CHAPTER

INTEL THREADING BUILDING BLOCKS

11

11.1 OVERVIEW

The Intel Threading Building Blocks (TBB) library has been available in the public domain as open source software for several years. Easily installed and configured (see Annex A), this library outfits C++ for multicore parallelism. It comprises complete documentation and a substantial number of code examples illustrating its most important and subtle aspects:

- The documentation accompanying the release is accessed from /docs/index.hmtl in the TBB installation directory. It contains complete information on all the library components.
- The examples accompanying the release are in the /examples directory.
- The online documentation at the website. Three links are particularly important: the Reference Manual [17], the tutorial TBB Getting Started [32], and the User Guide [33]. All contain detailed discussions and examples of the TBB operation, with the Reference Manual including complete information on all library components.
- James Reinders book, *Intel Threading Building Blocks*, develops a complete and pedagogical presentation the TBB parallel programming environment [34].
- The *Structured Parallel Programming* book by M. McCool, A. D. Robison and J. Reinders [35] discusses in detail the way TBB—as well as OpenMP and Cilk Plus [1]—cope with a large variety of parallel contexts. This book, an advanced tutorial in parallel programming, contains detailed and useful information on the way these three programming environments operate.

The core of the library is a task-centric programming and execution environment, implementing, as it is also the case of OpenMP and Cilk Plus, refined strategies for task scheduling. The TBB shared memory programming API operates at three levels of abstraction:

- At the highest level, TBB provides powerful STL-like algorithms, implemented as C++ template functions or classes, that drive parallel processing while hiding from the programmer the underlying operation of the thread-pool scheduler. Many of these high-level algorithms adopt a fork-join strategy at the task level, designed to optimize load balance and cache memory efficiency. They can be seen as automatic parallelization engines that extend in several directions the functionality provided by the parallel for directive in OpenMP.
- At the next level, TBB enables limited direct access to the task scheduler. A simple and elegant task scheduler API is provided by the task_group class, enabling immediate access to the most basic scheduler services. To put it in a nutshell, this programming interface is a simplified handle

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00011-3

to a set of features integrating the taskwait (tasks waiting for direct children) and taskgroup (tasks waiting for all descendants) synchronization constructs, as well as the task group cancellation features, discussed in the previous chapter on OpenMP.

- Finally, there is the complete task scheduler API, providing access to several additional features, the most important being the establishment of hierarchical relations among tasks to drive the operation of the task scheduler. A preview of this feature was given when discussing the depend task clause in OpenMP. Indeed, OpenMP 4.0 has recently incorporated hierarchical task dependency features. In TBB, the hierarchical relations among tasks are at the core of the scheduler architecture.

The first two levels of abstraction: high-level STL like parallel algorithms and the simplified access to the task scheduler are the subject of this chapter. More advanced use of the TBB task scheduler is discussed in Chapter 16.

TBB is at the core of Intel's Parallel Suite. Intel also proposes a C++ extension called Cilk Plus [1] with an underlying execution model close to TBB. By just adding three new keywords to C++, Cilk Plus provides a simple way of expressing potential parallelism. However, Cilk Plus requires compiler support, while TBB runs as a standalone library. We will come back to this point later in this chapter.

11.2 TBB CONTENT

In addition to the task management algorithms and APIs, TBB proposes several other standalone utilities worth keeping in mind, even if other environments (OpenMP, native threads) are adopted to manage threads. In fact, the TBB documentation underlines the fact that TBB has been designed to interoperate with OpenMP and the native threads libraries. A few interoperability examples have already been given in previous chapters, and a few others will come in the following ones.

Here is a summary of the additional utilities provided by the TBB programming environment:

- Concurrent containers with built-in thread safety (discussed in Chapter 5).
- Mutual exclusion tools: several types of mutex and scoped locks, discussed in Chapter 5, as well as the tbb::atomic<T> class, discussed in Chapter 8.
- Improved memory allocation tools that replace the standard memory allocation routines (malloc-free, new-delete) by more efficient and scalable allocators that enhance memory allocation performance in a multithreaded environment.
- Thread local storage: TBB proposes two classes to implement thread local storage:
  - combinable
  - enumerable_thread_specific (discussed in Chapter 4)
- TBB also proposes an implementation of an important subset of the C++11 multithreading library, basically, the thread management features in the <std::thread> header, and the event synchronization features in the <condition_variable> header.

A few comments on the status of these utilities may be useful at this point. Remember that the TBB programming model is a *task-centric* model, where the mapping of tasks to threads is not one-to-one. Some of these standalone utilities (concurrent containers, mutual exclusion, memory

allocation routines) can safely be used in the TBB task-centric environment, but others (thread local storage, C++11 event synchronization) must be handled with care. It has already been emphasized in previous chapters that thread local storage *does not* guarantee thread safety if the task to thread mapping is not one-to-one, and that synchronization constructs based on condition variables synchronize threads, not tasks. In a task-centric environment, event synchronization must be implemented directly on tasks by using the depend clause in OpenMP or the hierarchical relations among TBB tasks to be discussed in Chapter 16.

In fact, TBB introduced from the start its own mutual exclusion tools, however, *no explicit tools for event synchronization were available for a long time* simply because they are not compatible with the task-centric programming model. Thread-centric utilities (thread local storage, C++11 synchronization) were added later to enlarge the programmer's toolbox, and they are, indeed, useful. An example is given in Chapter 14, in which an application with a substantial amount of barrier contention exhibits very poor parallel scalability when Pthreads barriers are used. An alternative, more efficient barrier algorithm is available, involving atomic variables and thread local storage [25]. However, Pthreads not having atomic utilities, the algorithm is implemented in the TBarrier class—discussed in Chapter 9—using tbb::atomic<T> and one of the TBB thread local storage classes. When this new barrier class is used, the performance of the application significantly improves.

The rest of this chapter reviews in detail the high-level TBB parallel algorithms. Simple examples of their operation are presented, and Chapters 13–15 discuss the TBB implementation of several complete parallel applications. Chapter 16 explores the details of the TBB scheduler. Our intention is to enable readers to develop clear insights on the code development options that TBB provides, and on its complementary role with respect to other programming environments.

11.2.1 HIGH-LEVEL ALGORITHMS

Here is a list of the high-level parallel algorithms proposed by TBB. They are all referenced in the "Algorithms" topic in the Reference Guide [17].

Parallel_for: The parallel_for algorithm is used to implement a map operation, in which a specific action described by a function object is applied to a collection of elements in a container. This algorithm implements a *divide and conquer* strategy that performs a recursive domain decomposition of the data set.

Parallel_reduce: When, in addition, reductions must be performed in order to collect partial results from each parallel task—as was the case in most the data parallel examples we discussed—the parallel_reduce algorithm is required. Again, this algorithm automatically performs the domain decomposition of the data set, setting up an elegant way of performing internally the reduction operations.

Parallel_scan: This algorithm is more subtle. Its relevance can be usefully illustrated with a simple example. Consider the computation of the following loop:

```
int X[N];
...
for(int n=1; n<N, ++n)
    X[n] += X[n−1];
```

LISTING 11.1

Using the parallel_scan algorithm.

in which the value of each vector component X[n] is replaced by the accumulated sum of all the previous components. This scan operation for the addition operation exhibits a data dependency that blocks its immediate parallel computation. However, *if the operation is associative*, it is possible to organize the computation by performing two passes over the data set. In this case, the amount of computational work is increased, but the work can be done in parallel. The TBB tutorial has a very clear description of the way this algorithm operates. The parallel scan concept can be extended to any associative operation that has an identity element (like the 0 for addition).

Parallel_for_each: A simpler version of parallel_for, which avoids the weaponry required to automatically perform a recursive domain decomposition.

Parallel_do: This algorithm performs, like parallel_for, a parallel iteration over a range, with an additional feature: the possibility of optionally adding more work at each iteration, with the help of the auxiliary class parallel_do_feeder.

Parallel_invoke. Receives as arguments pointers to functions (up to 10 functions) that are invoked in parallel.

Parallel_sort: Parallel generalization of the sort algorithms proposed by the STL, discussed in Chapter 10.

Pipeline This algorithm is a class template. A producer-consumer synchronization is implemented via a data item (a token) that runs along the pipeline. It is discussed in detail in Chapter 15.

There is substantial added value provided by the TBB parallel algorithms. The implicit work-sharing strategies are somewhat more diversified than the ones proposed in OpenMP.

11.3 TBB INITIALIZATION

All the TBB software components are defined in the tbb namespace. As usual in C++, this namespace may be directly accessible as if it was the global namespace with the traditional using namespace tbb declaration, and we will assume in what follows that this is always the case. Otherwise, the tbb:: scope must be explicitly used in all TBB components in the program. The TBB documentation on this topic is in the "Task Scheduler" topic in the Reference Guide [17].

The initialization of the library requires the creation of a tbb::task_scheduler_init object in the application. The constructor of this object performs the required initialization, and the destructor shuts down the task scheduler. Moreover, the switch -ltbb—in Linux—or /link tbb.lib—in Windows—must be included in the linker in order to access the runtime TBB library.

The task scheduler constructor accepts three possible arguments that specify the number of threads managed by the scheduler:

- task_scheduler_init::automatic. In this case the number of threads in the application is equal to the number of cores available in the computer platform running the program.
- A *positive integer* defining the number of threads to use.
- task_scheduler_init::deferred. In this case, the task scheduler is created but the number of threads is not defined. It is specified later on with a call to task_scheduler_init::initialize (nThreads).

The very simple example TbbInit.C exhibits these initialization options by getting them from the command line and printing a message indicating the initialization strategy adopted. With no command line arguments, the initialization is automatic. Otherwise, the number of threads is read from the command line. If the value 0 is passed, the initialization is deferred and the code prompts the user later to specify a positive number for the number of worker threads. Listing 11.2 illustrates how the task scheduler object TS—the object name is obviously arbitrary—is created following these rules.

```
#include <tbb/task\_scheduler\_init.h>
...
using namespace tbb;
...
int main(int argc, char **argv)
   {
   int nTh;
   ...
   if(argc==1)
      task_scheduler_init TS(
          task_scheduler_init:: automatic);
   else if(argc==2)
     {
     nTh = strtol(argv[1], 0, 0);
     if(nTh==0)
       task_scheduler_init TS(
           task_scheduler_init:: deferred);
     else  task_scheduler_init TS(nTh);
     }
   ...
   return 0;
   }
```

LISTING 11.2

TbbInit.C source code.

The task scheduler TS does not need to be explicitly destroyed because the destructor is implicitly called when the object goes out of scope, in this case at the end of main(). Explicit destruction in the middle of the program is also possible [34].

Example 1: TbbInit.C

To compile, run make tbbinit. The initialization option is passed from the command line (see text above).

The automatic initialization is quite interesting. One of the purposes of TBB is to simplify as much as possible the deployment and operation of multicore applications. With this initialization, the application adapts automatically to the number of cores available in the computing platform. This means that an application that has been running on a dual-core laptop will immediately run *as*

such, with no change and no recompilation, on a four-core laptop. However, when running applications on large shared memory multicore servers accessed by many users, one may not want to run on all the available cores in the SMP platform.

11.4 OPERATION OF THE HIGH-LEVEL ALGORITHMS

TBB relies on C++ semantics to propose an elegant way of handling domain decomposition and work-sharing utilities in a multithreaded environment. In the OpenMP parallel for directive, work sharing is controlled by the number of threads available in the parallel region. In some of the TBB algorithms, work-sharing is not directly related to the number of threads in the pool. Rather, it is handled recursively. The initial workload submitted to the pool is repeatedly split until a predefined minimal workload—called granularity—is reached. Another difference with OpenMP is that the work sharing algorithms are not restricted to loops. In fact, the abstractions involved in these algorithms enable at least parallel_for and parallel_reduce to handle contexts in which the domain decomposition is applied to a data set *of any kind*, as long as it is splittable. Some examples will soon follow.

Here is the declaration of the parallel_for template function. The declarations of the two other functions are identical to this one:

```
template<typename Range, typename Body>
void parallel_for(const Range& R, const Body& T);
```

LISTING 11.3

Parallel_for signature.

The template parameters Range and Body (names are arbitrary) refer to two different types—C++ classes—that are required to have very precise interfaces—i.e., specific public member functions—internally used by the algorithms:

- A Range type models an *iteration space*, namely, a data set that defines the iteration range of a loop, or, more generally, the space in which the algorithm operates. A fundamental requirement for this data type is that *it must be splittable*. The split operation on an object of type Range divides the object into two equal parts: one assigned to the original object (that shrinks to half its initial size) and the remaining one assigned to a new object.
- A Body type that encapsulates the algorithm operation on a Range object. This is, in fact, a function object class that defines the task function to be executed by threads on an arbitrary range that ultimately results from successive splittings of the original range passed to the function.

These two data types are defined by their behavior, namely, by the public member functions they implement. They may be considered as the answer to *callbacks* coming from the template function, who is asking the client code for the additional information needed to adapt its operation to the problem at hand. The same situation is encountered when a C library function asks for a pointer to a function to specify behavior. In fact, the Body class is really a function object, which is the C++ generalization of a pointer to a function (see more details on function objects in Annex B).

11.4.1 INTEGER RANGE EXAMPLE

Listing 11.4—from source file IntRange.h—is an example of a Range type that models an integer range employed in a standard for loop. This example is presented only for pedagogical purposes; TBB has built-in template range classes for integer ranges. The member functions defined in this example are the ones required by the algorithms using it. In modeling a range, additional functions may be defined if needed for code clarity and efficiency, but the ones defined here must in all cases be implemented.

```
#include <tbb/tbb_stddef.h>

class IntRange
   {
   private:
    int Beg;
    int End;
    int granularity;

   public:
    IntRange( int b, int e, int g)
       {
       Beg = b;
       End = e;
       if(b>e)
          {
          Beg = e;
          End = b;
          }
       granularity = (End—Beg)/g;
       }

    IntRange(IntRange\& R, tbb::split)
       {
       int middle = (R.End+R.Beg)/2;
       End = middle;
       Beg = R.Beg;
       granularity = R.granularity;
       R.Beg = middle;
       }

    bool empty() const
       { return (End == Beg); }

    bool is_divisible() const
       { return (End >(Beg+granularity+1)); }

    int begin() const
```

Continued

```
    { return Beg; }

  int end() const
    { return End; }
  };
```

LISTING 11.4

IntRange Range class.

This class describes the half-open integer range [Beg, End) using the STL conventions: Beg is the first index in the range, and End is the next after last index, not in the range. Remember that this STL convention is motivated by a convenient way of writing loops.

Besides Beg and End, the IntRange class incorporates another private data member, granularity. The range object will be split into two equal parts if its size is bigger than the granularity. When a range object is first constructed, its granularity is determined by an integer value n, as a fraction (End-Beg)/n of the initial size. The constructor of IntRange objects take three arguments: the limits of the initial range and the integer n that determines the requested granularity.

Other functions called by the algorithm are:

- empty(), which returns true if the range is empty.
- is_divisible(), which returns true if the range size is bigger than the granularity.
- begin() and end(), which return the limits of the range object.

Notice the const qualifier in the definition of these functions. This means they are read-only and do not modify the private data in the object. Never forget these qualifiers; C++ is very touchy when it comes to type checking, and const qualifiers are part of the data or member function types. The TBB library will not accept the range class if they are omitted.

The only subtle point in this class is the *split constructor* IntRange(IntRange& R, tbb::split) listed above, which receives as arguments a reference to an existing IntRange object, as well as a split data item, defined in the include file tbb/tbb_stddef.h. The introduction of this split data item—which is in fact an empty object—is just a trick that serves no other purpose than allowing the compiler to distinguish the split constructor from the standard copy constructor. Indeed, any C++ class has a copy constructor, either explicit provided by the programmer or implicitly defined by the compiler, that constructs a clone of the initial object passed as argument. In our case, the copy constructor has signature IntRange(IntRange& R), and it is present even if there is no explicit definition. Therefore, a spurious extra argument is needed in order to distinguish the split constructor—which also takes a reference to a target object—from the copy constructor.

The action of the split constructor is shown in Figure 11.1. It constructs a new range object with half of the target object range, and then it shrinks the target object range to the other half, so that we end up with two half-range objects. Notice that the split data item is ignored in the function body.

11.4.2 BUILT-IN TBB RANGE CLASSES

The class IntRange listed above is meant as an example to show the content of the Range concept; it will be used it in the examples that follow to show that it effectively works. TBB has two built-in template classes for integer-like ranges that are more sophisticated than the example above and

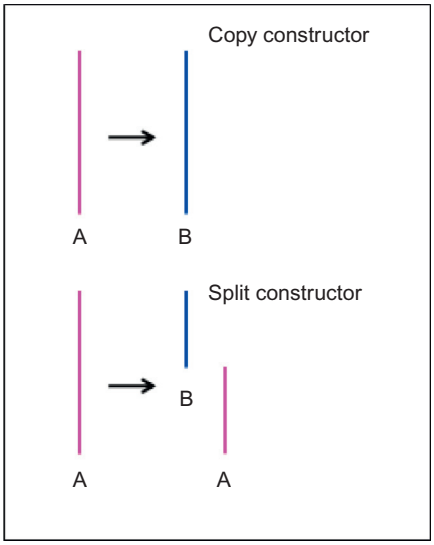

FIGURE 11.1

Difference between the copy and split constructors.

that should be adopted in real applications (see the "Algorithms->Range Concept" in the Reference Guide [17]).

- `blocked_range<T>` is a half-open range of elements of type `T` that can be recursively split. The type `T` must satisfy some requirements that include integer types, pointers, and STL random access iterators.
- `blocked_range2d<T>` is a two-dimensional extension of the previous class.

These classes follow the strategy outlined in the example, but they have several additional features. They trigger error-checking assertions when the debug version of the TBB library is used. Furthermore, the granularity argument in the constructor can be avoided when working with versions of the algorithms in which the granularity is dynamically determined from the task scheduler operation.

11.4.3 FORK-JOIN PARALLEL PATTERN

The basic task-scheduling mechanism implemented by the TBB divide and conquer algorithms for managing the recursive splitting of the initial range is the fork-join construct shown in Figure 11.2: an initial task 1 spawns a child task 2 and the control flow continues in the parent task 3. The parent task that triggers the fork will be referred to as the *continuation* task. At some point, tasks 2 and 3 are synchronized: the parent (continuation) task waits for its child. Then, the parent task continues execution as task 4. This process is recursive: child or continuation tasks can in turn trigger new, nested fork-join constructs.

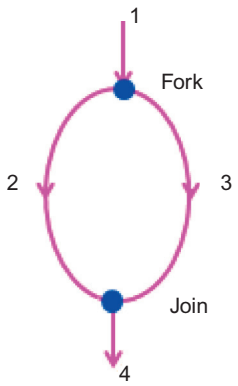

FIGURE 11.2

Fork-join construct.

The *Structured Parallel Programming* book [35] has a full chapter devoted to the fork-join parallel pattern, with a detailed discussion of its implementation not only in TBB but also in Cilk Plus [1]—based on new keywords added to the language—or in OpenMP—based on task spawn and wait. In all cases, *task stealing* is the mechanism that transforms the potential parallelism—expressed by the task spawn operation—into a mandatory parallel processing where other worker threads take over the task execution. The operation of the TBB scheduler will be described in detail in Chapter 16. For the time being, it is sufficient to know that.:

- Waiting tasks are queued waiting for execution. Each thread has its own, proprietary queue.
- A spawned task is queued in the proprietary queue of the thread executing the parent task.
- When a thread is idle and looks for a next task to execute, the first place it looks at is its proprietary queue.
- When a thread is idle and its proprietary queue is empty, *it steals a task from another, non-proprietary, queue* (Figure 11.3).

In this approach, real parallel processing is activated by task stealing, when an idle thread having no input from its own queue chooses a victim task from another queue. It is interesting to observe that TBB and Cilk Plus have different behavior in what concerns task stealing, due to different design options. In TBB, it is the child task that is queued and eventually stolen if there is a sufficient number of worker threads, and the parent task continues normal execution by its executing thread. In Cilk Plus, it is the *continuation* task that is queued and eventually stolen, and the thread that was initially executing the parent task starts executing the spawned task. To understand the difference, imagine that there is only one worker thread. In this case, the Cilk Plus choice corresponds to standard sequential execution of a function call: the spawned task is executed first, and the parent task suspends execution until the function returns. In TBB, it is the other way around: the spawned task will be dequeued and executed only when the continuation task reaches the join synchronization point. There are, of course,

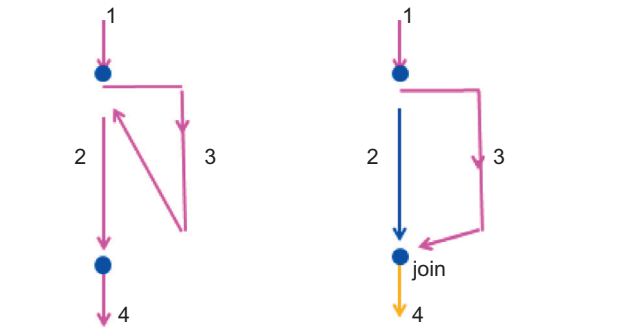

FIGURE 11.3

Modes of execution of the fork-join pattern.

different pros and cons for these different choices. The Cilk++ strategy is more difficult to implement, and requires compiler support. The TBB library, instead, works with any compiler.

Chapter 8 of the *Structured Parallel Programming* book [35] has a very detailed discussion of the fork-join pattern and of many of its implications in specific parallel contexts: the fact that task stealing in Cilk Plus and TBB effectively balance the fork-join workload, or the major impact on stack space usage of the different strategies: *continuation stealing* in Clik Plus, *child stealing* in TBB. It turns out that the Cilk Plus strategy can guarantee a more efficient stack space usage.

The fork-join strategy using the Body and Range types is also used in the parallel_reduce and parallel_scan algorithms. The Range classes are all the same, but the Body classes have specific requirements for each algorithm.

11.5 SIMPLE PARALLEL_FOR EXAMPLE

The parallel_for algorithm can be used when the parallel treatment involves the activation of a number of embarrassingly parallel tasks with no communications among them. In the example that follows, the initial range will be successively split into a series of nested fork-join constructs until sub-ranges reaches the limits tolerated by the given granularity, as sketched in Figure 11.4. Ultimately, we are left with a number N of equal size sub-ranges, so that N tasks are executed by the thread pool. Notice that N is in all cases a power of 2. It is obvious that if N is bigger than the number of threads in the pool, task stealing will be rare and in most cases the same thread will take care of different sub-ranges. As expected, too small a granularity induces excessive fork-join overhead and conspires again overall performance.

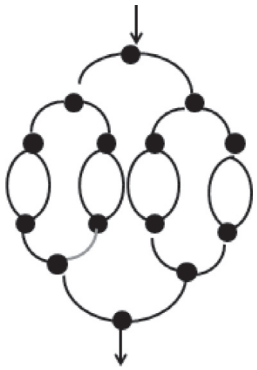

FIGURE 11.4

Nested fork-join patterns in the parallel-for algorithm.

A simple example of a Body class is provided by the parallel computation of the following loop:

```
double *X, *Y;
// X and Y, size N, are initialized to random values
for(int n=0; n!=N; ++n) X[n] += Y[n];
```

LISTING 11.5

Simple parallel loop.

i.e., adding vector Y to vector X. The range class in this example is the IntRange class defined above. Here is the Body function object class for this problem:

```
class AddVectorBody
   {
   private:
    double *X, *Y;

   public:
    AddVectorBody(double *x, double *y) : X(x), Y(y) {}
    void operator() (const IntRange& R) const
       {
       for(size_t n=R.begin(); n!=R.end(); ++n)
          X[n] += Y[n];
       }
   };
```

LISTING 11.6

Body class for the parallel loop.

This class has, as private internal data, the addresses of the vectors to be added. The constructor just initializes the two vector pointers. As in any C++ class, there is also in this case an (implicit) copy constructor that constructs a new object by copying the object data, i.e., the vector pointers.

This class is a *function object class*, because it overloads as a member function the function call operator operator(). This allows the client code using the class to instantiate an object with an arbitrary name, and use this object name as a function name, as shown in Listing 11.7.

```
double *X, *Y;
IntRange(0, 1000, 4);                // construct range
AddVectorBody my_name1(X, Y);        // define function object
...
my_name1(R);                         // calls operator()(R)
...
```

LISTING 11.7

Using function objects.

Given the importance of function objects and the extensive usage made by TBB, Annex B summarizes their properties and discusses their interest and relevance. A very important point to keep in mind is that function objects are objects, and that consequently they can be copied. As explained in Annex B, they generalize the concept of function pointer, enabling additional features that pointers to functions do not have.

The body of the operator() function defines the task to be performed by the executing worker thread on an arbitrary range, ultimately resulting from successive splittings of the initial range. *Notice the* const *specifier in the definition of this function*: this means the private function object data cannot be modified by this member function. This const specifier is required by the library, and in the present case it just guarantees it is not possible for the operator() function to modify the vector pointers. The reason is easy to understand: the algorithm will use the copy constructor to produce copies of this function object to handle them to different threads, and it is necessary to guarantee that they all refer to the same vectors. This, however, does not prevent each operator() function from modifying *the data the pointers point to*, as it is the case of the vector X, whose components are indeed updated by this function.

11.5.1 PARALLEL_FOR OPERATION

The parallel_for algorithm implements the fork-join strategy discussed above. Obviously, the worker threads that receive a task to execute start by taking a look at the range. If it is splittable, the executing thread splits it into two halves, spawns a child task dealing with one half of the range, and continues the execution of the continuation task dealing with the other half. If the range is not splittable, the initial task is executed.

Let us go back to Figure 11.3 that shows the two options available to the scheduler to execute a fork-join pattern. When the range is splittable, the following actions take place:

- The executing thread puts *the child task* in its own task queue and starts executing the continuation task.
- The child task can eventually be stolen by another idle thread in the pool.
- If no other idle thread in the pool steals the queued child task, the current thread dequeues and executes it when the continuation task terminates.
- If another available thread steals the child task, *it gets a copy of the parent thread task function object* and uses it to concurrently execute the task. The copy of the function object is obtained, naturally, from the copy constructor.

- A synchronization between parent and child operates at the joining point. Notice the thread that continues with the execution of task 4 is in all cases the same as the one executing initially task 1. We will come back to this point in Chapter 16.

11.5.2 FULL ADDVECTOR.C EXAMPLE

The full example is in file AddVector.C. The code reads the input data from the addvec.dat file: vector size N, number of threads nTh and granularity Gr. It initializes the vector components to uniform random values in [0, 1]. Then it applies the parallel_for algorithm as described below:

```
int main(int argc, char **argv)
   {
   double *X, *Y;

   // allocate and initialize X, Y
   // read from file N, nTh, Gr
   // print X[0], X[N/2], Y[0], Y[N/2]

   task_scheduler_init init(nTh);
   IntRange R(0, N, Gr);
   AddVectorBody B(X, Y);
   parallel_for(R, B);

   // print again X[0] and X[N/2], to check
   return 0;
   }
```

LISTING 11.8

AddVec.C main function.

Example 2: AddVec.C

To compile, run make addvec. Input data is read from file addvec.dat.

11.6 PARALLEL REDUCE OPERATION AND EXAMPLES

Embarrassingly parallel contexts are rare because most of the time some kind of information must be passed or shared among tasks. Common concurrency contexts require, for example, the accumulation or the comparison of partial results provided by each task. This is the kind of concurrency pattern handled by the parallel_reduce algorithm.

The parallel_reduce algorithm also accepts Range and Body types as template parameters. The Range classes model range values and do not require any further modification. The Body classes, however, *are not* the same as the classes for parallel_for algorithm. Indeed, the necessity of performing a reduction operation requires a more refined implementation of the fork-join pattern, and some modifications are introduced to the way the body classes operate.

11.6.1 FIRST EXAMPLE: RECURSIVE AREA COMPUTATION

In order to exhibit the flexibility of the TBB basic algorithms, a first example of the usage of the parallel_reduce algorithm is proposed next, adapted to a problem *not dealing with a parallel loop*, namely, the parallel computation of the area under a curve already discussed several times in previous chapters.

The first issue is to define a new Range class on which the algorithm operates. This is done in the include file RealRange.h. This class models a splittable interval on the real axis. It has exactly the same member functions described in the listing of IntRange.h, but with very mild modifications adapting it to a real interval rather than a succession of integer values.

The structure of the body class, defined in Listing 11.9, has a few modifications required by the necessity of managing the partial results computed by each task. This class refers to the auxiliary library function area() discussed in previous chapters that receives as arguments the limits of integration and a pointer to the function to be integrated, and returns the area with a given precision.

```
double area(double a, double b, double(*func)(double));

class AreaBody
   {
   private:
    double (*fct)(double x);

   public:
    double partial_area;

    AreaBody(double (*F)(double)) : fct(F),
                    partial_area(0.0) {}

    AreaBody(AreaBody& A, split) : fct(A.fct),
                    partial_area(0.0) {}

    void operator() (const RealRange& R)
       {
       partial_area += area(R.begin(), R.end(), fct);
       // print debug message
       }

    void join(const AreaBody& A)
       {
       partial_area += A.partial_area;
       }
   };
```

LISTING 11.9

Body class for the area computation.

New, public data items

This class stores a private pointer to the function to be integrated, and a public double partial_area that is used to store partial results. The ordinary constructor initializes the function pointer to the passed argument value, and also initializes partial_area to 0. Each task function object constructed and used by the algorithm carries internally this information. Notice that in this class the member operator() function no longer has the const qualifier, because this function updates the partial_area value. Notice also that partial_area is public, simply because ultimately it will be read by the client code in order to retrieve the final result of the reduction.

Additional, "split" constructor

The body class listed above has the standard constructor that initializes the function pointer, as well as the initial value of partial_area. There is also the implicit copy constructor built by the compiler that copies to a new object the values of all the private and public data items stored in the target object.

A new, additional constructor, different from the copy constructor, is needed by the algorithm. The split argument is, again, just a trick to distinguish it from the copy constructor. However, in this case, the "split" constructor does not split anything: it just constructs a new function object from a reference to another target function object. The main point is that the new object is not an exact clone of the original one: the split constructor *does not copy* the value stored in partial_area; rather, it reinitializes this data item to the original value 0.

This new constructor is needed for a correct management of partial results. If there is no task stealing and the same thread is handling different ranges, the partial area results keep accumulating in the partial_area value, which is what we want. However, when there is task stealing, the new stealing thread gets a copy of the function object of the parent task. But it may happen that the parent task function object has *already* in its guts partial area results, because it has already explored other sub-ranges. If the normal copy constructor was used to provide a copy to the new thread, this new thread would inherit these values, and this is definitely not what the algorithm wants, because previous values are already taken into account in the parent thread that continues execution of the continuation task. The new thread must start with a brand new function object with no memory of previous partial results. This is exactly what the split constructor does: it creates a new function object where only the function pointer is copied from the target, the partial_area data item taking the standard initialization values. The split constructor therefore erases the memory of previous computations by other threads stored in partial_area, and starts a brand new strand of partial reductions.

The conclusion is: the split constructor is used every time there is task stealing, to create a brand new concurrent task executed by another thread. If there is no task stealing the split constructor is not called: it is the original parent thread that takes over the continuation task after executing the spawned task.

Performing the reduction

At this point, the new join() member function enters the game. It is called by the algorithm to close a fork-join construct with task stealing, to perform the reduction between partial results held by the spawned and continuation tasks. Obviously, calls to join() are paired to calls to the split constructor at a task stealing fork-point.

The caller task adds to its internal partial_area value the value provided by the target task. At the end of the algorithm operation, all the copies of the task function object are destroyed, and the initial task function object passed to the algorithm holds internally the final result of the reduction.

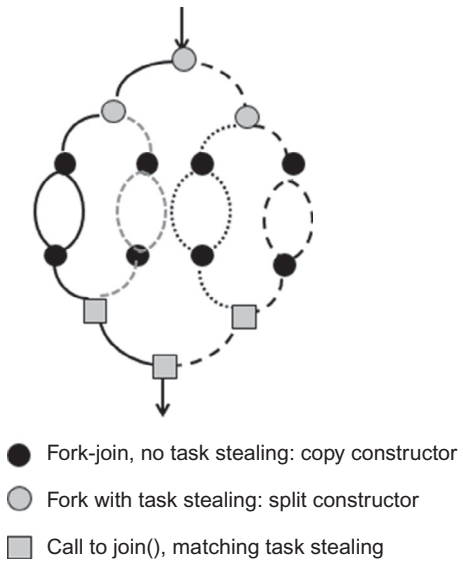

FIGURE 11.5

Nested fork-join patterns in the parallel-reduce algorithm.

This mechanism is illustrated in Figure 11.5. Flow lines with different structure correspond to different threads (4 threads in the figure). Fork vertices with lines of different structure indicate that task stealing has taken place. These vertices are paired with join vertices where the join() function is called. Some conclusions:

- The reduction is performed partly by the operator() function, and partly by the join() function.
- The operator() function accumulates the sub-range result to whatever previous result is already stored in the partial_area data. This guarantees that the reduction is correctly performed in the case of sequential execution without task stealing. Finally, the join() function performs the reduction at a joining point of two threads.

The code for the main() function is given below. There is nothing new besides the occurrence of the new Range object.

```
// Function to be integrated
double FCT(double x)
   {
   return sin(x);
   }

// The main function
int main(int argc, char **argv)
   {
```

Continued

```
double a, b;
int nTh, Gr;
InputData();
task_scheduler_init init(nTh);

RealRange R(a, b, Gr);
AreaBody B(FCT);
parallel_reduce(R, B);
std::cout << "\nFinal result for area is : "
          << B.partial_area << std::endl;
return 0;
}
```

LISTING 11.10

PRarea.C main function.

Example 3: PRarea.C

To compile, run make prarea. Input data is read from file prarea.dat.

11.6.2 SECOND EXAMPLE: LOOKING FOR A MINIMUM VALUE

The next example elaborates on an advanced example proposed in the "Parallelizing simple loops" section of the TBB User Guide [33]: the parallel search of the minimum value of a vector, as well as its vector index (location). This example is interesting because it shows in a different way the flexibility of the parallel_reduce algorithm. Only some minor modifications have been added here, showing in detail the way the algorithm operates. A size N vector X of doubles is initialized with uniform random values in the interval [-100, 100]. The purpose of the example is to search for the value and location of the minimum value.

The range class used here is the IntRange class introduced before. The body class stores a private copy of the target vector address, and two public data items: a double minval to store minimum values and an int minindex to store the corresponding vector index. In this example, the operator() function scans the given range looking for the minimum value, but the values of minval and minindex are updated only if minval beats the stored value holding partial results from previous sub-ranges. And, as shown in Listing 11.11, the join() function only updates its minval and minindex values if minval is beaten by the value held by the target task.

```
class MinvalBody
   {
   private:
    double *V;

   public:
    double minval;
    int    minindex;
   ...
```

```
    void join(const MinvalBody& A)
       {
       if(A.minval < minval)
          {
          minval = A.minval;
          minindex = A.minindex;
          }
       }
    };
```

LISTING 11.11

Partial listing of body class for the minimum search.

Complete MinVal.C example

The source file for the example we are currently discussing is MinVal.C. To compile, run make minval. The main() function is listed below:

```
int main(int argc, char **argv)
   {
   int N, Gr, nTh;
   Rand Rd(999);

   InputData();
   V = new double[N];  // allocate and initialize
   for(int n=0; n<N; ++n)
      V[n] = 200.0 * Rd.draw() - 100.0;

   task_scheduler_init init(nTh);
   IntRange R(0, N, Gr);
   MinvalBody B(V);
   parallel_reduce(R, B);
   std::cout << "\nOverall minimum value is : "
             << B.minval << "  at index " << B.minindex
             << std::endl;
   return 0;
   }
```

LISTING 11.12

MinVal.C main function.

The InputData() is the same as the one of the previous example: it reads N, nTh, and Gr from the file minval.dat. Then, it allocates and initializes the target vector V.

Notice that B is the initial function object, which, when passed to the algorithm, will start exploring the initial full range R. As we will soon verify, this function object will be copied at a fork with stealing, and then the copies will be destroyed at the corresponding join() call. At the end of the day, when the function returns, the initial function object B holds the final values of minval and minindex, which are read and printed to stdout.

Example 4: MinVal.C

To compile, run make minval. The input data is read from file minval.dat.

In the real code, some debug features are introduced in the body class in order to track the operation of the algorithm. A MinVal function object has another private internal integer variable called body_id that identifies its different split copies. When the initial function object B is created, its identity is 1. Each time the split constructor builds a new function object, it increases *a thread-safe counter* and assigns the next integer value to the new object. A thread-safe counter is provided by a simple class defined in the include file SafeCounter.h. In this way, all the different "split" copies of the initial function object created by the algorithm are identified by a unique integer rank.

Each time the operator() function is called, a debug message is printed before exit. This message includes:

- The identity of the function body that called the function.
- The identity of the executing thread. This is done by inserting a call to the appropriate library function (see Chapter 3).
- The sub-range that was explored, and the final values of minval and minindex.

The code employs a global mutex to serialize access to stdout, in order to print clear messages from each thread. Here are some typical program outputs with N=10000 and nTh=2. First, let us consider a granularity Gr=2:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Body ID : 1,  thread = 3077502720
Interval ( 5000, 10000 ), minval = -99.8998, at index 5184

Body ID : 2,  thread = 3076447040
Interval ( 0, 5000 ), minval = -99.9748, at index 4619

Overall minimum value is : -99.9748  at index 4619
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - --
```

This is clearly a straightforward fork-join with task stealing: there are two different function bodies executed by two different threads. The function body 1 split the initial interval and started execution of the spawned task covering the half-range [500, 1000). A second thread with a new function body executes the continuation task covering the half-range [0, 500). If a granularity Gr=4 is chosen next, the nested fork-join constructs are organized as follows:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Body ID : 1,  thread = 3078035200
Interval ( 7500, 10000 ), minval = -99.7154, at index 8499

Body ID : 2,  thread = 3076979520
Interval ( 2500, 5000 ), minval = -99.9748, at index 4619

Body ID : 1,  thread = 3078035200
Interval ( 5000, 7500 ), minval = -99.8998, at index 5184
```

```
Body ID : 2,  thread = 3076979520
Interval ( 0, 2500 ), minval = -99.9748, at index 4619

Overall minimum value is : -99.9748  at index 4619
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This output corresponds to the same initial fork-join with task stealing seen before, followed by a nested fork-join naturally without task stealing because there no other available idle threads. One thread uses body 1 to scan the interval [7500, 10000) followed by [5000, 7500), and the other thread uses body 2 to scan the interval [2500, 5000) followed by [0, 2500). Notice also that the second message sent by body 2 looks weird because it reports a minimum at an index 4519, which is not in the range being scanned. This is simply because body 2 has already found a minimum in another range—look at the previous message—and this value is still valid in the current sub-range. This is exactly the behavior we expect.

To be fair, I must admit that this execution mode is not exactly reproducible. It happens sometimes that there is no task stealing at all and that the execution is purely sequential with only one function body. Very occasionally you may find three function bodies resulting from two stealings, probably because one thread completed its two sectors very quickly and came back to steal the second sector of the other thread.

11.7 PARALLEL_FOR_EACH ALGORITHM

The parallel_for_each algorithm is a simplified version of the parallel_for algorithm, without the dynamic domain decomposition driven by a Range class. This algorithm is very useful when a specific action—a map operation—must be performed in consecutive elements stored in an arbitrary container. Its signature is given in Listing 11.13.

```
#include <tbb/parallel_fo_each.h>
...
template<typename Iter, typename Body>
void parallel_for_each(Iter first, Iter last, const Body& B);
```

LISTING 11.13

Parallel_for_each algorithm signature.

This algorithm operates as follows:

- Iter is an input iterator (iterator with reading capability) adapted to the target container.
- Body is a unary function object, which defines an operator() member function acting on container elements of type T, with the following signature:
  - void operator()(T& t) const
- The function object B defines an action on a container element. The template function above applies this action to all container elements in the half-open range [first, last).
- This action on the container range is done in parallel.
- The STL proposes an identical, sequential for_each() template function.

Notice that the Body class satisfies in this case the same requirements as the parallel_for body class, except that it acts on an individual container element, not on all elements in a range.

11.7.1 SIMPLE EXAMPLE: FOREACH.C

This example targets a STL std::vector<double> container, namely, a vector of doubles, of size 1000. The container elements are initialized to a uniform random number in [0, 1].

The action of the Body class is a map operation on the components of a vector of doubles, identical to the one used in the OpenMP example in Section 10.7: the Body function keeps calling a uniform random number generator and comparing with the element value, until the difference is smaller than a given precision. When this happens, the initial element value is replaced by the new, very close value. By increasing the precision, the amount of work done on each container element is increased.

As in the previous OpenMP example, there is a small subtlety in this code: the Body function object defines internally a local random number generator to perform its task. The parallel algorithm generates a different instance of the Body function object for each participating thread and, as in all the previous Monte Carlo examples, each task disposes of a *different* random number sequence. Therefore, different instances of the Body class must dispose of different initial seeds. Look at the sources to see how, as in the OpenMP example, this issue is handled using the SafeCounter auxiliary class.

Example 5: for_each.C

To compile, run make feach. This is the sequential code, using the STL for_each algorithm.

Example 6: ForEach.C

To compile, run make foreach. This is the parallel code, using the TBB parallel algorithm.

This example runs on two threads. For large precision (large amount of work done on each container element), the parallel speedup is quite acceptable.

11.8 PARALLEL_DO ALGORITHM

In its simplest form, this algorithm has the same signature as parallel_for_each in Listing 11.13: a function object Body class acts on all the elements of a container within the range defined by the two iterators. However, this algorithm is used when extra work must be optionally added. The TBB release has a complete example in the /examples/parallel_do directory—the parallel traversal of a directed acyclic graph—adapted to OpenMP and extensively discussed in Section 10.7.2.

Remember that in this example the starting point, after the construction of the graph, was the construction of a vector container including all the Cell pointers referring to root cells, with no ancestors. In the TBB implementation, the parallel_do algorithm starts by updating all the elements of this container. In updating a cell, reference counts of successors are decreased, and some may reach 0, in which case the successor is recursively updated. The capability of adding new work is used to

handle this recursive update. Readers are encouraged to look at the TBB example, easily accessible after all the explanations of the code provided in Section 10.7.2.

11.9 PARALLEL_INVOKE ALGORITHM

This algorithm invokes simultaneously several functions, and runs them in parallel. When the parallel_invoke template function returns, all the functions have completed their operation. The algorithm incorporates a set of overloaded functions that run between two and ten functions in parallel. Here are the template functions signatures:

```
#include <tbb/parallel_invoke.h>
...
template<typename Func0, typename Func1>
void parallel\_invoke(const Func0& f0, const Func1& f1);
...
template<typename Func0, ... ,typename Func9>
void parallel\_invoke(const Func0& f0, ... const Func9& f9);
```

LISTING 11.14

Parallel_invoke algorithm signature.

Notice that:

- Func0, Func1, ...Func9 template arguments are function objects that define an operator function with the following signature:
  - void operator()() const;
- As in the STL, pointers to functions can be used instead of function objects.

If pointers to functions are used, they must point to functions that receive no argument. According to the documentation, possible return values are ignored.

11.9.1 SIMPLE EXAMPLE: INVOKE.C

This example re-examines the Monte Carlo computation of π that was encountered several times in the past. Here, the idea is simple: a function is written that implements the Monte Carlo algorithm for a given number N of events, and returns in a global variable the acceptance count. If this is done for several functions simultaneously, the main() function can obtain an improved value of π by using the total number of events and the total acceptance.

Look at the source file Invoke.C. Two threads are used in this example. A unique function object class is defined that implements the Monte Carlo algorithm. Then, two different instances of this class provide the two independent function to be passed to parallel_invoke.

One of the problems of this algorithm is the impossibility of passing or returning values from functions. Notice, however, how the function object approach shortcuts this limitation in an elegant way. Parameters needed for the function operation can be passed via the object constructor, and return values can be stored internally as *public* data items to be read at the end of the day by the client code.

This is yet another simple example of the fact that function objects are *smart pointers to functions* that enable the existence of several identical functions with personalized internal state.

This example shows the same subtlety discussed before. The function object operator() function declares a local random number generator, and we want different instances of this class to use different random number sequences. Once again, the SafeCounter auxiliary class is used for this purpose.

Example 7: Invoke.C

To compile, run make invoke. This version of the code uses function objects.

Running the example, a very correct parallel speedup is observed, comparable to the other versions of this code. We have provided, for comparison, another version of the code using ordinary functions, to emphasize the fact that in this case one two different functions need to be defined, doing the same thing.

Example 8: InvokeFct.C

To compile, run make invokefct. This version of the code uses ordinary functions.

11.10 HIGH-LEVEL SCHEDULER PROGRAMMING INTERFACE

TBB has recently introduced the task_group class that implements an elegant and easy-to-use programming interface to the task scheduler. The official documentation is in the "Task Group" link of the Reference Guide [17].

This class can be used to run a group of tasks—which can be dynamically enlarged with new tasks once running—as a single, encapsulated parallel job. It implements the two basic synchronization patterns discussed for OpenMP tasks: waiting for direct children and waiting for all descendants. It also implements a task group cancellation utility, very much like the one discussed in the OpenMP environment.

The task_group class has two basic member functions: the run() member function that allocates and spawns a task, and the wait() member function that waits for the termination of all tasks previously spawned by the task_group object. It is important to keep in mind that this member function waits for all descendants, not just direct children. This section shows how this happens in detail, on the basis of the same examples already proposed in the OpenMP chapter.

11.10.1 DEFINING TASKS IN THE TASK_GROUP ENVIRONMENT

Tasks are in general defined in TBB as instances of a class derived from the basic task class, in which the execute() virtual function is defined. The specific derived task class must be defined, and objects instances of this class must be constructed and submitted for execution to the scheduler. This is what we will be doing in Chapter 16, when discussing the low-level scheduler API based on the task class.

This protocol has been enormously simplified in the task_group environment. In the first place, a task function is directly defined by a function object Func, which must provide the member functions listed below:

- Func::Func(const Func&): a copy constructor. If all the copy of the function object requires is copying data items, then the default copy constructor provided by the compiler is adequate.
- Func::~Func(): a destructor. Same comment as before.
- void Func::operator()() const: Task function executed by the function object.

In practice, all that is required is the definition of the constructor of the function object, as well as the operator() member function. Notice that whatever local data items in the parent task need to be captured by the new task must be passed via the constructor either by value or by reference. The operator() function itself receives no argument and returns void.

Once the function object class is defined, instances of this class must be allocated, to define a specific task. But the nice thing is that all these steps can be bypassed: the *lambda function syntax* for function objects discussed in Annex B enables the definition and allocation of function objects on the fly, at the place where they are needed. This leads, once the lambda syntax is understood, to very elegant and compact code.

11.10.2 TASK_GROUP CLASS

Here is the listing of the task_group class, defined in the tbb namespace:

```
class task\_group
   {
   public:
     task\_group();
     ~task\_group();

     template<typename Func>
     void run(const Func& f);           // spawn

     template<typename Func>
     void run_and_wait(const Func& f);  // spawn and wait

     task_group_status wait();          // wait for all

     bool is_cancelling();   // check if cancellation requested
     void cancel();          // request cancellation
   };
```

LISTING 11.15

Task_group class.

- The function run() receives a reference to a task function object. But the functor constructor, or a lambda expression, can equally well be passed. The examples in the next section show in detail how this works.

- The wait() function returns task_group_status, which is an enumeration with three symbolic values:
  - not_complete: not canceled, not all tasks in the group have completed.
  - complete: not canceled, all tasks in the group have completed.
  - canceled: task group received cancellation request.
- The function run_and_wait() is equivalent to run(); wait(); but guarantees that the task is run on the current thread.
- Two member functions, cancel() and is_cancelled(), enable programmers to cancel the operation of a group of tasks, in the way discussed below.

A few examples already encountered in the OpenMP context are discussed in the next section to show in detail how the task_group class monitors the execution of a group of tasks. Two important features will in particular be exhibited: the dynamic operation of task_group objects—tasks can be added when the group is already running—as well as the cancellation of a group of tasks.

The two cancellation member functions are very similar to the two OpenMP cancellation directives: requesting cancellation and checking if cancellation has been requested. But they work in a different way. In TBB, a task group object can be canceled at any time. If the cancellation request arrives before the tasks have been scheduled, they are not submitted for execution to the pool. On the other hand, the function is_cancelled() checks for cancellation and returns true of false, *but it does not cancel anything*. It is the programmer responsibility to implement the program logic that terminates a task when cancellation is requested.

11.11 TASK_GROUP EXAMPLES

The four examples developed in this section show in detail how function objects and lambda expressions are used to run and group of tasks, as well as various ways of using task_group objects to synchronize or cancel a group of tasks.

11.11.1 AREA COMPUTATION, WITH A FIXED NUMBER OF TASKS

The first example computes again the area of the function $4.0/(1 + x * x)$ in the interval [0, 1], whose value is π. This is done by activating Nth threads in the pool, and launching Ntk parallel tasks. The interval [0,1] is divided in Ntk sub-ranges of size 1.0/Ntk. Parallel tasks compute the area of each sub-range, and accumulate their partial results in a mutex protected global variable called result.

An orthodox approach is adopted in this case: a function object class AreaTask is explicitly defined, and its constructor is passed to the run() member function whenever a task is submitted for execution. Here is the listing of the source code, in file Tg1.C

```
double result;    // accumulates partial results
mutex  m;         // protects "result"
int    nTk, nTh;  // number of tasks and threads

double my_fct(double a);
double Area(double a, double b, double (*func)(double),
           double eps=EPSILON);
```

```
// Function object defining the task function
// – – – – – – – – – – – – – – – – – – – – – –
class AreaTask
   {
   private:
    int rank;

   public:
    AreaTask (int n) : rank(n) {}

    void operator() () const
       {
       double a, b, res;
       a = rank*(1.0/nTk);
       b = (rank+1)*(1.0/nTk);
       res = Area(a, b, my_fct, EPSILON);
          {
          mutex::scoped_lock slock(m);
          result += res;
          }
       }
   };

// – – – – – – – – – –
// The main function
// – – – – – – – – – –
int main (int argc, char *argv[])
   {
   nTh = 2;
   nTk = 4;
   // Override the number of threads and/or the number
   // of tasks from the command line.

   task_scheduler_init init(nTh);
   // – – – – – – – – – – – – – – – – – – –
   task_group tg;
   for(n=0; n<nTk; ++n) tg.run(AreaTask(n));
   tg.wait();
   // – – – – – – – – – – – – – – – – – – –
   std::cout << "\n Area result is : " << result
             << std::endl;
   }
```

LISTING 11.16

Area computation, fixed number of tasks.

In Listing 11.16, the AreaTask constructor takes an integer argument, which is the rank of the sub-interval whose area is computed. The task function just computes the area by using our traditional Area() library function, and accumulates the partial result in the mutex-protected global variable result.

The main() function code is straightforward. After initialization, a task_group object local to main() is created and used to submit the Ntk tasks and wait for their termination.

Example 9: Tg1.C

To compile, run make tg1. This example spawn four tasks running on two threads. The number of threads and the number of tasks can be overridden from the command line.

11.11.2 AREA COMPUTATION, TASKS WAIT FOR CHILDREN

The next example deals with the recursive area computation in which tasks either compute the area of a sub-range and return the result to the parent task (if the sub-range is smaller than the granularity G), or they split their integration domain into two sub-domains, launch two child tasks, and wait for their return value. In this recursive algorithm, the only task synchronization required is spawning two child tasks and waiting for their termination. Therefore, each task that needs to do so defines its own, local task_group object and uses it to run the two child tasks and wait for their completion.

If the standard procedure of writing explicitly a function object class to define the task function was adopted, an AreaTask class should be defined, whose constructor takes three arguments: the limits [a, b] of the target sub-range, and a double *s pointing to an address in the parent stack where child tasks deposits the return value. In other words: a and b are captured by value, but s is captured by reference.

This is not, however, what is done in this case. Rather, a recursive function is defined to compute the area of a given range, and a lambda expression is used whenever this function needs to be transformed into a function object. Here is the listing of the code source for this example, in file Tg2.C.

```
int    nTh;        // number of threads
double G;          // granularity

// This is a recursive function, not a class
// — — — — — — — — — — — — — — — — — — —
double AreaTask(double a, double b)
   {
   double result;
   if(fabs(b—a)<G)
      result = Area(a, b, my_fct);
   else
      {
      double x, y;
      double midval = a+ 0.5*(b—a);
      task_group tg;
      tg.run([&]{x = AreaTask(a, midval);});
      tg.run([&]{y = AreaTask(midval, b);});
      tg.wait();
```

```
        result = x+y;
        }
    return result;
    }

// – – – – – – – – – –
// The main function
// – – – – – – – – – –*/
int main (int argc, char *argv[])
    {
    nTh = 2;        // default values
    G = 0.4444;

    task_scheduler_init init(nTh);
    double result = AreaTask(0, 1);
    std::cout << "\n Area result is : " << result << std::endl;
    }
```

LISTING 11.17

Area computation, waiting for children.

Let us first examine the recursive AreaTask(a, b function. It returns the area of the range passed as argument. This function operates in the traditional way: if the sub-domain size is smaller than the granularity, it returns directly the area. Otherwise, it splits the range into two sub-ranges and computes the area recursively. In this case, this function defines two local variables x, y that recover the recursive results of the two sub-ranges, *and calls itself recursively*. Our intention, however, is to run child tasks to do this job. Therefore, the function:

- Defines a local task_group object, used to run and wait for the two child tasks.
- Runs the tasks. But now, the run() member function requires a function object as argument, not an ordinary function. Therefore, a lambda expression is used to transform the function into a function object. The lambda expression syntax is clear: the variables x, y that receive the return values are captured by reference, and the limits of integration are captured by value (see Annex B).
- The function, as usual, calls wait() before returning x+y.

Finally, look at the main() function: it calls directly the function AreaTask(0,1). Indeed, all the taskwait synchronizations in this problem are local and internal to each recursive task. This function does not return before all the recursive partial results have been correctly returned to the parent tasks. It is indeed possible to introduce yet another task_group local to main() to run AreaTask(0,1) as a task, but this is superfluous in this case (it will not be superfluous in the next example).

Example 10: Tg2.C

To compile, run make tg2. This example runs four threads, with G=0.24. The number of threads and the granularity can be overridden from the command line.

11.11.3 AREA COMPUTATION, NON-BLOCKING STYLE

Next, the traditional non-blocking style is implemented, in which tasks spawn child tasks and terminate. Tasks that finally compute areas of sub-ranges accumulate their partial results in a mutex protected global variable, and are called result in Listing 11.18.

In this example, only a unique, overall wait of the complete computation is needed. Therefore, a unique, global task_group tg is introduced. New tasks will be dynamically added to this already running task group, who will be accessed by all tasks, but only to run their children. The main() thread is the only one that waits on tg. Here is the listing of the source code, in file Tg3.C

```
int   nTh;
double G;
task_group tg;
mutex  m;          // protects result
double result;     // accumulates partial results

// This is a recursive function, not a class
// – – – – – – – – – – – – – – – – – – – –
void AreaTask(double a, double b)
   {
   double res;
   if(fabs(b–a)<G)
      {
      res = Area(a, b, my_fct);
         {
         mutex::scoped_lock lock(m);
         result += res;
         }
      }
   else
      {
      double midval = a + 0.5*(b–a);
      tg.run([=]{AreaTask(a, midval);});
      tg.run([=]{AreaTask(midval, b);});
      }
   }

// – – – – – – – – – – –
// The main function
// – – – – – – – – – –*/
int main (int argc, char *argv[])
   {
   double res;

   nTh = 2;     // default values
   nTk = 4;
   G = 0.2;
```

```
    task_scheduler_init init(nTh);

    tg.run([=]{AreaTask(0, 1);});
    tg.wait();
    std::cout << "\n Area result is : " << result << std::endl;
    }
```

LISTING 11.18

Area computation, non-blocking style.

The previous strategy has been maintained, of defining an AreaTask(a,b) function—which now returns void—and using a lambda expression to transform it into a function object when needed. Notice that, since there are no return values, all the variables in the lambda expression are now captured by value.

In this recursive computation, the global task_group tg is used to run new tasks, but nobody waits. Only the main thread that starts the recursion waits on tg, after running AreaTask(0,1) as a task, not as a function.

Example 11: Tg3.C

To compile, run make tg3. This example runs four threads, with G=0.24. The number of threads and the granularity can be overridden from the command line.

11.11.4 CANCELING A GROUP OF TASKS

Let us now look again at the database search emulation example: a set of Ntk tasks keeps calling a random number generator providing uniform deviates in [0, 1], until one of them gets a value close to a given target with a predefined precision. Each task disposes of a local generator delivering a personalized random suite.

A task function SearchTask(int n) is defined. This function will be, again, transformed into a function object by using a lambda expression. The integer argument is a rank assigned to each task, used to initialize its local random number generator. Here is the listing of the source code, in file DbTbb.C.

```
struct Data       // output passed to main thread
   {
   double d;
   int    rank;
   };

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

Continued

```
task_group tg;    // global task_group

// The workers thread function
// - - - - - - - - - - - - - --
void SearchTask(int n)
    {
    double d;
    Rand R(999*(n+1));
    while(1)       // infinite loop
       {
       if(tg.is_canceling()) break;   // check for cancellation
       d = R.draw();
       if(fabs(d-target)<EPS)
          {
          D.d = d;
          D.rank = n;
          tg.cancel();                // request cancellation
          }
       }
    }

// - - - - - - - - - - - - - - - - - - - - - - - -
// The main function gets the number of threads
// from the command line.
// - - - - - - - - - - - - - - - - - - - - - - - -
int main(int argc, char **argv)
    {
    int nTh = 2;
    int nTk = 4;

    // override from command line

    task_scheduler_init init(nTh);
    for(int n=0; n<nTk; ++n) tg.run([=]{SearchTask(n);});
    tg.wait();

    // Print search result
    // - - - - - - - - - --
    std::cout << "\n Task " << D.rank << " found value "
              << D.d << std::endl;
    return 0;
    }
```

LISTING 11.19

Emulation of database search.

Notice that the task_group tg is global: it is used by main() to run the worker tasks, but each task must also access tg to check or to request cancellation. Remember that the TBB runtime system will directly

cancel the task group if the tasks are not scheduled. Otherwise, the programmer must provide the termination code if a running task must be canceled. This is exactly what happens in the SearchTask(n) function: the task activity is an infinite while(1) loop that breaks if the group cancellation has been requested.

Example 12: DbTbb.C

To compile, run make dbtbb. This example runs four threads and four tasks.

11.12 CONCLUSIONS

This chapter has covered the high-level programming interfaces for running parallel applications in the TBB task-centric environment. It can be observed that, at this level, there is a strong correspondence between the OpenMP task API and the TBB taskgroup class features: waiting for direct children, waiting for all descendants, and canceling a parallel construct. What is left is the option or ordering the way tasks are executed by establishing hierarchical dependencies among them. In OpenMP this issue is controlled by the depend clause when a task is created. TBB is much more explicit, by exhibiting in more detail the inner workings its task scheduler, whose enlarged features are discussed in Chapter 16.