CHAPTER

CONCURRENT ACCESS TO SHARED DATA

5

5.1 FIRST COMMENTS ON THREAD SYNCHRONIZATION

The examples in previous chapters have shown that, even in the simplest parallel contexts, some degree of synchronization among the running threads is in general required. This is a very general issue: in most parallel applications, the underlying algorithms require that the threads execution be correlated at some points. Embarrassing parallel applications, where the tasks executed by threads are totally independent, are rare. Furthermore, it will be shown in Chapter 7 that synchronizations are required to ensure memory consistency: a thread reading a data value in memory must be guaranteed that the related write operations have already taken place, and that it is reading the correct values because all the expected previous writes have been correctly registered into the main memory.

This chapter and the next one develop a discussion of the most basic thread synchronization issues. Parallel languages and multithreading libraries provide direct support for a few *synchronization primitives*, which are basic mechanisms from which more elaborate synchronization tools can be built, better adapted to the high-level concurrency patterns that occur in real applications. Besides the natural synchronization mechanisms occurring at thread creation and termination discussed in Chapter 3, *there are essentially two basic synchronization mechanisms for threads:*

Mutual exclusion, in which threads are forced to execute a code block one at a time, and **event synchronization**, in which threads are forced to wait until some event occurs (an event being, for example, a data item reaching a specific value).

Indeed, *any* parallel synchronization pattern can be handled with these two basic synchronization primitives. They are, of course, explicitly or implicitly present in all multithreading programming environments. But they present different levels of programming difficulty for application programmers. Mutual exclusion is rather high level and easy to use. Event synchronization is a wait-notify mechanism in which a thread decides to wait until another thread notifies the occurrence of an expected event, and this is, instead, a rather low-level primitive. An application programmer equipped with only these tools will have a substantial amount of work to do to cope with the specific high-level concurrency patterns that appear in applications. Over the years, however, native libraries like Pthreads and Windows have been extended to incorporate a number of additional high-level synchronization tools based on these primitives.

The good news is that the basic synchronization concepts are universal, and the programming interfaces are similar in all multithreading libraries. When high-level tools are developed, it is not

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00005-8

difficult to migrate them to a different programming environment. The utility classes in the vath library, providing high-level synchronization services, are implemented in Pthreads, Windows, and C++11.

The fact that any synchronization context can be implemented using mutual exclusion and event synchronization does not mean that these basic primitives provide the most efficient solution in all cases. The applications examples presented in Chapters 13, 14, and 15 indicate that high-level utilities constructed with these basic primitives do not always provide the best possible performance in contexts with very significant synchronization overhead. In those cases, custom synchronization interfaces constructed with the atomic utilities improve in general the code performance.

The rest of this chapter deals with the different ways of implementing mutual exclusion among threads. Besides the basic mutual exclusion primitives, all programming environments—except Pthreads—also implement the atomic variable concept. Atomic variables can efficiently replace the basic mutual exclusion protocols in some specific contexts. They also provide tools to set up efficient custom synchronization utilities, and a few examples in Chapters 7 and 8 illustrate their relevance.

5.2 NEED FOR MUTUAL EXCLUSION AMONG THREADS

Let us consider a multithreaded application where all worker threads occasionally increment a shared counter. The thread function executed by all worker threads would look like this:

```
int counter;
void *thread_function(void *arg)
   {
   ...
   counter++;
   ...
   }
```

LISTING 5.1

Threads incrementing a global counter

This increment operation looks very innocent, but in fact it is not. Current microprocessors implement *load/store architectures*, where direct memory operations are not supported. To increment the counter, the processor *reads* its value from memory to an internal register, increments it, and finally *writes* the new value back to memory. The increment operation is therefore not *atomic*, in the sense that it involves several successive complex operations. Nothing prevents two threads running on two different cores from reading the same value at almost the same time, before the first reader had time to increment the counter and write it back to memory. In this case, both threads will increment the same initial value. The final value of the counter is increased by 1, in spite of the fact that it has been incremented twice.

Things can also go wrong even if the two threads are sharing cycles on the same core. Indeed, nothing prevents thread A from being preempted by the operating system precisely in the middle of the increment operation, after the read and before the write. Then thread B is scheduled and increments the counter, after reading the same, old value not yet updated by thread A. When thread A is rescheduled, it will complete its ongoing update. The end result is the same as before: the counter is increased by one, in spite of the fact that two threads have increased it.

This is a classical example of a *race condition*: the outcome of the computation depends on the way threads are scheduled by the operating system.

A RACE CONDITION occurs when several asynchronous threads are operating on the same shared global data item, and the result of the operations depends on the way the threads have been scheduled.

In concurrent or in parallel programming, threads require a mechanism that locks the access to shared global variables, allowing them to safely complete some operations, avoiding race conditions. A mutual exclusion mechanism is needed, in which a well-identified code block cannot be executed by more than one thread at the same time.

A CRITICAL SECTION is a code block that must be executed by one thread at a time. A thread cannot enter a critical section if another thread is inside. This mechanism is used to exclude possible race conditions.

The way to implement critical sections is indeed very simple. Threads entering a critical section are forced to acquire ownership of a shared global resource that can only be owned by one thread at a time. Such a resource is called a *mutex*, a short name for *mutual exclusion*. Before entering the critical section, a thread acquires ownership of a mutex by *locking* it. When the mutex is locked, other threads that reach the critical section code are forced to wait for it to be released. A thread exiting the critical section unlocks the protecting mutex. If there are other threads waiting for it, the operating system reschedules one of them, which then locks and takes ownership of the mutex.

This mechanism preventing race conditions in parallel programming also operates correctly when threads are over-committed. If the operating system preempts a thread in the middle of a critical section, the thread moves to the ready queue waiting to be rescheduled, *keeping the mutex ownership*. It is therefore impossible for other threads to enter the mutex protected code block. They have to wait until the owner thread is rescheduled, completes the critical section treatment, and releases the mutex.

Critical sections are required whenever thread safety needs to be enforced when accessing shared variables. Imagine, for example, the case of several threads adding elements at the tail of a shared vector container. They do so by retrieving a pointer to the end of the array, storing the new value and increasing the pointer. However, if two threads retrieve the same pointer value—which may occasionally happen if the code is not protected by a critical section—they will end up storing two values at the same place. The data set is corrupted: it does not contain the data elements it is supposed to.

5.3 DIFFERENT KINDS OF MUTEX FLAVORS

All multithreading libraries provide a basic mutex service implementing the mutual exclusion protocol enabling the creation of a critical section. Mutexes can be locked and unlocked and, when locked, they are owned by only one thread. Then, all libraries broaden the service scope by introducing mutexes of different types, having extended capabilities that can be used by the programmer to fine-tune the application performance.

Private versus public mutexes:

A mutex is in general private to a process, in which case it is only accessed by the process threads. This is the standard case. In Windows, however, mutex and other synchronization objects are handled by the kernel and are public in the sense that they can also be shared by different *processes*. In this case, the mutex is allocated in a shared memory block, and used to protect access to shared memory data. Pthreads also provides public mutexes that can be shared by several processes. The public mutex can be locked by the threads in the different processes that are accessing a shared memory block.

Public mutexes are designed to be used in inter-process communications via shared memory. Only the native Pthreads and Windows libraries support them. This mechanism, not directly related to multithreading, is outside the scope of this book.

Standard versus spin mutexes:

When a thread calls the function that locks and acquires a mutex, the function does not return until the mutex is locked. This is typical for event synchronization: a thread waiting for an event, namely, the fact that it has acquired mutex ownership. There are two ways in which this can happen:

- An *idle wait*: the thread waiting to lock the mutex is blocked in a wait state as explained in Chapter 2. It releases the CPU, which can then be used to run another thread. When the mutex becomes available, the runtime system wakes up and reschedules the waiting thread, which can then lock the now available mutex.
- A *busy wait*, also called a *spin wait*, in which a thread waiting to lock the mutex *does not release the CPU*. It remains scheduled, executing some trivial do nothing instruction until the mutex is released.

Standard mutexes normally subscribe to the first strategy, and perform an idle wait. But some libraries also provide mutexes that subscribe to the spin wait strategy. The best one depends on the application context. For very short waits spinning in user space is more efficient because putting a thread to sleep in a blocked state takes cycles. But for long waits, a sleeping thread releases its CPU making cycles available to other threads.

Fair versus unfair mutexes:

When a mutex is released and there are several threads waiting to lock it again, the operating system selects the next thread that takes ownership. A fair mutex lets threads acquire the mutex in the order they requested the lock. This strategy avoids starving threads, because each one of them will get in due time its turn. Unfair mutexes do not respect the order in which the mutex lock is requested. They can be faster, because they allow threads that are ready to run to jump over the waiting queue and take ownership of the mutex if the thread that is next in line to acquire it is in a blocked state for other reasons.

Recursive mutexes:

When a thread owning an ordinary mutex tries to lock it again, the result is undefined and the lock call returns an error code. *Recursive mutexes* allow *the owner thread* to lock them recursively several times. A recursive mutex has internally a counter that counts the number of nested locks. It will only be released after being unlocked by its owner thread as many times as it was locked.

Recursive mutexes are needed, as you can guess, in recursive algorithms. Imagine a library function that internally manipulates complex data structures that need exclusive access for thread safety. The function will then lock and unlock an internal mutex to protect access to the data structure. However, if the function is recursive and needs to call itself, the mutex that protects the internal data structure must be a recursive mutex.

Another context that sometimes requires recursive mutexes are C++ classes with internal private data that, for thread safety reasons, require exclusive access. In this case, a mutex is defined inside the object to protect the private internal data. This mutex will be locked and unlocked by the member functions of the class to manipulate the critical internal private data. Imagine now that a derived class is defined with member functions that, again, need to lock the internal mutex. If, in addition, these member functions call member functions of the parent class that also lock the mutex, then the mutex needs to be recursive. This pattern is very common in Java.

Try_lock() functions:

All libraries propose, besides the lock() and unlock() functions to lock and unlock a mutex, a trylock() function that enables a thread to *try to lock a mutex*. This function *never waits*. If the mutex is available, it returns 1 after locking the mutex. Otherwise, it returns 0 meaning *mutex not available*. The intention is to allow programmers to optimize code by testing the mutex availability. If the mutex is not available, the caller thread can proceed to do something else and come back later on to try again.

Timed mutexes:

Some libraries also propose mutexes supporting a *timed trylock* feature. In this case the lock function does not return immediately, and performs a timed wait for a number of milliseconds passed as argument in the function call. Then, it proceeds as before: it returns 1 (or 0) if it succeeds (or fails) in locking the mutex.

Shared mutexes:

A shared mutex has two lock modes: shared and exclusive. In shared mode, several threads can take simultaneous ownership. This feature, introduced for optimization purposes, seems to contradict the very nature of the mutex operation. But the point is that, when several threads are accessing a data set, the operations performed can be classified as *write operations*, which modify the data set, and *read operations*, which don't. It is true that write operations need exclusive access to the data set. Read operations, instead, only need the guarantee that the data set will not be modified while they are reading, so they need to exclude simultaneous writes. But they do not need to exclude simultaneous reads, and can share the mutex with other readers.

This is a feature introduced to optimize performance in some specific cases. Obviously, critical sections conspire against parallel performance—we will see this happening in some examples in this chapter—and the motivation is to reduce excessive mutual exclusion contention. Shared mutexes are also called read-write mutexes. This is a subtle subject, and a detailed discussion is postponed to Chapter 9.

Table 5.1 shows the mutex flavors proposed by the different libraries. It is interesting to observe that TBB and C++11 expand the basic mutual exclusion service in different directions. TBB adds options

Table 5.1 Mutex Flavors in Different Programming Environments

	Mutex Flavors				
	Pthreads	**Windows**	**C++11**	**OpenMP**	**TBB**
mutex	X	X	X	X	X
recursive	X		X	X	X
timed	X	X	X		
timed recursive			X		
spin fair	X				X
spin unfair					X
shared	X	X			X

for fair and unfair spin locks, which, as we will see, aims at enhancing mutual exclusion performance. C++11, instead, proposes a refined locking strategy by incorporating timed standard and recursive mutexes.

NOTE: As indicated in Table 5.1, C++11 does not propose a ready-to-use spin mutex. However, we will see in Chapter 8 that the std::atomic class provides the tools needed to construct a spin lock.

The mutual exclusion programming interfaces of the different environments are discussed next, paying particular attention to the basic mutual exclusion services. The trylock() (return immediately if the mutex is locked) or the timed lock (wait to lock for a fixed amount of time) options will not be discussed in detail. In practically all the examples developed in this book, every time a mutex needs to be locked, there is nothing else that can be done in the meantime if the mutex is not immediately available. However, keeping in mind that these options are useful when occasionally meeting a context in which they may help to improve the code.

5.4 PTHREADS MUTUAL EXCLUSION

Pthreads introduces two types of mutexes, distinguished by the wait policy (idle or spin) when threads are trying to lock them:

- A standard mutex type pthread_mutex_t. Threads using these mutex objects perform an idle wait. This mutex is, by default, an ordinary mutex: private, fair, and non-recursive.
  - This mutex is accompanied by an associated attribute variable of type pthread_mutexattr_t that can be used to modify its properties.
  - With the help of this attribute variable, this mutex can in principle be made public or recursive. In application programming, changing to a recursive mutex is the only interesting option. This feature may depend on the particular Pthreads implementation. Readers are advised to consult the man pthread_mutex_init manual page.

  - The attribute variable that modifies the mutex properties is passed as an argument to the initialization function (see below). If the attribute variable is NULL, the default initialization is adopted.
  - This mutex also admits a default static initialization when the object is declared (see Listing 5.2).
- A spin lock of type pthread_spinlock_t. Threads using these mutex objects perform a spin wait. *There is no attribute data type associated to this mutex*. The spin lock is never recursive, and the only attribute a programmer can choose is its process scope. This is done directly, without an attribute data type.

5.4.1 MUTEX-SPIN-LOCK PROGRAMMING INTERFACE

Listing 5.2 shows how to declare a mutex or a spin-lock, as well as a mutex attribute. It also shows how to perform a default initialization at the moment the mutex object is declared, and how to use mutex attributes to make a recursive mutex.

```
pthread_mutex_t      my_mutex;      // declare mutex
pthread_mutexattr_t  attr;          // declare mutex attribute

pthread_spinlock_t      my_splock;  // declare spinlock

// Default mutex initialization
// - - - - - - - - - - - - - - - -
pthread_mutex_t  my_mutex = PTHREAD_MUTEX_INITIALIZER;

// Explicit mutex initialization
// - - - - - - - - - - - - - - - -
pthread_mutex_init(&my_mutex, NULL);

// spin_lock initialization
// - - - - - - - - - - - - - - - -
pthread_spin_init(&my_splock, PTHREAD_PROCESS_PRIVATE);

// Initializing "my_mutex" as a recursive mutex
// - - - - - - - - - - - - - - - - - - - - - - - - -
pthread_mutexattr_init(&attr);     // initialize attribut
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&my_mutex, &attr);
```

LISTING 5.2

Mutex and spin-lock declarations

As indicated in Listing 5.2, it is possible to initialize a mutex when it is declared by using a symbolic constant defined in pthread.h. In this case, the mutex takes the default attributes: private to the process, and *not recursive*. For spin locks, there is no static initialization.

PTHREADS MUTEX-SPIN-LOCK INTERFACE

```
int pthread_mutex_init(&my_mutex, &attr
int pthread_spin_init(&my_splock, PTHREAD_PROCESS_PRIVATE)
```

- Mutex and spin-lock initialization.
- If attr=NULL, mutex takes default private attributes.
- An explicit attribute variable is needed to construct a recursive mutex.
- Spin locks are always fair and non-recursive.

```
int pthread_mutex_destroy(&my_mutex)
int pthread_spin_destroy(&my_splock)
```

- Destroys the mutex or spinlock whose address is passed as argument.

```
int pthread_mutex_lock(&my_mutex)
int pthread_spin_lock(&my_splock)
```

- Locks the mutex or spin lock whose address is passed as argument.
- If the mutex is already locked, this function blocks until the caller thread succeeds in locking the mutex.

```
int pthread_mutex_unlock(&my_mutex)
int pthread_spin_unlock(&my_splock)
```

- Unlocks the mutex or spin lock whose address is passed as argument.
- Called by the thread that owns the mutex.

```
int pthread_mutex_trylock(&my_mutex)
int pthread_spin_trylock(&my_splock)
```

- Tries to lock the mutex or spin lock whose address is passed as argument.
- This function always returns immediately. It never waits.
- If the mutex is available, it lock the mutex and returns 1.
- If the mutex is not available, it returns 0.

Pthreads also disposes of the pthread_mutex_timedlock() function, in order to limit the wait for the mutex availability. This requires using the Unix time structures defined in the <time.h> header. Look at the pthread_mutex_timedlock manual page for details.

For spin locks, there is no default initialization and no usage of attribute variables. The spin-lock initialization is always done by a call to pthread_spin_init() by passing a symbolic constant that defines its process scope. Public spin locks across processes will never be used. The behavior of the remaining Pthread functions is exactly the same as in the mutex case. Some examples later in this chapter as well as in Chapter 13 provide comparisons of the relative performances of mutexes and spin locks.

5.4.2 SIMPLE EXAMPLE: SCALAR PRODUCT OF TWO VECTORS

A first simple example of mutex usage is provided by the computation of the scalar product of two long vectors. The number of worker threads is hardwired to 2. The partial results obtained by each thread are directly accumulated in a mutex protected shared global variable. This first example is very inefficient, and it is proposed only for pedagogical reasons, in order to have a measure of the negative impact of excessive mutual exclusion contention. A more efficient way of handling this issue follows next.

5.4.3 FIRST, VERY INEFFICIENT VERSION

Here is the listing of the first version of this example, in source file ScaProd1.C.

```
#define VECSIZE  10000000
double A[VECSIZE];       // Global variables
double B[VECSIZE];
double dotprod;
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
SPool TH(2);     // two worker threads

void *thread_fct(void *P)
   {
   double d;
   int beg, end;
   beg = 0;                    // initialize [beg, end) to global range
   end = VECSIZE;
   TH.ThreadRange(beg, end);   // now [beg, end) is the sub-range for
                               // this thread
   for(int n=beg; n<end; n++)
      {
      d = A[n]*B[n];
      pthread_mutex_lock(&mymutex);
      dotprod += d;
      pthread_mutex_unlock(&mymutex);
      }
   }


int main(int argc, char **argv)
   {
   CpuTimer TR;       // object to measure execution times
   Rand R(999);       // random generator used to initialize vectors

   dotprod = 0.0;
   for(int n=0; n<VECSIZE; n++)
      {
      A[n] = -1.0 + 2.0 * R.draw();      // value in [-1, 1]
      B[n] = -1.0 + 2.0 * R.draw();      // value in [-1, 1]
      }

   TR.Start();
   TH.Dispatch(thread_fct, NULL);
   TH.WaitForIdle();
   TR.Stop();

   std::cout << "\n Scalar product is = " << dotprod << std::endl;
   TR.Report();
   }
```

LISTING 5.3

Scaprod1.C

The code listed above is a straightforward extension of the dot product algorithm to a shared memory parallel programming environment. The full range of vector indices is divided into subranges allocated to each worker thread. Notice in particular the following features of this example:

- The purpose of the mymutex mutex is to protect the global variable dotprod where the worker threads accumulate their partial results. This mutex is initialized to default attributes with a symbolic constant, defined in pthread.h.
- The main() function initializes dotprod to zero, initializes the A and B vector components to random values in $[-1, 1]$ to keep the scalar product reasonably bounded, and runs the threads in the usual way.
- The worker threads accumulate partial results directly in the global variable dotprod. All updates of this variable are protected with the locked mutex.
- In this approach, the mutex is locked and unlocked a very large number of times (as many times as vector components). This is highly inefficient, and can easily be avoided.

Example 1: ScaProd1_P.C

To compile, run make scp1_p. The number of threads is 2. The number of times the scalar product is computed is read from the command line (the default is 1000000).

When the code is executed on two cores, practically no parallel speedup is observed: the wall and user times are practically the same (about 0.9 s each) and, even worst, there is a huge system time of about 0.7 s. The code spends a huge amount of time in the operating system kernel moving the threads back and forth to a blocked state every time they wait for a locked mutex.

5.4.4 ANOTHER INEFFICIENT VERSION USING SPIN LOCKS

In the next example, file Scaprod1S.C, the code is exactly the same, except that now a spin lock is used, instead of an ordinary mutex. In this case the application profile is different. The system time is zero, and the wall time (0.6 s) is half the user time (1.2 s). This shows very clearly that the two threads are active all the time, sharing the application workload, and that the spin lock is a better choice when there is a lot of mutual exclusion and threads can keep running without releasing its CPU. Nevertheless, the last word has not yet been said: there is still a much better way of handling this issue.

Example 2: ScaProd2_P.C

To compile, run make scp2_p. The number of threads is 2. The number of times the scalar product is computed is read from the command line (the default is 1000000).

5.4.5 CORRECT APPROACH

It should by now be obvious what is wrong with the previous examples: there is excessive mutual exclusion among the worker threads. Mutual exclusion *serializes* the access to a shared variable, and in so doing it conspires against the parallel performance searched for when introducing threads.

Excessive mutual exclusion spoils parallel performance. Mutual exclusion must be restricted to the strict minimum required to preserve the integrity of the algorithms.

There is in our case an obvious way to proceed to avoid excessive mutual exclusion. Each thread can accumulate its partial results *in a local variable*. Then, before termination, each thread accumulates only once its computed contribution to the scalar product in the global variable scalarprod. In this way, the mutex is only locked a number of times equal to the number of threads (2 in our case). A more efficient version of the code is given in the source file Scaprod2.C; the only difference is in the thread function, listed below (the auxiliary local variable is d).

```
void *thread_fct(void *P)
   {
   double d = 0.0;
   int beg, end;
   beg = 0;                    // initialize [beg, end) to global range
   end = VECSIZE;
   TH.ThreadRange(beg, end);   // [beg, end) is the range for this thread

   for(int n=beg; n<end; n++) d += A[n]*B[n];

   pthread_mutex_lock(&mymutex);
   scalarprod += d;
   pthread_mutex_unlock(&mymutex);
   }
```

LISTING 5.4

Thread function in Scaprod2.C

Example 3: ScaProd3_P.C

To compile, run make scp3_p. The number of threads is 2. The number of times the scalar product is computed is read from the command line (the default is 1000000).

Running the program, one can observe that the performances are much superior to the previous versions, and that the system time is practically absent. In the first version, there was excessive overhead generated by excessive mutual exclusion. In the spin-lock version, threads are never blocked, but they spend a large amount of time turning around doing nothing, waiting for the mutex. The technique of introducing auxiliary local variables to reduce mutual exclusion is so obvious that it is used implicitly in programming environments with automatic parallelization capabilities, like the data-sharing directives of OpenMP that will be discussed in Chapter 10.

5.5 OTHER SIMPLE EXAMPLES

Mutexes are always associated with a shared variable—or a set of shared variables—they protect, and it is good programming practice to declare them together in order to improve the source code clarity.

In C++, it is possible to do better by *encapsulating* shared variables together with the protecting mutex inside an object that provides thread-safe services to its clients. Here are a couple of simple examples:

5.5.1 REDUCTION<T> UTILITY

Reduction operations of the kind performed in most of the previous examples, where partial results coming from different threads are accumulated in some global variable, are so common in application programming that we have set up a simple template class called Reduction<T>, where T represents a generic type for which the addition operation is defined. This class encapsulates a variable of generic type T and the mutex that guards the increment operations performed on this variable. It is used to easily perform mutex-protected reduction operations on variables of this generic type.

Remember that template library utilities are *text libraries*. The compiler cannot produce code until the user specifies what the generic data type T is. Then, all one has to do is to include the source files in the client code, and the library code is compiled with the application. No specific libraries need to be specified to the linker.

The source file Reduction.h is very simple. The constructor initializes the internal mutex, and the destructor of the class destroys it. Listing 5.5 shows how the class is used.

```
#include <Reduction.h>
...
Reduction<double> RD;   // creates object to perform reductions on a double
...
RD.Accumulate(d)        // accumulates double d inside RD. Called by workers
double d = RD.Data()    // returns the internal data to caller
RD.Reset();             // resets to zero the internal data, so that RD can be
                        // used to perform a new reduction.
```

LISTING 5.5

Interface of the Reduction<T> template class

The source code ScaProd4_P.C uses a Reduction<double> object to accumulate the partial results of threads. This reduction facility will often be used in the forthcoming examples. This class, as all the classes in the vath library is portable. There are also a Windows and C++11 implementations using the mutual exclusion interfaces discussed next.

Example 4: ScaProd4_P.C

To compile, run make scp4_p. The number of threads is 2. The number of times the scalar product is computed is read from the command line (the default is 1000000).

5.5.2 SAFECOUT: ORDERING MULTITHREADED OUTPUT TO STDOUT

The following chapters propose lots of examples in which several threads are created which, among other things, write messages to stdout. Since threads are asynchronous, these messages are most often all mixed up if written directly to stdout. The solution to this problem is to use a mutex to provide to

each thread exclusive access to stdout. A very simple class that encapsulates the protecting mutex and implements this idea is discussed next.

In order to reduce mutual exclusion to the strict minimum, it is important to separate the construction of the complete output message—which is done locally by each thread—from the actual flushing of the message to stdout, done with a locked mutex. To construct the message, a local std::ostringstream object is used, which, instead of writing to stdout, employs a C++ strings as output device. When using the standard operators for formatted IO to write to an ostringstream object, a string is constructed. Then, the whole message, when ready, is sent to stdout using a SafeCout object.

The very simple source code is in file SafeCout.h. This class has a unique member function Flush(ostringstream& os) that does three things:

- Extracts the string from os.
- Locks the mutex, writes the string to stdout *appending an end of line*, and unlocks the mutex.
- Finally, it resets os with an internal null string, so that the client thread function can use it again to prepare the next message.

Listing 5.6 shows the simple usage of the SafeCout class:

```
#include <SafeCout.h>
#include <sstring>
using namespace std;
...
SafeCout  SC;         // creates global object, flushes strings to stdout
...
void *thread_function(void *P)
   {
   ostreamstring os;
   int rank;
   ...
   // Here, thread wants to write to stdout.
   // Write instead to os
   // - - - - - - - - - - - - - - - - - - - - -
   os << "This is the output message from thread " << rank;

   // Now, make a mutex protected flush to stdout
   SC.Flush(os);
   ..
   }
```

LISTING 5.6

Using the SafeCout class

The two classes discussed above are portable. Besides the Pthreads implementation, there are also Windows and C++11 implementations using the mutual exclusion interfaces discussed below.

5.6 WINDOWS MUTUAL EXCLUSION

Multithreading interfaces in Windows are an integral part of the operating system. Besides threads, Windows has a mutex and a few other kernel objects whose usage parallel the thread creation protocol. The creation of an specific object is demanded, and a HANDLE is returned that allows the client code to access the object. The Windows mutex is therefore a public mutex that can be used to protect access to memory shared between processes. The Windows mutex also supports timed waits for the mutex availability.

For ordinary, private mutual exclusion operations Windows has introduced another lightweight interface more efficient than the mutex object, because kernel participation is restricted to the strict minimum: the CRITICAL_SECTION object, used to construct a critical section in the same way the ordinary Pthreads mutex was used before. An object is declared, and its address is passed to specific functions for initialization, lock, unlock, and destruction of the object. These objects do not support timed waits, but we will never perform timed waits in locking a mutex anyway. Here is the CRITICAL_SECTION programming interface:

WINDOWS CRITICAL SECTION INTERFACE

CRITICAL_SECTION CS

– Declares a CRITICAL_SECTION object, called CS (the name is arbitrary).

void InitCriticalSection(&CS)

– Initializes the CriticalSection object CS.

void EnterCriticalSection (&CS)

– Locks the associated mutex.

void LeaveCriticalSection (&CS)

– Unlocks the associated mutex.

void DeleteCriticalSection (&CS)

– Destroys the CRITICAL_SECTION object CS.

Migrating the previous Pthreads codes to Windows is straightforward. Take a look for example at the Windows implementations of the SafeCout and Reduction<T> classes, as well as the Windows version of the scalar product of two vectors:

Example 5: ScaProd2_W.C

To compile, run make scp2_w. This example migrates ScaProd2.C to Windows by using the Windows critical sections in the thread functions.

5.7 OpenMP MUTUAL EXCLUSION

Critical sections can be constructed in OpenMP in two ways: by locking a mutex with library function calls, or by using the critical directive. The first option, which closely parallels the Pthreads and Windows interfaces, is described here. The critical directive (which in fact encapsulates a mutex lock) is discussed in Chapter 10, devoted to OpenMP.

The header file omp.h, which must be included in any code using OpenMP, defines two types of mutexes:

- An ordinary mutex of type omp_lock_t
- A recursive mutex of type omp_nest_lock_t

No attribute variables are needed, because the mutex behavior is fully specified by its type. All mutexes are private; OpenMP is not concerned with interprocess communications. All mutexes are fair. Finally, OpenMP does not explicitly discriminate between idle and spin waits for threads waiting to lock a mutex. But the OpenMP standard introduces an environment variable that allows programmers to select the wait strategy for the whole application, discussed in Chapter 10. It is not clear that all OpenMP versions actually implement this feature. Listing 5.7 indicates how to declare OpenMP mutexes:

```
omp_lock_t         my_lock;     // object declaration
omp_nested_lock_t  my_nlock;    // object declaration
```

LISTING 5.7

Declaration of OpenMP mutexes

The OpenMP library functions for mutex initialization, locking, unlocking, and destruction are given below. They behave exactly like the corresponding Pthread functions, with minor differences: mutexes must be initialized explicitly before usage with a function call (there is no default initialization) and mutex attributes are not needed in OpenMP.

OpenMP MUTEX INTERFACE

void omp_init_lock(&my_lock)
void omp_init_nest_lock(&my_nlock)

– Mutex initialization

void omp_set_lock(&my_lock)
void omp_set_nest_lock(&my_nlock)

– Locks the mutex whose address is passed as argument
– If the mutex is already locked, this function blocks until the caller thread succeeds in locking the mutex

void omp_unset_lock(&my_lock)
void omp_unset_nest_lock(&my_nlock)

– Unlocks the mutex whose address is passed as argument
– Called by the thread that owns the mutex

int omp_test_lock(&my_lock)
int omp_test_nest_lock(&my_nlock)

– Tries to lock the mutex whose address is passed as argument.
– This function always returns immediately. It never waits.
– If the mutex is available, it lock the mutex and returns 1.
– If the mutex is not available, it returns 0.

void omp_destroy_lock(&my_lock)
void omp_destroy_nest_lock(&my_nlock)

– Destroys the mutex whose address is passed as argument.

Notice that all functions return void except the test functions, which are the OpenMP version of trylock, that return an integer. Here are some further comments on this mutex interface:

- The init functions must be called to create the mutex. After creation, the mutex is of course in an unlocked state.
- For nested locks, the test function returns the new nesting count if the lock is available or already owned by the caller thread. This function returns 0 is the lock is not currently owned by the caller thread.

Example 6: ScaProd_Omp.C

To compile, run make scap_omp. This example migrates ScaProd2.C to OpenMP by using the OpenMP locks in the thread functions and the parallel directive inside the main thread to create the parallel section where the scalar product is computed.

5.8 C++11 MUTUAL EXCLUSION

The C++11 thread library defines four different classes corresponding to four mutex types. As is the case for all objects with exclusive ownership, mutex objects cannot be copied or assigned. But they can be moved.

- The std::mutex class, a normal, non-recursive mutex.
- The std::recursive_mutex class, a recursive mutex.
- The std::timed_mutex class, a normal, non-recursive mutex that allows for timeouts on the lock functions.
- The std::recursive_timed_mutex class, a recursive mutex that allows for timeouts on the lock functions.

In C++11, there is no option related to the wait strategy. There is, however, a finer control of the wait duration in lock functions that is not present in the other programming environments (except Pthreads). An example in Chapter 8 shows how a spin mutex class can be constructed using the std::atomic class.

All four mutex classes have a standard, default no-argument constructor, as well as three basic member functions for locking and unlocking: lock(), unlock() that returns void, and try_lock() that returns true when returning with a locked the mutex, and false otherwise. The timed mutex classes have two additional member functions for timed mutex waits:

- try_lock_for(delay) that will try to lock the mutex for a given duration delay.
- try_lock_until(date) that will try to lock the mutex until the absolute date passed as argument.
- Both functions return true if they return with a locked mutex, or false if they have been timed out and the mutex is not locked.

5.8.1 SCOPED LOCK CLASS TEMPLATES

C++11—as well as other C++ libraries like TBB and Boost—also implements an alternative method for locking mutexes: the *scoped locking method*. The basic idea is that, rather than locking the mutex with a member function call, a different class is used for this purpose, for reasons that will soon be clear. In C++11—and Boost—there are two classes whose role is to lock and unlock mutexes: the

lock_guard<T> and the unique_lock<T> classes. The purpose of the first one is discussed next. Later on, we will explain why the second one is also needed.

- Different mutex types have different features, and locking-unlocking requires different functions. Remember the OpenMP or Pthreads interfaces: there are different functions for different mutex types.
- In C++11, the std::lock_guard<T> template argument T defines the mutex type on which the class operates. This template argument is one of the four mutex classes. This is the way the mutex type to be locked is selected.
- The std::lock_guard<T> constructor receives a mutex of the corresponding type as argument. When the object is created, *a reference to this mutex is stored, and the mutex is automatically locked*.
- The std::lock_guard<T> destructor unlocks the stored mutex.

Listing 5.8 shows how to construct a critical section with a mutex and a lock_guard object:

```
std::mutex  MyMutex;     // declare mutex
...
   {
   std::lock_guard<mutex>  MyLock(MyMutex);   // construct and lock
   // - - - - - - - - - - - - -
   //  critical section code
   // - - - - - - - - - - - - -
   }    // here, the destructor of Mylock is implicitly called
```

LISTING 5.8

Critical section with scoped locking

The braces delimiting the mutex protected code block are critical, because they define the scope of the critical section. The mutex is locked when MyLock is created on an entry in the code block. On exit of the code block, the MyLock object goes out of scope and its destructor is called, unlocking the mutex. There are two advantages in this critical section programming style:

- There is no risk of forgetting to unlock the mutex, in which case the code would deadlock because other threads will wait forever to acquire it.
- The code block is *exception safe*: the mutex is automatically unlocked even if the code returns because an exception is thrown inside the critical section. Indeed, when an exception is thrown a normal function return follows that destroys all the function local objects. Even if it is very rare to explicitly throw exceptions inside a critical region, library functions called at these places may eventually throw them.

This scoped lock mechanism is just a special case of a very common C++ idiom called RAII (Resource Allocation Is Initialization) where, instead of directly allocating a resource (pointer, file) a new object is introduced to perform the allocation in its constructor and the deallocation in its destructor. This is, for example, the case of smart pointers. This strategy guarantees that resources will in all cases be correctly restored to the system. When a smart pointer is destroyed, the referenced memory block is automatically deleted, so there in no risk of memory leaks in the application.

Why is another scoped locking class needed?

The std::lock_guard<T> class only provides strict scoped lock-unlock services for simple or recursive mutexes. Let us be more precise. A lock_guard object can be used when the locking needs are just a simple lock at the beginning of the critical section, terminating unconditionally by an unlock at the end. A lock_guard locks the target mutex when created, and never releases the lock during its lifetime. There is no way of interrupting the mutex lock and then locking it again. This lock_guard class can only be used as shown in Listing 5.8.

There are, however, programming contexts in which this direct and straightforward critical section protocol is not sufficient. Consider a context in which a function is called inside the critical section. This function needs to relax momentarily the mutual exclusion constraint in order to give other threads the chance of operating on the data items protected by the mutex. This scenario cannot be implemented if the critical section is constructed with a lock_guard object. C++11 introduces another more sophisticated class, std::unique_lock<T>, to cope with this context.

The std::unique_lock<T> class implements the additional amount of flexibility required by the programming context described above. It allows for deferred locking: it is possible to construct the lock, telling him that the mutex will be locked later on. It is also possible to unlock the mutex before the lock object is destroyed. Obviously, this class is tailored to be used with the timed mutexes. However, if the unique_lock is used to construct the critical section, it can then be passed as argument to a function that needs to unlock the mutex, so that other threads can operate on the data, and lock the mutex again before returning, so as to continue the normal execution of the critical section. This concurrency pattern—which may look weird at first sight—will play a fundamental role when discussing event synchronization. The event synchronization primitive is built on a construct of this kind.

The only usage of the unique_lock that we will make in this book is in the event synchronization protocol discussed in Chapter 6. Examples of simple mutex locking in C++11 are provided by the C++11 implementations of the SafeCout and Reduction<T> classes, in which mutex locking proceeds as indicated in Listing 5.8, using the std::lock_guard<std::mutex> class. There are also the C++11 versions of the scalar product examples developed initially in the Pthreads environment.

5.9 TBB MUTUAL EXCLUSION

The TBB mutual exclusion interfaces are designed with an special care for simplicity and performance. As in C++11, mutex objects of different types are instances of different TBB classes, and all these mutex types share a similar locking-unlocking programming interface. Mutual exclusion performance is fine tuned simply by selecting different mutex types. The TBB documentation on the mutex interfaces can be found in the "Synchronization" topic in the Reference Guide [17].

5.9.1 MUTEX CLASSES

TBB introduces six classes to declare mutex objects, each one having its own characteristics:

- The mutex class. Objects of this class are non-recursive, and perform an idle wait. The include file is tbb/mutex.h.
- The recursive_mutex class. Objects of this class are recursive, and perform an idle wait. The include file is tbb/recursive_mutex.h.

- The spin_mutex class. Objects of this class perform a spin wait. They are very fast for short waits, but they are not fair or scalable (a discussion of scalability follows next). The include file is tbb/spin_mutex.h.
- The queuing_mutex class. Objects of this class also perform a spin wait, but they are fair and scalable. The include file is tbb/queuing_mutex.h.
- The spin_rw_mutex class. A shared (read-write) spin_mutex. The include file is tbb/spin_rw_mutex.h.
- The queuing_rw_mutex class. A shared (read-write) queuing_mutex. The include file is tbb/queuing_rw_mutex.h.

As indicated in the TBB documentation, the first two mutex types—the normal mutex and the recursive mutexes—are just wrappers on top of the mutexes provided by the native operating system libraries. In Unix-Linux systems, they are therefore implemented on top of the Pthreads mutex. Then, TBB adds two kinds of mutexes—spin and queuing mutexes—where threads wait for mutex availability by spinning in user space. Finally, there are the read-write spin and queuing mutexes that will be discussed in more detail in Chapter 9.

Spin and queuing mutexes have different scalability properties. The concept of *scalability* for a mutex may at first sight seem odd, because mutual exclusion serializes the thread's execution and in all cases conspires against parallel performance. In the TBB mutex context, scalability means that, in serializing code execution, the mutex will never do worst than restoring single thread performance. Now, we may wonder how performance could possibly be worst, since, after all, all mutexes are supposed to do is to serialize code execution. The point is that, when spin waiting, threads are continuously monitoring some memory location for a value change. This adds workload to the initial application, as well as network contention. Important mutual exclusion overhead may induce a global workload significantly bigger than the initial, sequential workload. In this case, we say that the mutex is non-scalable.

The two flavors of TBB spin mutexes are complementary. The spin_ mutex is supposed to be very fast in lightly contended situations, but it is not scalable. Moreover, it is unfair, probably because the next thread in line waiting for the lock will not get immediate ownership if for some reason it has been preempted by the operating system. On the other hand, the queuing_mutex is not as fast as the previous one in lightly contended contexts, but it is scalable and fair.

Selecting the correct mutex flavor may help to optimize a multithreaded code. However, no matter what kind of mutex is used, mutual exclusion is to be limited as much as possible. If an application requires excessive contention on a mutex, the best strategy is to reconsider the algorithms, as was done with the scalar product example, where the naive excessive contention was disposed of by the introduction of new local variables in the thread function.

5.9.2 SCOPED_LOCK INTERNAL CLASSES

As C++11, TBB also implements the scoped lock pattern for mutex locking and unlocking. There are three major differences between the TBB and C++11 approaches:

- TBB does not have timed mutexes. The different mutex flavors are designed to optimize the performance of unlimited waits when locking a mutex. C++11, instead, focuses on timed mutexes.
- TBB has a unique class, scoped_lock to lock mutexes, whose functionalities parallel those of the unique_lock in C++11.

- C++11 mutexes can be locked directly. This is not the case in TBB. Mutex objects cannot be locked or unlocked directly. All the locking-unlocking goes through the scoped_lock interface.

Rather than using class templates to select the mutex type, TBB provides six different implementations of a unique tbb::scoped_lock class, by making this class *an internal class* inside each one of the six mutex classes. The name of the internal class as well as the names of their member functions are the same, but they are implemented differently inside each mutex class. In fact, they are really *different* classes, because the internal class name must in all cases be qualified with the name of the container class.

tbb::MX::scoped_lock is the name of the TBB scoped lock inner classes, where MX is one of the mutex classes mutex, recursive_mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex.

Besides two different constructors, these classes have two member functions:

- acquire(mutex_name), that locks mutex_name
- release(), that unlocks a previously locked mutex

Listing 5.9 shows how the TBB mutual exclusion interface is used.

```
using namespace tbb;

MX    MyMutex;       // MX is a mutex class

// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
MX::scoped_lock  Slock;     // first constructor
                            // no mutex referenced
Slock.acquire(MyMutex);     // lock MyMutex
SLock.release();            // unlocks MyMutex
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -

MX::scoped_lock  MLock(MyMutex);   // scoped locking
                                   // MyMutex is locked
```

LISTING 5.9

TBB critical section interface

Listing 5.9 shows how to create mutexes and the corresponding locks that act on them. The scoped_lock objects can be used to lock mutexes in the usual way, by calling the acquire() and release() member functions, or in the scoped lock way by passing a mutex to the constructor.

The TBB scoped_lock objects allow the mutex they own to be unlocked and locked again during their lifetime. Consider again the case of a function called inside the critical section, that needs to relax momentarily the mutual exclusion constraint in order to allow other threads operate on the data items protected by the mutex. As shown in Listing 5.10, the scoped_lockobject is passed as argument to the function, which can use it to unlock and then lock again the mutex before returning.

```
using namespace tbb;

mutex  MyMutex;     // declare mutex
```

```
...
   {
   tbb::mutex::scoped_lock  Slock(MyMutex);   // construct and lock
   // - - - - - - - - - - - - - - - - -
   //  Critical section code
   //  Call here a function that needs to access the data
   //  protected by MyMutex. Pass the scoped lock to the
   //  function:
   Fct(Slock);
   // - - - - - - - - - - - - - - - - -
   }    // here; destructor of Slock is implicitly called
```

LISTING 5.10

Construction of a critical section with a scoped lock

Example 7: ScaProd_Tbb.C

To compile, run make scap_tbb. This example is identical to ScaProd2.C in which threads are run using the SPool utility, except that the critical section is now constructed using a TBB mutex, following the strategy described in Listing 5.10. This is an example of interoperability of different programming environments.

TBB provides also more general Reader-Writer locks. They will be discussed with synchronization tools in Chapter 9.

5.10 FIRST LOOK AT ATOMIC OPERATIONS

Mutexes are not the only way of implementing thread safety in concurrent accesses to shared variables. A useful and fast alternative to locked mutex access to simple shared variables is provided by a limited set of *atomic operations*, which are guaranteed to be performed in an indivisible way, as if they were a unique machine instruction. Any atomic operation executed by a thread is seen as an *instantaneous* operation by the other active threads. We have already seen that increasing or updating a counter is a complex operation that involves several steps: load, increment, and store. If the counter is implemented by an atomic data type, its update is guaranteed to be seen as a compact operation in which a memory value is instantaneously replaced by a new memory value.

Atomic operations are very efficient because they are implemented by low-level hardware and/or operating system instructions specially designed to support thread synchronizations. The TBB, Boost, and C++ thread libraries provide C++ classes defining *atomic data types*, on which a limited set of atomic operations are defined. Besides implementing an alternative to mutex locking, these classes also provide tools for implementing custom synchronization protocols, as will be shown in Chapter 8. OpenMP does not have specific atomic data types, but enables a variety of atomic operations via the atomic directive. Windows does not have an explicit atomic data type either, but there are a substantial number of library functions that provide an equivalent service.

However, atomic operations have limitations and cannot replace mutexes in all cases. They cannot, for example, be used to create a critical section involving a complex code block. In OpenMP, atomic

operations are defined for fairly small data sizes, the largest size being the size of the largest scalar, typically a double-precision floating-point number. TBB restricts explicit atomic operations to integer or pointer types. C++11 deals with bool, integer, and pointer types, and partially extends direct atomic operations to any *suitable* generic type T (we will clarify later on what suitable means).

To sum up, the status of atomic operations in the programming environments discussed in this book is the following:

- Pthreads does not support atomic operations.
- OpenMP supports atomic operations for basic data types via an atomic directive.
- TBB support atomic operations for integer and pointer types.
- C++11 standard—and Boost—support atomic operations for integer, boolean, enum, pointer, and in addition generic types satisfying specific conditions.

5.10.1 ATOMIC OPERATIONS IN OPENMP

There are no new atomic data types in OpenMP. The atomic keyword is used to qualify an operation. This keyword is used in a directive that specifies: *the operation that follows in the following line must be performed atomically*. Here is the general form of the OpenMP atomic construct:

```
#pragma omp atomic { read | write | update | capture } [seq_cst]
   expression
```

LISTING 5.11

OpenMP atomic construct

The atomic directive is qualified with one of the four clauses listed above, which describe the nature of the atomic operation. There is an additional seq_cst clause whose meaning is explained below. The qualifying clauses operate as follows:

- The atomic construct with the read clause performs an atomic read of the memory location designated by x, regardless of the native machine word size.
- The atomic construct with the write clause performs an atomic write of the memory location designated by x, regardless of the native machine word size.
- The atomic construct with the update clause performs an atomic update of the memory location designated by x, using the designated operator.
- The atomic construct with the capture clause performs an atomic update of the memory location designated by x, using the designated operator. In addition, the initial or final value of x is stored in the memory location y, according to the form of the statement following the directive (see Table 5.2). Notice, however, that only the read or write of x are performed atomically. Neither the evaluation of expr nor the write to y need to be atomic with respect to the read or write of x.
- *The* seq_cst *clause, which means sequential consistency is an OpenMP 4.0 extension*. When inserted, the atomic operation is forced to include an implicit memory fence, i.e., an implicit flush operation without a list.

Table 5.2 lists the expressions that correspond to each one of the four qualifying clauses. The conventions adopted are the following:

- x and y are scalar data types.

Table 5.2 Directive Clauses and Related Atomic Operations

Atomic Directive Qualifying Clauses	
read	y=x;
write	x = expr;
update	x++;
	++x;
	x–;
	–x;
	x binop= expr;
	x = x binop expr;
	x = expr binop x;
capture	y = x++;
	y = ++x;
	y = x–;
	y = –x;
	y = x binop expr;
	y = x = x binop expr;
	y = x = expr binop x;

- expr is an expression with scalar type that does not reference x or y. Notice also that expr *is not computed atomically*.
- binop is one of the following binary operations: +, *, -, /, &, ^, |, <<, or >>.
- Notice that the last two update and capture expressions are OpenMP 4.0 extensions.

AtomicTest.C example in OpenMP

Listing 5.12 shows how to replace the mutex locked increment of a shared variable by an equivalent atomic update of a long counter.

```
#include <omp.h>
...
long nsamples;            // number of MonteCarlo events per thread
long C;                   // used to accumulate acceptances

void *thread_fct(void *P)
   {
   double x, y;
   int rank;

   rank = TH.GetRank();
   Rand R(rank*999);
   for(int n=0; n<nsamples; n++)
      {
```

Continued

```
            x = R.draw();
            y = R.draw();
            if((x*x+y*y) <= 1.0 )
               // - - - - - - - - - - - - - - - - - - -
               #pragma omp atomic update
                  ++C;  // atomically accumulate in  C
               // - - - - - - - - - - - - - - - - - - -
            }
         }
```

LISTING 5.12

AtomicTestOmp.C (partial listing)

Besides the replacement of the SPool utility by the OpenMP parallel section, we observe minor differences in the task function. The variable C used to accumulate acceptances is now an ordinary long, and its increment in the task function is preceded by the qualified atomic directive.

Example 8: AtomicTestOmp.C

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

5.11 CONTAINER THREAD SAFETY

Containers are commonly used data structures in applications programming. They store and manage collection of objects of the same type. Different types of containers store elements in different ways, providing different levels of performance for various insertion, traversal, and extraction operations. Programmers can therefore select the type of container most adapted to the problem at hand. The C++ Standard Template Library provides a number of very efficient containers: vectors, queues, linked lists, sets, and maps. All of them support, of course, a number of insertion or deletion operations that modify the data set.

STL containers, however, are not thread-safe. Consider, for example, an insertion operation of a new vector element at the tail of a STL std::vector<T> container, where elements are stored in consecutive positions in memory. The vector container maintains internally a pointer to the vector end. A thread that invokes the insert_last() operation on the container launches an operation in which the end pointer is read, the new vector element is copied to this memory address, and the pointer is incremented to point to the new vector end. Obviously, this operation is not thread-safe: a new thread can start the same operation *before* the end pointer is updated, and the two threads end up storing the vector element in the same place. The final result is that two or more threads stored data, but there is only one new data value in the container. The STL containers must therefore be used with care in a multithreaded context.

When using non-thread-safe containers, the only option for programmers is to use mutual exclusion and perform the container operations that are open to data races inside a critical section. However, container operations are complex in general, and the critical sections that protect them may end up implicitly involving reasonable big code blocks, with a potential degradation of parallel performance in case of significant contention in the containers access.

5.11.1 CONCURRENT CONTAINERS

A better option is to use if possible *concurrent containers*, i.e., thread-safe containers that encapsulate internally whatever is required to control race conditions. Concurrent containers use two basic techniques to enforce thread safety:

- *Restricted locking*. Critical sections are used internally to protect only the sensitive parts of the code, and mutex locking is restricted to a strict minimum. We will meet in Chapter 9 a thread-safe queue class—`ThQueue<T>`—that build upon the STL queue to make it thread safe.
- *Lock-free algorithms*, in which race conditions are controlled without serializing the thread operation. The basic concepts behind lock-free algorithms are discussed in Chapter 8.

5.11.2 TBB CONCURRENT CONTAINER CLASSES

TBB is a powerful programming environment for multithreaded processes. Besides the thread management services, based on an implicit thread pool, which will be discussed in Chapters 11 and 16, TBB also incorporates a large number of standalone utilities that can profitably be used in *any* programming environment, like the concurrent containers classes. As a general rule, these classes support concurrent insertion and traversal, but not concurrent erasure. According to the TBB documentation—"Containers Overview" topic in [17]—thread safety is implemented using both fine locking—involving highly optimized internal critical sections—and lock-free algorithms of the kind discussed in Chapter 8.

- **Sequential containers**:
  - `concurrent_vector`:
  - `concurrent_queue`:
  - `concurrent_bounded_queue`:
  - `concurrent_priority_queue`:
- **Associative container**:
  - `concurrent_hash_map`:
- **Unordered containers**:
  - `concurrent_unordered_set`:
  - `concurrent_unordered_map`:

These classes can be considered as thread-safe versions of many of the STL container classes. The main difference is that in TBB there is a unique template parameter `T`, which is the type of the objects stored in the container, while the STL containers have a second optional template parameter that allows the user to specify a memory allocator class. But this STL feature—overriding the default STL memory allocator—is rarely used in practice.

Sequential containers are containers in which the position of an element depends on the order in which it has been inserted, not on its value. The STL has vectors, queues, and priority queues, but it does not have a bounded queue. Bounded queues have a finite capacity and many features that come with it: producers wait if the queue if full, consumers wait if the queue is empty. The `ThQueue<T>` class discussed in Chapter 9 is a bounded queue, and TBB's bounded queue is very close in essence to this class.

Associative containers are ordered in the sense that the position of an element depends on its value. They are often implemented as a binary tree. The STL has two kinds of associative containers: maps and sets. They are very efficient for search operations, but there is a rather expensive prize to pay for insertions and deletions, which enforce a reorganization of the container. The concurrent_hash_map container is a thread-safe version of the STL map. Recently, the STL has introduced unordered sets and maps: search operations are less efficient, but they have better performance for insertions and deletions. The TBB classes for unordered containers are thread-safe versions of the corresponding STL classes.

An example involving the usage of TBB's bounded queue will be presented in Chapter 15. Readers are referred to the TBB reference guide [18] for the concurrent containers whose usage is accessible to anybody having minimal experience with STL containers.

5.12 COMMENTS ON MUTUAL EXCLUSION BEST PRACTICES

This chapter has presented a broad description of mutual exclusion programming interfaces. Different programming environments exhibit slightly different interfaces and functionalities, but the basic concepts and strategies are strictly the same. Locks are the fundamental tools for implementing mutual exclusion to prevent race conditions. Atomic operations, as will be shown in Chapter 8, are powerful tools for implementing custom synchronization utilities, but they can replace mutex locking only in some specific cases.

Restricted scope of atomic operations

The OpenMP example shows that atomic operations are very efficient for guarding an integer variable or a pointer. But they cannot replace mutex locking in more complex circumstances. Indeed, *a code block composed of a suite of atomic operations is not atomic*. Therefore, it is not possible to construct a critical region just by piling up atomic operations. Very often, maintaining a locked mutex for the duration of the compound action cannot be avoided.

How many different mutexes do we need?

Consider an application written in C style, with all the shared variables accessible to all threads defined as global variables. Let us also imagine that two critical sections need to be introduced in the code, and call them A and B. The question is: how many mutexes are needed? One or two?

The answer is: it depends. If both critical sections are modifying the *same* data, then obviously *the same* mutex must be used in critical sections A and B. Indeed, having a thread executing critical section A and another thread executing critical section B at the same time is not thread safe, because the two threads are accessing the same data. Mutual exclusion requires the same mutex in A and B. If, instead, the two critical sections are uncorrelated in the sense that different data items are modified, then *two different mutexes are more efficient*, because in this case two different threads can safely execute the two critical sections simultaneously. If only one mutex is used, the application is still thread-safe, but suffers from unnecessary additional contention. *Increasing the number of mutexes decreases the contention*.

The conclusion is: *each state variable should be guarded with a dedicated mutex*. By *state variable* we mean either a unique data item or a complex data set that as a whole specifies some specific state of the application.

Encapsulation helps in controlling complexity

Rather than using the C programming style, encapsulating data in C++ objects makes it more practical to manage thread safety. An object can encapsulate a lot of internal data, and shared objects can of course be accessed by many different threads. If an object is *immutable*, in the sense that its internal state is not changed by member functions, then it is thread-safe. If an object is mutable, then the simplest thing to do is to incorporate a mutex inside the object, guarding the object state. If the member functions lock and unlock the mutex whenever appropriate, the object will be thread-safe. If a lot of objects are created, there will be of course one mutex per object: there is no mutual exclusion contention among different objects. Mutexes are initialized in the object constructor, and destroyed in the object destructor.

This strategy is so efficient that Java automatically includes a hidden mutex in each object. Programmers do not need to declare it.

Compound thread-safe operations are not thread-safe

Operations of type *check and act*, where if something is true an action is taken, are very common in any kind of programming. Let us consider an example: a vector container class with thread-safe member functions. Consider two of them: contains(T), which returns true if T is a vector element and false otherwise, and add(T), which inserts T somewhere in the container. Even if each member function is thread-safe, the following check and act construct is not, and a mutex lock is required for thread safety:

```
vector V;
...
if(!V.contains(d)) V.add(d);
```

LISTING 5.13

Compound thread-safe operations

This observation is the same made before about atomic operations: a compound operation made of atomic operations is not atomic.

This page intentionally left blank