CHAPTER

USING THE TBB TASK SCHEDULER

16

16.1 INTRODUCTION

The power and the flexibility of the TBB high-level algorithms, including the task_group interface, have been discussed in Chapter 11. They are expected to operate efficiently in most practical cases. Nevertheless, some more advanced features of the TBB task scheduler may be needed, for example, when the application profile demands specific *event-like synchronizations among tasks*, implemented by the introduction of hierarchical relations among them. A first look at this feature was taken when discussing in Chapter 10 the depend clause recently introduced in OpenMP 4.0. The TBB environment, however, has been in operation for several years and it is more general. Besides task synchronizations, several refined services proposed by TBB—such as thread affinity or task recycling—are only accessible through the basic scheduler API.

This chapter contains rather advanced material, and as such it will probably not be of central interest to all readers. Moreover, this material is well described in the official TBB documentation—TBB tutorial and reference guide—and in James Reinder's book [34]. Nevertheless, a pedagogical effort has been made to summarize it here because these subjects are deeply connected to many other issues discussed in previous chapters, and because we wanted to offer a self-contained discussion of task-centric programming environments. Some of the advanced examples proposed at the end of the chapter are not explicitly discussed in the TBB documentation.

In order to facilitate a selective reading of its content, the different sections are classified as follows:

- Qualitative overview of the operation of the TBB thread pool: Sections 16.1 and 16.2.
- Full scheduler API: Sections 16.3 and 16.4. This is the complete API, based on the task class, that implements a very refined control of the scheduler, like establishing hierarchical relations among tasks that synchronize their activity, bypassing the scheduler by forcing the execution of a specific task, and much more.
- Using the TBB scheduler: advanced examples: Sections 16.6 and 16.7.

16.2 STRUCTURE OF THE TBB POOL

TBB proposes a sophisticated and powerful task scheduler API, inspired by Cilk's programming model. Tasks are registered for execution in a way to be discussed, and refined scheduling algorithms select the task that an available thread executes next, searching to optimize in particular cache memory efficiency

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00016-2

and load balance of parallel workloads. TBB also gives programmers the possibility of *bypassing* the scheduler, because a task has the option of enforcing the immediate execution of a successor task without submitting it to the task pool.

16.2.1 TBB TASK QUEUES

The structure of the TBB pool is shown in Figure 16.1. The pool is composed of a fixed number of worker threads, each one run by an available core. There are two kinds of data structures acting as *ready pools*, containing the tasks already submitted for execution:

- *Thread-owned ready pools*—one per worker thread—implemented as deques. Deques are double-ended queues in which elements can be retrieved either from the head (the oldest element) or from the tail (the most recent element).
- *Global task queue*, which looks like an ordinary first-in, first-out (FIFO) queue, where tasks are enqueued at the tail and dequeued from the head of the queue, and are therefore served on a fair FIFO basis. In fact, it would be more appropriate to speak of a quasi-FIFO queue because, for sake of scalability, TBB does not promise exact FIFO order, only fairness that no task will starve. This feature has been incorporated in recent TBB releases to offer programmers a way to deal with simple parallel contexts that do not require more than straightforward fair scheduling, like, for example, a server dispatching single tasks to manage individual client requests.

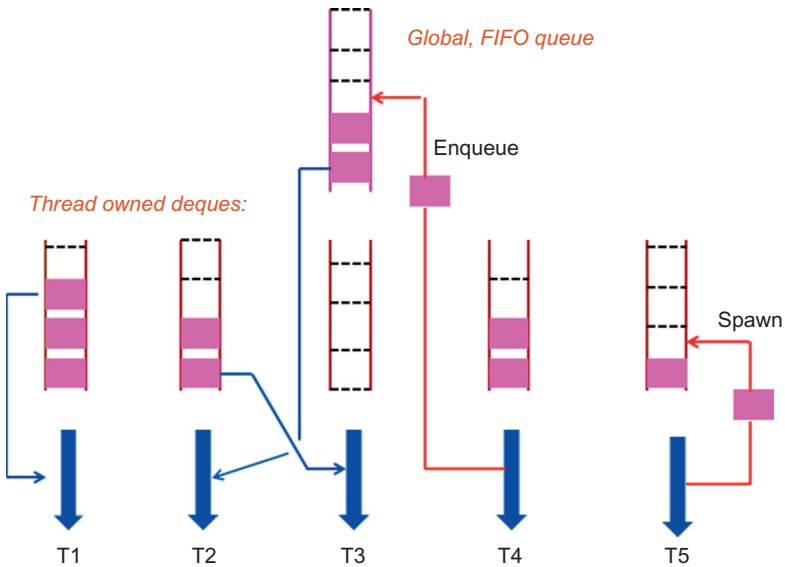

FIGURE 16.1

Structure of the TBB thread pool.

TBB establishes a simple convention for feeding these ready pools:

- A running task *spawning* a new task places it *at the tail of the executing thread deque*, as is the case of thread T5 in Figure 16.1.
- A running task *enqueuing* a new task places it at the tail of the global queue, as is the case of thread T4 in Figure 16.1.

The TBB pool implements therefore two different scheduling strategies. Usage of the global pool emphasizes fairness: tasks are executed in the order in which they are submitted; this is very natural in a throughput computing, embarrassingly parallel context in which tasks are totally independent. Use of the thread's deques, instead, emphasizes locality of memory accesses and optimizes the overall load balance of a parallel application, by activating a parallel treatment whenever there are resources available to implement it. This is the place where the added value provided by the built-in intelligence of the TBB scheduler operates. The operation of the global pool will not be discussed any further in the rest of this chapter.

16.2.2 FIRST LOOK AT THE SCHEDULING ALGORITHM

It is possible at this point to present a first simplified description of the scheduling algorithm by discussing how a thread proceeds to select a task for execution from the ready pools:

- First of all, the thread pops a task *from the tail of its own deque*, as is the case of thread T1 in Figure 16.1.
- If its own deque is empty, it pops a task *from the head of the global queue*, as is the case of thread T2 in Figure 16.1.
- Finally, if the global queue is empty, it steals a task from another randomly chosen thread's deque by popping it *from the head of a neighbor deque*, as is the case of thread T3 in Figure 16.1.

These rules are not exhaustive. Later on, we will see that in some cases the scheduler can be bypassed by forcing a thread to execute next a specific task not popped from the ready pools. This may be decided by the programmer, or it may be done implicitly by the scheduler itself when executing tasks with hierarchical relations among them.

It is not difficult to understand why the thread-owned deques are set to operate in the way described above. Task stealing converts the potential parallelism expressed by the decomposition of a job into tasks, into a mandatory parallel treatment. In the absence of task stealing, a task ends up being executed by the same thread that submitted it. In this case, when the task submission mechanism is undone, it is very natural to select *the most recent task* in the deque, which may benefit from the presence of the local L2 cache of data items recently accessed by the immediate predecessor task. Executing the most recently spawned task exploits data locality and optimizes L2 cache access.

Task stealing, by forking a task to another thread—that is, most probably to another core—gives up all possible cache advantages. Task stealing operates to keep busy idle threads with no target tasks available in their own deques. Since any possible locality advantages are lost anyhow, there is no reason to steal the most recently spawned tasks. This is why task stealing targets the oldest tasks in the victim deques.

16.3 TBB TASK MANAGEMENT CLASSES

The task scheduler API is composed of three basic task management classes, and a few others that implement specific services.

- The task_scheduler_init class, discussed in Chapter 11.
- The task_group class, a high-level programming interface to the scheduler also discussed in Chapter 11.
- The task class, which constitutes the basic, low-level programming interface for task management and synchronization. It incorporates a substantial number of services that enable refined control of the task scheduler.
- A few other subsidiary classes complete the task services (for details, see the Reference Guide) [18]:
  - The task_list class, useful to spawn several tasks at once, rather than doing it one by one.
  - The task_group_context class, a collection of tasks that maintains its identity at execution time. The library allows programmers to set up a common priority level for them all. Tasks in the group can be explicitly canceled by the programmer, or can be implicitly canceled if the code raises an exception.
  - The empty_task class, representing a task that does nothing. This is a very useful tool for task synchronization that will be used intensively in our examples.

16.4 COMPLETE SCHEDULER API: THE TASK CLASS

From now on, we focus on the complete TBB task programming interface, based on the task class. Chapter 11 has shown how the task_group class enables the submission, execution, and possible cancellation of a group of tasks. The task interface provides the programmer with more refined control of the scheduler operation, in particular by enabling the explicit management of task dependencies, which are a key element in the TBB scheduler design.

This section develops the following:

- How to instantiate TBB tasks.
- How tasks are mapped to threads.
- The basic hierarchical relationships among tasks.
- The different ways of allocating TBB tasks, in order to take into account their hierarchical dependencies.
- Two different TBB paradigms for spawning and waiting for a group of child tasks.
- The complete task scheduling algorithm.

16.4.1 CONSTRUCTING TASKS

Task objects are constructed using exactly the same procedures already used in the NPool utility (obviously inspired by TBB):

- Define a class derived from the base tbb::task class.
- This derived class may implement as many data members and auxiliary member functions as needed for the task operation.
- External data arguments needed for the task operation can be captured through the class constructor.
- The derived class must override the execute() virtual function of the base task class. This defines the task operation.
- Task objects are allocated by using overloaded versions of the new operator, in the way described in the following sections.

The listing below shows a task class, the MyTask class that will be used in many simple examples that follow. Task objects of this class have an integer rank and a vath Timer utility as private data members. The rank is a task identifier initialized by the constructor. The Timer is used to block the running thread for a given number of milliseconds, to simulate a long operation. These task objects print an identification message, wait in a blocked state for 1 s, and terminate.

```
class MyTask : public task
   {
   private:
   int    rank;
   Timer  T;

   public:
   MyTask(int r) : rank(r) {}
   task *execute()
      {
      std::cout << "\n Executing MyTask task, rank = "
                << rank << std::endl;
      T.Wait(1000);
      return NULL;
      }
   };
```

LISTING 16.1

MyTask class

The basic class task has a large number of hidden data items and public member functions that implement all the scheduler services, and which are naturally inherited by the derived class. The execute() function implementing the task operation takes no arguments and, as was the case with the function objects used in the task_group environment, any needed information required for the task operation must be passed via the task constructor.

The execute() function returns a task*, namely, a task reference. *This implements a scheduler bypass*. If the returned task* is not NULL, the executing thread executes the returned task as soon as the execute() function exits. The returned task is not submitted to the pool.

16.4.2 MAPPING TASKS TO THREADS

TBB tasks are executed by threads in the following way:

- *Tasks are mapped to physical threads in a nonpreemptive way*. This means tasks are tied to a specific thread for the whole duration of their lifetime. When the execute() function starts, the task is bound to the thread until execute() returns. There is no task migration across threads (as may be the case of untied tasks in OpenMP).
- When running the execute() function of a task, a thread may suspend its execution and service another task *only when it waits on child tasks*. In this case, the thread may run one of the child tasks or, if there are no pending children, it may steal and run a task created by another thread.
- If a task blocks for some reason other than waiting for children—locking a mutex, or performing a spin or idle wait—the executing thread remains blocked. This is the normal behavior of threads.

Note that this mechanism was borrowed in the NPool utility (except that there is no task stealing in NPool). Suspending tasks is needed to avoid deadlocks that may occur when there are more active tasks than threads in the pool.

16.4.3 HIERARCHICAL RELATIONS AMONG TASKS

The TBB scheduler sets up the tools needed to cope with task synchronizations, by establishing hierarchical relations of the parent-child kind among tasks.

In the vath thread pools as well as in OpenMP, we think of a parent task as a task that spawns a few child tasks, and then, optionally, either terminates or waits for its children. In these cases, the parent-child relationship can be thought of as a describing a transition from a sequential to a parallel region, the child tasks eventually joining the parent tasks if they are waited for.

In TBB, this picture is somewhat different. It is more appropriate to think of the transition from a parallel to a sequential region. Parents are, in fact, identified to successors, corresponding to the continuation task that takes over when a parallel section terminates, and children are called predecessors, corresponding to the parallel region tasks. TBB tasks have, among others, two internal properties:

- *successor* (this property applies to child tasks in a parallel region). This is a pointer to the successor task that will be executed after the current task and its partners terminate. This pointer is either set:
  - When the task is allocated as a child of the successor task (see the next section).
  - By the user code, by calling the set_parent() member function on the target task (see the "synchronization" part of the task class Listing 16.16 at the end of the chapter).
- *refcount* (this property applies to successor tasks). A reference count is an integer that counts the number of tasks that have this one as successor (parent). This reference count is decreased by the scheduler every time one of the child tasks terminates.
  - The successor task reference count is set by the programmer, by a call to the set_ref_count(int n) member function (see Listing 16.16). Indeed, no one else knows how many children have been attached to the successor task.

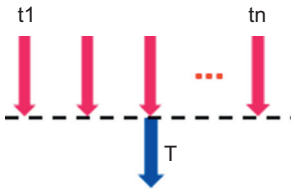

FIGURE 16.2

Successor (parent)-predecessor (children) relationships.

– Decreasing the reference count is done directly by the child tasks when they terminate. Indeed, the child task knows who its parent (successor) is.

Imagine now that n tasks, t1, t2, ..., tn are allocated as children (predecessors) of a task T. Figure 16.2 shows the predecessor-successor mechanism:

- The reference count of the successor task T must be set to the number n of its children.
- The n predecessor tasks are spawned by some other task, not by T. *We will soon explain the slight change that must be made to this protocol when the children tasks are spawned by T itself.* As they terminate, the reference count of T is atomically decreased.
- The thread executing the last terminating predecessor finds that the T reference count has reached zero, and executes T. *We have here again a scheduler bypass*: the successor task is immediately executed, without being spawned into the task queues.

In Figure 16.2, the parent task is not necessarily the task that spawned t1, ..., tn. It is the task T declared as parent of its children, either when the children were allocated (see next section) or by a call to the set_parent() member function by each one of the child tasks.

16.4.4 ALLOCATING TBB TASKS

Task objects must naturally be created (allocated) before they are activated, and TBB allows programmers to define the hierarchical role of a task at this time. Three hierarchical allocation options are available:

- Tasks can be allocated as root tasks (no parent task). The allocation of root tasks is naturally performed by a static function, not related to any task object. The root allocation of a task reference *T, instance of the MyTask class, is made by the following function call:

```
MyTask *T = new(allocate_root()) MyTask(n);
```

- Tasks can be allocated as children of a given successor task. The allocation of a child task tn as a child of T in Figure 16.2 is performed by a allocate_child() member function called by the successor task T:

```
MyTask *tn = new(T.allocate_child()) MyTask(n);
```

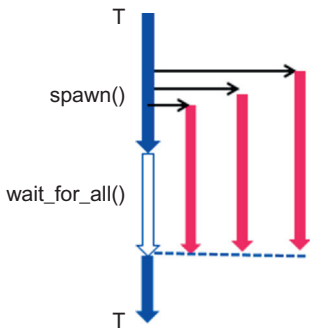

FIGURE 16.3

Task spawning children and waiting for termination.

- Finally, tasks can be allocated as continuations of other tasks. This is a subtle mechanism deployed by TBB to implement a fork-join pattern in which a task spawns several child tasks and terminates, not waiting for children, and is replaced by a continuation task succeeding the child tasks. This mechanism is introduced to enable a continuation stealing like Cilk, instead of the natural child stealing in TBB, as discussed in Chapter 11. This subject is discussed in Chapter 8 of [35]. The allocation of a continuation task Tc is also performed by the task object T that will be replaced, and which *transfers its own parent task to the newly allocated continuation task*. This mechanism is shown in Figure 16.3:

```
MyTask *Tc = new(T.allocate_continuation()) MyTask(n);
```

Why should the task that allocates a continuation task transfer its parent (successor) pointer to the new task? Since the continuation task that takes over when the children terminate is replacing the initial task that spawned them, it is natural that it inherits its successor, in order to preserve the enclosing hierarchical task organization in which the fork-join pattern is inserted.

Listing 16.16 of the task class in Section 16.9 at the end of the chapter shows several allocation functions that return internal data types called proxyN, with N=1, …, 4. The point is that the TBB library needs to adapt the action of the new operator to the type of allocation requested. New overloaded versions of the new(proxyN) operator are defined, which take these data types as input and perform the memory allocation in the appropriate way. In practice, these data types can be bypassed by directly passing to the new() operator the allocation function call, as shown above.

16.4.5 TASK SPAWNING PARADIGMS

These are the two fork-join TBB paradigms for spawning tasks:

- A task spawns the children and at some point blocks waiting for their termination, as shown in Figure 16.3.
- A task spawns the children and terminates, and a different continuation task is adopted to succeed the children, as shown in Figure 16.4. A number of examples will soon follow.

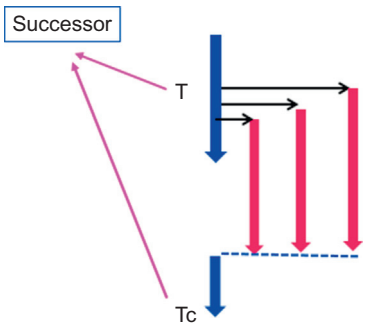

FIGURE 16.4

Allocating a continuation task.

Spawning tasks with no parent (successors) and terminating—as was done in one of the task_group examples in Chapter 11—is also possible, but the problem is that in the low-level task API there is no explicit task_group synchronization pattern that enables a task to wait *for all its descendants*. Only the taskwait pattern, waiting for direct children, is immediately available. Waiting for all descendants can be implemented in this context, using advanced synchronization techniques to be discussed later on.

16.4.6 SPAWNING TASKS WITH CONTINUATION PASSING

Continuation passing is a straightforward application of the predecessor-successor mechanism discussed above. The spawn mechanism—shown in Figure 16.2—proceeds as follows:

- First, the continuation task Tc is allocated as a continuation of T.
- Next, the parallel region tasks tn are allocated as children of Tc.
- The reference count of Tc is set to the number of children n. The successor task will be run directly by the thread executing the last terminating child.
- T spawns the child tasks and terminates.

As the examples will show, this task spawning paradigm requires some tricky coding, and should really be considered as an optimization option, because there are differences that may impact performance. As stated before, this issue is related to the different stealing strategies mentioned in the fork-join discussion in Chapter 11: child stealing in TBB versus continuation stealing in Cilk Plus, which implements a more efficient stack usage. Continuation passing allows TBB to implement the Cilk Plus continuation stealing strategy, at the price of more complex coding.

This is a complex issue, and a complete discussion is available in Chapter 8 of [35]. A simple observation can be given, to give an idea of the issues involved. In continuation passing, the initial task terminates, and its executing thread is released after cleaning all the task local variables from its stack. The executing thread just liberated can proceed to execute another task. If, instead, a task waits for children, its executing thread is not released because the task remains active, and its local data remains alive in the executing thread stack. A thread executing a task waiting for children is not blocked: TBB allows the executing thread to suspend the task, execute another task, and resume later on execution of

the initial task. However, the thread stack grows more than in the continuation passing case, and the risk of stack overflow is larger when running tasks that allocate a large number of local variables.

16.4.7 SPAWNING TASKS AND WAITING FOR CHILDREN

TBB also proposes naturally the classical task spawn mechanism: the successor task T spawns its children and eventually blocks waiting for them before resuming execution. This is shown in Figure 16.3. The protocol is now different, because the successor task is already running:

- The reference count of the task T is set to n+1, n being the number of children.
- Then T spawns its n predecessor tasks. At some point, T executes a member function call that waits for them to terminate, and resumes execution.

Setting the successor reference count to n+1 instead of n—by adding +1 for the wait—is critical. In the basic protocol in which the predecessors *are not* spawned by the successor task, the successor is automatically launched when the reference count reaches 0. However, in this case, *another thread is already waiting to run the successor* so the successor cannot be launched by the last child who terminates. Launching a running task crashes the code. The extra value in the reference count takes into account the fact that the successor is waiting for termination of children, not waiting to start execution.

16.4.8 TASK SPAWNING BEST PRACTICES

There are several member functions that can be used to spawn child tasks and eventually wait for their completion, as shown in the "synchronization" part of Listing 16.16. Note that the spawning function are always static: they can be directly called either from the main() function—which is not a TBB task—or from inside a task function code. They are not associated with a particular task object.

When several tasks need to be spawned, the choice is offered of spawning them one at a time, or first encapsulating them in a task_list object, and then spawning them all in a single function call: the spawn() functions take as arguments either a task reference or a task_list reference. Consider the case of a task T that needs to spawn n child tasks t1, t2, …, tn and wait for their termination. In all cases:

- T must allocate the child tasks t1, …, tn by using allocate_child() inside the new() call.
- T must call set_ref_count(n+1), to set the initial value of its reference count before spawning the child tasks.

After doing that, T has several options for spawning the n child tasks:

- Pack the n tasks inside a task_list TL, and call spawn_and_wait_for_all(TL).
- Spawn the n tasks—either one by one or via a task list—and then call wait_for_all().
- Spawn directly the first n-1 tasks and then call spawn_wait_for_all(Tn) for the last one.

As emphasized in the TBB documentation, the spawning mechanism just described is not, however, the most efficient way of spawning several tasks and waiting for their termination in the TBB environment. The point is that, as explained in Section 16.2.1, they are all queued in the deque of the running thread that allocates and spawns them. They will, of course, be stolen by other idle threads if available. But it is more efficient, from the point of view of L2 cache reusage, if they are directly

created and placed in the deques of other threads that may run them. This is why a recursive task-generation mechanism, in which a task spawns two children and waits, which in turn spawns children and wait, and so on and so forth, is to be preferred if possible.

Examples dealing with these task spawning best practices will be proposed in Section 16.6.

16.5 MISCELLANEOUS TASK FEATURES

16.5.1 HOW THREADS RUN TASKS

When a thread runs a task, the following steps are performed:

- The thread invokes execute() and waits for its return.
- If the task has not been marked by a "recycle" method, requesting that the task is preserved to be reused, as explained in the following Section 16.5.3:
  - The executing thread destroys the task.
  - If the parent task is not null, the thread atomically decrements the successor reference count. If the reference count becomes zero, it executes immediately the successor task.
- If the task has been marked for recycling (as child or continuation) nothing is done. *The successor reference count is not decreased*, and the task is not destroyed.

Keep in mind—for future discussion about task recycling in Section 16.7—that the predecessor-successor protocol of decreasing the successor reference count *only applies if the task is destroyed*; it does not apply if the task is recycled. By default, tasks are destroyed after execution.

16.5.2 COMPLETE TASK SCHEDULING ALGORITHM

Section 16.2.2 presented a first preliminary overview of some of the priorities adopted by an idle thread to select a task to execute next from the available ready pools. The picture can now be completed by listing all the actions taken by a thread to decide on the next task to execute. When a thread finishes executing a thread T, the next task to be executed is obtained by the first rule below that applies:

- The task returned by the execute() method, if the returned pointer is not null.
- The successor of T, if T was the last completed predecessor.
- A task popped from the tail of the thread's own deque.
- A task with affinity for the thread (see the later discussion on memory affinity).
- A task popped from the head of the shared queue, as explained before.
- A task popped from the head of another randomly chosen thread's deque (task stealing).

16.5.3 RECYCLING TASKS

By default, a task object is destroyed when its execute() function terminates. Since TBB is tailored for a microtasking programming style, the memory management overhead related to task creation and destruction can become important in some cases. Consider, for example, the molecular dynamics

example MdTBB.C. Lightweight tasks are created and destroyed at each time step in the computation of the particle trajectories: first the acceleration tasks, and then the momentum and coordinate update tasks. It makes sense to try to reuse the same tasks at each iteration, when dealing with a large system with a substantial number of parallel tasks and time steps. This problem is examined in detail in Section 16.9.

16.5.4 MEMORY AFFINITY ISSUES

The importance and relevance of thread affinity issues have been discussed in the OpenMP chapter. TBB provides some support for thread affinity, well described in the documentation, but the memory affinity strategy is different. The OpenMP thread affinity strategy is tailored for a thread-centric environment, where tasks are uniquely mapped to threads. In this case, the thread affinity interfaces select a placement of threads on cores. In a task-centric environment, this approach does not apply. The TBB thread affinity service focuses on establishing a togetherness relation between tasks, by informing the runsystem that they should, in possible, be run by the same thread.

Indeed, a task accessing some global data may want to spawn another task that accesses the same global data. In this case, memory access performance will be enhanced if the task is executed by the same thread. TBB therefore gives the programmer the possibility of defining the *affinity identity* of the initial thread—which can be thought of as some kind identification of the executing core—and passing it to a spawned thread. Then, as discussed before, the affinity identity can be used, if other high-priority criteria fail, to select the task that has the same identity as the terminating task. The TBB documentation insists on the fact that task affinity is a hint for the scheduler: it is not a forcing criteria, and in some cases it may be ignored. Basically, there is a fundamental conflict between affinity and load balancing.

16.6 USING THE TBB SCHEDULER

A few examples follow that illustrate the operation of the TBB scheduler, looking at a few different programming style options for accomplishing some simple tasks.

16.6.1 BLOCKING VERSUS CONTINUATION PASSING EXAMPLES

Consider a code where some initial sequential computation is done first, then a parallel region is activated, and finally the code returns to a sequential region to complete the computation. This example is organized in the way shown in Figure 16.5: the main thread launches the initial sequential task (called B in the figure), which in turn spawns three worker tasks (called M in the figure) that perform some parallel treatment, and finally B takes over after waiting for its children.

Main thread code

Consider first the main thread code. The important observation here is the fact that the main() function *is not running as a TBB task object*. Therefore, this function cannot use the member functions of the task class; it can only access the *static* functions of the class, which behave as ordinary global functions. Taking a look at the task class, we observe that there is not much choice: the only option available to main() is to use spawn_root_and_wait() to spawn either a single task or a task list. Note also that, since tasks must be allocated before being spawned, the main() function can only allocate and spawn root tasks.

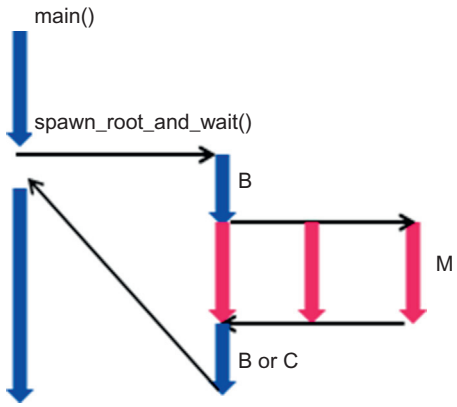

FIGURE 16.5

Forking and joining a parallel section.

We then define a BasicTask class—to be defined later—to construct the object B spawned by main(), which sets the stage for the computation and spawns in turn the three worker tasks of the parallel section as instances of the MyTask class discussed above. Worker tasks are identified by their integer rank. The code for the main() function is given below. This code is generic, and it will be the same for all the examples discussed in this and the following section (as we said, there is not much the main function can do).

```
int main(int argc, char **argv)
   {
   int n;
   if(argc==2) n = atoi(argv[1]);
   else n = 2;
   Timer T;

   task_scheduler_init init(n);
   std::cout << "\nMain starts "<< std::endl;

   BasicTask *B = new ( task::allocate_root() ) BasicTask();
   task::spawn_root_and_wait(*B);
   std::cout << "\nMain ends " << std::endl;
   }
```

LISTING 16.2

Main() code.

The main thread, therefore, spawns the task or tasks that drive the job, and waits. Note that the main thread is in principle doomed to remain blocked doing nothing. For a TBB task, it is possible to spawn the child tasks, and then do some other useful work before waiting for their termination. However, main() is not running as a TBB task. We will see later on how it is possible to modify this context, by setting up a stage in which main() spawns a task, continues to do something else and only then waits for

the termination of the spawned task. But this requires advanced synchronization tricks to be discussed later on.

The two examples that follow differ only in the implementation of the BasicTask class that drives the computation.

Example 1: blocking on children

The definition of the BasicTask class is given below, for the case in which the B task waits for its children after spawning the parallel region.

```
class BasicTask : public task
   {
   private:
   Timer  T;

   public:
   BasicTask() {}
   task *execute()
      {
      // Allocate three MyTasks as my children
      // - - - - - - - - - - - - - - - - - - - -
      MyTask* B1 = new( allocate_child() ) MyTask(1);
      MyTask* B2 = new( allocate_child() ) MyTask(2);
      MyTask* B3 = new( allocate_child() ) MyTask(3);

      set_ref_count(4);  // refcount=(3+1) because I wait

      spawn(*B1);        // spawn childs
      spawn(*B2);
      spawn_and_wait_for_all(*B3);
      return NULL;
      }
   };
```

LISTING 16.3

BasicTask class, blocking style.

This class has a trivial constructor because no data items need to be passed to the tasks. Note that:

- The three MyTask workers are allocated as children of the running BasicTask object, called B. This automatically makes B the successor of the three MyTask workers.
- The reference count of the running BasicTask object is set to 4, one more than the number of spawned children. This is because B waits for its children. If the reference count was set to 3, it would reach zero when all children terminate, and the running task would be automatically spawned. However, spawning a running task crashes the code (try).

In this case, although the thread running task B is not released, it can suspend B while waiting in order to service another thread. This avoids wasting CPU resources on inactive tasks (and avoids deadlocks, as often emphasized before).

Example 1: Blocking.C

To compile, run make block. The number of threads in the pool is passed via the command line (the default value is 2).

Example 2: continuation passing

Next, the definition of the BasicTask class is given for the case in which the B task terminates, and is replaced by a continuation task. The class ContTask is again trivial; the task just writes an identification message and terminates.

```
class BasicTask : public task
   {
   private:
   Timer  T;

   public:
   BasicTask() {}
   task *execute()
      {
      // Allocate the continuation task C
      // – – – – – – – – – – – – – – – – – –
      ContTask* C = new(allocate_continuation()) ContTask();

      // Allocate three MyTasks as children of C
      // – – – – – – – – – – – – – – – – – – – –
      MyTask* B1 = new(C–>allocate_child() ) MyTask(1);
      MyTask* B2 = new(C–>allocate_child() ) MyTask(2);
      MyTask* B3 = new(C–>allocate_child() ) MyTask(3);

      C–>set_ref_count(3);  // C is spawned by last child

      spawn(*B1);        // spawn childs
      spawn(*B2);
      spawn(*B3);
      return NULL;
      }
   };
```

LISTING 16.4

BasicTask class, continuation passing style.

- The three MyTask workers are now allocated as children of the continuation task, called C. This makes the continuation task C the successor of the three MyTask workers.
- The reference count of the successor task C object is set to 3, the number of spawned children. The continuation task C is automatically executed when all children terminate and its reference count reaches zero.

Remember that the main() function spawns the initial task B and waits for its termination. In this case, B terminates when the spawned tasks M start execution, and its executing thread is released.

However, in executing the example code you will observe that main() prints its termination message *after the continuation task* C *is finished*. This continuation task, probably run by another thread, is fully playing its role of continuation of B, and taking its place in the wait mechanism of the main thread.

Example 2: Continuation.C

To compile, run make cont. The number of threads in the pool is passed via the command line (the default value is 2).

16.6.2 RECURSIVE COMPUTATION OF THE AREA UNDER A CURVE

The traditional example of the recursive area computation, already discussed in the context of the OpenMP, task_group and NPool environments, is reexamined next, looking at this problem from the point of view of the task API.

The first thing to be observed is that there is no direct way of waiting for all descendants. Therefore, the nonblocking programming style in which tasks either generate child tasks or compute and store partial results in a global variable is not directly accessible here. The next section will discuss how this programming style can be implemented, using more advanced synchronization techniques. For the time being, children have to return partial results to the successor (parent) task. The two options discussed in the simple preceding example are therefore open to us: tasks recursively spawn children and wait for their termination, or tasks recursively spawn children and terminate, being replaced by a continuation task. *It is the continuation task that recovers the partial results returned by the child tasks*.

The main() function, listed below, is the same for the two cases. But there are differences in the recursive task function code.

```
int main(int argc, char **argv)
   {
   double area;
   InputData();     // read Nth and G
   task_scheduler_init init(Nth);
   AreaTask& A = *new(task::allocate_root()) AreaTask(0, 1, &area);
   task::spawn_root_and_wait(A);
   std::cout << "\nArea is : " << area << std::endl;
   }
```

LISTING 16.5

Main function for the recursive area computation.

The first case, in which tasks block waiting for children, has exactly the same structure as the NPool code discussed in Chapter 12. The AreaTask constructor receives as argument the return value address in the parent task. The execute() function declares two local variables x and y where the child partial results are retrieved, and the task returns (x+y) after waiting for its children. The complete listing is in the source file AreaRec1.C.

Example 3: AreaRec1.C

To compile, run make arearec1. The number of threads in the pool as well as the granularity are read from the file arearec.dat.

Continuation passing case

The necessity of handling return values introduces some subtleties in the recursive AreaTask class. These issue are well described in the TBB documentation [18, 34] and the discussion that follows just underlines the main points. The full source code, in file AreaRec2.C, incorporates a number of useful comments. First, a continuation task is defined in the listing below.

```
class AreaContinuationTask : public task
   {
   public:
   double *sum;
   double x, y;

   AreaContinuationTask(double *sum_) : sum(sum_) {}
   task* execute()
      {
      *sum = x+y;
      return NULL;
      }
   };
```

LISTING 16.6

AreaContinuationTask class.

All the continuation task does to collect two partial results from its children (predecessors) and return their sum to its parent. This class has three data items: the doubles x and y, where the predecessor (child) tasks deposit their partial results, and a pointer to a double sum where the continuation task returns its partial result (x+y) to its own successor (parent). This address sum is passed via the constructor when the continuation task is allocated by the task to be replaced, which transfers the return address received when it was allocated by its own successor.

In the initial AreaTask class, the changes to be made are the following (see the source AreaRec2.C):

- The initial task allocates its continuation C, passing the return value address.
- The two child tasks are allocated as children of C, passing the addresses of C.x and C.y, where they have to deposit their return values.
- The two children are spawned, as before. The reference count of C is set to 2, because this task is not waiting: it will run when the children terminate.

Example 4: AreaRec2.C

To compile, run make arearec2. The number of threads in the pool as well as the granularity are read from the file arearec.dat.

16.7 JOB SUBMISSION BY CLIENT THREADS

Our attention moves next to an interesting issue: how to implement, using the TBB scheduler API, the programming style adopted by the task group utilities in OpenMP or TBB, as well as in the NPool task pool: a thread submits a complex parallel job, does something else and, when the time comes, waits for the job termination. This issue is cleanly handled by TBB with the task_group class, as discussed in Chapter 11. However, this class does not provide access to all of the task scheduler services. It is useful to see how a taskgroup-like feature can be implemented by using the task scheduler API. This is possible by using some advanced synchronization techniques, well described in the TBB documentation, which are the subject of this section. The discussion that follows is in fact a refined exercise in task synchronization: how, because of the hierarchical relations, tasks can be used to synchronize instead of compute.

In order to have a clear understanding of the issues at hand, take a look at the "synchronization" part of the task class listing in Annex, Section 16.10, where all the available functions for spawning and waiting for child tasks are declared. Remember that ordinary member functions can only be called by TBB tasks (and not by the main function, which is not running as a TBB task). Static functions, instead, are global functions that can be called by anyone. We observe that:

- For TBB tasks, there are independent spawn() and wait_for_all() function calls available, which implement the standard taskwait mechanism: a task spawns as many tasks as needed, continuing to do something else and then, at the right time, performs the wait call to wait for direct children. *However, there is no immediate way of waiting for all possible descendants*.
- *There are no static functions for an isolated wait call*. Therefore, the main function can only call spawn_root_and_wait(). It cannot directly perform other activities between the spawn and the wait.

It would be useful, instead, to have the possibility for the main function or an arbitrary TBB task, of launching a complex job, eventually doing other things, and waiting when the time comes for the termination of all the tasks generated by the job. The software architecture required to handle this problem, shown in Figure 16.6, is relatively simple and based on the fact that the wait for the end of the job can be delegated to another task—an empty task—whose only role is to wait, and which does not even need to run. This mechanism is a critical part of the TBB design, well advertised in the TBB documentation.

The software architecture shown in Figure 16.6 is implemented in the following way:

- The main thread allocates an empty task E as a root task. *This task will never run*.
- An auxiliary task called B in the figure is allocated by main as a child (predecessor) of the empty task E. Its role is to trigger the parallel job represented by the grey box.
- This auxiliary task B will henceforth be called the *work-holder* task. It encapsulates the gray box parallel job, as a successor of the complete job. This means the initial tasks of the job, spawned by B, will be allocated as children of B.
- The reference count of the empty task E is set to 2=1+1, so that this task *waits but is not executed when its predecessor* B *completes*.
- The empty task spawns its child B task.
- When the time to wait comes, main calls E.wait_for_all(). When this function returns, the gray box has been completed.

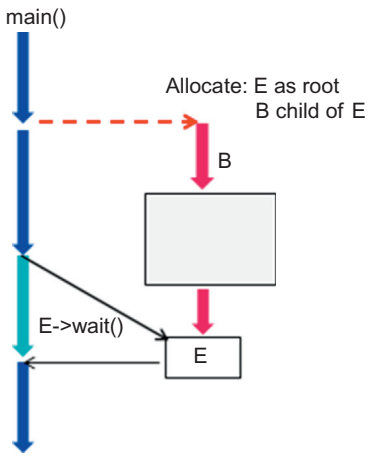

FIGURE 16.6

Job submission by the main thread.

16.7.1 GENERIC INTERFACE FOR JOB SUBMISSION

The previous software organization can be implemented in a generic programming interface allowing the main function of a TBB task to submit a complex job for execution, do something else, and then wait for termination:

- empty_task* Submit(): This is a template function, where the template parameter T encapsulates the submitted job. It returns an empty task reference that acts as a job identifier.
- void Wait_For_Job(empty_task *e): Waits for the job corresponding to the empty task e.

Here is the definition of these functions.

```
template <typename T>
empty_task* SubmitJob()
   {
   empty_task* E = new( task::allocate_root() ) empty_task;
   E->set_ref_count(2);
   T *t = new( E->allocate_child() ) T();
   E->spawn(*t);
   return E;
   }

void WaitForJob(empty_task* E)
   {
   E->wait_for_all();
   E->destroy(*E);
   }
```

LISTING 16.7

Generic functions for job submission.

Note that in defining the generic SubmitJob() function, it is explicitly assumed that the work-holder task class T has a trivial, no argument constructor T(). Note also that, in the WaitForJob() function, once the empty task has finished waiting for the job termination, it is explicitly destroyed by a call to the member function destroy(). The reason is that this task has only one predecessor, and that its reference count has been set to 2, as in the case of a running task waiting for children. However, *this task was never run*, so it was never deleted by the task scheduler. This is why it is good programming practice to explicitly destroy it.

The job ID returned by the SubmitJob() function is passed to the second function to wait for the job. This works exactly like the job submission interface of the NPool utility in Chapter 11. Jobs can be submitted and then waited for later on *in any order*.

16.7.2 SUBMITTING AN IO TASK

The code for this very simple example is in the source file SynchIO.C. The IOTask class submitted for execution has a simple execute() function that writes an initialization message, waits for a given number of milliseconds, and writes a termination message. The time-wait duration is read directly from a global variable iowait, so this class does not need an explicit constructor to capture a data value. Here is the listing of the class.

```
class IOTask : public task
   {
   private:
   Timer  T;

   public:
   task *execute()
      {
      std::cout << "\n IOtask starts " << std::endl;
      T.Wait(iowait);
      std::cout << "\n IOtask ends " << std::endl;
      return NULL;
      }
   };
```

LISTING 16.8

IOTask class.

Since this class already has a default no-argument constructor, it is not necessary to encapsulate it in a work-holder class. It can be passed directly as a template argument to the SubmitJob() function.

The code for the main function is listed below. The main() function launches the IO task, waits for a given number of milliseconds, and then waits for the completion of the IOtask. The wait times, iowait for the IO task and mainwait for the main function, are read from an input file. By modifying these timed waits we can check that the synchronization between the main task and the IO task works properly.

```
int main(int argc, char **argv)
   {
   Timer T;
   InputData();            // read Nth, mainwait, iowait
   task_scheduler_init init(Nth);

   std::cout << "\nMain starts "<< std::endl;
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   empty_task *e = SubmitJob<IOTask>();    // submit
   T.Wait(mainwait);                       // do something else
   WaitForJob(e);                          // wait for job
   // - - - -  - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   std::cout << "\nMain ends " << std::endl;
   }
```

LISTING 16.9

Main() function.

Example 5: SynchIO.C

To compile, run make synchio. The timed waits for the main function and the IO task are read from the file synchIO.dat.

In the vath-based utilities or in OpenMP, the synchronizations required for launching a task, doing some more work, and finally waiting for the child task were handled by using a Boolean lock. This example shows how the same issue can be handled, in the TBB environment, using the built-in synchronization properties of the scheduler, together with empty tasks used as a synchronization tool.

16.7.3 SUBMITTING COMPLEX, RECURSIVE JOBS

In order to present a more sophisticated example of generic job submission, we discuss next how to submit one of the three area computation jobs discussed in Chapter 11: the case in which a recursive AreaTask function, which returns an area value, blocks waiting for children. Internally, the AreaTask function uses task_groups to generate the recursive tasks, but we do not need to be concerned with this issue here. Listing 11.17 shows that the main function blocks on the AreaTask call, waiting for the returned area value. It is shown next how to modify this code in a simple way, in such a way that the main function can now do other work concurrently with the area computation, before waiting for termination and using the result.

The job submission interface provides a simple way of implementing this strategy. A work-holder class encapsulates the AreaTask() call, waits for the return value, and initializes a global result variable, on behalf of the main function. The main function submits the work-holder class, performs other activities, and waits for the job termination before using the value of result.

```
double result;
...

double AreaRec2(double a, double b)
   {
   // Same function as in listing 6, chapter 11.
   }
class WH : public task
   {
   public:
   task *execute()
      {
      result = AreaRec2(0, 1);
      }
   };

int main(int argc, char **argv)
   {
   result = 0.0;
   ...
   task_scheduler_init init(Nth);
   std::cout << "\nMain starts "<< std::endl;
   ...
   ...
   // Launch area job
   // - - - - - - - - -
   empty_task *e = SubmitJob<WH>();
   // - - - - - - - - - - - - - - - - -
   // here main can do other things
   // - - - - - - - - - - - - - - - - -
   WaitForJob(e);    // e is destroyed here
   std::cout << "\nThird job done. result is " << result <<  std::endl;
   }
```

LISTING 16.10

Submitting a recursive area job.

All main() has to do is to pass the WH class as template parameter to the SubmitJob() function and, when the time comes, wait for the job termination. This example is developed in file Submit.C.

Example 6: Submit.C

To compile, run make submit. The number of threads in the pool as well as the number of tasks are read from the file submit.dat.

Comment on empty tasks

Before moving ahead, it is useful to come back to the role played by the empty task. This task, never spawned and never executed, is in fact acting as a barrier that signals the end of the parallel region computation. Because the empty task is the successor of the work-holder class, its reference count

(initially equal to 2) is decreased when the parallel job terminates. The wait_for_all() function, when called on the empty task object, returns when the reference count reaches 1. That is all this empty task is needed for: just synchronization. Using an empty task with a trivial execute() function is very convenient, but not really necessary. Any nontrivial task can also be used to implement the required barrier synchronization. Using the library provided empty task is just a convenient way of remembering that the task is being used only for synchronization purposes.

16.8 EXAMPLE: MOLECULAR DYNAMICS CODE

The original MdTbb.C code in Chapter 13 computes the trajectory of a large number of particles with long-range forces, moving in one dimension. A trajectory sample is computed by iterating a large number of times over small time steps. Then the application prints the final kinetic and total energies of the system, and starts again the computation of a new trajectory sample. The code organization is shown in Figure 16.7. In the initial TBB version in Chapter 13, the code remains in a sequential mode, using the parallel_for algorithm to compute the accelerations and to perform the trajectory updates.

In order to provide a simple example of advanced synchronizations using the TBB scheduler, a new version of the code is now deployed, where the calls to parallel_for are replaced by a direct spawning of a set of parallel tasks. The discussion that follows concentrates on the trajectory computation part of the code, because the occasional energy computations have very little impact in performance. Energies are therefore computed in sequential mode.

Two basic issues are discussed in-depth in this section:

- How to use empty tasks to implement barrier-like synchronizations among tasks.
- How to use empty tasks to recycle computing tasks, avoiding repeated memory allocations resulting from the fact that the computational tasks are created and destroyed at each time step.

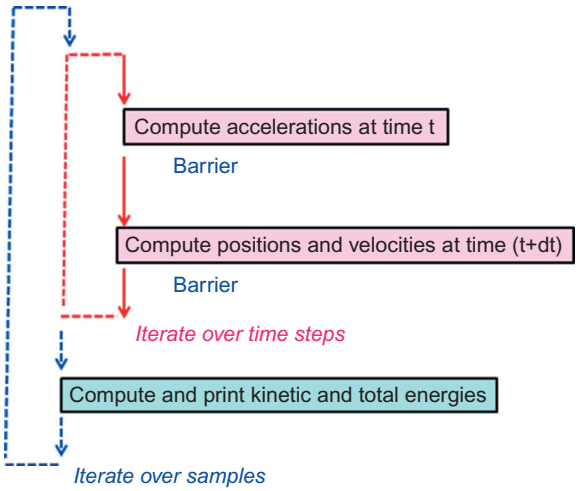

FIGURE 16.7

Code MdTBB: computation of a molecular dynamics trajectories.

16.8.1 PARALLEL TASKS FOR THIS PROBLEM

The programming strategy proceeds as follows: for each time step in the trajectory computation, Ntk tasks are spawned to perform a parallel computation of the acceleration, followed by a successor task whose only role is to enforce a barrier synchronization point among the tasks. Then, Ntk tasks are spawned to update the coordinates and momenta, again followed by a synchronizing successor task. Since there is no sequential work to be performed in the time-step loop, these successor tasks will just be empty tasks.

Here is the listing of the task class used to compute accelerations.

```
class Accelerations : public task
   {
   private:
    int rank;
    int beg, end;

   public:
   // Constructor
   Accelerations(int n) : rank(n)
      {
      int SZ = 3*N;
      int step = SZ/NTk;
      beg = rank*step;
      end = (rank+1)*step;
      if(rank==(NTk-1)) end = SZ ;
      }

   // compute acceleration
   // - - - - - - - - - - -
   task *execute()
      {
      for(size_t n=beg; n!=end; ++n)
          {
          a[n] = 0.0;
          for(size_t j=0; j<SZ; ++j) a[n] += D[n][j] * q[j];
          }
      return NULL;
      }
   };
```

LISTING 16.11

Acceleration task.

The constructor receives an integer rank in $[0, Ntk - 1]$ that identifies the task. Then, from the value of this rank, the constructor computes the range $[beg, end)$ of particle indices that are controlled by this task. The execute() function simply computes the accelerations for all the particles in the subrange allocated to this task.

The task class used to compute the momentum and coordinates updates, called TimeStep, has exactly the same structure.

16.8.2 COMPUTING PARTICLE TRAJECTORIES

The code is basically the sequential code, except that the main() function calls an auxiliary function RunTrajectory that executes the time step loop. When this function returns, a new trajectory sample is available. It is important to observe that the Ntk tasks that compute the accelerations and time steps are all allocated and spawned by main(). This means they are all queued in the main thread personal deque, waiting to be stolen by other threads. This is not the optimal way of using the task scheduler, which is more efficient when, as discussed in Chapter 11, a nested fork-join pattern operates, spreading tasks over different deques. This is what naturally happens when tasks are dynamically generated in a recursive pattern, but in this particular problem we want to address task recycling issues and dynamically generated tasks with a limited lifetime that are not useful. Therefore, we have settled for simplicity in the code structure listed below.

Here is the listing of the RunTrajectory() auxiliary function.

```
void RunTrajectory()
   {
   Accelerations*   x[Ntk];
   TimeStep*        y[Ntk];
   empty_task   *e1, *e2;

   // Allocate empty tasks
   // – – – – – – – – – – –
   e1    = new( task::allocate_root() ) empty_task();
   e2    = new( task::allocate_root() ) empty_task();

   for(int k=0; k<nSteps; ++k)    // time step loop
      {
      // Allocate worker tasks
      // – – – – – – – – – – –
      for(int n=0; n<NTk; ++n)
         x[n] = new( e1->allocate_child() ) Accelerations(n);
      for(int n=0; n<NTk; ++n)
         y[n] = new( e2->allocate_child() ) TimeStep(n);

      // Set reference counts of successor tasks
      // – – – – – – – – – – – – – – – – – – –
      e1->set_ref_count(NTk+1);
      e2->set_ref_count(NTk+1);

      // Spawn NTk Accelerations, and wait for them to complete
      // – – – – – – – – – – – – – – – – – – – – – – – – – – – –
      for(int n=0; n<NTk; ++n) task::spawn(*x[n]);
      e1->wait_for_all();

      // Spawn NTk TimeSteps, and wait for them to complete
      // – – – – – – – – – – – – – – – – – – – – – – – – – – – –
```

Continued

```
    for(int n=0; n<NTk; ++n) task::spawn(*y[n]);
    e2->wait_for_all();
    }

  // destroy tasks
  // - - - - - - - -
  task::destroy(*e1);
  task::destroy(*e2);
  }
```

LISTING 16.12

Computation of trajectories.

A few comments are useful at this point:

- This function has, as local variables, two arrays of references to Acceleration and TimeStep tasks, as well as two empty tasks. These pointers have yet to be allocated.
- The first thing that happens, before the time step loop starts, is the allocation of the two empty tasks as root tasks. This can be safely done here because these tasks are never spawned, executed, and destroyed. Then, their lifetime is not restricted to each loop iteration.
- The computing tasks, instead, must be allocated, as children of the empty tasks, at each loop iteration, because they are executed and destroyed.
- The reference count of the empty tasks is set to Nth + 1, so that these tasks wait for their children and are not spawned.
- Then, the acceleration tasks are spawned with the empty task `e1` waiting for them, which implements the barrier required at this point. The time step tasks follow in the same way.
- When the trajectory computation is finished, the empty tasks are explicitly destroyed.

The rest of the code is straightforward. Performance is comparable to the parallel_for version.

Example 7: MdShed.C

To compile, run make mdsch. Molecular dynamics code using the task scheduler. The input parameters are, as for all the other versions of this code, read from the file md.dat.

Example 8: MdShedBis.C

To compile, run make mdschbis. Same as before, but using an improved recursive spawning of the computational tasks.

16.9 RECYCLING PARALLEL REGION TASKS

When looking at the previous versions of the molecular dynamics code—both high-level algorithms and task scheduler—one observation comes to mind: in this microtasking approach, *the computational tasks are allocated and destroyed at each loop iteration*. For systems of moderate complexity, this may not be a major issue. But imagine the case for a huge number of particles, and a code running on a 60 core Intel Xeon Phi platform, supporting, because of hyperthreading, up to 240 threads. That makes

480 memory allocations per time step. If, in addition, the complete run involves millions of time steps, it is legitimate to explore the possibility of getting rid of memory management overhead by moving task allocations *outside of the time step loop*, as was the already case for the empty tasks. This is, in fact, the most obvious optimization that could justify the use of the task scheduler to squeeze some extra performance for the code.

TBB does provide a task recycling interface. A task that wants to be recycled calls, inside its execute() function, one of the two member functions available for recycling: recycle_as_continuation() or recycle_as_child_of(task *t). This marks the task for recycling, and prevents the scheduler from destroying it when the execute() function terminates. Then, the task can be spawned again for running. There is an obvious *nonoverlap rule* to be respected when recycling tasks: the recycled task should not be spawned again before the execute() function that marks it for recycling terminates. In other words, the next execution of the recycled task cannot overlap with the current one.

Going back to the molecular dynamics code, let us see how the acceleration tasks could be recycled (the time step tasks would work in the same way). The first thing that comes to mind is to modify the execute() function with one extra line of code, as follows:

```
class Accelerations : public task
   {
   ...
   // compute acceleration
   // - - - - - - - - - - -
   task *execute()
      {
      for(size_t n=beg; n!=end; ++n)
          {
          a[n] = 0.0;
          for(size_t j=0; j<SZ; ++j) a[n] += D[n][j] * q[j];
          }
      recycle_as_child_of(*parent());  // recycling
      return NULL;
      }
   };
```

LISTING 16.13

Attempt to recycle Acceleration tasks.

This extra line of code would, indeed, recycle the acceleration tasks as children of their successor, the empty task e1, returned by the parent() call. Clearly, the nonoverlap rule is respected. Because of the barrier induced by the empty task e1, these tasks will never be spawned again before all the execute() functions that mark them for recycling terminate.

Unfortunately, *this simple scheme does not work, for the reason already mentioned in Section 16.5.1*: recycling the tasks breaks down the implicit barrier mechanism induced by the successor task. If a task is marked for recycling, *the scheduler does not destroy the task when* execute() *terminates, but it does not decrease the parent reference count either*. In this case, the e1 reference count never decreases and reaches 1, the e1 task never returns from the wait, and the code deadlocks at the first loop iteration.

We have concocted a simple example, RecycleBad.C, to show how this happens. Take a look at the source, which is well documented. This code prints the parent reference count, and shows that when the task is recycled the reference count is not decreased.

Example 9: RecycleBad.C

To compile, run make recbad. The number of threads in the pool is 2, and the number of parallel tasks is 4.

16.9.1 RECYCLING PROTOCOL

The previous discussion shows us that the barrier synchronization and the recycling operations do not coexist peacefully when driven by the same task e1. Therefore, *two empty tasks are used, one for synchronization and one for recycling*. The procedure is the following:

- An empty task E is introduced for recycling. The parallel tasks for the parallel region are allocated as children of E, and are recycled as, again, children of E. When these tasks run, the E reference count will not be implicitly decreased, as was shown before. But we do not care, *because* E *is not used for synchronization, and it never waits*. In fact, E does nothing at all—never waits, never runs—except helping to manage the task recycling protocol.
- Another task P is used for barrier synchronization. It is this task that waits for completion of the N parallel tasks, and its reference count is set to (N+1) before the parallel tasks are spawned. However, now P is not their successor, so P's reference count is not automatically decreased by the scheduler when the worker tasks terminate. For this reason, the worker tasks need to have a reference to P to be able to *explicitly* decrease its reference count before terminating.

The listing below shows how the Acceleration class must adapted to this protocol. The task objects now receive an extra argument in the constructor: a pointer to the fake successor task that will be used for barrier synchronization. Then, there is the execute() function, an explicit call to decrease_ref_count() on the fake successor task.

```
class Accelerations : public task
   {
   private:
    int rank;
    int beg, end;
    empty_task* successor;   // NEW

   public:
   // Constructor
   Accelerations(int n, empty_task *t) : rank(n), successor(t)
      {
      int SZ = 3*N;
      int step = SZ/NTk;
      beg = rank*step;
      end = (rank+1)*step;
      if(rank==(NTk-1)) end = SZ ;
      }
```

```
    // compute acceleration
    // – – – – – – – – – – – –
    task *execute()
       {
       for(size_t n=beg; n!=end; ++n)
           {
           a[n] = 0.0;
           for(size_t j=0; j<SZ; ++j) a[n] += D[n][j] * q[j];
           }
       successor–>decrement_ref_count();
       recycle_as_child_of( *parent() );
       return NULL;
       }
    };
```

LISTING 16.14

Modified Accelerations class.

The new version of the function RunTrajectory() called by main(), performing the loop plus sequential region iteration, is listed next. Note that:

- All allocations and deallocations take place outside of the time step loop.
- The same empty task dummy is used to perform the two successive barrier synchronizations.

```
void RunTrajectory()
   {
   Accelerations*    x[NTk];
   TimeStep*         y[NTk];
   empty_task* dummy;
   empty_task* e1, *e2;

   // Allocate tasks
   // – – – – – – – –
   dummy = new( task::allocate_root() ) empty_task();
   e1    = new( task::allocate_root() ) empty_task();
   e2    = new( task::allocate_root() ) empty_task();

   for(int n=0; n<NTk; ++n)
       x[n] = new( e1–>allocate_child() ) Accelerations(n, dummy);

   for(int n=0; n<NTk; ++n)
       y[n] = new( e2–>allocate_child() ) TimeStep(n, dummy);

   // HERE STARTS THE RECURRENT PART
   // – – – – – – – – – – – – – – – –
   for(int k=0; k<nSteps; ++k)
```

Continued

```
      {
      // Set reference counts of successor tasks
      // – – – – – – – – – – – – – – – – – – –
      dummy–>set_ref_count(NTk+1);
      e1–>set_ref_count(NTk+1);

      // Spawn NTk Accelerations, and wait for them to complete
      // – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
      for(int n=0; n<NTk; ++n) task::spawn(*x[n]);
      dummy–>wait_for_all();

      dummy–>set_ref_count(NTk+1);
      e2–>set_ref_count(NTk+1);

      // Spawn NTk TimeSteps, and wait for them to complete
      // – – – – – – – – – – – – – – – – – – – – – – – – – – – –
      for(int n=0; n<NTk; ++n) task::spawn(*y[n]);
      dummy–>wait_for_all();
      }

   // destroy tasks
   // – – – – – – –
   task::destroy(*dummy);
   task::destroy(*e1);
   task::destroy(*e2);
   for(int n=0; n<NTk; ++n)
      {
      task::destroy(*x[n]);
      task::destroy(*y[n]);
      }
   }
```

LISTING 16.15

New version of the RunTrajectory() function.

Example 10: MdRec.C

To compile, run make mdrec. Molecular dynamics code using the task scheduler, with task recycling. The input parameters are, as for all the other versions of this code, read from the file md.dat.dat.

The code works correctly, reproducing the results of the other versions. For a small system, there is not much difference in performance with the versions without task recycling.

16.10 ANNEX: TASK CLASS MEMBER FUNCTIONS

The listing below shows most of the member function of the task class, the ones that are most commonly used. A complete list of member functions and a very detailed description of each one of them is

available in the Reference Guide that comes with the TBB release. Note that many of these member functions are qualified as static. Remember that these are ordinary global functions associated with the class and not to a particular object. They must be called by qualifying them with the class name: task::member_function().

```
class task
   {
   protected:
     task();

   public:
     virtual ~task() {}

     virtual task* execute() = 0;

     // Allocation:
     // – – – – – –
     static proxy1 allocate_root();
     static proxy2 allocate_root(task_group_context\&);
     proxy3 allocate_continuation();
     proxy4 allocate_child();
     static proxy5 allocate_additional_child(task\&);

     // Explicit destruction
     // – – – – – – – – – –  –
     static void destroy(task\& victim);

     // Recycling
     // – – – – –
     void recycle_as_continuation();
     void recycle_as_safe_continuation();
     void recycle_as_child_of(task\& new_parent);

     // Synchronization
     // – – – – – – – –
     void set_ref_count(int count);
     void increment_ref_count();
     void decrement_ref_count();
     // – – – – – – – – – – – – – – – – – – – – – – – – – – –
     static void spawn_root_and_wait(task\& root);
     static void spawn_root_and_wait(task_list\& root);
     static void spawn(task\& t);
     static void spawn(task_list\& list);
     // – – – – – – – – – – – – – – – – – – – – – – – – – – –
     void wait_for_all();
     void spawn_and_wait_for_all(task\& t);
     void spawn_and_wait_for_all(task_list\& list);
```

Continued

```
// – – – – – – – – – – – – – – – – – – – – – – – – –
static void enqueue(task\& t);

// Task context
// – – – – – – –
static task\& self();
task* parent() const;
void set_parent(task *p);
bool is_stolen_task() const;
task_group_context* group();
void change_group(task_group_context\& ctx);

// Cancellation
// – – – – – – –
bool cancel_group_execution();
bool is_cancelled() const;

// Affinity
// – – – – –
typedef (implementation–defined–unsigned–type) affinity_id;
virtual void note_affinity(affinity_id id);
void set_affinity(affinity_id id);
affinity_id affinity() const;

// debugging
// – – – – –
int ref_count();
...
};
```

LISTING 16.16

Task class.