CHAPTER

FURTHER THREAD POOLS

12

12.1 INTRODUCTION

This chapter focuses on two thread pool utilities provided by the vath library. First, the SPool class—already introduced in Chapter 3 and frequently used in the examples developed in previous chapters—is reviewed, in order to better understand its design requirements. Then, another thread pool class called NPool, with complementary capabilities, is introduced. In fact, the SPool implements a slightly different design for the OpenMP *thread-centric* execution model, and the NPool class does the same thing for the OpenMP and TBB *task-centric* execution models.

The development of these utilities was initially motivated by the interest in simplifying the programming of some specific parallel contexts. The intention was not to compete with professional, well-established programming environments, or to replace widely adopted standards. The thread pools discussed in this chapter operate correctly, but they do not have the very high level of sophistication of the professional standards. However, the slightly different software architecture adopted in these utilities may constitute an interesting option in some specific programming contexts, as is shown in two examples deployed at the end of the chapter.

12.2 SPooL REVIEWED

As observed in past examples the SPool class operates like an OpenMP parallel region. When a job is submitted for execution, a task function encapsulating the computational work to be performed is passed to the pool. All worker threads in the pool execute the same task function. This utility implements a strict thread-centric environment, in which all the worker threads have a precise preassigned task to perform. There are, however, two differences with OpenMP:

- The SPool pool is explicitly created by the client threads that require its services. The OpenMP pool is implicit: there is a unique worker thread pool in the application that can be stretched by adding new threads as discussed in Chapter 10. In our case, there can be several independent pools operating in an application. An example was provided in Section 11.11.4, when discussing a reader-writer lock example: two different pools were used to run the writer or reader threads accessing a shared vector container.
- Unlike OpenMP, the client thread that launches a parallel job is in all cases external to the pool. The client thread does not join the set of worker threads. Instead, it is correctly synchronized with

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00012-5

the pool activity. The client thread can continue to do other things and, when the time comes, wait for completion of the submitted job.

The SPool class implements practically all the thread-centric OpenMP features. Consider, for example, the nested parallel region structure shown; Chapter 10, Figure 12.1. Since client threads must necessarily be external to the pool, it is obvious that a worker thread participating in an ongoing job cannot submit a nested job *to the same pool*. However, pools are now explicit, and nothing prevents worker threads from creating on the fly a new temporary, local pool, submit a job to it, and, again, when the time comes, wait for the job termination and destroy the local pool. Remember that OpenMP dynamically adds new threads to its pool to fit the nested parallel region requirements. In our case, rather than stretching the existing pool, a new one can be created. An example of this mechanism will be proposed later on.

There are two specific parallel contexts in which this pool architecture may be helpful, as shown in Figure 12.1:

- *Decoupling MPI and multithreading*. This was the motivation for introducing the SPool class. In a hybrid application, in which each MPI process internally activates shared memory computations, the main thread handles the MPI communications and all the details of the large-scale distributed memory code. This MPI thread does not get involved in shared memory computations. Rather, it offloads to one or more pools the internal shared memory parallel tasks, continues to handle its MPI workload, and ultimately waits for the submitted internal jobs. This effectively decouples the MPI and shared memory environments. Of course, a precise synchronization between the client MPI thread and internal worker threads executing the shared memory subroutines is required, but this is precisely what is incorporated by the SPool utility. An explicit example of this programming style is proposed at the end of the chapter.

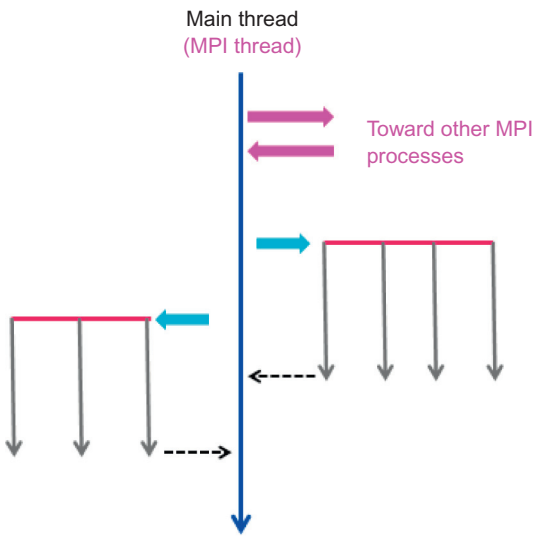

FIGURE 12.1

Typical usage of explicit pools in hybrid applications.

- *Running parallel subroutines in parallel*. Note that a client thread can simultaneously address several independent pools in the application. Indeed, as seen in Figure 12.1, the main thread has created two pools executing *two parallel tasks in parallel*. The asynchronous concurrent execution of parallel jobs is very easily implemented in the SPool environment. This feature first looked initially a bit far-fetched. Today, with the availability of Intel Xeon Phi architectures with 60 cores in a socket, running up to 240 threads, a programming style enabling the execution of several parallel jobs in parallel may be attractive in some cases, as it will be shown at the end of the chapter.

The SPool utility implements practically all the OpenMP thread-centric features. However, the OpenMP and TBB task-centric features discussed in the previous two chapters are missing. One option is to extend SPool, but it is much simpler to introduce another thread pool—called NPool—implementing a task-centric execution environment, while preserving the two basic features mentioned above: explicit pool construction, that is, client threads not members of the pool working team. The point is that, since pools are explicit, an application can simultaneously operate both kinds of pools. And a worker thread in a SPool—or NPool—can always construct a NPool—or SPool—on the fly to run a nested job that requires a different environment.

The rest of the chapter concentrates on the NPool class. We will be particularly clear on two specific points:

- The OpenMP and TBB features that *are not* implemented in NPool, as well as the possible performance impact of the simpler task scheduling strategies.
- The added value that the slightly different software architecture can eventually provide to some specific applications. A complete example that demonstrates how a significant performance boost can be obtained in a specific application by exploiting the possibility of running parallel subroutines in parallel—something not easily implemented in OpenMP and TBB—is discussed in Section 12.7 at the end of the chapter.

12.3 NPooL FEATURES

Besides the two basic features mentioned above—explicit pool, external client threads—a third one was adopted: making a parallel job an explicit, well-identified C++ object, with a well-defined identity returned to the client threads when the job is submitted for execution. There are two reasons for that; First, client threads must be to be able to query the pool about the job status. Second, client threads should be able to submit any number of different jobs *and wait for them in any order*.

The parallel job concept used in the NPool pool is close to the taskgroup concept implemented in OpenMP and TBB. There are, however, some differences that will show up as we go. The main building block for constructing a parallel job is, naturally, an object instance of a Task class that encapsulates, among other things, the specific function to be executed by the parallel task on the target data set. This class operates in the same way as the function object classes used by TBB in the task_group environment, to define specific tasks. It is also totally analogous to the TBB task class used by the full TBB task scheduler discussed in Chapter 16.

To define a parallel job, another simple class called TaskGroup (what else?) is introduced as a container of Task references. This class is used to pack the tasks that initially define the parallel job. This *must* be done because the parallel job must be submitted as a single entity, in one shot. Indeed,

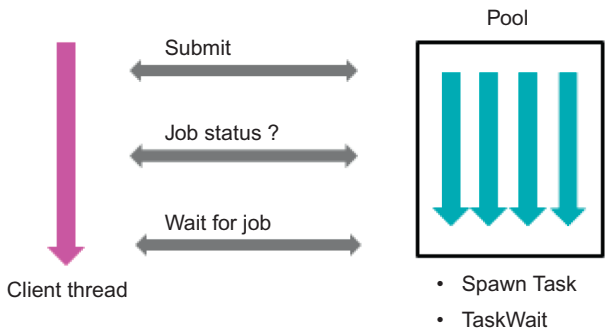

FIGURE 12.2

NPool job submission and management.

client threads get in return a job identifier that allows them to track the job execution or wait for its termination.

The submitted TaskGroup remains alive after submission, and it evolves dynamically. It helps to track the execution of the job, incorporating new tasks spawned by the worker threads, or destroying the terminating tasks. The basic job management features implemented by the NPool utility are summarized below. They are also shown in Figure 12.2:

- Submission of a task as an initially sequential job by a client thread. Nothing forbids this unique task from spawning further tasks when running.
- Submission of a TaskGroup object encapsulating a group of tasks as a unique parallel job, by a client thread.
- Job submission (initially sequential or parallel) returns a unique job identifier that can be used by client threads to inquire about job status or to wait for job termination.
- Jobs can grow dynamically. Tasks being run by the worker threads—either the initial tasks or their children—can spawn new child tasks. In all cases, tasks spawned by worker threads are automatically incorporated to the ongoing job to which the parent tasks belong.
- An *end of job* event is broadcasted to client threads when all the tasks, *including all the spawned tasks*, terminate.
- Spawning dynamical tasks and waiting for children operate in exactly the same way in OpenMP and TBB.
- Note that *only client threads can launch a new job*. Worker threads can only enlarge the ongoing job with new tasks.
- Nested parallelism is supported, in the same way as in the SPool utility. It follows from the statement above that *worker threads cannot submit a nested job to the pool they belong*. But they can legitimately be clients of *another* pool, and this is the way nested jobs are supported. It will be clear later on why NPool is tailored to work in this way.

Every time a job is submitted, an internal job manager object is created incorporating, among other data items, the initial TaskGroup and a Boolean lock in charge of the synchronization with the client tasks. There is one job manager per submitted job, and the job identifier returned to the client tasks targets the relevant job manager and Boolean lock.

It should be noted here that the mechanism of launching an arbitrary parallel job, getting a job identifier, and using it to wait for the job termination can also be implemented in TBB, using advanced programming techniques provided by the TBB scheduler API. A discussion of this feature will be presented in Chapter 16. The main difference with TBB is that in TBB, as in OpenMP, there is a unique, implicit TBB pool in an application.

After describing the basic NPool features, a more detailed discussion of this utility follows, organized as follows:

- A presentation of the user interface of the NPool class—simple and compact—for managing and running parallel jobs.
- A presentation of the Task and TaskGroup classes, needed to define, prepare, and submit a parallel job.
- Finally, a very qualitative discussion of the operation of the thread pool, and a comparison with OpenMP and TBB features.

12.4 NPool API

The NPool public interface is defined in the NPool.h file. The public member functions are simple—there are only seven of them—and intuitive. They are described in detail below, and shown in Figure 12.2.

NPOOL PUBLIC INTERFACE

1. Constructing an NPool Object

```
ThPool(int nTh)
```

– Constructor of the thread pool.
– Constructs a pool with nTh worker threads.
– Called by client threads.

2. Job Submission and Management

```
int SubmitJob(Task* T)
int SubmitJob(TaskGroup& TG)
```

– Submits a sequential or parallel job for execution by the thread pool.
– Returns 0 if the submission failed (submission by worker thread).
– Returns an integer bigger than 0 if the submission is successful.
– In this case, the integer so returned is a *job identifier*.
– Called by client threads.

```
bool JobStatus(int id)
```

– Returns the status of the job identified by id.
– If return value is true, job is running or waiting to run.
– If return value is false, job is terminated.
– Called by client threads.

```
void WaitForJob(int id)
```

– Waits until all the tasks in the job identified by id are finished.
– Prints message and aborts program if job identifier is invalid.

- Called by client threads.

`void WaitForIdle()`

- Wait until the whole pool is idle.
- Called by client threads.
- When this function returns, all the worker threads are idle.

3. Spawning and Waiting for Child Tasks

`int SpawnTask(Task& T, bool iswaited=true)`

- Spawns a child task for execution by the thread pool.
- Returns 0 if the submission failed.
- Returns 1 if the submission is successful.
- The second argument informs the runtime system if the parent task will wait for its child.
- If the parent task waits, there is no need to pass the second argument.
- If the parent task does not wait, the false value must be passed.
- Called by worker threads in the pool.

`void TaskWait()`

- Waits for the termination of all the child tasks previously spawned.
- Called by worker threads in the pool.
- Only *direct* child tasks are synchronized.

Notice that job management functions are always called by external client threads, and dynamic task spawning and taskwait synchronization functions are always called by worker threads in the team.

Client threads can call `WaitForJob()` to perform an idle wait until the job is finished, or they can call `JobStatus()` and return to some other activity if the job is still running. If these calls are made by a worker thread in the pool, the program is aborted with an error message.

The `WaitforIdle()` call means: wait until the pool is totally inactive, that is, there are no running jobs, or jobs waiting to run. This is a very handy utility when, for example, a large number of simple jobs has been submitted, and the client code only needs to know they are all finished. It is useful to keep in mind that this function call does not interfere with the tracking of individual jobs. It is possible to wait for some individual jobs, and at some point wait for all the remaining ones.

A `WaitForIdle()` call made by a worker thread is ignored, to avoid a deadlock. Indeed, a worker thread making this call goes to sleep while executing a task in a job. Therefore, the job will never be finished and the final result is a circular wait context.

The two functions for spawning and waiting for child tasks work exactly the same way as the `task` and `taskwait` directives in OpenMP, discussed in Chapter 10, or the similar TBB utilities discussed in Chapter 11. We underline once again the fact that *spawned tasks are incorporated into the parallel job of the parent task*.

12.5 BUILDING PARALLEL JOBS

Having a global overview of the `NPool` environment, a more detailed discussion follows concerning the construction of a parallel job. The methodology adopted is very close to the one implemented by the TBB task scheduler discussed in Chapter 16. This section paves the way for the construction of TBB tasks.

12.5.1 TASK CLASS

In C++, most natural implementation of the task concept is a C++ object instance of a task class. The advantages in implementing a task as a C++ function object have been underlined, when discussing the TBB task_group class. Here, rather than function objects, the more basic TBB approach is paralleled, in which explicit task classes as derived classes from a base Task class, defined in the include file Task.h as follows:

```
class Task
   {
   protected:
    // data items and member functions not accessible
    // to users, but accessible to derived classes
    ...
   public:
    ...
    Task();                              // constructor
    virtual ~Task();                     // destructor
    virtual void ExecuteTask() = 0;      // interface
   };
```

LISTING 12.1

Task interface class.

This class contains private internal data items and member functions needed for task management and parent-child synchronization, but users of the utility are not concerned by them. Besides the constructor—which, among other things, initializes the private data items hidden to the user—and the destructor, there is the ExecuteTask() function, which is the task function to be called by the executing thread.

Note, however, that the ExecuteTask() function has been declared *virtual* and set equal to zero. This means this function will only be defined in derived classes. The Task class cannot be used as such to create task objects. This class is really just an interface that declares the existence of a service and provides a toolbox to implement it, once the precise task to be executed is specified in a derived class.

Therefore, in order to create real tasks, users must:

- Derive another task class from Task (the class name is arbitrary). This class naturally inherits all the protected member functions that are not visible to end users.
- Define the constructor and, if needed, the destructor of the derived class.
- Define the ExecuteTask() function for the derived class.

The derived class is only limited by the user's imagination. It may contain all the additional specific data items required by the task function, and eventually new auxiliary member functions if needed. The additional data items needed by the task to operate are, of course, initialized via the object constructor. The listing below displays a derived task class MyTask in which the task receives an integer rank as argument:

```
class MyTask: public Task
    {
    private:
     int rank;

   public:
    MyTask(int r): Task(), rank(r) {}

    void ExecuteTask()
       {
       // here comes the specific
       // task function code
       }
    };
```

LISTING 12.2

Derived task class.

As it is obvious from the example above, the derived class constructor can be adapted to the user's needs. Several different data arguments can be passed to the new task via the object constructor.

After a real Task class like MyTask is available, and the task is submitted to the pool for execution, the standard virtual function mechanism of object-oriented programming operates. The run-time code manipulates Task pointers. However, when a Task* has been initialized with the address of a MyTask object, it is the derived class ExecuteTask() function that is called.

This approach is the same as the one implemented in TBB, where there is a pure interface task class—more sophisticated than ours—with, again a task function as a virtual member function execute() returning a task*. The reason for returning a task* is that the TBB task management strategy is more sophisticated, as it allows a task to bypass the standard operation of the task scheduler, by returning the address of the task to be executed next. This issue is discussed in detail in Chapter 16.

12.5.2 TaskGroup CLASS

A TaskGroup is a container of Task* objects. In fact, this object is just an STL list of task pointers std::list<Task*>. A TaskGroup object encapsulates the initial set of tasks that defines a parallel job. It collects all the tasks that start with the job, and enables the submission of the parallel job as a single entity, related to a unique identifier. Here is the public interface:

TASKGROUP PUBLIC INTERFACE

TaskGroup()

– Constructor.
– Creates a empty TaskGroup object.

~ TaskGroup()

- ~TaskGroup()
  - Destructor.
  - Destroys the TaskGroup object.
- void Attach(Task *T)
  - Inserts Task T in the TaskGroup container.
- void Clear()
  - Empties the TaskGroup container.

Users declare first an empty TaskGroup object, and then allocate each individual task and attach its address to it. In the listing below, MyTask1, MyTask2, and MyTask3 are three task classes, taking each an arbitrary number of arguments in their constructors. As was the case for tasks, TaskGroups are allocated in the heap, and the programming interface manipulates their addresses. This is the way to construct a group of these three tasks:

```
TaskGroup *TG = new TaskGroup()     // declare and initialize TG
MyTask1 *t1 = new MyTask1( args );  // construct task
TG->Attach(t1);                      // insert in container
MyTask2 *t2 = new MyTask2( args );  // construct task
TG->Attach(t2);                      // insert in container
MyTask3 *t1 = new MyTask3( args );  // construct task
TG->Attach(t3);                      // insert in container
```

LISTING 12.3

Constructing a TaskGroup.

When the task group pointer TG is submitted for execution, the pool returns, as stated before, a job identifier that can be used later on to inquire about the job status, or to put a client thread to wait until all the tasks in the group are finished.

Internally, the ThPool system associates a job manager object to each job. The job manager contains, among other objects, a Boolean lock. This job manager tracks the execution of the tasks in the job and uses the Boolean lock to signal to client threads that the job activity is completed. This establishes a powerful synchronization service between the client and worker threads.

Note that task groups are dynamic. When a sequential task is submitted directly to the NPool pool, the runtime code always constructs implicitly a one task TaskGroup, and returns its identifier. The reason is twofold:

- The Boolean lock associated with a TaskGroup is always needed to signal to the client code that the task is finished.
- Task groups are dynamic: A task in a parallel job can in turn submit child tasks, which joins the ongoing job and must therefore be incorporated into the parent task group.

12.5.3 MEMORY ALLOCATION BEST PRACTICES

A few basic facts concerning the allocation of tasks and task groups should be kept in mind, in order to avoid unexpected bugs and to make correct use of the NPool service.

Notice that the internal run-time code manipulates *pointers* to tasks and task groups. Therefore, tasks and task groups are created *on the heap* by initializing pointers with the new operator, as shown in the listing above.

The NPool code uses internally STL containers—lists, queues, maps—in several places, to store task or task group pointers, and *STL containers use copy semantics*: insertion operations copy the inserted object into the container. Then, when a task pointer object is attached to a TaskGroup object (as in the listing above), it is copied inside the internal linked list. Similarly, when Task* or TaskGroup* objects are submitted to the pool, they are copied inside the appropriate queues or placeholders.

This means that, after submission, the original Task* and TaskGroup* used to construct and submit a job are no longer needed. They can be destroyed or used again and reallocated by the client code, because the run-time code has already copied them at the right places. *However, they should never be deleted*, because this operation destroys objects that the pointers copied by the run-time code are pointing to. They become invalid, and the program crashes. The rule to keep in mind is the following:

Tasks and TaskGroups are allocated by client code submitting parallel jobs. These objects are deallocated by the internal run-time code, when the associated jobs are finished.

These observations justify the programming style that will be used in the examples. Very often, a parallel job is constructed where the tasks are identical, with possible differences in the values of the arguments passed to the constructor. Here is the way to construct and submit a parallel job composed of four identical MyTask tasks, passing a rank to each task and waiting for its completion:

```
ThPool *TP;
...
int main(int argc, char **argv)
  {
  ...
  // at this point, this task submits a job
  // and waits for termination
  TaskGroup *TG;
  for(int n=0; n<4; n++)
     {
     MyTask *t = new MyTask(n);
     TG->Attach(t);
     }                              // here, t destroyed
  int ID = TP->SubmitJob(TG);       // TG can be reused
  TP->WaitForJob(ID);
  }
```

LISTING 12.4

Submitting a parallel job with identical tasks.

The point to be observed in this listing is that, once the MyTask *t has been inserted into the TaskGroup container, it can be reused in the next loop iteration.

12.6 OPERATION OF THE NPooL

A qualitative understanding of a few details of the inner workings of the NPool utility is very useful to understand some of the pitfalls and best practices that will be discussed next. The management of

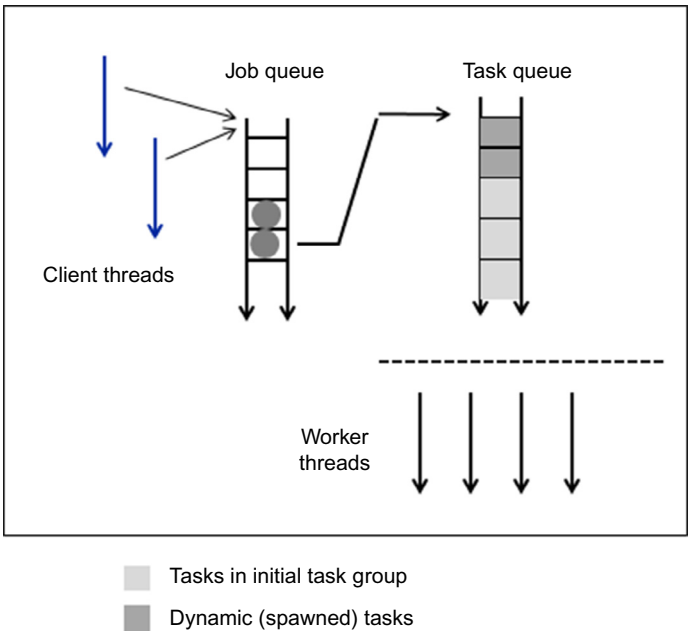

FIGURE 12.3

Schematic operation of the ThPool thread pool.

submitted jobs is in principle straightforward: tasks are queued in a task queue, and worker threads service them as they become available. If threads are sleeping because there was no work available, the queue wakes them up as soon as new tasks are ready to run.

In practice, the operation of the pool, as shown in Figure 12.3, is slightly more involved, for reasons that will soon be clear.

- There are in fact two queues: one in which the job requests (represented by a `TaskGroup*`) are queued, and a task queue, where the `Task*` involved in a specific parallel job are queued.
- Job requests are buffered in the `TaskGroup*` queue, and their tasks are flushed into the task queue on a job-per-job basis: only when a job is finished are the initial tasks of the next waiting job flushed to the task queue.
- Dynamic tasks spawned by worker threads during job execution are naturally queued in the task queue. Then, we have the situation represented in Figure 12.3: the initial tasks (in light gray in the figure) are the oldest tasks in the queue, and the dynamic tasks, if any (in dark gray in the figure) are at the back of the queue.
- Finally, the task queue is really a thread-safe *double-ended queue*. We will see that, in order to avoid deadlocks, tasks can be dequeued for execution from the head of the queue (oldest tasks) or from the tail (youngest tasks).

12.6.1 AVOIDING DEADLOCKS IN THE POOL OPERATION

The operational model of thread pools must implement ways of avoiding potential deadlocks related to dynamic task generation, which can occur when running parallel jobs. These potential deadlocks must be controlled by the operational design of the pool. We know that deadlocks can also occur when using event synchronizations in task-centric environments. They must be controlled by the programmer, as discussed in the next section.

The pool deadlocks if *all the worker threads* are waiting for events that never happen, because they are triggered by tasks that are still queued and not executed, precisely because all the worker threads are waiting and there are no further threads available to dequeue and execute them.

The fact that the thread pool has a fixed number of worker threads may cause problems if the threads are over-committed. Look at the parallel pattern shown in Figure 12.4. The main thread launches a four-task parallel job. One of the parallel tasks spawns a child task and waits for it. Finally, all the tasks are synchronized at a barrier call, before the job ends.

This application will run smoothly with five or more threads in the pool. However, *in the absence of a task-suspension mechanism* enabling threads to suspend the execution of a task in order to service another tasks, this code deadlocks with less than five threads in the pool. At some point in the job execution, three threads are waiting at the barrier synchronization point for the arrival of the fourth thread (the one that submitted the child task). However, this thread keeps waiting for the completion of its child, which is never executed because there are no threads available in the pool to dequeue and execute the task.

A mechanism is therefore needed to allow threads to suspend the execution of a task and service another tasks.

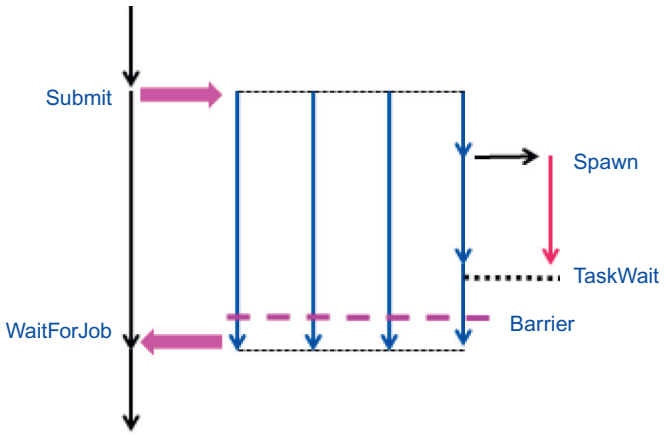

FIGURE 12.4

Spawning one child task.

Mapping tasks to threads

In this utility, the mapping of tasks to threads proceeds as follows:

- The mapping is *non-preemptive*: A task is tied to a specific worker thread for the whole duration of its lifetime. No matter what happens, the thread that initially removed the task from the ready pool and started execution is the one that will complete its execution if the initial execution is suspended.
- In addition, at one specific synchronization point—a TaskWait() function call—when a task decides to wait for all the child tasks it has previously spawned—the executing thread can suspend its execution and service another task in the way discussed next.

The mapping of tasks to threads just discussed is identical—and inspired from—the one adopted by TBB. OpenMP also allows a thread to suspend a task in some specific places, service other tasks, and resume later on the execution of the suspended task. In OpenMP, tasks are also tied to a thread by default. But as we have seen in Chapter 10 the user can declare them *untied*, allowing the suspended task to be resumed by another thread.

Running recursive tasks

When a task decides to wait for its children, the running thread checks if there are waiting tasks in the queue. If this is the case, then the worker threads are most likely over-committed. In this case, the running thread suspends —before waiting for children—the execution of the current task and *it dequeues for execution a task from the back of the queue*. In doing so, it picks the most recently spawned child task. It may even happen that it picks one of its own children, in which case the task submission operation is simply undone. This mechanism is coded inside the TaskWait() call. This function finally waits for children only when there are no further unscheduled tasks in the pool.

It is now clear why job requests are buffered, and only tasks belonging to a unique job are flushed into in the task queue: we know that in this way threads suspending the execution of a task will service child tasks *of the same job*. Indeed, servicing tasks of other jobs does not help prevent the deadlocks we are trying to dispose of.

Without a task suspension mechanism—which exists in all programming environments—it would be unsafe to run recursive parallel jobs where the number of tasks is not known from the start. Several of the examples proposed in the last two chapters show the robust execution of recursive applications with heavily over-committed worker threads resulting from important recursive task generation.

12.6.2 NESTED PARALLEL JOBS

As was stated before tasks executed by worker threads can only spawn new tasks attached to the ongoing job. The reason why worker threads cannot submit a completely different job to the pool they belong, and wait for its termination, results from the fact that job requests are buffered, as shown in Figure 12.5. The task group submitted by the worker thread will be queued in the task group queue and, at some point, the worker thread will start waiting for its termination. *However, at this point, the system deadlocks* because of a circular wait: the nested job will never be executed, because the ongoing job, waiting for its termination, will never complete.

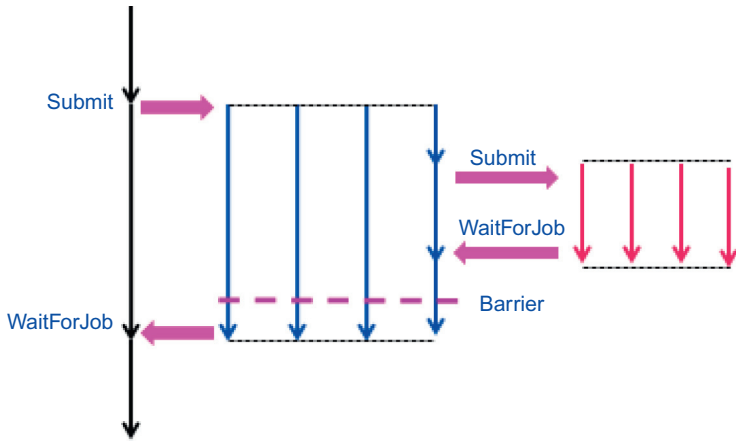

FIGURE 12.5

Nested parallel jobs.

Best practice for nested jobs: If a worker task wants to submit a nested job and dispose of all the synchronization benefits that come with a real parallel job encapsulating all its tasks, *it is always possible to create a new, local, short-lived* NPool *object, not known by other threads, and submit the job to it in the normal way*. See Section 12.6.3.

The subsidiary NPool object, being local to the submitting task, will be automatically destroyed when the task terminates. This is obviously very close to the OpenMP strategy of dynamically adapting the number of worker threads to the parallel context. This way of handling nested jobs is simple and robust, and specific examples will be presented in the following sections.

12.6.3 PASSING DATA TO TASKS

The interest and the necessity of passing to a child task data values that are local to the parent task has been discussed in previous chapters. In the NPool environment, passing data to child tasks follows the same protocols adopted by the TBB tasks in the task_group environment. The required data values are passed by value via the task constructor, which initializes the corresponding internal data items of the child task. To recover a return value from a child, data items are passed by address. A task allocates the local data item that will receive the value, and passes a pointer to its child. The child task will use this pointer, when the time comes, to write directly the return value to the target data item sitting in the parent task stack.

Remember, however, that *in order to recover the return value, the parent task must wait for its child* with a TaskWait call, because the target data item that receives the return value must still be alive when the child writes to it. If the parent task does not wait for its child, the task most likely terminates and, when the child task returns the value, the target data item has been destroyed from the running thread stack. Then, the pointer passed to the child task points to an invalid memory address, and the code crashes.

12.6.4 ASSESSMENT OF THE NPooL ENVIRONMENT

As stated before, the NPool environment operates correctly. With a few exceptions, all the task-centric OpenMP and TBB examples proposed in the previous chapters are correctly run by this utility. The vath library proposes a large number of tests to be executed after compilation that control the correct behavior of the utility. And, in addition, there are further examples that follow in this chapter.

The OpenMP and TBB features that *are not* incorporated in this utility are those related to the hierarchical relations among tasks used to implement event synchronization of tasks, not threads. An OpenMP example was proposed in Chapter 9, when discussing the OpenMP 4.0 depend clause. In TBB, hierarchical relations among tasks are accessible via the complete task scheduler API. They will be discussed in Chapter 16. Thread-affinity utilities are also missing, because they are just beginning to emerge in the basic libraries underlying vath.

Even if the NPool motivation is not to compete with OpenMP or TBB, it makes sense to ask how good this environment is, considered as an ordinary thread pool. The NPool scheduling strategies, tailored to track job execution, are less ambitious than the ones adopted in OpenMP and TBB based on the *task-stealing* approach described in Chapter 16. The adoption of a unique task queue with a first-last in, first-out operation certainly introduces a serialization point in the parallel operation, which, as it is well known, does not optimize—among other things—memory usage or cache reusage. The task-stealing strategy deploys one task queue per thread, and operates in such a way to optimize private cache reusage and load balance. This point has been strongly emphasized by the TBB architects.

These observations are of course quite correct. We would expect, however, to see a significant performance difference in irregular codes deploying a large amount of task generation. In all the examples, we have looked at, involving a moderate amount of recursive task generation, the NPool performances are comparable to OpenMP or TBB. The full applications deployed in the three chapters that follow will confirm this observation. Even the irregular, unbalanced, recursive example of a directed acyclic graph traversal discussed in Chapter 10—to be discussed later on, in Section 12.7.7—shows performance comparable to the OpenMP implementation, for the moderately small data set selected for the test.

It is our intention intend to modify in the future the NPool implementation by keeping the job buffering feature and adopting a task-stealing approach—as described in Chapter 16—for the execution of the job tasks.

12.7 EXAMPLES

There are several example codes provided with the library, for testing all the NPool features. We discuss first a few test codes that demonstrate all the features concerning task submission and management, nested parallel jobs, and child task spawning. Code listings will not be presented in full detail here; the sources are clear and well-documented. We just underline instead the few key points on which each example focuses.

Some of the examples described here have already been discussed in the OpenMP and TBB chapters, and minor modifications are needed to adapt them to the peculiarities of the NPool environment.

12.7.1 TESTING JOB SUBMISSIONS

The different features of the NPool class are tested: submitting a single task job, constructing and submitting a parallel job, waiting for an individual job, and waiting for all running jobs. In all cases, a task class defined in Listing 12.8 is used, consisting of:

- A start message printed to stdout, including a task identifier.
- A timed wait for a random time interval, simulating some significant job activity.
- A termination message printed to stdout, always including a task identifier.
- Tasks are identified by passing them an integer rank in the task class constructor.

```
RandInt  R(3000);   // produces random integers in [0, 3000]
NPool   *TP;        // reference to NPool

class TestTask: public Task
   {
   private:
    int rank;
    long timewait;
    Timer T;

   public:
   TestTask(int r): Task(), rank(r) {}

   void ExecuteTask()
      {
      cout << " TASK " << rank << " START" << endl;
      timewait = 2000 +(long) R.draw(); // random integer in [2000, 5000]
      T.Wait(timewait);
      cout << " TASK " << rank << " END" << endl;
      }
   };
```

LISTING 12.5

Simple task class.

The global variables in the listing above are:

- A reference to the NPool object, initialized by main() with the number of requested threads for the pool.
- A RandInt object R, whose member function R.draw() returns uniformly distributed random integers in the interval [0, N], N being passed as a constructor argument. This is an easy way to choose a random number of milliseconds in [0, N] for the timed wait.

The task function prints the start message, waits for a random time interval between 2 and 5 s, prints the termination message and returns. The messages output track how the submitted tasks have been executed.

Example 1: TNPool1.C

To compile, run make tp1. No command line arguments are required for the execution.

12.7.2 RUNNING UNBALANCED TASKS

The example discussed in Section 12.7.1 is reconsidered: a map operation on all the elements of a vector container, involving a very unbalanced action on each container element. Container elements are first initialized to a random double value in [0, 1]. Tasks acting on container elements keep calling a local random number generator until it retrieves a value close to the target value, within a given precision. The precision itself decreases as the map operation moves along the container.

This code is adapted to the NPool environment. In the OpenMP environment, a function Replace(int n) was used as a task function for the task acting on container element n. Here, a task class is needed, given in the listing below. The listing also shows the modifications made in the main() function.

```
class ReplaceTask : public Task
   {
   private:
   int n;

   public:
    ReplaceTask (int nn) : n(nn) {}

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


// The main function
// - - - - - - - - --
int main(int argc, char **argv)
   {
   ...
   // NPool computation. Construct first a huge
```

Continued

```
// TaskGroup encapsulating the N tasks
// - - - - - - - - - - - - - - - - - - - - - -
TaskGroup TG;
for(int k=0; k<N; k++)
   {
   ReplaceTask *t = new ReplaceTask(k);
   TG.Attach(t);
   }

T.Start();
// - - - - - - - - - - - - - - - - - - - - - -
jobID = TP->SubmitJob(TG);   // submit
TP->WaitForJob(jobID);       // wait for job
// - - - - - - - - - - - - - - - - - - - - - -
T.Stop();
...
// print results and exit
}
```

LISTING 12.6

Foreach.C (partial listing).

The number of threads and the number of tasks (i.e., the container size) are hard-coded to 4 and 100,000, respectively, as in the OpenMP example discussed in Chapter 10. Note that, in the listing above, a TaskGroup incorporating the 100,000 task pointers is constructed before submitting the job. The performance of this code as compared with the OpenMP implementation is shown in Table 12.1. There is clearly slightly more overhead but, given the huge number of unbalanced tasks, the results are satisfactory.

Example 2: Foreach.C

To compile, run make feach. There are 4 threads and 100000 tasks in the parallel job. The value of the precision can be over-ridden from the command line.

Table 12.1 100,000 Tasks, 4 Worker Threads: Execution Times in Seconds

Performance of Foreach.C			
Environment	**Wall**	**User**	**System**
sequential	9.25	9.25	0
OpenMP	2.54	10.13	0.02
NPool	2.9	10.8	0.15

12.7.3 SUBMISSION OF A NESTED JOB

In this example, the main() thread launches a four-task parallel job, instances of the class OuterTask. One of the outer tasks launches, in turn, an inner, nested parallel job of, again, four tasks, instances of the class NestedTask. These inner tasks are identical to the outer ones: start message, timed wait, and termination message. The OuterTask class participating to the external job launched by main is listed below.

```
class OuterTask: public Task
   {
   private:
    int rank;

   public:
    OuterTask(int r): Task(), rank(r) {}
    void ExecuteTask()
       {
       cout <<  " START of Task " << rank << endl;
       if(rank==3)
          {
          // – – – – – – – – – – – – – – – – – – – – – – –
          // Creation of new task pool, job construction,
          // submission, wait for termination and implicit
          // destruction of the new task pool
          // – – – – – – – – – – – – – – – – – – – – – – –
          ThPool INPool(4);     // new, inner thread pool
          TaskGroup *TG;
          for(int n=0; n<4; n++)
             {
             NestedTask *t = new NestedTask(n);
             TG.Attach(t);
             }
          int ID = INPool.SubmitJob(TG);
          INPool.WaitForJob(ID);
          // – – – – – – – – – – – – – – – – – – – – – – –
          }
       B–>Wait();
       cout << " END of task " << rank << endl;
       }
   };
```

LISTING 12.7

Modified OuterTask class.

This example demonstrates how to launch a nested parallel job: *the task that launches the nested parallel job does not submit it to its own thread pool*. Rather, it creates on the fly a new local thread pool to which the new job is submitted. This task pool has a limited lifetime and it is automatically destroyed when the task function returns. The listing below shows the new version of the OuterTask class.

Example 3: Nested.C

To compile, run `make nt2`. No command line arguments required. The inner nested job is correctly executed.

12.7.4 TASKGROUP OPERATION

The purpose of this example is to confirm the fact that child tasks are correctly incorporated into the ongoing job and tracked by the job manager. For this reason, a parent task will be spawning a bunch of child tasks *and not waiting for them*. The code workflow proceeds as follows:

- A one-task job is launched, and the main thread waits for its termination.
- The task in the submitted job spawns in turn four child tasks, which last between 3 and 5 s each, *and terminates, not waiting for its children*.

The code output shows clearly that the parent task terminates before its children. However, the job submitted by `main()` does not terminate because the child tasks are still active. The job termination message comes after the termination messages of the four child tasks.

Example 4: Spawn1.C

To compile, run `make sp1`. The number of threads in the pool can be chosen via the command line. The default is 4 (no command line argument).

12.7.5 PARENT-CHILDREN SYNCHRONIZATION

This example demonstrates parent-children synchronization via the `TaskWait()` function call. As in the previous example, a one-task job is launched, and the main thread waits for its termination. The task in the submitted job:

- Spawns five child tasks that last between 3 and 5 s each, and calls `TaskWait()` to wait for their termination.
- Spawns again two child tasks, and calls again `TaskWait()` to wait for their termination.
- Finally, it spawns again two tasks and returns, not waiting for them.

The code output shows the correct ordering of all the tasks. Again, the initial one task job terminates after the last two child tasks are finished.

Example 5: Spawn2.C

To compile, run `make sp2`. No command line arguments are required.

12.7.6 TASK SUSPENSION MECHANISM

This example demonstrates the task suspension mechanism. A two-task parallel job is launched. Each one of these two tasks spawns a child task and waits for its termination. There are therefore, at some

point, two tasks are blocked waiting for their children, and the two child tasks to be executed. This code seems at first glance to require four threads.

In fact, even in the absence of task suspension, it only requires three threads to run properly, because the extra, nonblocked, worker thread can successively deque and execute the child tasks waiting in the task queue. In our case, since the two threads waiting for their child can suspend the ongoing task to run another task in the queue, the code runs properly with only two threads. It even runs correctly in sequential mode, with only one worker thread in the pool.

Example 6: Spawn3.C

To compile, run make sp3. The number of threads in the pool can be chosen via the command line. The default is 4 (no command line argument).

12.7.7 TRAVERSING A DIRECTED ACYCLIC GRAPH

The graph traversal example discussed in detail for the OpenMP environment in Section 12.7.1 is examined again in the NPool context. This is an irregular, unbalanced, and recursive example; that is, a good testing ground to check whether the NPool performance can sustain the comparison with the standard programming environment. The code sources are in file Preorder.C.

The code starts by constructing the graph G, and by extracting from a Graph member function a vector of pointers to the root cells, namely, the cells that have no ancestor and that consequently can be immediately updated. Then a recursive task function is defined such that, after updating a cell passed as an argument to the task, also recursively updates all the successor cells whose update has been enabled. The UpdateCell class that implements this task function is given in the listing below.

```
int  nTh;          // global variables
Graph G;
NPool *NP;

class UpdateCell : public Task
   {
   private:
     Cell *C;

   public:
    UpdateCell(Cell *c) : Task(), C(c) {}

    void ExecuteTask()
       {
       C->update();
       C->ref_count = C->op;     // reinitialize ref_count

       // Access successors and decrease their reference count.
       // - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Continued

```
        for(size_t k=0; k<C->successor.size(); ++k)
           {
           Cell *successor = C->successor[k];
           if( 0 == --(successor->ref_count) )
              {
              UpdateCell *T = new UpdateCell(successor);
              NP->SpawnTask(T, false);
              }
           }
        }
     };


  // Auxiliary function that does the job
  // ----------------------
  void ParallelPreorderTraversal(std::vector<Cell*>& root_set)
     {
     int jobid;
     std::vector<Cell*>::iterator pos;

     TaskGroup TG;    // This TaskGroup contains the root cells
     for(pos=root_set.begin(); pos!=root_set.end(); pos++)
        {
        Cell *ptr = *pos;
        UpdateCell *T = new UpdateCell(ptr);
        TG.Attach(T);
        }

     jobid = NP->SubmitJob(TG);   // submit job and wait for it
     NP->WaitForJob(jobid);
     }
```

LISTING 12.8

Traversal task and auxiliary function.

The ExecuteTask() member function of the UpdateCell class follows the same logic than the OpenMP task function. Objects of this class receive a Cell* via the constructor. After updating the target cell, the successor cells are visited and their ref_count decreased. Then, when ref_count reaches zero, a new UpdateCell task is spawned. As was the case in the OpenMP version, we are using a tbb::atomic<int> to represent ref_count to enforce thread safety in its update. Note the second argument in the SpawnTask() function, which means the spawned task is not waited on by its ancestor.

Listing 12.12 also displays the auxiliary function that performs a graph traversal. In OpenMP, a parallel loop is run, by applying the update task function to the root cells, inside a taskgroup block. Here, a parallel job TG is constructed and submitted, whose initial tasks are the updates of all the root cells, the remaining cell updates following recursively.

The listing shows the main() function.

```
int main(int argc, char **argv)
   {
   int n, jobid;
   Centimeter T;
   int nTh, n Nodes, n Swaps;

   // Get command line input, initialize pool
   NP = new NPool(nTh);

   // Setup the acyclic graph
   //  - - - - - - - - - - - --
   G.create_random_dag(nNodes);
   std::vector<Cell*> root_set;
   G.get_root_set(root_set);
   root_set_size = root_set.size();

   // Do the traversal
   //  - - - - - - - - -
   T.Start();
   for(unsigned int trial=0; trial<nSwaps; ++trial)
      {
      G.reset_counter();
      ParallelPreorderTraversal(root_set);
      }
   T.Stop();
   T.Report();
   }
```

LISTING 12.9

Main function for graph traversal.

The main function is practically identical to the one used in the OpenMP code. The code performance, for a configuration identical to the one used in Section 12.7.1 for OpenMP, is given in Table 12.2. Performance is comparable, within a few percent, to the performance of the OpenMP version.

Table 12.2 Graph With 4000 Nodes, and 30 Successive Traversals (GNU 4.9.0 Compiler)

Graph Traversal Performances			
n_threads	**Wall**	**User**	**System**
1	58.3	58.3	0.0
2	29.3	58.9	0.06
4	14.9	59.2	0.18
8	7.92	61.8	0.45
16	4.53	66.53	0.8

Example 7: Preorder.C

To compile, run make preorder. The number of threads in the pool, the number of cells in the graph, and the number of traversals can be chosen via the command line. To run, execute preorder nTh nCells nSwaps. The default values are 4, 1000, and 5.

12.7.8 FURTHER RECURSIVE EXAMPLES

This section discusses the NPool versions of the recursive examples proposed in the OpenMP and TBB chapters. The recursive computation of the area under a curve will not be discussed in detail: barring some minor points, the codes are very close to the TBB task_group implementations in Chapter 11. Again, a programming style may be adopted in which tasks return a partial area return value, and they wait for children when the area range is split into two halves. Alternatively, since the job submission interface acts in fact like a task_group, a nonblocking style may be chosen in which tasks that generate children terminate immediately, and children accumulate partial results in a global variable.

In the nonblocking case, the recursive AreaTask class receives the integration range in the constructor parameters. Child tasks that compute partial results store their output in a global Reduction<double> object RD. Here is the listing of the main function driving the computation.

```
int main (int argc, char *argv[])
   {
   int n, jobID;
   double result;

   InputData();    // read Nth and G from file

   // Initialize the thread pool
   // - - - - - - - - - - - - - -
   TP = new NPool(Nth);

   // Submit task, and wait for idle
   // - - - - - - - - - - - - - - - --
   AreaTask *T = new AreaTask(0, 1);
   jobID = TP->SubmitJob(T);
   TP->WaitForJob(jobID);
   result = RD.Data();
   cout << "\n result = " <<  result << endl;
   return 0;
   }
```

LISTING 12.10

Main function, nonblocking style.

Example 8: AreaRec1.C

To compile, run make arec1. This code takes as input the number of threads in the pool Nth as well as the granularity G from the file arearec.dat.

In the blocking style case, the AreaTask recursive class constructor takes an extra parameter, the address of the data item where the area result must be returned. Here is the main function listing in this case.

```
int main (int argc, char *argv[])
   {
   int n, jobID;
   double result;

   InputData();    // read Nth and G from file

   // Initialize the thread pool
   // - - - - - - - - - - - - - -
   TP = new NPool(Nth, 20);

   // Submit task, and wait for idle
   // - - - - - - - - - - - - - - -
   AreaTask *T = new AreaTask(0, 1, &result);
   jobID = TP->SubmitJob(T);
   TP->WaitForJob(jobID);
   cout << "\n Result is = " <<  result << endl;
   return 0;
   }
```

LISTING 12.11

Main function, blocking style.

Example 9: AreaRec2.C

To compile, run make arec2. This code takes as input the number of threads in the pool Nth as well as the granularity G from the file arearec.dat.

Finally, the PQsort.C source file contains the NPool version of the parallel quicksort algorithm discussed in Chapter 9.

Example 10: PQsort.C

Qsort.C is a sequential version of this example. To compile, run make qsort. **PQsort.C** is the parallel version discussed above. To compile, run make pqsort.

12.8 RUNNING PARALLEL ROUTINES IN PARALLEL

This section focuses on the added value provided by the basic architecture of the vath thread pools—explicit pool, client threads external to the pool—when the parallel context can benefit from simultaneous access to different pools running several parallel routines inside an application.

This feature may be relevant in certain cases with the advent of SMP computing platforms—Intel Xeon Phi, IBM Power 8—disposing of hundreds of hardware threads.

The example developed next shows how it is possible to boost the performance of an application, bypassing the limitations imposed by its limited scalability as a function of the number of threads. The application is a computation that requires intensive access to a parallel utility that generates vectors of *correlated Gaussian fluctuations*. It is not important, for the time being, to have a precise idea of what this means; this point is clarified in the next chapter, in Section 13.2.1. The important point is that these vectors are generated by a Monte-Carlo algorithm that requires a significant amount of computation, and that this algorithm can be easily parallelized in an single program, multiple data (SPMD) style.

12.8.1 MAIN PARALLEL SUBROUTINE

The most important ingredient in the correlated generator algorithm can be rephrased as a molecular dynamics problem dealing with the computation of the trajectories of a set of N interacting particles moving in a line. The following chapter is entirely devoted to a thorough analysis of this molecular dynamics problem, and many of the statements that follow will be fully understood and justified. For the time being, we will consider that we dispose of a C++ class, called GaussVec, that acts as a provider of vectors of correlated Gaussian fluctuations, of size N. The listing below shows a partial listing of the class declaration, in GaussVec.h.

```
class GaussVec
   {
   private:
    double  **D;       // external matrix
    double   *V;       // external vector, return value
    ...
    SPool   *TP;
    Barrier *R;
    ...
    // several private member functions

   public:
    GaussVec(double **d, double *v, int vSize, int nThreads,
             double dt, double prob, long seed);
    ~GaussVec();
    void Print_Report();
    void Reset();

    void MCTask();
    void Request_Vector(int steps);
    void Wait_For_Request();
   };
```

LISTING 12.12

GaussVec class.

We should consider this class as a library routine; understanding its precise mode of operation is not necessary. However, a few comments are useful. This class incorporates internally a N×N matrix D,

which defines the imposed correlations among the vector components, as well as a vector V where the return values are stored. Both data items are pointers passed by the client code via the class constructor. The class constructor also incorporates other external parameters, as well as the number of threads to be used in the parallel computation.

Note that this class includes an internal SPool reference, initialized by the constructor when the number of threads is known. In fact, a GaussVec object disposes of a private thread pool to run its parallel computation. There will be as many independent thread pools in the application as GaussVec objects. All these pools will be able to run asynchronously, providing independent correlated Gaussian vectors. An internal Spool utility is used here, rather than the NPool one, because the parallel algorithm is an SPMD computation perfectly adapted to a thread-centric environment. This particular point will be better understood in the next chapter.

There are a few public member functions:

- void McTask(): This function encapsulates the task function to be passed to the thread pool. It is never called by external clients, but it has to be public and not private. To understand why, look at the class implementation in GaussVec.C, where this issue is discussed.
- Request_Vector(int nsteps): This function call submits a new parallel job to the pool that computes a new correlated vector sample, stored in V. The integer argument controls how long the internal Monte-Carlo computation will be running to provide a new sample totally disconnected from the previous one.
- void Wait_For_Request() just calls the Wait_For_Job() SPool member function. This function returns when the previous vector request has completed.

12.8.2 CLIENT CODE

The full application is very simple: it is just a test of the quality of the correlated Gaussian vector generator. The main code requests correlated Gaussian vectors from a GaussVec provider and accumulates all the products Vi.Vj and their squares into two matrices M1 and M2. At the end of the run, an auxiliary routine uses the collected data to compute all the Vi.Vj mean values and their variances, compare with the requested mean values, and provide information about the generator quality, which turns out to be quite good for the specific parameters selected. External parameters are read from a data file cgauss.dat.

Clearly, the code performance is controlled by the GaussGen performance. The analysis developed in the next chapter shows that the algorithm involves parallel patterns of simple vector operations, either vector additions or matrix-vector multiplications, distributed across the worker thread team. This leads to a large succession of lightweight parallel sections separated by barrier synchronizations. For huge values of the vector size N and a reasonable number of threads, the thread computational workload dominates over the barrier synchronizations and the code exhibits good scaling properties. However, as the number of worker threads is increased, their computational workload decreases, synchronization overhead takes over, and the scalability limit is reached. The point to be kept in mind is that, for a given vector size, a scalability limit is reached at some point, such that it does not make sense to keep adding worker threads. In the example that follows, for vectors of size N = 900, this limit is reached for six threads.

Imagine, however, that there are lots of cores available in our computing platform. It is very easy, in this case, to boost the application performance *by using two or more asynchronous provider routines,*

each one in its scaling region. In the code that follows, there are two GaussVec objects, GV1 and GV2, running asynchronously and acting as vector providers. The complete application is in file TwoGen.C.

```
int main (int argc, char *argv[])
    {
    ...
    GV1->Request_Vector(nSteps);
    for(sample=1; sample<=nS; sample++)
        {
        GV1->Wait_For_Request();
        GV2->Request_Vector(nSteps);
        // - - - - - - - - - - - - -
        // ...
        // Treat data provided by GV1
        // ...
        // - - - - - - - - - - - - --
        GV1->Request_Vector(nSteps);
        GV2->Wait_For_Request();
        // - - - - - - - - - - - - --
        // ...
        // Treat data provided by GV2
        // ...
        // - - - - - - - - - - - - - -
        if( sample%50 == 0 )
           printf("\n sample %d done", 2*sample);
        }    // end of samples loop
     ...
     // post-treatment is performed here
     }
```

LISTING 12.13

Main function.

In the listing above, the main code runs a loop that generates two vector samples at each loop iteration, as follows:

- The main code waits for the request submitted to GV1, and then it submits a request to GV2, before treating the data received from GV1.
- When the data treatment is finished, the main code waits for the GV2 request, and then submits a new request to GV1 before treating the data produced by GV2.

For the vector size chosen in this example (N = 900) the GaussVec routine shows good scaling properties up to about six threads. From eight threads on, no further performance speedup is obtained. We have compared the performance of two codes using one or two generators—the source files are OneGen.C and TwoGen.C. The resulting performance is given in Table 12.3, and the results speak for themselves. For a few threads, there is no difference. However, there is a very significant performance difference when using two generators running on 6 threads each (in their scaling regions) instead of one generator running on 12 threads.

Table 12.3 Wall Execution Times in Seconds, 900 × 900 Matrix, 2000 Samples, With Intel C-C++ Compiler, Version 13.0.1

One vs Two Generators			
Threads	**Wall**	**User**	**System**
1	211		
1	211		
2	107.9	210.9	0.06
1+1	–	–	–
4	61.3	234	0.09
2+2	67.5	212.5	0.06
8	46.9	351	0.15
4+4	38.2	237	0.09
12	46.9	509	0.32
6+6	29.5	271.9	0.08

Example 11: OneGen.C and TwoGen.C

To compile, run make onegen or make twogen. Input data is taken from file cgauss.dat.

This example underlines the potential usefulness of running parallel routines in parallel when there is a sufficient number of cores available. And the important point to keep in mind is that this software architecture is very easily implemented with our explicit thread pools.

12.9 HYBRID MPI-THREADS EXAMPLE

This section reconsiders the same problem just discussed—the parallel generation of samples of correlated Gaussian vectors—by using MPI to implement a distributed memory software architecture. An MPI program involves a number of *processes* running concurrently, exchanging data values via explicit message passing. Each MPI process can, of course, internally activate threads, but they are a private affair: a process is not aware of the threads activated in other partner processes. The example in the previous section is reformulated by introducing N MPI processes: one master process that collects vector samples, and (N-1) slave processes that compute vector samples using the multithreaded GaussGen utility and then send them to the master process.

In the computationally intensive phase of the code, the master process just receives data packets from the slaves. The slaves, instead, have two things to do: compute the vector samples by running the GaussGen routine, and send them to the master. The main point of this example is to show how the design of the vath pools—keeping client threads outside of the workers teams—helps to overlap computations and communications in the slaves codes.

MPI has been designed to interoperate with multithreading libraries. The MPI library function calls used by processes to communicate are therefore thread-safe: there are no hidden internal states that determine the outcome of future calls. Any thread in a process can in principle send or receive messages from other processes, and this feature is often useful. But for a large number of applications it is sufficient to limit the MPI function calls to the main thread in each process, and this is indeed our case. Therefore, the main thread manages the MPI structure of the code, and offloads to SPool or NPool pools whatever shared memory parallel activity needs to be run in each MPI process. In MPI jargon, this is called *funneled multithreading support*, and it is good practice to request it explicitly in the MPI initialization steps, as discussed in detail in the example source MpiGauss.C.

MPI adopts a single program, multiple data style in which a unique main() function is executed by all the MPI processes. Each process receives an integer rank identity in $[0, N-1]$. In our case, the rank 0 process will be the master. MPI processes need to allocate buffers for sending or receiving data. In our case, the master process allocates a vdata vector of doubles for receiving the vector samples from the slaves. The slaves, instead, allocate two vector buffers: a ivector buffer for retrieving the correlated Gaussian vectors computed by the GaussGen utility, and a ovector buffer for sending them to the master process. Two buffers are needed because these two operations overlap. At each step, these buffers are swapped, so a buffer that is retrieving data at one step will be used to send data to the master at the next step.

Listing 12.14 is a partial listing of the source for this example, MpiGauss.C. This file contains a number of additional comments that clarify some basic MPI issues, not reproduced here. Let us go directly to the computational part of the example, after all initializations.

- The master process executes a loop over the number of vector samples requested. For each sample iteration, it addresses to each slave a Mpi_Recv() call to get a vector sample, and accumulates the data needed for the final computation of mean value and variances.
- In the slave process, GV is the address of the GaussVec object providing the correlated Gaussian vector service. The slave process starts by filling the ivector buffer with a vector sample. Then, it enters a loop on the number of samples where, at each iteration it:
  - Swaps the *ivector and *ovector pointers. At this point, data to be sent to the master is in ovector.
  - Requests to GV the next data sample in ivector.
  - Before waiting for the return value, sends the data in ovector to the master process.
  - Waits for the GV return value.

```
// Global data
// - - - - --
double *vdata;          // owned by master
double *ivector;        // one per slave
double *ovector;        // one per slave

int main(int argc, char **argv)
   {
   MPI_Status stat;     // used by MPI
   int myid;            // this process rank
   int numprocs;        // number of MPI processes
```

```
int vsize;           // vector size
CpuTimer *T;         // only master will measure times
...
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_support);
MPI_Comm world = MPI_COMM_WORLD;
MPI_Comm_size(world, &numprocs);  // get numprocs
MPI_Comm_rank(world, &myid);      // get myid
int nSlaves = numprocs-1;         // number of slaves
int nSamples = 2000;              // number of samples per slave

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Initialization steps. See details and comments in MpiGauss.C
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if(myid==master)
    { // master initialization }
else
    { // slave initialization }
MPI_Barrier(world);

// - - - - - - - - -
// ENTER COMPUTATION
// - - - - - - - - -

if(myid == 0)  // MASTER CODE
   {
   T->Start(); // master measures execution times
   for(int sample=0; sample<nSamples; sample++)
      {
      for(int sl=1; sl <=nSlaves; sl++)
         {
         MPI_Recv(vdata, vsize, MPI_DOUBLE, sl , MSG, world, &stat);

         // Collect data for for later analysis
         // - - - - - - - - - - - - - - - - - -
         for(int i=0; i<vsize; i++)
            for(int j=0; j<vsize; j++)
               {
               double scr = vdata[i]*vdata[j];
               MV[i][j] += scr;
               MV2[i][j] += scr * scr;
               }
          }

      if( sample%100 == 0 )    // report message
         std::cout << "\n sample " << sample << " done" << std::endl;
      }
```

Continued

```
   T->Stop();   // top measuring execution times
   }    // end of master code
else
   {  // SLAVE CODE
   GV->Request_Vector(ivector, nSteps);
   GV->Wait_For_Request();
   for(int sample=0; sample<nSamples; sample++)
      {
      SwitchVectors();  // exchange ivector <-> ovector
      GV->Request_Vector(ivector, nSteps);

      // Before waiting, send previous vector to the master
      // - - - - - - - - - - - - - - - - - - - - - - - - - -
      MPI_Send(ovector, vsize, MPI_DOUBLE, 0, MSG, world);
      GV->Wait_For_Request();
      }
   }  // end of slave code

MPI_Barrier(world);   // a safe net

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Master performs statistical analysis and reports results on
// on the generator quality, comparing computed and exact values
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

MPI_Finalize();
return 0;
}
```

LISTING 12.14

Partial listing of MpiGauss.C.

Inside each slave process, the computation of the next vector sample and the transfer to the master of the currently available vector sample run concurrently. In fact, all the complex shared memory computation is entirely encapsulated in the GV object which, as we know, has its own internal SPool for shared memory parallel processing.

In this code, the execution times of the master process are measured. Executing the code shows that the execution times (wall, user, system) are very close to the execution times of the OneGauss.C example of the previous section. This is to be expected: execution times are dominated by the cost of the shared memory vector samples computation, and each process is running one Gaussian generator.

Note that the total number of vector samples accumulated by the master process is equal to nSamples*nSlaves. By increasing the number of slaves, the amount of data produced for the final statistical analysis is increased, and it is observed that the execution times remain approximately constant. In fact, this code is a nice example of *weak scaling*: the problem size grows linearly with

the number of MPI processes, and optimal parallel performance is traduced by a constant execution time. At some point, however, communication overhead will start to take over and weak scaling will be lost.

Example 11: MpiGauss.C

To compile, run `make mpig`. The makefile assumes the **mpich** library is adopted for MPI, but this can be easily modified to run with other libraries, like **openmpi**. Input data is taken from the file `cgauss.dat`.