - - - - - - - -
   for(unsigned int trial=0; trial<nSwaps; ++trial)
      {
      #pragma omp parallel
         {
         #pragma omp single
            { ParallelPreorderTraversal(root_set); }
         }
      }
   // - - - - - - - - - - - - - - - - - - - - - - - - - -
   T.Stop();
   // Report results
   }
```

LISTING 10.20

PreorderOmp.C: the main function

Table 10.4 Graph With 4000 Nodes and 30 Successive Traversals (GNU 4.9.0 Compiler)

Graph Traversal Performances			
n_threads	**Wall**	**User**	**System**
1	58.1	58.1	0.0
2	29.16	58.16	0.0
4	14.77	58.92	0.0
8	7.65	60.94	0.0
16	4.13	66.53	0.1

The main() function is listed above. The number of threads nTh, the total number of cells in the graph nCells, and the number of successive traversals nSwaps are read from the command line. The default values are (4, 1000, and 5). The graph is traversed several times in order to achieve reasonable execution times with a small graph. Once the graph G is constructed, a parallel section is created to dispose of a team of worker threads, and a unique implicit task is created under a single directive to execute the loop on the root cells.

Table 10.4 shows the wall user and system times as the number of threads is doubled at each step. We observe the very high quality of the results for this highly unstructured, recursive problem. Perfect scaling starts to weaken between 8 and 16 threads, but the performance is still quite acceptable. Notice the absence of system activity, as well as the very modest amount of synchronization overhead, as reported by the user execution time. We have, for 16 threads, a small 10% increase of total CPU time.

Example 10: PreorderOmp.C

To compile, run make ompreorder. To execute, run ompreorder *nTh, nCells, nSwaps*. The default parameter values are 4, 1000, and 5.

10.7.3 TRACKING HOW TASKS ARE SCHEDULED

This example demonstrates that a new explicit task can be executed by another threads if available, or by the parent thread itself if nobody else is immediately available. The scenario, as shown in Figure 10.11, is described.

- A two-thread parallel region is created.
- The rank 1 thread waits for 1 s, creates a new task (in red in the figure), and continues by writing an identification message.
- The rank 0 thread is kept out of the way for a lapse of time T, by using a timer.
  - If T is much longer than 1 s, the rank 0 thread is not available to execute the new task. In this case, the scenario "b" is chosen: the rank 1 thread suspends its ongoing task, executes the new one, and finally resumes its original task.

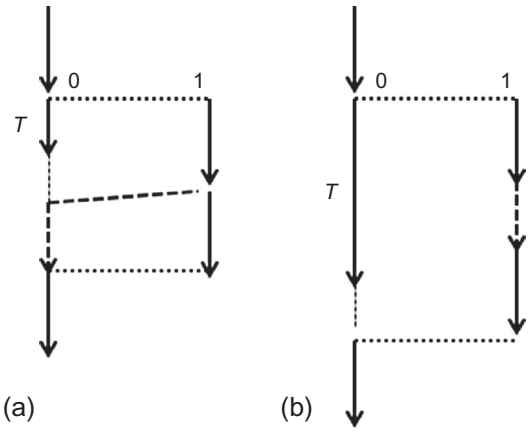

FIGURE 10.11

Example 11: scheduling of the red explicit task.

– If T is much smaller than 1 s, the rank 0 thread is available and executes the new task, according to scenario "a."

Tasks print messages indicating which thread is in charge of its execution. These scenarios are validated by the example code.

Example 11: Task2.C

To compile, run `make task2`. The number of threads is hardwired to 2. The time interval T is 500 ms by default. This can be overridden from the command line, by passing an integer value in milliseconds.

10.7.4 BARRIER AND TASKWAIT SYNCHRONIZATION

The following example shows that explicit tasks are terminated at the next (implicit or explicit) barrier. The scenario is very simple: a parallel region with N worker threads is launched. All but one immediately hit a barrier. The remaining thread spawns a long level 1 explicit task, before hitting the barrier. This child task in turn spawns another even longer level 2 explicit task. The example verifies that the N threads are released only after all the explicit tasks launched before the barrier are completed.

Example 12: Task3.C

To compile, run `make task3`. The number of threads is hardwired to 2.

The next example demonstrates task synchronization. In the previous example, the level 1 task, shorter than the level 2 one, terminated before. A taskwait directive is now added in the level 1 task, forcing it to wait for its level 2 child. The code execution verifies that this task terminates later. The source file is `Task4.C`.

Example 13: Task4.C

To compile, run make task4. The number of threads is hardwired to 2.

10.7.5 IMPLEMENTING A BARRIER AMONG TASKS

This example shows the way OpenMP 4.0 establishes dependencies among tasks, with the depend clause involving a *dependence type* and a list of shared data items on which the task depends. In Intel TBB, tasks are C++ objects and hierarchical dependencies are established when they are allocated (they can, e.g., store a pointer to another parent task). In OpenMP, tasks are implemented as code blocks attached to the task directive, and shared variables are used to establish dependencies among them. The depend clause operates as follows:

- depend*(dependence-type, list of data items)*. Dependencies are derived from the information conveyed by the dependence type and the list of data items.
- Dependence types can be in, out, or inout:
  - in dependence-type: the generated task depends on all previously generated sibling tasks that reference at least one of the items in an out or inout dependence-type list.
  - out and inout dependence-types: the generated task depends on all previously generated sibling tasks that reference at least one of the items in an in, out, or inout dependence-type list.

These rules are indeed very flexible. They allow programmers to establish flow dependencies (some tasks executing after others have terminated), anti-dependencies (some tasks executing before other tasks are scheduled to run), or more complex dependencies in which tasks that need the results of other tasks are ordered in order to guarantee the integrity of the underlying algorithms.

We start with a simple example of flow dependency. Let us go back to the computation of the area under a curve, but this time the computation is not recursive: the integration domain is split by hand in Ntk equal subdomains, and an explicit task is spawned for each subdomain. Partial results coming from the computing tasks are collected in a mutex-protected global variable.

A parallel region with Nth threads is first created. As discussed in Chapter 3, the area could be easily computed in the SPMD way, by assigning one parallel task to each thread. However, for the purposes of this example, a number of parallel tasks different from Nth will be used. Again, this is easily done: using the single directive, only one implicit task is activated, which spawns the Ntk computing tasks and waits for their termination with a taskwait directive. After that, a global variable where partial results have been accumulated holds the final value of the computation.

In the example that follows, a small change is introduced in this scenario by imagining that, after the area is computed, something must be done with the area result *before* the parallel job terminates, that is, before the initial implicit task takes over after the taskwait directive. An extra task is therefore launched in the parallel job, and dependencies are used to make sure it is scheduled after the parallel area computation terminates. *This extra task is in fact establishing a barrier among the computing tasks*.

Here is the listing of the example, in source file Depend.C. The function integrated is the good old function $4.0/(1.0 + x^2)$, whose area in the interval [0, 1] is π. The subsidiary task that executes after the parallel computation just prints some information to stdout.

```
#define EPSILON  0.0000001

using namespace std;

double result;    // global variable
int    synch;     // used for synchronization

// The function to be integrated
// ---------------------
double my_fct(double a)
   {
   double retval;
   retval = 4.0 / (1.0+a*a);
   return retval;
   }

// Generic, sequential integration routine
// Integrates func(x) in [a, b] with precision eps
// -------------------------*/
double Area(double a, double b, double (*func)(double),
            double eps=EPSILON);

// The main function
// -----------
int main (int argc, char *argv[])
   {
   int nTh = 2;          // default value, number of threads
   int nTk = 4;          // default value, number of tasks
   result = 0.0;
   synch  = 0;

   // override from command line
   // -------------
   if(argc==2) nTh = atof(argv[1]);
   if(argc==3)
       {
       nTh = atof(argv[1]);
       nTk = atof(argv[2]);
       }

   // Set the number of threads
   // -------------
   omp_set_num_threads(nTh);

   // -------------------------
   // Create the parallel region, and activate only
```

Continued

```
// one implicit task.
// – – – – – – – – – – – – – – – – – – – – – –
#pragma omp parallel
   {
   #pragma omp single
      {
      // Here starts the implicit task
      // Spawn the nTk computing tasks
      // – – – – – – – – – – – – – – –
      for(int n=0; n<nTk; ++n)
         {
         #pragma omp task depend(in: synch)
            {
            double a = n/nTk;    // global range is [0, 1]
            double b = (n+1)/nTk;
            double res = Area(a, b, my_fct, EPSILON);
            synch++
            #pragma omp critical
               result += res;
            }
         }
      // Next, spawn the printing task
      // – – – – – – – – – – – – – – –
      #pragma omp task depend(out: synch)
         {
         synch++;
         cout << "\n Area result from inner task is " << result;
              << "\n Value of synch is " << synch << endl;
         }
      #pragma omp taskwait
      }
   }
return 0;
}
```

LISTING 10.21

Depend.C

The variable result is used to accumulate partial results, and the variable synch is used to establish the dependencies among tasks. The code establishes some default values for Nth and Ntk, and reads eventual new values from the command line. Then, the parallel section is created, and the single implicit task launches first the Ntk computing tasks, followed by the additional printing task. Note that:

- The first Ntk computing tasks have an in dependence on synch. Since there is no previous sibling task with out or inout dependency on synch, these tasks are totally independent and can execute concurrently.
- The printing task has an out dependency on synch. It is therefore dependent on all the previous computing tasks and will execute after them.
- The printing task prints the result for the area. If everything executed as expected, the result must be π.

- All tasks increase the value of sinch (this is not really needed, as we discuss next). The value printed by the printing task must be Ntk+1.

A few further comments are in order. First, notice that referencing the synchronization variable inside the tasks is not required. The correct dependencies are established if synch is referenced only in the depend clauses. This can easily be checked by suppressing the synch++ instruction in the dynamic tasks. Second, ordinary variables can also be used to establish dependencies. In the example above, the synch variable can be completely disposed of and result can be used instead in the dependency clauses. Third, a task can have more than one depend clause with different dependence types for different variables. I strongly recommend looking at the matrix multiplication example proposed in the official OpenMP examples document [29].

Example 14: Depend.C

To compile, run make depend. The number of threads is hardwired to 2.

10.7.6 EXAMPLES INVOLVING RECURSIVE ALGORITHMS

We turn next to the discussion of the OpenMP version of two recursive examples: computation of the area under a curve and the parallel version of the quicksort algorithm.

Consider first the recursive computation of the area under a curve. The algorithm is implemented as follows: if the range of integration is bigger than a given granularity, a task splits the domain into two halves and launches two child tasks for each subdomain. Otherwise, when the integration range is finally smaller than the granularity, the task computes the area and returns in some way the partial result. Two programming styles are possible:

- Blocking on children. Tasks spawning two children wait for their return value. This is the natural implementation in OpenMP 3.1, using the taskwait directive.
- Nonblocking style. Tasks that split the domain do not wait for return values, they terminate immediately after spawning the two child tasks. Final tasks that do the real computational work accumulate partial results in a mutex-protected global variable. This style is only possible with the OpenMP 4.0 taskgroup construct.

The next two sections develop in detail these two programming options.

Area under a curve, blocking style

The basic computational tool in this example is a library function Area(a, b, fct, eps)—taken from Numerical Recipes in C [16]—that computes the area under the function fct(x) in the interval [a, b] with precision eps. Then, a recursive task function is constructed for the problem at hand. If the integration domain [a, b] is larger than a given granularity G, the task splits it into two halves and launches two identical recursive tasks for each subdomain. When the domain size is finally smaller than the granularity, the area is computed and the results are returned to the parent tasks in the following way.

Figure 10.12 shows the code organization in this example. A team of N worker threads is created in a parallel region, but only one initial implicit task A is activated, under the single directive. This task recursively generates all the other tasks in the problem. In Figure 10.12, an initial domain of integration of size 1 and a granularity of 0.25 are assumed. Task A spawns tasks B1 and B2, which operate on domains of size 0.5, and blocks on a taskwait directive, waiting for their return values. Tasks B,

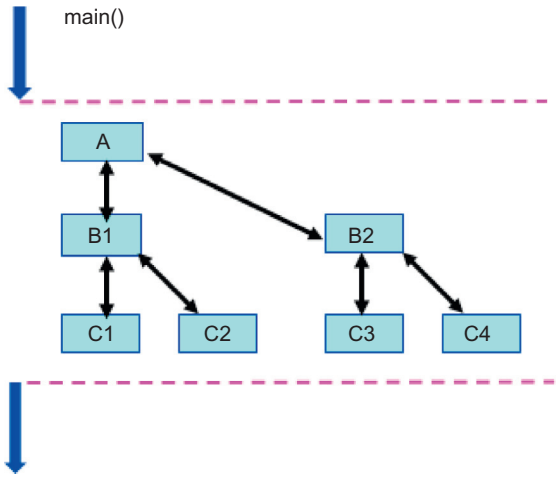

FIGURE 10.12

Area computation, blocking style.

in turn, spawn the four-level C tasks and wait for their return. Tasks at level C stop the recursive domain decomposition, compute the partial results, and return their values to the parent tasks. Here is the listing of the recursive task function:

```
double AreaRec(double a, double b, double L,
            double (*func)(double), double eps=EPSILON)
   {
   double x, y, medval, retval;
   // identification messages here

   if( fabs(b–a) > L)
      {
      medval = 0.5*(b–a);

      #pragma omp task untied shared(x)
         { x = AreaRec( a, a+medval, L, func); }
      #pragma omp task untied shared(y)
         { y = AreaRec( a+medval, b, L, func); }

      #pragma omp taskwait
      retval = x+y;
      }
   else retval = Area(a, b, func, eps);
   return retval;
   }
```

LISTING 10.22

Recursive task function for the area computation

This code deserves a careful discussion to illustrate, once again, how clauses are used to transfer information between parent and child tasks. Note that *the recursive area function returns to its parent task the value of the area for the subdomain passed as argument*. If the task is a final compute task in the task tree hierarchy that *does not* spawns other child tasks, the situation is clear: the task function just returns its computed value. The point that needs further discussion is, what happens when the executing task needs to recover return values from its two child tasks?

In the code above, there are several data items in the enclosing environment of the parent task, a, b, L, medval, func, that are needed in the child tasks. They are by default firstprivate, and they are captured by value by the child task functions, which is what we want.

The parent task function declares in addition two local variables x, y, which will receive the return values of the two subdomain areas if the problem is split into two halves. These variables, being declared shared, are passed by reference to the child tasks. When the child tasks terminate, they hold the expected return values.

Finally, the parent task returns to its parent (x+y), after the taskwait directive, needed to ensure that all children have terminated and that x and y hold the correct return values.

The complete example code, in file AreaOmp.C, includes an identification message every time the AreaRec() function is entered, so that the task generation and execution process can be followed in the output. The listing below shows the way the parallel execution is triggered in the function main():

```
int main (int argc, char *argv[])
   {
   int n, nTh;
   double result;
   double A = 0.0;
   double B = 1.0;
   double G = (B–A)/5;

   // get nTh from command line
   omp_set_num_threads(nTh);
   #pragma omp parallel
      {
      #pragma omp single
         {
         result = AreaRec(A, B, G, my_fct);
         }
      }

   cout << "\n result = " << result << endl;
   return 0;
   }
```

LISTING 10.23

Driver for the recursive area computation

There are in this code local variables in the main() function like result, A, B, and G that are referenced in the parallel region and in the task code block. This code works "as such" because, for the parallel directive, variables are shared by default in C-C++. We do not need to specify their scope. Arguments

are passed to AreaRec by address, and this is the reason the local variable result will ultimately hold the correct return value at the end of the parallel section. Try adding the private(result) clause to the parallel directive, verify that in this case the result is 0, and convince yourself that this is what you should expect.

Example 15: AreaOmp.C

To compile, run make areaomp. Computational parameters are hardwired in the code (see the listing above). The number of threads is read from the command line (the default is 2).

Area under a curve, nonblocking style

This example shows an alternative strategy for performing the recursive area computation, based on the OpenMP 4.0 taskgroup directive. Figure 10.13 shows the code organization in this example. Again, an N worker threads team is created in a parallel section, and only one initial implicit task A is activated under the single directive. Again, this task recursively generates all the other tasks in the problem. But now, there are no return values to parent tasks. If a task gets an assignment on a domain bigger than the granularity, *it spawns two child tasks for each half subdomain, and terminates*. At the bottom of the task tree, the C tasks compute their partial results and accumulate them in a mutex-protected global double variable.

Here is the listing of the modified recursive task function:

```
double AreaRec(double a, double b, double L,
               double (*func)(double), double eps=EPSILON)
```

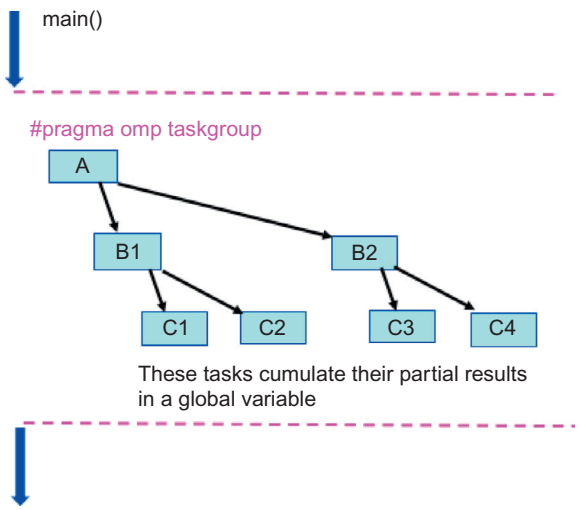

FIGURE 10.13

Area computation, nonblocking style.

```
{
medval, retval;
// identification messages here

if( fabs(b—a) > L)
   {
   // spawn two tasks and terminate
   // — — — — — — — — — — — — — —
   medval = 0.5*(b—a);

   #pragma omp task untied shared(x)
      { AreaRec( a, a+medval, L, func); }
   #pragma omp task untied shared(y)
      { AreaRec( a+medval, b, L, func); }
   }
else
   {
   Timer T;
   T.Wait(1500);
   retval = Area(a, b, func, eps);
   RD.Accumulate(retval);    // global Reduction<double>
   }
}
```

LISTING 10.24

Modified recursive task function for the area computation

In this listing, RD is a global Reduction<double> object used to accumulate the partial results provided by the worker threads. A timer is used to delay the worker threads performing the partial area computations, to explicitly show that the task A effectively waits for all its descendants under a taskgroup construct. Now let us look at the taskgroup construct by examining the main()function code:

```
Reduction<double> RD;
...
int main (int argc, char *argv[])
   {
   // same as before
   // ...
   #pragma omp parallel
      {
      #pragma omp single
         {
         // Here starts task A
         // — — — — — — — — —
         #pragma omp taskgroup
```

Continued

```
            {
            AreaRec(A, B, G, my_fct);
            // This task A could do, if needed, other things here,
            // before waiting for descendants at the end of this
            // code block.
            }
         }
      }

   cout << "\n result = " << RD.Data() << endl;
   return 0;
   }
```

LISTING 10.25

Driver for the nonblocking area computation

As before, only one implicit task A is activated in the parallel region. Inside this task, a taskgroup construct is inserted, before calling the recursive task function, that returns immediately after launching the recursive child task generation process. At this point, task A could do other useful things if needed before waiting for the termination of all its descendants at the end of the taskgroup region. Note the difference with the taskwait directive, which blocks the tasks at the place where the directive is inserted.

Example 16: AreaOmpBis.C

To compile, run make areabis. Computational parameters are hardwired in the code (see the listing above). The number of threads is read from the command line (the default is 2).

10.7.7 PARALLEL QUICKSORT ALGORITHM

Quicksort is a recursive algorithm that operates on containers of any type for which the LessThan operation is defined. The container is sorted in increasing order. The recursive sort algorithm is well adapted for recursive task parallelism. The sequential version is reviewed first, focusing on the type of integer array. The algorithm, however, manipulates pointers and works for arrays of any type equipped with the $<$ operation.

A function qsort is defined next that sorts an integer array in the range [*beg*, *end*) where *beg* is an integer pointer to the first element of the array, and *end* is a past the end pointer that points to the memory position where a new integer could be inserted. These are the standard STL conventions for describing container ranges. Remember that C-C++ accepts past the end pointers as long as they are only used in pointer arithmetic and never referenced (because they are not pointing to a valid container element). The function qsort() listed below will be called by recursive tasks whenever they have real work to perform, as was the case for the Area() function in the previous example. When examining the qsort() code, keep Figure 10.14 in mind.

```
void qsort(int *beg, int *end)
   {
   if(beg != end)
```

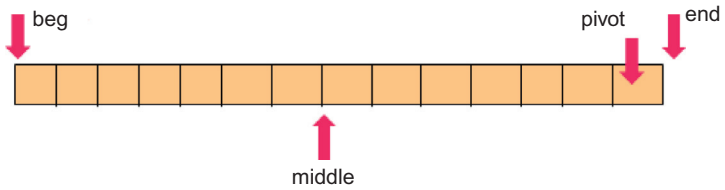

FIGURE 10.14

Operation of the quicksort algorithm.

```
        {
        --end;
        int pivot = *end;
        int *middle = partition(beg, end, LessThan(pivot));
        swap(*end, *middle);
        qsort(beg, middle);
        qsort(++middle, ++end);
        }
    }
```

LISTING 10.26

Sequential quicksort algorithm

Here is how way this code operates:

- If beg != end, this means, with our range conventions, the container is not empty and we can proceed.
- The last container element is then read by decreasing the end pointer and referencing it. Now end has been moved one step to the left and points to the last array element, whose value is pivot.
- Our intention next is to reorganize the container by placing the last integer in the array, pivot, in its final position. This is done by the partition STL algorithm. Given a range and condition, this algorithm reorders the container, putting at its head all the elements that satisfy the condition. The algorithm then returns a pointer middle to the position of the first element that *does not* satisfy the condition.
- In our case, the condition is LessThan(pivot). This is a function object defined in the source file that will be used by the algorithm to compare all the integers in the array to pivot. When the partition algorithm returns, all the integers less than pivot are stored before the position pointed by middle.
- It is now obvious that middle is the final position of pivot. The integer sitting at this position, that is, *middle, is swapped with pivot, which is sitting at *end.
- Next, the same quicksort algorithm is recursively applied to the array to the left of middle, with range [*beg*, *middle*), and to the array right of middle, with range [*middle* + +, *end* + +). In this second case, middle is incremented to jump over pivot, which is already at its place, and end is increased to place it as the past the end pointer.

Note that middle is not necessarily at the middle of the initial container. The sizes of the two leftover containers can be anything, which leads us to slightly modify the recursion strategy of the previous

example. The container ranges will be passed to the child tasks, very much as the integration domain ranges were passed in the recursive area computation. This example is similar in spirit to the previous one, and only reports the parts of the code that are different.

Recursive task function

A recursive task function is now constructed to be used in the OpenMP parallel version. This function receives as arguments the range pointers of the domain to be sorted. The task function starts developing the quicksort algorithm until pivot is at the right place. Then, it enters the recursive part and successively examines the two domains to the left and to the right of pivot. If the subarray size is smaller than the granularity G the task function calls directly qsort to complete the sorting of the subdomain. Otherwise, it spawns a recursive task to sort the subdomain. In our example code, there is a message printed to the screen every time a child task is spawned.

```
void Pqsort(int *a, int *b)
  {
  cout << "\n Inside Pqsort" << endl;
  – – b;
  int pivot = *b;
  int *middle = partition(a, b, LessThan(pivot));
  swap(*b, *middle);

  if( (middle–a) < G) qsort(a, middle);
  else
     {
     #pragma omp task untied
        {
        cout << "Child task 1 submitted " << endl;
        Pqsort(a, middle);
        }
     }

  ++middle;
  ++b;
  if( (b–middle) < G) qsort(middle, b);
  else
     {
     #pragma omp task untied
        {
        cout << "Child task 2 submitted " << endl;
        Pqsort(middle, b);
        }
     }
  }
```

LISTING 10.27

Parallel quicksort task function

This function starts by placing at its final position the last element of the container. Then the function repeats itself for the two subranges before and after this element. If the residual container sizes are larger than the granularity, a new recursive task is launched. Otherwise, there is a direct call to the sequential qsort() function.

This algorithm performs an *in place* modification of the container, and no return values need to be recovered. Notice that different threads are simultaneously accessing and modifying a shared container, a situation that in general requires mutual exclusion. However, it is not difficult to convince oneself that in this algorithm different tasks—possibly executed by different threads—are modifying in place *nonoverlapping subranges of the container*. Therefore, this algorithm does not require mutual exclusion for thread safety.

Another detail to be observed: a and middle are integer pointers so, according to the pointer arithmetic, (middle-a) with a pointing to the first element and middle as past the end is the number of elements in the left container. The same applies to the right container.

```
int main(int argc, char* argv[])
   {
   int n = 100000;
   G = 20000;
   if(argc>1) n=atoi(argv[1]);

   int *a = new int[n];
   for(int i=0; i<n; i++) a[i]=i;
   random_shuffle(a, a+n);

   #pragma omp parallel num_threads(2)
      {
      #pragma omp single
         Pqsort(a, a+n);
      }

   CheckSort(a, n);
   return 0;
   }
```

LISTING 10.28

Parallel quicksort driver

As far as the main() function is concerned, all steps are by now standard: data input, pool initialization, and submission of a single task parallel job, which triggers the recursive operation. Finally, main() checks if the final array is properly ordered.

Example 17: QsortOmp.C

To compile, run make pqsort. The number of threads is read from the command line (the default is 2).

With the initial data n=100000, G=20000, and C=20, the code runs smoothly.

10.8 TASK BEST PRACTICES

It is useful to draw attention to some pitfalls that may eventually occur in a task-centric environment. They arise from the use of thread-specific utilities in an environment in which threads may be servicing different tasks.

10.8.1 LOCKS AND UNTIED TASKS

As was stated before mutual exclusion can be safely used in a task-centric environment. This is in general true, but OpenMP has a special feature that may, in some particular contexts that are admittedly very rare, require particular care.

In general, the mapping of tasks to threads in a task-centric environment is *nonpreemptive*. This means a thread that starts executing a task will keep ownership of the task until its termination. The running thread may suspend the task execution to serve other tasks, but the task execution will always be resumed by the same owner thread. This is the case of the thread pools discussed in the next chapters: the TBB programming environment and the NPool class. OpenMP, instead, supports *untied* tasks; namely, tasks that, when suspended, can be resumed by another thread. This feature is obviously introduced to optimize task scheduling.

In the presence of untied tasks, mutex locks require special care. When a mutex is locked in a task, it is the executing thread that takes ownership of the mutex. Let us now imagine a scenario in which an untied task creates a critical section and spawns another task *from inside the critical section* (a highly unusual but legitimate thing to do). Now, OpenMP allows a task to be suspended at certain places called task synchronization points, and task spawn is one of them. Therefore, the parent task may be suspended and, being untied, resumed later in the middle of the critical section by another thread. At the end of the critical secretion, *the new thread will try to unlock a mutex that it does not owns*, and the code deadlocks.

This scenario is very unlikely but possible. It may also occur by accident when using a programming style that does not encapsulate tasks and keeps nesting directives and code blocks. The lesson here is that care should be exercised with critical sections and locks when using explicit untied tasks.

10.8.2 THREADPRIVATE STORAGE AND TASKS

As discussed in Chapter 4 threadprivate variables are static variables—they live across function calls—that are stored on a *per thread* basis. A function accessing a threadprivate variable recovers at each function call the value belonging to the executing thread. Threadprivate variables were used in Chapter 4 to enforce thread safety when using a global random number generator accessed by several threads.

Let us now imagine that explicit tasks are used to run a parallel Monte-Carlo calculation similar to the computation of π discussed in Chapters 3 and 4. The basic idea is to have *each task* participating in the parallel treatment accessing a totally independent random number generator. In Chapter 4, a thread-safe, global generator function was used, incorporating a threadprivate seed initialized in a thread-dependent way. This code organization implements consistent, independent random suites *for each thread*. Now, in the absence of a one-to-one mapping of tasks to threads, thread safety is not so obvious.

Indeed, let us now imagine that N tasks are sharing the work of this Monte-Carlo computation. Three cases are possible:

- There are at least as many worker threads in the pool as tasks, and tasks are tied. In this case, nothing is wrong, each task is uniquely mapped to a thread, and the code is thread-safe. Results are reproducible.
- Tasks are tied, but there are fewer worker threads than tasks. There is therefore no unique mapping between threads and tasks, as some threads will execute more than one single task. In this case, *different tasks will be accessing the same threadprivate variable*. Thread safety is broken, because the results will depend on the particular way in which tasks are scheduled.
- Tasks are untied. Even if there are more threads than tasks, thread safety cannot be guaranteed if there is other activity in the parallel region that may force the suspension of tasks.

In all these cases, the code will always run, but thread safety is broken. Therefore, we come to the same conclusion as before: threadprivate storage should be used with care in the presence of explicit tasks. For the particular problem discussed here, the solution of using local C++ objects as random number generators is better and fully thread-safe in all cases, because a consistent random number generator is available *per task, not per thread* and there is no thread-safety problem.

Notice that this discussion applies to any thread local storage utility, in any programming environment.

10.9 CANCELLATION OF PARALLEL CONSTRUCTS

OpenMP 4.0 incorporates a new service enabling the cancellation of parallel constructs. This is a very useful extension of OpenMP. An example was proposed in Chapter 3, simulating a database search, which demonstrated the interest of canceling thread activities using a thread cancellation service incorporated in the SPool utility. In this example, a team of worker threads is launched to concurrently explore different sectors of a data set, and the first one that finds the target data terminates gracefully after forcing the termination of the other members of the team. The same strategy can be used, for example, to cancel a parallel job if an error occurs.

OpenMP 4.0 introduces a new feature, focused on the cancellation of a parallel treatment. This service, easy to use, is based on two directives: one to request the cancellation, and the other to insert checks to detect if a cancellation has been requested.

10.9.1 CANCEL DIRECTIVE

This is a standalone directive, with the following form:

```
...
#pragma omp cancel [construct—type] [if(expression)]
...
```

LISTING 10.29

Cancel directive

This directive must specify the type of parallel construct (parallel, sections, for, taskgroup) whose cancellation is requested. When this directive is encountered, the cancellation of the innermost enclosing region of the type requested is activated. An optional if(scalar-expression) clause is possible, in which case the cancellation request is ignored if the scalar expression evaluates to false.

The cancel directive is operational only if the *cancel-var* ICV is true, otherwise it is ignored. This ICV is set by the OMP_CANCELLATION environment variable, and a call to the omp_get_cancellation() library function returns its value. However, there is no way to modify it at runtime. Therefore, codes activating the cancellation service must necessarily run with OMP_CANCELLATION set to true.

When a thread hits a cancel directive, it signals to the runtime system that the cancellation request must be implemented on the other tasks involved in the target parallel construct and terminates the current task.

10.9.2 CHECKING FOR CANCELLATION REQUESTS

This is, again, a standalone directive, with the following form:

```
...
#pragma omp cancellation point [construct–type]
...
```

LISTING 10.30

Cancel directive

When cancellation is enabled, threads check in all cases for cancellation requests at all implicit or explicit barriers. This directive introduces additional user-defined points where cancellation requests are checked. When a thread encounters one of these points, it checks for eventual cancellation activation by a cancel request. If this is the case, the thread continues execution at the end of the canceled parallel construct in the way we discuss next.

It is interesting to observe that in the OpenMP cancellation mechanism it is the parallel tasks, not the running threads, that are canceled. Consider, for example, the cancellation of a taskgroup, where a task driving this construct waits for the termination of all its descendants. The task that encounters the "cancel taskgroup" construct jumps to its termination point. Any other task that would normally run to completion, runs until a cancellation point is reached and then it jumps to the end of its taskgroup region (i.e., it terminates). Finally, any task that has not begun execution is discarded, which implies its completion. An example of taskgroup cancellation will soon follow.

When a parallel construct is canceled, the master thread resumes execution at the end of the canceled region after a cancellation point is encountered. Any explicit tasks that may have been created by a task construct and their descendants are canceled according to the taskgroup semantics discussed above.

In the OpenMP environment, programmers are responsible for the correct management of the resources allocated to the parallel tasks. If a task is canceled while holding a locked mutex, the mutex will remain locked and nobody else will be able to lock it again. In C++, it is possible to make use of the scoped locking utilities proposed by C++11 or TBB to avoid this problem, because a canceled task terminates normally, and the mutex is automatically released. The Pthreads thread cancellation service has special interfaces to enforce automatic restitution of resources to the system in the C language context.

The examples that follow reconsider the database search example of Chapter 3 in the OpenMP 4.0 context. The first one, in source file DbOmp4-A.C, is exactly the example already discussed in Chapter 3, but now using the parallel region cancellation mechanism. In the second example, in source file DbOmp4-B.C, the same code is reorganized by using a taskgroup construct instead of a parallel region, and the cancellation of the taskgroup construct is demonstrated.

10.9.3 CANCELING A PARALLEL REGION

As in Chapter 3, a parallel region is launched where tasks keep calling a local random number generator trying to get a value equal to a given target value, with a precision EPS. The first task that hits the target cancels the parallel region. Here is the listing of the task function executed by the worker threads:

```
const double EPS = 0.00000001;
const double target = 0.58248921;
Data D;

// The workers thread function
// – – – – – – – – – – – – – ––
void worker_fct()
    {
    double d;
    int rank = omp_get_thread_num();
    Rand R(999*(rank+1));
    std::cout << "\n Thread " << rank << " starts" << std::endl;

    while(1)       // infinite loop
       {
       #pragma omp cancellation point parallel    // check for cancellation
       d = R.draw();
       if(fabs(d–target)<EPS)
          {
          D.d = d;
          D.rank = rank;
          #pragma omp cancel parallel
          }
       }
    }
```

LISTING 10.31

DbOmp4-A.C

Tasks enter an infinite loop in which they first check if a cancellation has been requested. Then, they get a random number from the generator. If the target is found, the task writes the search result in a global structure D and cancels the parallel region.

Example 18.A: DbOmp4-A.C

To compile, run make dbomp4-a. The number of threads is read from the command line (the default is 2).

10.9.4 CANCELING A TASKGROUP REGION

Next, the previous computation is reformulated by using explicit, dynamically generated tasks rather than the implicit tasks associated with a parallel region. The basic idea is to recursively generate a set of tasks that perform the database search. One initial task is launched inside a parallel region that spawns two child tasks and terminates. The child tasks will in turn spawn two children each and terminate, and so on, until the recursive task generation process comes to an end. The final tasks execute the database search code. To control the task generation process, an integer deepness is read from the command line, which determines the number of recursive task generation steps performed before the actual computation starts. Here is the listing of the recursive task function used in this example:

```
const double EPS = 0.00000001;
const double target = 0.58248921;
Data D;
SafeCounter SC;
int   deepness;

void task_fct(int deep)
    {
    int rank;
    double d;
    rank = SC.Next();
    std::cout << "\n Task " << rank << " starts" << std::endl;

    if(deep < deepness)
       {
       // launch two child tasks
       #pragma omp task
       task_fct(deep+1);
       #pragma omp task
       task_fct(deep+1);
       }
    else
       {
       Rand R(999*rank);
       // perform the search
       while(1)
          {
          #pragma omp cancellation point
          d = R.draw();
          if(fabs(d-target)<EPS)
             {
             D.d = d;
```

```
            D.rank = rank;
            #pragma omp cancel taskgroup
            }
         }
      }
   }
```

LISTING 10.32

DbOmp4-B.C: the task function

Note the following issues in the task function above:

- The task function receives as argument an integer deep.
- If deep<deepness, the task terminates after spawning two child tasks with argument (deep+1).
- If deep==deepness, the search is performed, after creating a local random number generator with a task-dependent initial seed based on the task rank.
- Since we need to identify *tasks, not threads*, a unique rank value for each task is obtained from our SafeCounter utility.
- The while loop that performs the search is identical to the one used before in a parallel region, except that the cancellation protocol refers now to a taskgroup.

The listing below shows the way the parallel computation is launched from the main() function:

```
int main(int argc, char **argv)
    {
    int nTh;

    // get number of threads and deepness from command line:
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    nTh = 2;
    deepness = 2:
    if(argc==2) nTh = atoi(argv[1]);
    if(argc==3)
        {
        nTh = atoi(argv[1]);
        deepness = atoi(argv[2]);
        }

    omp_set_num_threads(nTh);
    #pragma omp parallel
       {
       #pragma omp single
          {
          #pragma omp taskgroup
             {
             #pragma omp task
             task_fct(0);
```

Continued

```
            }
        }
    }

std::cout << "\n Thread " << D.rank << " found value "
          << D.d << std::endl;
return 0;
}
```

LISTING 10.33

DbOmp4-B.C: the main function

First, default values for the number of threads and the deepness are defined. They can be overridden from the command line, by passing either the number of threads or the number of threads and the deepness. Then, a parallel region is established with the given number of threads in order to set up a pool of worker threads to execute the dynamically generated tasks that will follow.

Because of the single directive that follows, the parallel region activates only one initial implicit task, which executes the single block code. This initial implicit task immediately encounters a taskgroup code block. Inside the taskgroup region, it then launches an initial explicit task with deep argument equal to 0, and waits at the end of the taskgroup region for the termination of all the generated descendant tasks.

Example 18.B: DbOmp4-B.C

To compile, run make dbomp4-b. The number of threads as well as the deepness can be modified from the command line (the default is 2 threads and deepness 2). By increasing the deepness, the number of active tasks is increased.

10.10 OFFLOADING CODE BLOCKS TO ACCELERATORS

One of the major evolutions in OpenMP 4.0 is support for heterogeneous computing platforms involving different computational devices: standard CPUs, associated with GPU accelerators or co-processors like the Intel Xeon Phi platform. The OpenMP code is compiled and launched in a host device, and a set of device directives can be used to offload code blocks for execution in partner accelerator devices. These directives manage both the offloading of executable code blocks as well as the necessary data movements between the host and external devices. Given the current impact of accelerator devices in high-performance computing, this is a major evolutionary step taken by OpenMP.

At the time of writing this utility is not yet fully implemented in the available 4.0 compliant compilers (like GNU 4.9). The code is correctly compiled, but the directives are ignored and the code blocks are executed in the host device. The discussion that follows describes the basic code offloading protocols adopted by OpenMP 4.0, which are simple, compact, and easy to understand. Their impact will be determined by the quality of the implementations, when they do become available.

10.10.1 TARGET CONSTRUCT

The fundamental offloading tool in OpenMP 4.0 is the target directive that specifies that the following code block must be executed in a target device characterized by an integer identifier.

```
pragma omp target device(int) map(map-type : list) if(scalar expression)
  {
  // structured block, executed in
  // target device
  }
```

LISTING 10.34

Target directive

This directive accepts three optional clauses: device, map, if, and it operates as follows:

- The executable code block is executed in the device identified by the device clause. If the clause is absent and there are no available devices, the executable code block is executed in the host.
- Note that *there is no concurrent activity between host and target devices*: the encountering task in the host *waits* for the device to complete the target region. This is likely to evolve in future OpenMP releases.
- If the scalar expression passed in the if clause evaluates to false, the executable code block is executed in the host device. This clause operates as all the similar if clauses in OpenMP: a logical test that maintains or invalidates the directive.
- The map clause controls the way data is moved between host and target device. This clause is useful to reduce to a strict minimum the data transfers between target and host. It involves a type of map action and a list of associated data items that are to be mapped according to the declared map type. Map types are:
  - *alloc*: on entry to the target region each list item has an undefined initial value. In other words, they can be directly allocated in the target device.
  - *to*: each list element is initialized in the target device with the original item value in the host. Data is copied from host to target.
  - *from*: on exit from the target device, list item values are assigned to the original items in the host. Data is copied from target to host.
  - *tofrom*: data is copied from host to target on entry, and from target to host on exit. This is the default behavior in the absence of a map directive: any variable referenced in a target construct that is not declared in the construct is implicitly treated as if it had appeared in a map clause with a tofrom map type.

It may happen that, in a target section, different data items need to be mapped with different map types. In this case, several different map directives may be used in the target construct. The official examples document [29] proposes a few clear examples for mapping arrays.

10.10.2 TARGET DATA CONSTRUCT

In the target construct previously discussed, a map strategy must be specified every time a computational kernel is offloaded for execution in a given target device. Always focused on minimizing as much as possible data transfers between host and target device, OpenMP offers the possibility of mapping data in a way that can be useful to several successive computational kernel offloads. This is the purpose of the target data directive, which only defines a map action without an accompanying executable code offload.

```
pragma omp target data device(int) map(map–type : list) if(scalar expr)
  {
  // structured block starts execution in host
  ...
  #pragma omp target [clauses]
     {
     // offloaded to target
     }
  // back to host
  ..
  #pragma omp target [clauses]
     {
     // offloaded to target
     }
  ...
  }
```

LISTING 10.35

Target data directive

As shown in the listing above, all the target data directive does is prepare a data environment in the target device. Then, the encountering task continues execution of the code block that follows in the host, until different target directives that force an offload are encountered. These directives benefit from the target data environment set up by the enclosing target data region. They may include additional map clauses referencing data items not included in the enclosing target data directive. Again, the official examples document proposes a few clear examples of the target data construct [29].

10.10.3 TEAMS AND DISTRIBUTE CONSTRUCTS

Another new feature of OpenMP 4.0 is the teams construct, which launches several independent teams of worker threads. This is a kind of super-parallel directive encapsulating several parallel directives executing autonomous parallel treatments. The operation of the teams directive activating four teams of three threads each is shown in Figure 10.15, and the code is given in the listing below.

```
...
#pragma omp teams num_teams(4) thread_limit(3)
   {
   // code here is executed only by the master threads of each team
   ...
   ...
   #pragma omp parallel
      {
      // code here is executed by the full team
      ..
      }
   }
...
```

LISTING 10.36

Operation of the teams directive

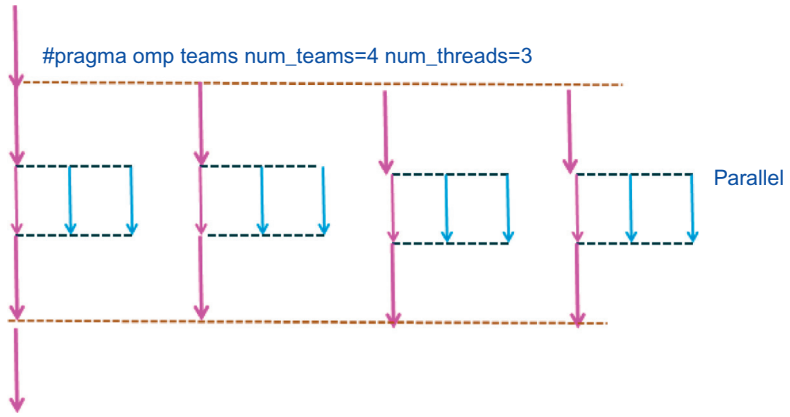

FIGURE 10.15

Operation of the teams construct.

Other clauses used to establish parallel regions—default, private, firstprivate, shared, reduction—are also accepted. This is natural, because the teams directive can be seen as establishing a parallel region for the master threads of each team. The number of teams can be selected by the nteams clause, or by library function calls, as was the case for the number of threads. The official documentation states that the number of threads in each team is implementation defined, but in all cases it is less or equal to the value specified in the thread_limit clause. Threads now have two identifiers: the team number (starting from 0) and the traditional thread number, which identifies the threads inside a team. When the code enters the teams region, all the teams are ready, *but only the master threads of each team (thread number 0) start the execution of the encountered code*. The remaining threads in each team are activated only when the team encounters a parallel region.

A few comments are in order here. First, observe that the teams construct in not very different from the operation of nested parallel regions, apart from the fact that the number of threads in each team is not strictly enforced. The main advantage is that the number of nested parallel regions that are activated is dynamic in the sense that it is not hard-coded, but determined by a clause or a library function call. This provides an additional flexibility that is welcomed in the context in which this construct is expected to be used.

The second observation is that the teams construct is only accepted in offloaded code. Indeed, it is rejected by the compiler if it is not inside a target region. And, if the target code is executed in the host, the teams construct is ignored. This will probably evolve in future OpenMP releases, but, for the time being, the teams construct is only a way of adapting a given parallel treatment to the memory hierarchy of accelerator devices.

Accelerator devices dispose of hundreds of cores, with a hierarchical organization. Cores are grouped in medium-sized SMP blocks sharing a common memory for the block. Then, there is an overall global memory shared by all blocks. Memory accesses are more efficient if they are restricted to the innermost shared block memory. Adjusting the thread activity and the data placement to this

memory organization is the real purpose of the teams construct. This is probably the reason the number of threads in each team is not strictly enforced.

Finally, OpenMP 4.0 also has the distribute directive, which spreads the iterations of one or more loops across the master threads of all the teams that execute the binding teams region. The master threads can in turn activate the intrinsic tasks of their team to execute the subset of iterations they are in charge of. Here is the syntax of the construct:

```
...
#pragma omp distribute [clauses]
for—loops
...
```

LISTING 10.37

Distribute directive

Here follows an example, taken from the official examples document [29], showing how the computation of the scalar product of two vectors is offloaded to an accelerator device using all the directives discussed so far. The function dotprod() receives as arguments the pointers B and C to the vector arrays, the vector size N, the size block_size of the vector chunks that will be distributed across the different teams, the number of teams, and the number of threads per team.

```
double dotprod(double *B, double *C, int N, int block_size,
               int n_teams, int block_threads)
    {
    double sum = 0;
    int i, i0;

    #pragma omp target map(to: B[0,N], C[0,N])
       {
       #pragma omp teams num_teams(nteams) thread_limit(block_threads)  \
                         reduction(+:sum)
          {
          // – – – – – – – – – – – – – – – – – – – – – – –
          // This is the master threads code; sum is now
          // a private variable inside each master thread.
          // – – – – – – – – – – – – – – – – – – – – – – ––
          #pragma omp distribute
          for(i0=0; i0<N; i0 += block_size)
             {
             // – – – – – – – – – – – – – – –
             // launch the parallel sections
             // – – – – – – – – – – – – – – –
             #pragma omp parallel for reduction(+=:sum)
             for(i=i0; i<min(i0+block_size, N); i++)
               sum += B[i]*C[i];
             }
```

```
        }     // teams terminate here
    }         // end of offloaded code
return sum;
}
```

LISTING 10.38

Scalar product of two vectors

Let us carefully examine this code:

- The function enters a target region in which the target vectors are mapped to the target device. The variable sum has, by default, a map tofrom: its final result will be recovered on the host at the end of the target region.
- Inside the target region, teams of worker threads are created, and the master threads are activated. The variable sum becomes by default a private variable inside each master thread. The reduction clause in the teams directive performs the reduction of the local values of src in each master thread.
- Next, in the master threads code, there is an external for loop with a block_size stride. If there was no distribute directive, all the master threads would execute the complete loop. However, because of the distribute directive, successive iterations of the loop are executed by successive master threads in round-robin order.
- Finally, each master thread activates the partner worker threads in the team with a parallel for directive. Again, the sum variable becomes a private variable for each worker thread, and the inner reduction clause accumulates the workers' partial results into the sum variable of the corresponding master thread.
- On exit from the target region, the final value of sum is mapped to the host.

The previous discussion summarizes the basic concepts in OpenMP 4.0 dealing with the offloading of code blocks to target accelerator devices. The devices API also proposed several ways of merging directives (e.g., target and teams can be merged in target teams).

10.11 THREAD AFFINITY

Thread affinity issues deal with the mapping of threads to cores as a means of optimizing access to a hierarchical memory system. If, for example, two threads are systematically accessing the same shared data, obviously it pays in memory performance to have them running in two cores that share the same L2 cache. Likewise, we know that accesses to the main memory are not uniform, because a given core may have different access times for different main memory banks. Performance benefits can be obtained if some kind of *togetherness* can be established between the core executing a thread and the data set the thread is acting upon, by placing the data in memory banks close to the executing core.

How is this togetherness with main memory data implemented by the operating system? Let us consider the memory allocation of a long vector. The fact that the whole vector is a unique consecutive array in logical memory does not necessarily mean it is a unique, consecutive array in physical memory.

The virtual memory systems of current computing platforms map blocks of contiguous logical memory addresses (called *memory pages*) to physical memory pages in different banks. And the way this mapping distributes consecutive pages in different banks can be decided by the operating system at execution time, using the traditional first touch policy. In the case of the long vector mentioned earlier, the operating system will try to map the page that contains the vector to a memory bank close to the core that first accesses it for initialization. If the vector spreads over several logical pages, and different cores initialize different sections of the vector, the operating system will do its best to establish a map from logical to physical memory addresses in such a way that different subarrays sit as close as possible to the cores that first initialized them. In this way, data can be distributed in main memory so as to optimize the nonuniform access times.

Notice, however, that paying attention to data placement in memory or to optimized usage of shared L2 caches is useless if threads can migrate across cores, which is in fact a very common event. When threads are preempted and moved to a blocked state, either because they are over-committed sharing CPU cycles or because an idle wait has been requested by the programmer, there is absolutely no guarantee that, when rescheduled, they will be run by the same core. *By default, threads are not tied to cores*.

> The OpenMP thread affinity interface guides the placement of specific threads on specific cores. Threads remain tied to the core they are placed on the whole duration of their lifetime.

This guidance is implemented by:

- An environment variable, OMP_PLACES, describing the hardware configuration on which the threads must be placed, and providing the value for the place_partition_var ICV. This description is performed once at the start of the code execution, and there is no way of retrieving the place_partition_var value, or of modifying it.
- When a parallel region starts, a description is provided of the placement policy requested for the worker threads. This placement strategy is defined by the OMP_PROC_BIND environment variable, which sets the value of the bind_var ICV. Its values can be retrieved by a library function call. However, the default thread affinity strategy established by the bind_var ICV can be overridden in a particular parallel region by the proc_bind clause associated with the parallel directive that defines the placement policy for the threads activated in the corresponding parallel region.
- The proc_bind() clause receives one policy placement argument that can be either scatter, close, or master.
- Placement policies applies to *threads* executing the initial implicit tasks in the parallel section. This means we know where the initial implicit tasks are executed. But there is absolutely no placement policy for the eventual explicit tasks generated in the parallel region, other than the fact that they will be executed by the placed threads in the worker team.

Figure 10.16 shows an SMP node made out of two quadricore sockets allocated to the application, which we will use to illustrate the thread placement policies. There are in this case eight cores available altogether. We assume that hyperthreading is enabled in the architecture, and that each core runs two hardware threads (a common situation today). In this context, parallel sections will be launched with different numbers of threads and different placement policies. The first step is to set the environment variable that describes the hardware configuration. This is done as follows:

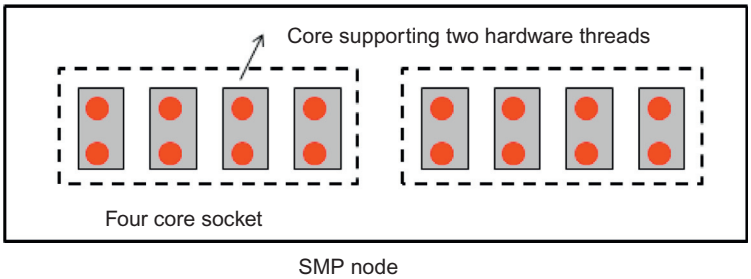

FIGURE 10.16

SMP node based on two quadricore sockets.

- We start by numbering—from 0 to 15—the hardware threads that can be run in the configuration.
- Cores are described by a sequence of integers corresponding to the associated hardware threads, enclosed in braces. For example, successive cores in the configuration of Figure 10.9 can be described as {0,1}, {2,3}, . . . , {14, 15}.
- An initial_index : length : stride notation can also be used to describe sequences of integers. If the stride is omitted, it is 1 by default. Therefore, the cores in the configuration of Figure 10.9 can also be described as {0:2}, {2:2}, {4:2}, ..., {14:2}, the stride being 1 by default in this case.
- The initial_value : length : stride notation can also be extended to describe sequences of cores: the initial value is a core rather than an integer, and we add the number of cores that follow and the stride for the first hardware thread in each core. Therefore, the whole sequence of cores in Figure 10.9 can be described as {0:2}:8:2. This means the initial core repeats 8 times, with the initial hardware thread label jumping with stride 2.

With these conventions, the OMP_PLACES environment variable is defined by a string that describes the hardware configuration in the way described above. Here are three equivalent ways of setting OMP_PLACES for the example of Figure 10.9:

```
export OMP_PLACES = "{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"
export OMP_PLACES = "{0:2},{2:2},{4:2},{6:2},{8:2},{10:2},{12:2},{14:2}"
export OMP_PLACES = "{0:2}:8:2"
```

LISTING 10.39

Setting OMP_PLACES

It is important to keep in mind that for OpenMP places are cores. There are therefore eight places in the example above. Hyperthreading, if enabled, will be activated when more than one thread is placed in a core. When several threads are placed in the same core, the system will use time slicing to run them.

10.11.1 SPREAD POLICY

Let us consider that the spread policy is applied to T threads on P places, and that T<=P. The runtime system constructs P'=P/T subpartitions and places a thread in each one of them. Figure 10.17

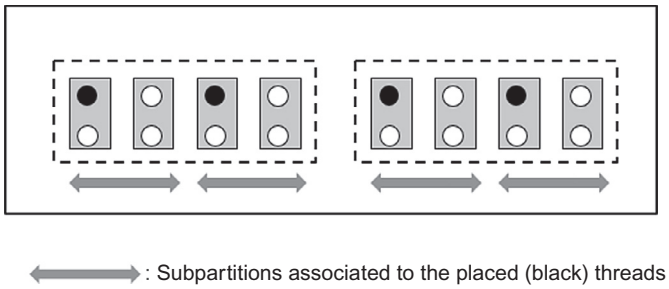

FIGURE 10.17

Scatter policy for four (black) threads.

shows how four threads are spread on the eight places corresponding to the configuration shown in Figure 10.16.

It is very important to keep in mind that the four subpartitions have a real role in the operation of the system, besides defining the placement of the threads. Indeed, these subpartitions correspond to the places available if the threads launch in turn a nested parallel region. In other words: for the placed threads, the subpartition becomes the new value of the placement description environment variable.

Placement policies are chosen to fit the application profile. If we are running an SPMD computation in which each thread acts mainly on its own data subset, with limited access to nonproprietary data items, then the spread policy is appropriate. Leaving four unused places in each subpartition may be very useful if, later on, the initial tasks launch a nested parallel section that access the parent data set, because they will be sharing the L2 cache.

You can easily guess what happens in some special cases. If we have five threads rather than four in our example, the size P' of the subpartitions is reduced to only one core. There will be five threads placed in consecutive cores, and eventual nested parallel regions will be run in the core of the master thread. If we have more threads than cores—let us say, ten threads—then the partition size is by default 1, threads are placed in consecutive cores and then wrap around eight, so that two cores will run two threads.

10.11.2 CLOSE POLICY

Let us first consider the case T<=P. When the number of threads does not exceed the number of cores, threads are placed in successive cores. But there is a difference with the spread policy: in this case, there are no subpartitions. The threads in the team will inherit the same, initial value of the place-partition ICV.

When T>P, the first T/P threads of the team (including the master thread) execute on the parent's place. The next T/P threads execute in the next place in the place partition, and so on, with wraparound. Let us go back to our example: consider that we are placing 16 threads, and that the parent thread that launches the parallel section is running on the place 2. Threads 0 and 1 run at place 3, threads 2 and 3 run at place 3, and so on. After the wraparound, threads 13 and 14 run at place 0 and threads 15 and 16 run at the place 1.

The close strategy is optimal if the eight threads are all acting on the *same data set*. In this case, because of their placement inside a unique socket, the share the L2 cache and access to shared data is optimized.

10.11.3 MASTER POLICY

In the master strategy, all threads in the parallel region execute at the place of the parent thread. Note that this will activate hyperthreading if enabled, but in our example time slicing will operate if more than two threads are placed according to this policy. The master policy may be useful if the parallel region launches occasionally subsidiary explicit tasks that must share data with the master thread.

10.11.4 HOW TO ACTIVATE HYPERTHREADING

In the example discussed above, you will notice that hyperthreading is activated, in the spread or close policies, only if threads are over-committed. In general-purpose computing platforms, hyperthreading helps to optimize CPU resources, but it is not a major performance factor. However, in highly specialized computing platforms (like IBM Blue Gene of Intel Xeon Phi) hyperthreading is mandatory to achieve optimal core performance.

Let us go back to our example, with four threads running on the hardware architecture of Figure 10.9. The question is, how can we force the placement of two threads per core? The solution is in the definition of the OMP_PLACES environment variable: we are not obliged to initialize it with all the places available in the hardware configuration. We just define a place partition of only two cores:

```
export OMP_PLACES = "{0:2}:2:2"
```

LISTING 10.40

Restricted number of cores

This solves the problem of enforcing hyperthreading for our four threads. Notice, however, that this place partition remains valid for the whole code, and cannot be changed.

10.12 VECTORIZATION

A very important extension of OpenMP 4.0 is the SIMD construct, which enforces vectorization of loops. As discussed in Chapter 1, vectorization is a parallelization technique designed to boost *single core performance* by implementing a single instruction, multiple data (SIMD) paradigm in which basic arithmetic instructions operate on vector chunks rather than on individual operands. Vectorization comes after multithreading in the parallelization hierarchy. The parallel for directive distributes different loop subranges across different worker threads. Inside each parallel task, the simd directive instructs the compiler to use native SIMD instructions to compute the assigned loop. Modern architectures rely on vectorization to deliver a substantial fraction of the full core performance.

10.12.1 SIMD DIRECTIVE

The simd directive has the following form.

```
...
#pragma omp simd [clauses]
for—loops
...
```

LISTING 10.41

Simd directive

In the vector addition example discussed in Chapter 3, the simd directive can be applied as follows:

```
...
double A[VECIZE], B[VECSIZE], C[VECSIZE];
...
#pragma omp parallel for
  #pragma omp simd
  for(int n=0; n<VECSIZE; n++) C[n] = A[n] + B[n];
...
```

LISTING 10.42

Adding two vectors

In this case, the parallel for directive will first split the vector operation in subranges allocated to different threads, and then every worker thread will compute its vector addition using the native SIMD instructions that concurrently add small chunks of vectors, the chunk size depending on the number of SIMD lanes available in the underlying hardware. In this case, one should observe a double speedup, coming from multithreading and vectorization. In an ideal case, this speedup is the number of threads times the number of SIMD lanes.

Simd clauses

Here are the clauses accepted by the simd directive:

- private, lastprivate, and reduction operate in the same way as in the multithreaded parallel constructs.
- collapse(n) also operates in the same way. The n nested for loops that follow the directive are merged in a larger, unique iteration space, which is next computed using native SIMD instructions. The order of iterations in the merged iteration space is determined by the natural order implied by the sequential execution of the loops.
- safelen(n): This clause indicates that two iterations executed concurrently cannot have a distance larger than n in the logical iteration space. Each concurrent iteration is executed by a different SIMD lane. In other words, the optimal value of n is the number of SIMD lanes available in the underlying hardware.
- linear*(argument list: constant linear step)*: This is a modified form of the firstprivate clause used in other parallel constructs (and not available here). Logical iterations of the SIMD loop are numbered $(0, \ldots, N)$. Variables in the argument list are initialized to a value depending on the loop iteration, equal to the value when the simd directive is encountered, plus the iteration number times the linear step.

- aligned*(argument list: alignment)*: The argument list contains arrays or pointers to arrays in C-C++. Variables in the argument list are aligned to the number of bytes expressed in the alignment parameter. This parameter is optional, and its default value is implementation dependent.

Notice that OpenMP 4.0 accepts pointers to arrays by default, leaving to the programmer the responsibility of guaranteeing the absence of *aliasing*, that is, the fact that the pointers involved in the vectorized algorithm are pointing to nonoverlapping memory blocks. Aliasing is a traditional problem with pointers and vectorization. Consider the following vector computation of A += B:

```
...
double A[VECIZE];
double *B;
...
// initialize B: this is aliasing
B = A-1;
...
#pragma omp simd
for(int n=1; n<VECSIZE; n++) A[n] += B[n];
...
```

LISTING 10.43

Aliasing issue

The simple appearance of the for loop above is deceptive: this loop cannot be vectorized. Because B[n]=A[n-1], there is a *data dependency* among consecutive iterations. To compute an iteration, we need the result of the previous one. If we insist in vectorizing the loop, we get a result that is not what we want. In OpenMP 4.0, it is the programmer's responsibility to make sure this does not happen. Intel compilers, instead, will not vectorize a loop involving pointers if it is not totally clear that there is no aliasing. Programmers dispose of directives to tell the compiler that vector dependencies can be safely ignored. We will come back to this point in Chapters 13 and 14.

10.12.2 DECLARE SIMD DIRECTIVE

Let us assume we want to vectorize the following loop corresponding to a map parallel pattern, where the same operation is applied to all the elements of a collection:

```
...
double *B;
double *A;
...
#pragma omp simd
for(int n=0; n<VECSIZE; n++) A[n] = f(B[n]);
...
```

LISTING 10.44

Need for SIMD functions

where $f(x)$ if a function that takes, in this example, a double argument, and returns a double. The declare simd directive enables the creation of SIMD versions of a given function the compiler can use

to process multiple arguments—sitting on different SIMD lanes—with a single invocation. There are no restrictions on the signature of the function, which can take several arguments.

```
...
#pragma omp declare simd [clauses]
function declaration or definition
...
// In example above:
//  - - - - - - - - -
#pragma omp declare simd
double f(double x);
```

LISTING 10.45

Declare simd directive

Declare simd clauses

Here are the clauses accepted by the declare simd directive. They provide information about the nature of the function arguments and the way they have to be initialized, as well as information on the way the function is intended to be used.

- simdlen(n): number of concurrent arguments in the function.
- uniform(argument list): the data items in the list have an invariant value for all concurrent executions of the SIMD loop.
- linear(argument list: linear step): the data items in the argument list must be initialized, when the function is invoked, in the way already discussed for the simd directive.
- aligned(argument list: alignment): operates as in the simd directive.
- inbranch: the function will always be called from inside a conditional statement.
- notinbranch: the function will never be called from inside a conditional statement.

Further discussion of the usage of vectorization directives will be developed in Chapters 13 and 14.

10.13 ANNEX: SafeCounter UTILITY CLASS

In a thread-centric environment, worker threads have an intrinsic integer identifier, that is, *rank*. In a task-centric environment, it is often useful to have similar integer identifier *for tasks*. This can be handled by passing a rank value to each task when they are created. But another useful alternative is to dispose of a thread-safe global counter returning successive integer values to tasks asking for the next integer. In this way, each task disposes of a unique integer identifier. This is the purpose of the SafeCounter class, which obviously encapsulates a mutex-protected integer counter. Here is its public interface:

SafeCounter PUBLIC INTERFACE

SafeCounter()

– Constructor.
– Creates a SafeCounter object. Internal integer is 0.

~ SafeCounter()

- Destructor.
- Destroys the SafeCounter object.

int Next()

- Returns the next integer.
- The first returned integer is 1.

void Reset()

- Resets the internal counter for a new use.