CHAPTER

PIPELINING THREADS 15

15.1 PIPELINE CONCURRENCY PATTERN

In the traditional data parallel concurrency pattern, different threads perform simultaneously the same action on different subdomains of a data set. The *pipeline concurrency pattern* is an alternative organization of the worker threads operation. Worker threads act one after the other on the same data target, and parallelism occurs when their action on a collection of data targets is organized in an assembly line fashion, as discussed below. Rather than establishing a parallel pattern in space (memory) threads establish a parallel pattern in time. We speak, in this case, of *control parallelism*.

The pipeline concurrency pattern applies when:

- A large collection of identical data sets must be processed.
- The treatment to be performed on each member of the collection can be split in a set of N consecutive independent actions, called *stages*. This is shown in Figure 15.1(a).

The data samples (light gray boxes) shown in Figure 15.1 can be, for example, a collection of two-dimensional images generated by an application—our first example in this chapter—or the rows of a matrix—our second example. Figure 15.1(a) shows a single thread performing three successive operations (stages) on each data sample. As shown in Figure 15.1(b), parallel speedup can be generated if the different stages are executed by three different threads on three successive data samples. The team of worker threads is then organized as an *assembly line*, each worker thread acting on all the members of the data set, performing its assigned stage task. This parallel pattern requires, of course, a careful synchronization to make sure that a thread starts operating on a given member of the data set only after the thread executing the previous stage has completed its assignment. A thread that has just finished executing its partial task on a given sample must pass this information to its successor thread in the assembly line. This chapter discusses different ways of implementing this transfer of control information.

Worker threads overlap their operation in time, and if all the stages are of comparable complexity, involving sufficient computation so that the synchronization overhead is limited, then the optimal parallel speedup is equal to the number of stages. Otherwise, if the pipeline is not well balanced, the overall execution time per data element is the execution time of the longest stage (which also provides parallel speedup).

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00015-0

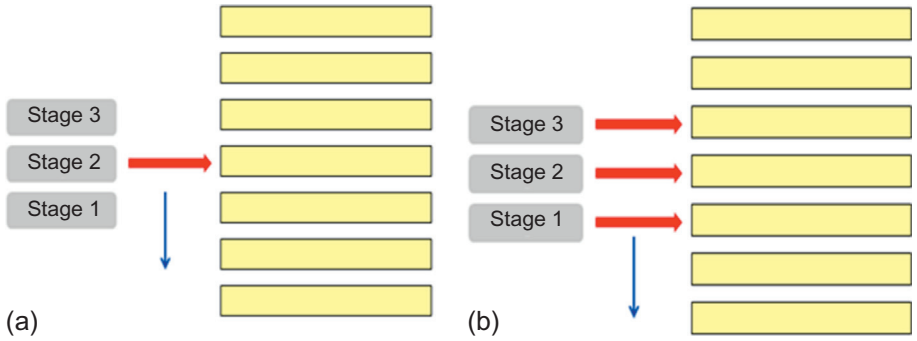

FIGURE 15.1

Pipeline concurrency pattern.

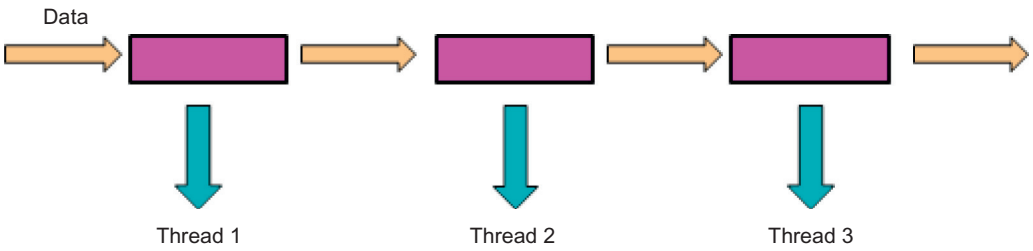

FIGURE 15.2

Another representation of the pipeline concurrency pattern.

The assembly line nature of the pipeline pattern can also be represented as shown in Figure 15.2. Rather than imagining the threads sweeping the elements of the data set collection, we can imagine these elements flowing along the pipeline and being acted upon by the successive stages. This image is fine, but it should not inspire the wrong idea concerning the applicability of the pipeline concurrency pattern:

Pipelines are efficient when the role of successive stages is to perform *in-place* modifications of the data set.

Indeed, the data samples on which the worker threads operate should be allocated in shared memory, avoiding copies from one stage to the next. Introducing lengthy memory operations along the pipeline completely spoils in most cases the benefits of concurrency. The only data that should really flow along the pipeline from one worker thread to the next is synchronization or control information, as the examples that follow illustrate.

15.2 EXAMPLE 1: TWO-DIMENSIONAL HEAT PROPAGATION

This first example focuses on an image production and treatment problem. A large number of two-dimensional data sets (images) are generated, and the nature of the problem requires performing a Fast Fourier Transform (FFT) on each one of them. Multithreading is relevant because the computation of each two-dimensional FFT can be pipelined as two successive one-dimensional FFTs. This example focuses therefore on the simplest possible pipeline, involving only two stages, but examining in detail the way in which the two threads are synchronized.

15.2.1 PHYSICAL PROBLEM

Consider a rectangular conducting plate, with sizes L_x and L_y in the x and y directions, initially heated at the center, as shown in Figure 15.3. This plate is not isolated: heat can flow in and out across the borders. Our intention is to watch how the heat initially concentrated at the center of the plate diffuses until the plate temperature $T(x, y)$ reaches ultimately a constant temperature configuration.

In order to allow heat to flow across the borders of the plate, *periodic boundary conditions* are imposed. This means we are really dealing with an infinite system that repeats itself an infinite number of times in the x and y directions. In such a system, the temperature distribution $T(x, y; t)$ is clearly a periodic function both in x and y, with periods L_x and L_y, respectively. The temperature distribution satisfies the heat conduction equation:

$$\frac{\partial T}{\partial t} = -\kappa \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

where κ is the *heat diffusion coefficient*, whose physical dimensions are cm^2/s. Given an initial condition $T(x, y; 0)$ this equation determines the temperature distribution at a later time t. Our purpose is to compute the temperature distribution $T(x, y; t)$ at successive values of t until the temperature profile is completely flat. To simplify the output, the code shows the temperature profile at mid-height prints three temperature values at the front border, the center of the plate, and the back border along the x direction, as shown in Figure 15.3.

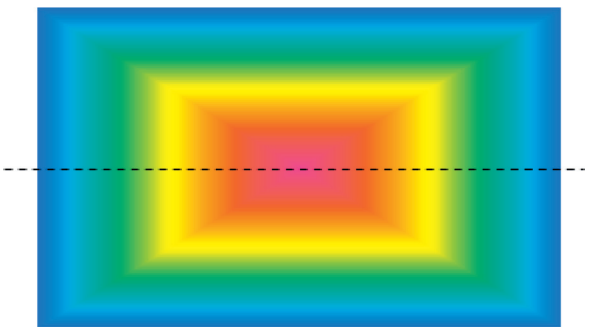

FIGURE 15.3

Heat diffusion in a rectangular plate.

15.2.2 METHOD OF SOLUTION

Section 15.9 summarizes the basic elements involved in the solution to this problem: finite-difference discretization of the heat diffusion equation, and the definition of the Fourier coefficients of the unknown solution. It is shown that the heat equation for the Fourier coefficients becomes a set of ordinary, uncoupled differential equations for each coefficient, immediately solved in terms of the known Fourier coefficients for the initial condition. The Fourier coefficients of the solutions as a function of time are therefore explicitly known, and given by Equation (15.8).

The procedure to solve our heat diffusion problem, shown in Figure 15.4, is straightforward:

- The Fourier coefficients of the initial temperature distribution are first computed and stored in a matrix `D`.
- Following Equation (15.8), the Fourier coefficients of the solution at time t are then obtained by multiplying each one of the `t=0` Fourier coefficients stored in D, by the damping factors defined in Equation (15.7). The resulting Fourier coefficients at time `t` are stored in a matrix `F`.
- Finally, the inverse Fourier transform of F is computed, to obtain the temperature profile at time t.

We will be using in this example *complex* Fourier transforms, which are linear maps among complex matrices, because they are easier to work with. While the temperature distribution $T_{m,n}(t)$ is a real matrix, its Fourier transform $\tau_{i,j}(t)$ is a complex matrix. When the inverse Fourier transform is performed, the result is naturally a complex matrix whose imaginary part is a null matrix (all matrix elements equal to zero). At this point, a simple trick can be used to avoid wasting CPU cycles in computing a result that we know has to be zero. Since the Fourier transform is a linear map, it is possible to compute two temperature profiles at two successive times in one shot, by proceeding as follows:

- The Fourier transform of the temperature distribution at time `t1` is computed from the matrix `D`. Results are stored in the matrix `F`.
- The Fourier transform of the temperature distribution at time `t2` is computed from the matrix `D`. Results are stored in the matrix `F2`.
- The matrix `iF2` is added to `F1`, and its inverse transform is computed. Then, the real part of the resulting `F1` matrix is the temperature profile at time `t1`, and its imaginary part is the temperature profile at time t2.

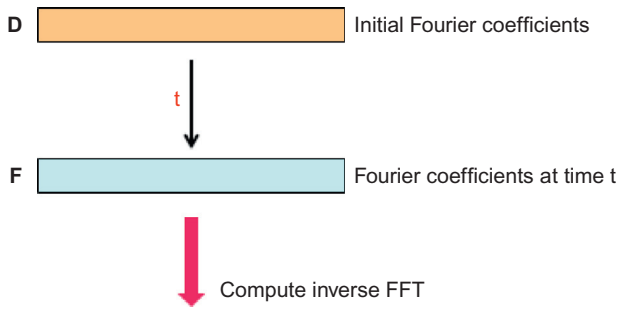

FIGURE 15.4

Data organization in sequential code.

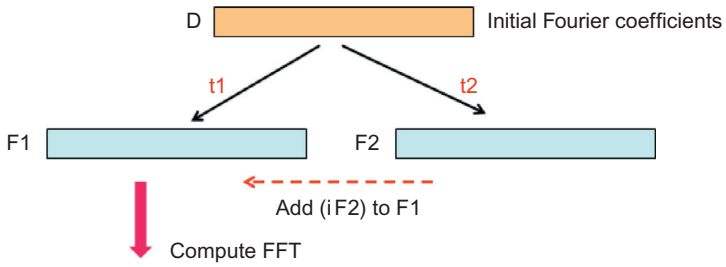

FIGURE 15.5

Data organization in sequential code.

Figure 15.5 shows the way the data structures are organized in the sequential version of the code. In the parallel version, all these preliminary steps are the same, the only difference being the fact that the final inverse FFT that provides the two temperature profiles at times t1 and t2 is pipelined over two threads.

One interesting observation following from Equation (15.8) is that it is possible to know at once the constant value T_∞ toward which the temperature will ultimately relax. Indeed, the coefficients $C_{j,k}$ are strictly positive, except for $j = k = 0$. Then, Equation (15.8) implies that, as t grows, all the Fourier coefficients tend to 0, except for $\tau_{0,0}(t)$, which remains equal to $\tau_{0,0}(0)$. Therefore, we know that:

$$T_\infty = \frac{1}{NM}\tau_{0,0}(0)$$

15.2.3 FFT ROUTINES

Section 15.10 at the end of the chapter contains a description of the FFT routines used in this example. The key routine is fft1, which computes the FFT of a one-dimensional complex vector of size N. The same routine computes the direct and inverse transforms. Since the FFT routines perform in-place modifications of the input data, the same vector that contains the initial input data contains its Fourier transform when the FFT routine terminates.

The listing below shows how the one-dimensional FFT routine operates. The vector size N must be a power of two (this is checked by the FFT routine).

```
std::complex<double> V[N];
...
fft1<double>(V, N, 1);  // direct FFT
//Here, V contains the Fourier coefficients of the initial vector
...
fft1<double>(V, N, -1); // inverse FFT
// The initial vector is now V/N
```

LISTING 15.1

Operation of fft1

Two-dimensional N×M matrices are allocated as a huge vector of size NM, holding the successive M rows, each one of size N, as explained in Section 13.7. The two-dimensional FFT routine fft2 computes the two-dimensional FFT as a succession of two one-dimensional FFTs, first in the *x* direction and then in the *y* direction. However, the one-dimensional routine fft1 *requires contiguous data* to operate. Therefore, it cannot be used directly to compute the FFT in the *y* direction, and an intermediate matrix transposition is necessary. The two-dimensional FFT routine fft2 therefore involves four steps: computing the one-dimensional FFT in the *x* direction for each row, transposing the matrix, computing again the one-dimensional FFT *in the new x direction*, and finally transposing again the matrix to recover the initial configuration.

It follows from this observation that the operation of fft2 is just the result of two successive, in-place, operations of a new routine called fft2h (half of fft2), which operates as follows:

- The one-dimensional FFT in the *x* direction is computed for each row.
- The matrix is transposed.

15.3 SEQUENTIAL CODE HEAT.C

Let us start by discussing the way the computational strategy just discussed is implemented. Input data is first read from a data file. Then, all the memory buffers are allocated, and the initial temperature distribution is defined, as indicated before, as a complex array D of size NM (with a null imaginary part). The first computational step passes this array to fft2, so that on return D contains, for the rest of the run, the Fourier coefficients of the initial condition.

The codes executes nSteps time steps, in which the initial time is incremented by a fixed amount δ at each step. The initial Fourier coefficients stored on D are propagated to time *t*, and the inverse FFT yields the corresponding temperature distribution. Two successive time steps are handled together—as discussed before—to avoid redundant computations. The resulting temperature distributions at mid-height are printed and compared with the (known) asymptotic value.

The listing below shows the set of global variables in the application.

```
std::complex<float> *D;      // holds FT of initial condition
std::complex<float> *F1;     // holds FT for time t1
std::complex<float> *F2;     // holds FT for time t2
float   *damp;               // array of damping factors, computed once.

// Data read from file fheat.dat
// ----------------------
int     N, M;                // sizes of 2D array
int     nB;                  // number of working buffers F[]
int     steps;               // number of time steps
float   deltaT;              // time step
```

LISTING 15.2

Global data set in Heat.C

The codes adopt the same notation used in Figures 15.4 and 15.5. In the parallel versions, the working buffer F1 needs to be replaced by an array of working buffers of size nB. In the sequential code that follows, nB is ignored.

Template functions appear here and there because the std::complex<T> STL class can take, in our case, either float or double as the template argument. We work here with floats, but the code works equally well with doubles. A few auxiliary functions are defined to simplify the listings. They are next described in the order in which they appear in the code:

- InitJob() reads the input data from the fheat.dat file, allocates the memory buffers, and prints the input data values. CloseJob() releases the memory buffers.
- CInitialCondition(D, N, M, FCT1, FCT2, -a, -a, a, a) initializes the complex matrix D holding the initial temperature configuration:
  - FCT1 and FCT2 are pointers to two functions $f(x, y)$ that describe the real and imaginary parts of the initial temperature distribution. In our case, FCT2 is a function that returns 0 everywhere. The discretized values of the temperature in the two-dimensional domain are computed in such a way that the lower-left corner corresponds to (-a, -a) and the upper-right corner to (a,a) in physical space.
- PrintStatus(F1, N, M, t1, t2) prints the temperature distribution at mid-height for the two results (real and imaginary parts) encapsulated in F1. The real(imaginary) parts correspond to times t1(t2).
- The two functions quoted above, which receive as arguments the template data items D and F1, are themselves template functions. They work equally well for any choice of template argument in D and F1, and they are defined in the include file HeatUtility.h.

The listing that follows is the complete main() function of Heat.C.

```
int main(int argc, char **argv)
   {
   int Ntot;
   float a = 2.0;
   int k, count;
   float t1, t2;
   std::complex<float> _Im(0.0, 1.0);
   CpuTimer TR;

   InitJob();
   Ntot = N*M;
   deltaT = 3000;
   F1  = new std::complex<float>[Ntot+1];

   CInitialCondition(D, N, M, FCT1, FCT2, -a, a, -a, a);
   PrintStatus(D, N, M, 0.0, 0.0);

   fft2<float>(D, N, M, 1);        // FFT of initial condition

   TR.Start();
   t2 = 0.0;
   for(count=1; count<steps; count++)
      {
      t1 = t2 + deltaT;
```

Continued

```
    t2 = t1 + deltaT;

    for(k=0; k<Ntot; ++k)  // Copy D to F1 and F2
       {
       F1[k] = D[k];    // will be FT at time t1
       F2[k] = D[k];    // will be FT at time t2
       }

    // damp F1 to time t1, F2 to time t2
    // - - - - - - - - - - - - - - - - - -
    Damp(F1, M, N, t1);
    Damp(F2, M, N, t2);

    // Construct F1 + i * F2, and insert time information.
    // - - - - - - - - - - - - - - - - - - - - - - - - - -
    for(k=0; k<Ntot; ++k)
        F1[k] += _Im * F2[k];
    std::complex<float> c(t1, t2);
    F1[Ntot] = c;

    fft2h(F1, M, N, -1);              // first FFT half
    fft2h(F1, M, N, -1);              // second FFT half
    for(k=0; k<Ntot; ++k) F1[k] /= (N*M);
    PrintStatus(F1, N, M, F1[Ntot].real(), F1[Ntot].imag());
    }
  TR.Stop();
  TR.Report();
  CloseJob();
  }
```

LISTING 15.3

Main() function in Heat.C.

For each pair of successive time steps, the initial condition Fourier coefficients are copied to the F1 and F2 buffers, and then the Damp() function constructs the Fourier coefficients at times t1 and t2, using Equation (15.8). Then, iF2 is added to F1, before transforming back to physical space. Note that F1 is allocated with an extra slot at the end (which does not participate to the FFT operation) where time information is stored in the form (t1+i*t2). This is redundant in this case, but will be needed in the pipelined versions to inform the second stage of the time tags.

Example 1: Heat.C

To compile, run make heat. Sequential version. The input data is read from file fheat.dat.

The initial configuration is a Gaussian distribution concentrated at the center of the plate. Running the example, we can see how the temperature profile at mid-height along the plate progressively flattens until the known asymptotic value is reached.

15.4 PIPELINED VERSIONS

The parallel versions of the code deploy a two-stage pipeline, in which the first stage performs exactly the same steps discussed before but, once the final value of F1 is known, only the first half of the FFT is computed by using fft2h. Then, the second stage is released to complete the second half of the FFT computation and prints the two temperature profiles.

This computational strategy introduces two basic issues:

- Additional data buffers are needed, to enable the simultaneous operation of both threads. Obviously, more than one working buffer F1 is required if the two threads produce in-place modifications of two successive data samples.
- The two threads must be synchronized, so that the second stage acts on a given data buffer only after the first one has completed its operation.

15.4.1 USING A CIRCULAR ARRAY OF WORKING BUFFERS

To cope with the first issue, *an array of* nB *buffers* F1[k]—where the index k is in the range k=0,1,…,nB-1—is introduced, as shown in Figure 15.6. The optimal values of the number (nB) of working buffers depends on the way the worker threads are synchronized, and will be discussed later. The first pipeline stage stores the Fourier coefficients at times t and $t+\delta$ in F1[k], computes the first half of the two-dimensional FFT, and moves to the next buffer F1[k+1] to repeat the computation for times $t+2\delta$, $t+3\delta$, and so on. The second stage follows behind by completing the second half of the FFT computation, and by printing the two temperature profiles. This array of buffers operates like a *circular array*: the index following the last one, nB-1, is the index 0. This is implemented as follows.

```
for(int n=0; n<nSteps; ++n)
   {
   int k = n%nB;      // k is in range [0, nB-1]
   ..
   // use buffer F1[k]
   ..
   }
```

LISTING 15.4

Operation of a circular array of buffers

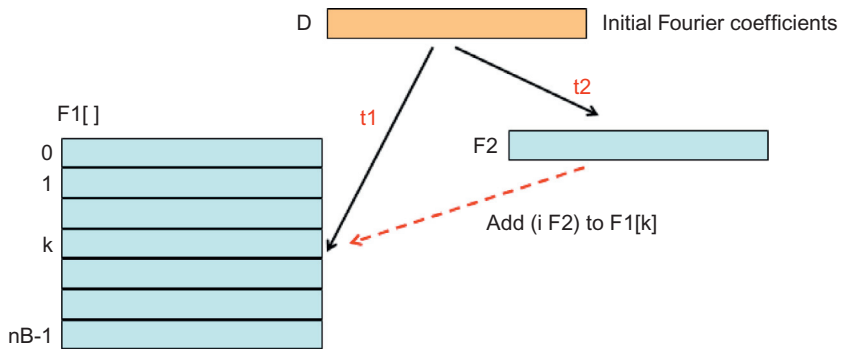

FIGURE 15.6

Data organization in pipelined code.

The circular array is useful because it would be wasteful to introduce as many buffers as time steps. The only requirement is that the first stage, which is leading the computation, does not wraparound and overwrites a buffer before the trailing second stage has finished with it. As we will see, this introduces constraints on the number of buffers nB, which depend on the way threads are synchronized.

One last comment about the data organization adopted in the code. Each one of the arrays F1[k] has size (NM+1), not NM. The first NM complex values are used as before to store the Fourier transforms. The extra complex component is used by the first stage to store the two times being handled in the form t1+it2. In this way, the second stage, when it completes the calculation of the two temperature profiles, knows the time dates to which they correspond.

The listing below shows the additional data items involved in the pipelined codes. The array pointer *F1 is replaced by **F1, a pointer to an array of pointers, allocated as shown in Section 13.7. The remaining data items are related to stage synchronization, and are discussed in the following sections.

```
std::complex<float> **F1;   // array of nB buffers

// Synchronization
// - - - - - - - --
OBLock BL;                            // for handshake synch
int     capacity;                       // for producer-consumer synch
tbb::concurrent_bounded_queue<int> Q;  // for producer-consumer synch
```

LISTING 15.5

Additional global data items in the pipelined codes

15.4.2 PIPELINE WITH HANDSHAKE SYNCHRONIZATION

There are two different ways to synchronize the two stages. In the first one, which we call *handshake synchronization*, both threads execute the loop over the time steps shown in Listing 15.2. But the first stage explicitly releases the second one after each iteration, and the second stage waits to be released by the first one to start a new iteration.

This synchronization mechanism, shown in Figure 15.7, is implemented with a Boolean lock, as follows:

- The initial state of the Boolean lock is false.
- Stage 1 toggles the state to true to inform stage 2 that it can proceed.
- Stage 2 waits for true and, when released, sets the state back to false to acknowledge the handshake.
- If, at the end of a loop iteration, stage 1 finds the state true, it knows that stage 2 has not yet acknowledged the previous handshake. Therefore, it waits for false before resetting the state to true.
- If, before starting a new iteration, stage 2 finds the state false, it knows that stage 1 is late and waits for true.
- To sum up, the synchronization proceeds as follows:
  - At the end of each iteration, stage 1 waits for false and sets the state to true.
  - At the beginning of each iteration, stage 2 waits for true and sets the state to false.

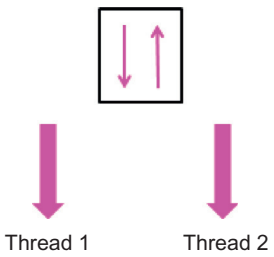

FIGURE 15.7

Thread synchronization with a Boolean lock.

What about the number of buffers nB? Looking back at the synchronization strategy discussed above, we can observe that, if the second stage is operating on F1[k], the first stage can at most operate on F1(k+2] but not beyond. Indeed, if stage 2 operates on F1[k], stage 1 naturally operates on F1[k+1]. However, it can happen that stage 1 completes its computation before stage 2, sets the state to true, and moves to F1[k+2]. Therefore, this scenario requires at least nB=3 buffers to operate correctly and avoid overwriting busy buffers. A larger number of buffers will do no harm.

> To avoid data corruption in this synchronization pattern, the number of buffers nB must be larger than 2.

15.4.3 EXAMPLES OF HANDSHAKE SYNCHRONIZATION

There are two examples, HeatOmpB.C and HeatThB.C, using OpenMP or SPool managed threads, and a OBLock or a BLock for synchronization, respectively. We describe next in some detail the OpenMP implementation. The SPool implementation is very similar, with a small number of differences that should, at this point, be familiar to the reader.

```
void HeatTaskFct()
   {
   int k, count, bindex;
   double t1, t2;
   std::complex<float> *F;    // pointer to working buffer
   int Ntot = N*M;

   int rank = omp_get_thread_number();
```

Continued

```
if(rank==0)    // THIS IS FIRST STAGE
   {
   t2 = 0.0;
   for(count=1; count<steps; count++)
      {
      bindex = count%nB;
      F = F1[bindex];    // select working buffer
      t1 = t2 + deltaT;
      t2 = t1 + deltaT;
      F[Ntot].real() = t1;
      F[Ntot].imag() = t2;

      // repeat steps shown in Listing 3:
      // – – – – – – – – – – – – – – – –
      // Copy D to F and F2
      // Damp F to time t1
      // Damp F2 to time t2
      // Construct F = F + i * F2.

      // Compute first half of inverse FFT
      fft2h(F, N, M, –1);         // half fft2

      BL.Wait_For(false);  // Release next stage
      BL.SetStare(true);
      }
   }

if(rank==1)   // THIS IS SECOND STAGE
   {
   for(count=1; count<steps; count++)
      {
      bindex = count%nB;
      F = F1[bindex];        // select working buffer

      BL.Wait_Until_True();  // synch with stage 1
      BL.SetValue(false);

      // Complete FFT computation, and print result
      fft2h(F, M, N, –1);
      for(k=0; k<Ntot; ++k) F[k] /= (N*M);
      PrintStatus(F1, N, M, F[Ntot].real(), F[Ntot].imag());
      }
   }
}
```

LISTING 15.6

OpenMP task function, handshake synchronization

It is easy to see how these task functions implement the protocol discussed above. The first stage starts by determining the next values of the two consecutive time steps $t1$ and $t2$, as well as the index bindex of the next working buffer F1, and initializes the address of the current working buffer F=F1[bindex]. Then, it proceeds as in the sequential code, except that at the end it computes only the first half of the FFT, stores in the last redundant component of F the value of $t1 + it2$, and releases the next stage.

The second stage has been executing the same time step loop and holds the same value of bindex. After being released, it identifies the current working buffer F, completes the second half of the FFT, and passes F, the plate sizes N and M, and the two time dates t1 and t2 to an auxiliary function PrintResult(), which, from the real and imaginary parts of F, prints the two temperature profiles at times $t1$ and $t2$.

The main() function goes through the same initialization steps as in the sequential code, but in addition allocates the Boolean lock BL initialized to false. Then, it sets the number of threads to two and launches a parallel section that executes the task function listed above.

Examples 2 and 3: HeatOmpB.C and HeatThB.C

To compile, run make heatompb or make heathb. Pipelined OpenMP version or SPool version, using handshake synchronization. The input data is read from file fheat.dat.

15.4.4 PIPELINE WITH PRODUCER-CONSUMER SYNCHRONIZATION

The second way of synchronizing the two threads uses a *thread-safe queue of integers*, shown in Figure 15.8. This synchronization pattern proceeds as follows:

- The loop over the time steps that defines the buffer index k on which the stages operates is only executed by the first thread—called in this context the *producer thread*—that drives the computation.
- When the first thread has finished operating on the buffer k, it queues the index k and moves to the next loop iteration.

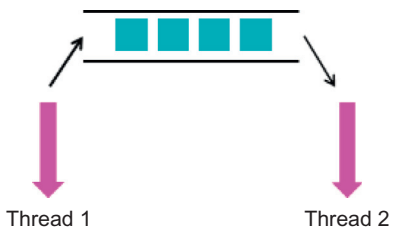

FIGURE 15.8

Thread synchronization with a TBB thread-safe queue.

- The second stage—called in this context the *consumer thread*—executes a very simple code: it dequeues the index k, completes the FFT computation on F1[k], and prints the temperature profiles. The second stage keeps doing this as long as the dequeued index value is non-negative. If the index value is negative, it stops.
- When the time step loop terminates, the first stage stops the pipeline by queuing a negative index.

This is a very loose synchronization. In the absence of handshake, the two threads are not necessarily working on neighboring buffers. Nothing forbids the first stage from running much ahead of the second one in the circular array of buffers. Because of the circular nature of the array indices, it is even possible for the first stage—if it works faster than the second—to make a complete turn, catch up the second stage from behind, and start overwriting a busy buffer. Given the asynchronous nature of the thread operation, this possibility must be excluded to dispose of a thread-safe application.

The solution to this problem is to use a thread-safe queue container, *with a finite capacity*, as was the case of our thread-safe queue class ThQueue<T> discussed in Chapter 9, or the RingBuffer class discussed in Chapter 8. Remember that, when the queue is full, producer threads wanting to queue an element wait for consumer threads to dequeue elements, making space for a new element insertion. In the same way, consumer threads wanting to deque an element from an empty queue wait for producer threads to insert new elements. The queue capacity can then be used to regulate the flow of information from producers to consumers and, in our case, to prevent the producer thread from overwriting a consumer busy buffer.

> To avoid data corruption in this synchronization pattern, the queue capacity must be smaller than nB. The optimum value is, obviously, nB-1.

Rather than using our thread-safe queue ThQueue<T> discussed in Chapter 9, we will use a similar queue container having the same capabilities, provided by the TBB library, to make once again the case for the interoperability of the different programming environments. The listing below summarizes the tbb:concurrent_bounded_queue<T> class interfaces that are needed in our application.

```
#include <tbb/concurrent_queue.h>
...
// Create an empty queue Q of objects of type T
tbb::concurrent_bounded_queue<T> Q;
...
// Set the queue capacity to N
Q.set_capacity(N);
...
// Push an element t of type T to the queue
Q.push(T& t);
...
// Pop an element t of type T from the queue
Q.pop(T& t);
```

LISTING 15.7

Use of the TBB concurrent queue class

15.4.5 EXAMPLES OF PRODUCER-CONSUMER SYNCHRONIZATION

As in the previous case, there are two examples, HeatOmpQ.C and HeatThQ.C, using OpenMP or SPool managed threads. As in the previous case, we discuss in detail the OpenMP example.

```
void HeatTaskFct()
   {
   // Same local data as in listing 5
   int rank = omp_get_thread_number();

   if(rank==0)    // THIS IS FIRST STAGE
      {
      t2 = 0.0;
      for(count=1; count<steps; count++)
         {
         ...   // Same as in listing 4
         ...
         // Compute first half of inverse FFT
         fft2h(F, N, M, -1);         // half fft2

         // Send index information to next stage
         Q.push(bindex)
         }

      // Send stop information to next stage
      bindex = -1:
      Q.push(bindex);
      }

   if(rank==1)   // THIS IS SECOND STAGE
      {
      Q.pop(bindex);
      while(bindex >= 0)
         {
         std::complex<float> *F = F1[bindex];
         fft2h(F, M, N, -1);               // complete fft computation
         for(k=0; k<Ntot; ++k) F[k] /= (N*M);
         PrintStatus(F1, N, M, F[Ntot].real(), F[Ntot].imag());
         Q.pop(bindex);
         }
      }
   }
```

LISTING 15.8

Producer-consumer OpenMP task function

Examples 4 and 5: HeatOmpQ.C and HeatThQ.C

To compile, run make heatompq or make heathq. Parallel OpenMP and SPool versions, using producer-consumer synchronization. The input data is read from file fheat.dat.

For the configuration proposed in the data file fheat.dat, all the versions of the code have similar performance. The speedup is close to 2, as expected. Notice that the pipeline is slightly unbalanced: the first stage computes much more than the second one, but the second one prints data to stdout.

15.5 PIPELINE CLASSES

The previous discussion, dealing with a pipeline with only two stages, showed in detail two different synchronization mechanism between the two worker threads. Dealing with N stages is in principle straightforward: (N-1) Boolean locks or concurrent queues are introduced, providing the synchronization objects between two consecutive stages. However, working with arrays of Boolean locks or concurrent queues soon becomes tedious and error prone. The vath library contains two simple classes that relieve programmers from this burden, by encapsulating the explicit manipulation of arrays of synchronization objects.

These classes are very simple. They have been designed to be used in any programming environment. In fact, they do little more than text processing, and they are declared and defined in the header files listed in Table 15.1. The template parameter T has the following meaning:

- In PipeBL<T>, T is the preferred connector Boolean lock class: either BLock—the idle-wait Boolean lock—or SBlock—the spin-wait Boolean lock— or the OpenMP class OBLock.
- In PipeThQ<T>, the connector is ThQueue<T>, T being type of the data item flowing along the pipeline.

The basic assumption is that these pipeline objects will be used in a context in which the N worker threads implementing the pipeline stages are identified by a set of N consecutive rank indices. This is the case for OpenMP (rank indices start at 0) and the SPool utility (rank indices start at 1). If, as is the case of the NPool pool, worker threads do not have an implicit rank index, it is easy to explicitly attribute each task a rank, as was done before.

These pipeline classes proceed as follows:

Table 15.1 In Column 3, r1 and r2 Are the First and Last Ranks of the Threads Participating to the Pipeline, and c Is the Capacity of the Related Connector Queue

Pipeline Classes		
Header	**Connector**	**Constructor**
PipeBL.h	Any BLock class	PipeBL<T> P(r1, r2);
PipeThQ.h	ThQueue<T>	PipeThQ<T> P(r1, r2, c)

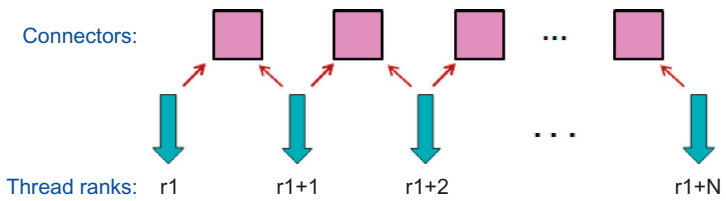

FIGURE 15.9

Structure of the pipeline objects.

- The first and last rank indices of the worker threads are passed as arguments to the pipeline constructors, as indicated in Table 15.1. The pipeline class using a concurrent queue receives optionally the queue capacity as a constructor argument, the default capacity value being 10.
- Once this is done, the pipeline constructor constructs an array of (N-1) connectors, either Boolean locks or concurrent queues. This is shown in Figure 15.9. The (N-1) connectors are identified by indices $(r_1, r_2, \ldots, r_{N-1})$.
- Threads of rank k in the pipeline act both as producer threads (except the last one) by sending synchronization information to the next stage via the connector k, and as consumer threads (except the first one) by receiving synchronization information from the previous stage via the connector (k-1). This is why threads pass their rank to the member functions used for synchronization.

15.5.1 SYNCHRONIZATION MEMBER FUNCTIONS

Table 15.2 shows the synchronization member functions used by the pipeline classes. In the PipeBL<T> class, the WaitForRelease(int r) function connecting with the previous stage returns immediately if called by the first stage. Similarly, ReleaseNext(int r) returns immediately if called by the last stage.

For the concurrent queue class, template parameter T is the type of the data item flowing along the connectors. Our previous example demonstrated how this data item is used to transfer control information to the next stage. Remember that this class implements a very loose synchronization, because threads along the pipeline do not wait for an acknowledgment from the next stage, so nothing prevents them from simultaneously operating on nonconsecutive data sets. We would expect this very

Table 15.2 In Columns 2 and 3, r Is the Integer Rank of the Caller Thread, and Flag Is a Boolean Acting as Output Parameter

Pipeline Synchronization Functions		
Class	**Previous Stage**	**Next Stage**
PipeBL<T>	WaitForRelease(r)	ReleaseNext(r)
PipeThQ<T>	T GetPrevious(r, flag)	PostNext(T token, r)

loose synchronization to be somewhat more efficient than the one based on Boolean locks, and this seems indeed to be the case in general. Some examples will be shown at the end of the chapter. In any case, we have never found an example in which this loose synchronization is less efficient that the handshake synchronization strategy.

Remember also that the ThQueue capacity must be carefully selected in order to ensure thread safety when there is the danger that producers corrupt consumer data, as was the case with circular arrays. The synchronization functions listed in Table 15.2 work as follows:

- `void PostNext(T elem, int rank)`, called by the producer thread, receives as arguments the data item to be queued in the next connector and the rank of the caller thread (to select the connector).
- `T GetPrevious(int rank, bool *flag)`, called by the consumer thread. This function dequeues and returns the first available element in the connector queue. It receives as arguments the rank of a caller thread (to select the connector) as well as the address of a Boolean flag, which is an *output parameter*. If this flag is `true`, the returned value is a valid one, queued by a producer. If the flag is `false`, the returned value is invalid because we are trying to dequeue from an empty and closed queue. We will not need to use this feature in practice.

15.6 EXAMPLE: PIPELINED SOR

We come back to the `Sor.C` code, discussed in the previous chapter. The purpose of this example is to show in detail how pipelining can efficiently handle the data dependency occurring in the Gauss-Seidel method. Indeed, by pipelining threads it is possible to implement a parallel version of this code *as such*, without using the white-black ordering trick. First of all, complete rows are no longer allocated to each thread. The data set is divided vertically rather than horizontally (see Figure 15.10). Each matrix row is then divided in equal length sectors allocated to successive pipeline stages, and each worker threads *updates its row sector in all the matrix rows*. In fact, the matrix rows are the data items flowing along the pipeline. Each thread controls the activity of the next thread in the hierarchy. In Figure 15.10, the first (black) thread controls the activity of the second (gray) thread, which in turn controls the activity of the third (white) thread.

15.6.1 SOR CODE

This application is identical to the one discussed in Chapter 14: initialization, I/O, result reporting, and all the auxiliary functions used in the code are strictly the same. The only modification is the parallel strategy, which induces some minor modifications in the `main()` function, and, of course, requires different tasks executed by the worker threads team.

There are several versions of this code, using either OpenMP or `SPool` class to manage threads, with different choices of the connector classes listed in Table 15.3. Our intention is to compare the relative performance of the different synchronization strategies in a context with significant synchronization contention. There is also a TBB version of the code, using the `tbb::pipeline` algorithm discussed in the next section. The OpenMP implementation using the queue connectors is discussed next, to show, once again, the interoperability of OpenMP with the `vath` tools.

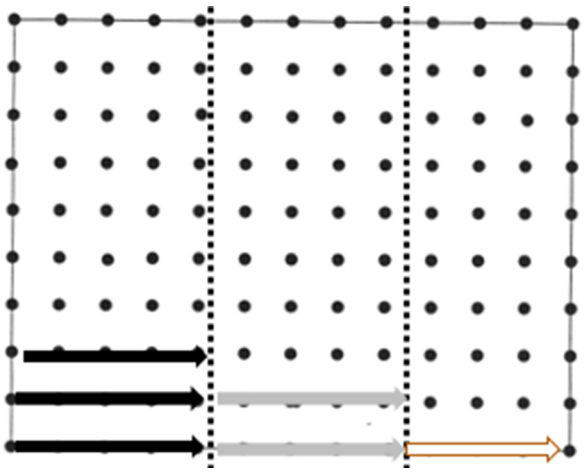

FIGURE 15.10

Pipelining three threads in the Sor code.

Table 15.3 Different Implementations of the Pipelined Sor Code, Using Different Thread Managers and Pipeline Connectors

Pipelined Sor Implementation			
Source	**Executable**	**Pool**	**Pipeline Class**
SorOmpBL.C	somp_bl	OpenMP	BLock
SorOmpSBL.C	somp_sbl	OpenMP	SpBLock
SorOmpQ.C	somp_q	OpenMP	PipeTh<int>
SorThBL.C	sth_bl	SPool	BLock
SorThSBL.C	sth_sbl	SPool	SpBLock
SorThQ.C	sth_q	SPool	PipeTh<int>
SorTBB.C	sortbb	TBB	tbb::pipeline

The listing below shows the main function in the SorOmpQ.C code. This function initializes a global pointer P to a PipeThQ<int> object. Since we are working with OpenMP, the main thread is included in the worker threads team, and the thread ranks start at 0. The correct thread rank range is passed to the pipeline constructor. The integer data item that flows along the pipeline is the matrix row index being processed. This code adopts a microtasking programming style in which the computation is driven by the main() function, and a parallel region is created each time a lattice update is executed.

```
PipeThQ<int>  *P;        // global data
Reduction<double>  R;
...
main(int argc, char **argv)
   {
   int n, m;
   double resid, error_norm;

   InputData();
   InitJob(M, N);
   CpuTimer TR;

   if(argc==2) nTh = atoi(argv[1]);
   P = new PipeThQ<int>(0, nTh-1);
   omp_set_num_threads(nTh);

   nIter = 0;
   omega = 1.8;
   TR.Start();
   do
      {
      #pragma omp parallel
         { TaskFct(); }
      anorm = R.Data();
      R.Reset();
      nIter++;
      if(nIter%stepReport==0)
          printf("\n Iteration %d done", nIter);
      }while( (anorm > EPS*anormf) && (nIter <= maxIts) );
   TR.Stop();
   // Print results, and exit
   }
```

LISTING 15.9

Main function in SorOmpQ.C

This examples uses the Reduction<double> class to perform the reduction of the partial errors accumulated by the different stages in the lattice update. The TaskFct() task function is defined in the listing below. These tasks are launched every time a new lattice update is required. They start by determining the rank of the executing thread, needed in the synchronization functions, as well as the row subsection under its control. This is done by using the same ThreadRangeOmp(int beg, int end) function used in Chapters 13 and 14 that receives in *beg, end* the global range for column indices, and returns the local range for the calling thread. Since there are N columns, the global column range is $[0, N-1]$. However, boundary values are not updated, so the range to be passed to this function is $[1, N-1)$, as the listing below shows.

- The data item flowing along the pipeline is an integer corresponding to a row index.

- The loop over rows is only executed by the first stage of the pipeline. Once the first stage has updated the row *n*, it queues this value on the next connector. When all rows have been updated, the first stage queues the value (−1) and terminates.
- The following stages dequeue the integer values inside a while loop that keeps running as long as the dequeued values are positive. Then, the corresponding matrix row is updated and, except for the last stage, the row index is queued again in the next connector. When the integer is negative, the while loop ends.

```
void TaskFct()
   {
   int n, m, col, rank;
   int nL, nH;
   double resid, error_norm;
   bool flag;

   rank = omp_get_thread_num();
   nL = 1;                          // determine [nL, nH)
   nH = N-1;                        // for this task
   ThreadRangeOmp(nL, nH);

   error_norm = 0.0;
   if(rank==0)          // This is first stage
       {
       for(m=1; m<(M-1); m++)
          {
          for(n=nL; n<nH; n++)
             {
             resid = U[m+1][n] + U[m-1][n] + U[m][n-1] + U[m][n+1]
                               - 4 * U[m][n] - F[m][n];
             error_norm += fabs(resid);
             U[m][n] += 0.25 * omega * resid;
             }
          P->PostNext(m, rank);
          }
       col = -1;
       P->PostNext (col, rank);
       R.Accumulate (error_norm);
       }
   else      // these are the following stages
       {
       col = P->GetPrevious (rank, flag);
       while (col>0)
          {
          for(n=nL; n<nH; n++)
             {
```

Continued

```
            resid = U[col+1][n] + U[col-1][n] + U[col][n-1] + U[col][n+1]
                          - 4 * U[col][n] - F[col][n];
            error_norm += fabs(resid);
            U[col][n] += 0.25 * omega * resid;
            }
         P->PostNext (col, rank);
         col = P->GetPrevious (rank, flag);
         }
      P->PostNext (col, rank);  // signal end of iteration
      R.Accumulate (error_norm);
      }
   }
```

LISTING 15.10

Task function in SorOmpQ.C

The negative index value that signals the termination of the pipeline operation must be propagated along the pipeline to ensure the correct termination of all the stages. It is clear that, given the way the code has been organized, there is no flow control problem in this example, and there is no harm in letting a stage to run ahead with its assignment. This is the reason the queue capacity is not specified in the constructor, and the default value is adopted.

The performance of the different implementations of the pipelined Sor code will be presented after discussing the TBB pipeline algorithm.

15.7 PIPELINING THREADS IN TBB

One of the most interesting advanced algorithms proposed by TBB is the pipeline class, whose official documentation is in the topic "Algorithms" in the Reference Manual [17]. The synchronization of the successive stages is implemented by a data item—a token—flowing along the pipeline. Rather than keeping the token type as a template parameter, TBB chooses a (void *) type as a universal token. This is fine, because a (void *) can point to any data type after an adequate cast, so there is no restriction on the nature of the control information flowing along the pipeline. The operation of the TBB pipeline is shown in Figure 15.11.

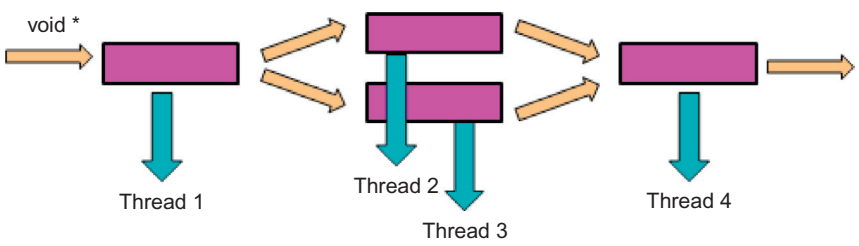

FIGURE 15.11

TBB pipeline.

As shown in the figure, the TBB pipeline algorithm is more sophisticated than the vath pipeline classes discussed before. This algorithm is indeed capable of implementing a nested parallel pattern in which stages can be concurrently executed by more than one thread. In other words: several threads can be put to execute *the same stage*. This is a very interesting feature, because it is possible in this way to accelerate the operation of a particularly slow stage, thereby improving the load balance of the pipeline. This feature is controlled by the user, who decides if a particular stage must be single-threaded or if several threads can execute it in parallel.

The TBB pipeline algorithm is based on two classes: a filter class that describes the operation of a stage, and a pipeline class that connects the filters and runs the pipeline. Note that there are no template parameters here, and that names are relevant: filter and pipeline are classes defined in the tbb namespace.

15.7.1 FILTER INTERFACE CLASS

Here is the declaration of the filter class inside the tbb namespace.

```
class filter
   {
   protected:
    filter(bool is_serial);

   public:
    bool is_serial() const;
    virtual (void *) operator() (void *token) = 0;
    virtual ~filter();
   };
```

LISTING 15.11

Filter interface class

This class defines the task function of a stage via the operator() function. The (void *) argument this function receives is the token coming from the previous stage. The (void *) return value is the token passed to the next stage. The virtual qualifier indicates, as usual, that this class is only *an interface* that defines a behavior to be implemented in derived classes.

This pure base class has a protected constructor accessible by the derived classes that receives a Boolean argument to decide about the parallel nature of the stage: true if the stage is single-threaded, false if the stage can be run in parallel by several threads. The precise level of parallelism will be fixed later on.

To implement a pipeline, filter classes are derived from the filter class. These derived classes may add whatever additional data items and internal support member functions are needed, and define the operator() function that implements the stage operation. Next, we provide a complete example for the pipelined Sor code.

15.7.2 PIPELINE CLASS

Here is the declaration of the pipeline class in the tbb namespace. This class is a packed service provided by the TBB library and is used *as such*.

```
class pipeline
   {
   public:
    pipeline();
    ~pipeline();
    void add_filter(filter\& f);
    void run(size_t max_tokens);
    void clear();
   };
```

LISTING 15.12

Pipeline class

These member functions behave as follows:

- The constructor pipeline() constructs a pipeline without stages. The stages are incorporated later on by another member function.
- The destructor destroys all the filters as well as the pipeline.
- add_filter(filter& f) adds the filter f to the pipeline. This function fails if f is already in the pipeline. Filters are connected in the order they are added to the pipeline.
- run(size_t maxtokens) runs the pipeline. The integer argument is the maximum number of filters that can be executed concurrently. If there are no parallel filters and there is one thread per stage, the optimal value of this argument is the number of threads. This value must be increased if there are parallel filters and enough threads to cope with that.
- clear() removes all the filters from the pipeline

When the run() methods is called, the pipeline starts operation, driven by the first filter (which obviously ignores its token argument). *It is important to keep in mind that the* run() *method keeps repeatedly activating the first filter until it returns* NULL. In writing code for the first filter, programmers must describe the action of the filter on a generic data item, and put some condition that decides when the filter returns NULL and stops the pipeline. But the repeated action of the filter on different objects is automatically fired by the run() method. The remaining filters just react to the token value they receive from the previous stage, and stop operation when they receive the NULL argument, after passing it to the next stage.

15.7.3 TBB PIPELINED SOR CODE

Let us now turn our attention to the TBB version of the pipelined Sor code. We have chosen to focus on this example to illustrate in detail the way the TBB pipeline algorithm works, because it raises more programming challenges than the FFT example involving only two stages. Our purpose is to discuss in some detail the way to write code where the number of stages (equal, of course, to the numbers of worker threads in the TBB pool) is only known at execution time. This is also an excellent example to understand the inner workings of the pipeline algorithm, and to illustrate the inclusion additional data items and member functions in the actual filter class derived from the TBB base class, to ensure its proper operation.

The whole issue here is the definition of the problem-specific filter class—called SorStage—derived from the TBB filter class. Since all filters operate in the same way on the data set—they all

update a given subrange of each matrix row—a unique SORstage class can be used to construct all the required chain of filter objects. However, the first filter, having a special role in driving the pipeline operation, needs to be distinguished from the remaining ones.

A microtasking style is adopted for this example: the sequential code runs the pipeline to perform one matrix update and get in return a new value of the global error. Then, main() performs the convergence test and decides if a new iteration is needed, in which case it runs the pipeline again. The fact that the pipeline filters must be reinitialized before each new iteration explains some of the features of the SORstage derived class that follows.

15.7.4 SORSTAGE CLASS

Here is the declaration of the SORstage class.

```
class SORstage: public tbb::filter
  {
  private:
   int beg, end;       // column subrange for this filter
   int rank;           // rank of this filter
   int m;              // row counter in range [2, M–1]
   void *arg;          // argument passed to next stage

   void  UpdateRow(int m);
   void* operator()(void*);

  public:
   double error_norm;  // stores accumulated partial errors
   SORstage() : filter(true), m(2), error_norm(0.0) {}

   void Reset();
      { m = 1;  error_norm = 0.0; }

   void SetFilter(int r, int nst, int NN)
      {
      rank = r;
      beg = 1; end = NN–1;
      StageRange(beg, end, r, nst);
      arg = static_cast<void *> (&nIter);
      }
  };
```

LISTING 15.13

SORstage class

This class has been equipped with several private data items, as well as a few auxiliary functions, to implement the correct operation of the pipeline.

Private data members

The meaning of the private data members is the following:

- error_norm is the internal variable used by the filter to accumulate the errors computed during the lattice swap of its data sector. At the end of the filter operation, this variable holds the partial error value provided by the stage. The global error must be obtained next by a reduction of the values reported by the different stages. This variable is public because its value will be read by main() to compute the global error.
- [beg, end) is the subrange of column indices where the stage operates.
- rank is the rank of the stage, which is equal to the order in which the filter has been inserted in the pipeline. This variable serves two purposes:
  - It distinguishes the first filter from the other ones.
  - It enables the initialization of the filter subrange [beg, end) from the knowledge of the global range [1, N-1) for the column indices.
- m is a running row counter swapping all the row indices to be updated. This counter starts from 1 and runs up to M-1 on successive row updates.
- arg is a void pointer, pointing to a valid address, passed along the pipeline. The pointed value is irrelevant, because we are not in this example sending explicit control information along the pipeline. Each filter knows what it has to do. The pointer is passed just to synchronize the filters.

Member functions

Member functions operate as follows:

- *SORstage()* is the constructor. It creates a filter and initializes the internal working variables m and error_norm. Note that this constructor calls the parent class constructor filter(bool) with a true argument, meaning that the filter is an ordinary sequential filter, and that it does not run in parallel. Then, it initializes the internal working variables.
- *UpdateRow()* is an auxiliary function that updates a subrange of the row defined by the internal row counter m. This function is introduced only to avoid code clutter.

```
void SORstage::UpdateRow(int m)
   {
   for(int n=beg; n<end; n++)
      {
      double resid = U[m+1][n] + U[m–1][n] + U[m][n–1] +
                     U[m][n+1] – 4 * U[m][n] – F[m][n];
      error_norm += fabs(resid);
      U[m][n] += 0.25 * omega * resid;
      }
   }
```

LISTING 15.14

UpdateRow() auxiliary function

- *SetFilter()* is an auxiliary function needed to complete the filter initialization. Indeed, filters have to be created before they are inserted in the pipeline. However, their rank is determined at the moment of its insertion, and this auxiliary function completes the filter initialization at insertion

time, determining at the same time the data sector [*beg*, *end*) allocated to the stage. This is done by calling another auxiliary function StageRange(), identical to the ThreadRange() function we have been using.

– *Reset()* is needed because the pipeline will be successively run a large number of times. When updating successive rows, the filters keep the memory of the next row to work upon, and the accumulated errors. However, before a new lattice swap starts, these internal working variables must be reset to start swapping rows from m=1 and to start a new accumulation of errors in error_norm.
– *operator()* defines the filter action. This function discriminates between the first and remaining filters. All filters update the row corresponding to the current value of m (this includes the accumulation of the partial errors), increase m for the next operation, and return a valid void * to the next stage. The only difference is in the way termination is handled.
  – The first filter—which must ignore its argument—terminates when the running row index m gets out of range. In this case, it accumulates partial error in R and returns a NULL pointer to the next filter.
  – The next filters keep running and returning a valid pointer as long as they do not receive a NULL pointer. When this happens, they also perform the global error reduction and return the NULL pointer to the next stages, as shown in the listing below.

```
void* SORstage::operator()(void* p)
    {
    if(rank==1)     // code for first filter
        {
        if(m==(M-1)) return NULL;
        else
           {
           UpdateRow();
           m++;
           return arg;
           }
        }

    if(rank>1)      // code for next filters
        {
        if(p==NULL) return NULL;
        else
            {
            UpdateRow();
            m++;
            return arg;
            }
        }
    }
```

LISTING 15.15

Filter action on data

This completes the discussion of the relevant features of the SORfilter class.

15.7.5 FULL PSORTBB.C CODE

Finally, we list below the main() function code. The auxiliary functions and variable identifiers are the same as those used in the previous versions of this example.

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

   InputData();
   InitJob(M, N);

   omega = 1.8;
   if(argc==2) nTh = atoi(argv[1]);     // override nTh
   tbb::task_scheduler_init init(nTh);  // start scheduler

   // construct the pipeline stages
   //  - - - - - - - - - - - - - - - --
   tbb::pipeline pipeline;
   SORstage ST[16];
   for(n=1; n<=nTh; n++) ST[n].SetFilter(n, nTh, N);
   for(n=1; n<=nTh; n++) pipeline.add_filter( ST[n] );

   tbb::tick_count t0 = tbb::tick_count::now();
   do
      {
      for(n=1; n<=nTh; n++) ST[n].Reset();
      pipeline.run(1);                    // lattice update
      nIter++;
      anorm = 0.0;
      for(n=1; n<=nTh; ++n) anorm += ST[n].error_norm;
      if(nIter%stepReport==0)
          {
          printf("\n Iteration %d performed", nIter);
          printf("\n Current error: %g\n", anorm);
          }
      }while( (anorm > EPS*anormf) && (nIter <= maxIts) );
   tbb::tick_count t1 = tbb::tick_count::now();

   // Print results, and exit
   }
```

LISTING 15.16

Main function

Our purpose in this example is to decide the number of stages at runtime, using as many stages as threads created in the TBB pool. A static array of 16 filters is declared. Then, the requested number (less

than 16) is set and inserted in the pipeline. At each iteration, main() resets the filters—which includes the internal error—and runs the pipeline. After running the pipeline, the total error is computed by main() from the partial errors stored in the filters.

15.8 SOME PERFORMANCE CONSIDERATIONS

This section focuses on the relative performance of the different implementations of the Sor code, in order to develop some insight about the efficiencies of the different synchronization strategies. Out the seven codes listed in Table 15.3, we will only report about the three OpenMP versions and the TBB one. Indeed, the code performance is essentially controlled by the pipeline synchronization methods. The three SPool versions have identical performance to the corresponding OpenMP versions, and show the same trends.

The performance tests are carried out in a Unix-Linux environment, because in this case wall, user, and system times are reported. Wall times are, of course, related to the parallel speedup and inform about the scaling quality of the code. In an ideal world with zero synchronization costs, the user time should be equal to the wall time times the number of stages, and equal in turn to the sequential execution time. Values of the user times significantly higher than the sequential execution time indicate significant synchronization overhead in user space, and, as we will see, this is sometimes the case with some implementations.

The cost of synchronization depends only on the internal behavior of the connectors and on the number of stages. Increasing the number of stages, increased synchronization overhead is expected. On the other hand, the ratio of computation to synchronization grows with the system size. Increasing the system size, a decreased impact of the synchronization overhead is expected.

As a first case of low synchronization overhead, we have taken a N=M=1000 configuration, with the optimal value of the over-relaxation parameter ω to benefit from fast convergence (sequential execution time is 27 s). The performance results are given in Table 15.4.

Table 15.4 shows a very honorable behavior of all implementations for two stages. When moving to four stages, the implementations based on the atomic Boolean lock (somp_sbl) and the concurrent queue (somp_q) show a better scaling than somp_bl, based on an idle wait Boolean lock. This was to be expected: the atomic Boolean lock avoid sustained mutex locking, and the concurrent queue implements a more relaxed synchronization.

Table 15.4 1000 × 1000 Configuration, Optimal Value of ω, 3425 Iterations, Sequential Execution Time 26.4 s

Pipelined Sor, 2 and 4 Stages						
Code	**Wall(2)**	**User(2)**	**Sys(2)**	**Wall(4)**	**User(4)**	**Sys(4)**
somp_bl	15.1	29.2	0.7	10.9	36.7	3.8
somp_sbl	14.4	28.5	0.1	8.5	33.3	0.1
somp_qq	14.5	28.8	0.6	8.9	33.7	1.1
sortbb	16.3	32.4	0.1	11.5	44.1	1.6

Table 15.5 600 × 600 Configuration, $\omega = 1.9$, 27,152 Iterations, Sequential Execution Time 74 s

Pipelined Sor, 2 and 4 Stages						
Code	**Wall(2)**	**User(2)**	**Sys(2)**	**Wall(4)**	**User(4)**	**Sys(4)**
somp_bl	41.3	81.5	0.9	47.2	123.4	34.2
somp_sbl	40.8	80.4	0.3	21.8	85.2	0.5
somp_q	41.2	81.2	0.9	23.5	90.3	2.5
sortbb	47.2	94.3	0.1	38.0	145.9	6.1

Next, we look at a smaller system (N=M=600), where synchronization overhead is expected to have a more significant performance impact. Using the optimal value of ω, the sequential code executes in only 7 s. In order to have more significant execution times, we used $\omega = 1.9$, slightly away from the optimal value. In this case, the sequential execution time is 74 s. This only increases the number of iterations required for convergence, without changing the ratio of computation to synchronization. The performance results are shown in Table 15.5.

Table 15.5 shows that somp_sbl and somp_q still show correct scaling when moving from 2 to 4 stages, but the behavior of somp_bl is somewhat outrageous. Remember that the idle wait Boolean locks keep preempting the executing thread to a wait state, and that this is a system activity. The significant system time reported in this case is not an accident. This incorrect behavior may seem suspicious, but it turns out that *exactly the same phenomenon happens with the SPool code* sth_bl. It was indeed to be expected that optimized spin locks in which threads are not preempted are much better adapted to a context of high synchronization activity with very short waits.

The TBB implementation does not show the best performance, but remember that the TBB pipeline is much more sophisticated than the simple connector classes, and that it allows a more flexible pipeline strategy with the activation of parallel filters.

15.9 ANNEX: SOLUTION TO THE HEAT DIFFUSION EQUATION

This annex summarizes the three steps that set the stage for the numerical solution of the heat diffusion problem used in the text:

- A finite-difference discretization of the heat diffusion equation.
- Definition of the one- and two-dimensional Fourier transforms.
- Analytic solution for the time dependence of the Fourier coefficients of the temperature distribution.

Finite-difference discretization

As in Chapter 14, the *finite-difference method* is used to obtain numerical solutions of the heat diffusion equation. The continuous (x, y) variables are discretized by introducing a two-dimensional grid, and by representing all continuous functions by their values at the discrete sets of points:

$$x_n = x_0 + (n-1)\Delta, \quad n = 0, \ldots, (N-1)$$

$$y_n = y_0 + (m-1)\Delta, \quad m = 0, \ldots, (M-1)$$

where Δ is the *grid spacing*. The point (x_0, y_0) corresponds to the lower-left corner of the rectangular domain, as shown in Figure 15.12. These naming conventions are the same as those used in Chapter 14. Using the notation $T_{m,n}(t)$ for $T(x_n, y_m; t)$, the discretized heat conduction equation becomes:

$$\frac{\partial T_{m,n}}{\partial t} = -\sigma \left(T_{m+1,n} + T_{m-1,n} + T_{m,n+1} + T_{m,n-1} - 4T_{m,n}\right)$$

where $\sigma = \kappa/\Delta^2$. The physical dimension of σ is $1/\text{s}$, namely, the inverse of a time. This parameter can be eliminated from the equation by using a dimensionless time variable $t' = \sigma t$. This is what we will do next. The final form of the discretized heat conduction equation is:

$$\frac{\partial T_{m,n}}{\partial t} = \left(T_{m+1,n} + T_{m-1,n} + T_{m,n+1} + T_{m,n-1} - 4T_{m,n}\right) \tag{15.1}$$

where it is understood that time is measured in units of $1/\sigma$, whatever this parameter is.

Fourier transforms

The initial value problem associated with Equation (15.1) can be solved exactly by using the Fourier transform. The original continuous temperature distribution has been replaced by the values of a function sampled on a discrete set of points. It is, however, evident that the periodic boundary conditions we adopted for the temperature distribution mean that $T_{m,n}(t)$ is, for any t, a periodic function of m and n with period M and N, respectively. It is at this point that the Fourier transform enters the stage.

Let us first consider the one-dimensional case, by considering a complex valued function H_n sampled on N values $n = 0, \ldots, (N-1)$, and periodic in n with period N. Its Fourier transform h_k is defined as:

$$H_n = \sum_{k=0}^{N-1} h_k \exp\left(\frac{2\pi i k n}{N}\right) \tag{15.2}$$

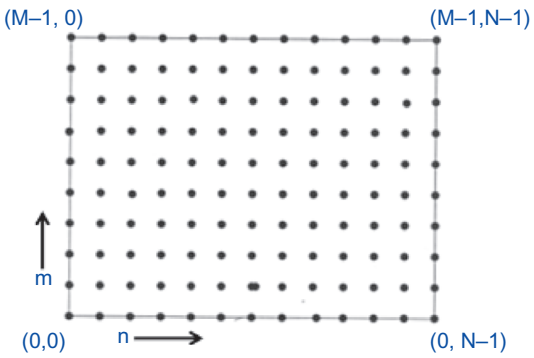

FIGURE 15.12

Discretization of the heat diffusion equation.

This expression defines a linear mapping of N *complex* values H_n into n complex values h_k. It can be immediately verified that $H_{n+N} = H_n$. In fact, this periodic function is being expanded as a superposition of periodic functions of period N (the exponential factors) with smaller and smaller wavelengths $N/2k\pi$. The inverse Fourier transform, that is, the coefficients h_k of the different periodic functions in the expansion, are given by:

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n \exp\left(-\frac{2\pi ikn}{N}\right) \tag{15.3}$$

Two-dimensional Fourier transforms

The definitions given above can be immediately generalized to two dimensions. The temperature distribution in our problem can be expanded as:

$$T_{m,n}(t) = \sum_{j=0}^{M-1}\sum_{k=0}^{N-1} \tau_{j,k}(t) \exp\left(\frac{2\pi ijm}{M}\right) \exp\left(\frac{2\pi ikn}{N}\right) \tag{15.4}$$

and the inverse transform is given by:

$$\tau_{j,k}(t) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} H_{m,n}(t) \exp\left(-\frac{2\pi ijm}{M}\right) \exp\left(-\frac{2\pi ikn}{N}\right) \tag{15.5}$$

Equation for the Fourier coefficients

Replacing the Fourier expansion (Equation (15.4)) of the temperature distribution in the original heat diffusion Equation (15.1), we obtain, with the help of some well-known trigonometric identities, the following simple differential equation for the Fourier coefficients $\tau_{j,k}(t)$:

$$\frac{\mathrm{d}\tau_{j,k}}{\mathrm{d}t} = -4C_{j,k}\,\tau_{j,k} \tag{15.6}$$

where

$$C_{j,k} = \sin^2\left(\frac{\pi j}{M}\right) + \sin^2\left(\frac{\pi k}{N}\right) \tag{15.7}$$

Equation (15.6) represents a set of single, uncoupled differential equations, one for each Fourier coefficient, which can be trivially integrated in terms of the Fourier coefficients of the initial condition, $\tau_{j,k}(0)$:

$$\tau_{j,k}(t) = \tau_{j,k}(0)\exp(-4C_{j,k}t) \tag{15.8}$$

15.10 ANNEX: FFT ROUTINES

The FFT algorithm optimizes the number of computational steps involved in such a computation. Let us consider first the one-dimensional case. The Fourier transform is a mapping from N complex numbers H_n to N complex numbers h_k, and it is not difficult to convince oneself that, in principle, there are N^2 operations involved in the computation. The FFT algorithm reduces the number of computations to $N \log N$.

The FFT algorithm is particularly simple to use when the sample size N is a power of two, $N = 2^p$. We will henceforth restrict ourselves to this case. The algorithm operates on a vector of size N. After reordering the vector elements, the algorithm performs $\log N$ successive *in-place* modifications of this array. What is left at the end is the required Fourier transform. Full details on the algorithm and on all the issues discussed here can be found in Ref [16].

Since the one-dimensional FFT algorithm is a succession of in-place transformations of a vector, it is by itself a good candidate for a pipelined operation. We have, indeed, written and tested pipelined versions of the one-dimensional routine that we will soon discuss, but we found that one has to go to enormous vector sizes N (several millions) to detect a significant enhancement in computational performance. Pipelining a two-dimensional FFT as a succession of two one-dimensional operations is much more efficient.

The FFT routines we will be using in this example are defined in the include file Fft.h. They are template C++ functions that rely on the std::complex<T> objects of the standard C++ library, T being the type adopted for the components of complex numbers (float or double). Here are the relevant definitions:

One-dimensional FFT

```
template<typename T>
void fft1( std::complex<T> *D, int sz, int isign)
   { ... }
```

LISTING 15.17

One-dimensional FFT

This function is the same that the one-dimensional complex FFT routine of Ref [16], except that it uses directly an array of complex numbers, and that it checks that the data size is a power of two. The conventions for using this routine are as follows:

- D, a pointer to an *offset 0 complex array*, is both an input and output parameter. On input, it contains the original data set, and on output it contains the result of the FFT operation.
- sz is the size N of the complex vector. If sz is not a power of two, the function returns with an error message.
- isign selects the direct or inverse transform. If isign=1, the routine returns the Fourier coefficients of the initial dataset. If isign=-1, the routine performs the inverse transform on the Fourier components.
- To recover the initial data samples from the inverse Fourier transform, the output vector components must be divided by sz.

To check the routine, take a complex vector of size N, perform the direct (isign=1) and the inverse (isign=-1) transforms, and divide the resulting vector by N. You should recover the initial vector.

Two-dimensional FFT

```
template<typename T>
void fft2( std::complex<T> *D, int szx, int szy,
           int isign)
   { ... }
```

LISTING 15.18

Two-dimensional FFT

This routine computes a two-dimensional FFT, on a data set of size szx in the x direction, and size szy in the y direction. The role of isign is the same as before. D is a pointer to an *offset 0 complex array* of size szx.szy that packs the two-dimensional matrix as a one-dimensional vector where the rows are placed one after the other. Again, the routine checks that sizes are powers of two.

It follows from Equations (15.4) and (15.5) that the two-dimensional FFT is just the result of two successive one-dimensional FFTs in the x and y directions. The problem at this point is that the one-dimensional fft1 routine requires the input to be consecutive vector elements. Given the way in which the two-dimensional data set is allocated, this is only true for the FFT in the x direction. In the y direction, column elements are not contiguous. Reference [16] has an in-depth discussion of this issue, with routines that generalize the FFT algorithm to any number of dimensions.

We have adopted here a simpler approach: the two-dimensional fft2 routine uses first fft1 to compute the FFT in the x direction, then it transposes the matrix, then it uses again fft1 to compute the FFT in the new x (old y) direction, and finally it transposes the matrix again to come back to the initial configuration. We have seen that performance is not very different from the refined two-dimensional FFTs of Ref [16]. On the other hand, there is huge benefit because the operation of this routine can be pipelined in a trivial way.

Half of two-dimensional FFT

```
template<typename T>
void fft2h( std::complex<T> *d, int szx, int szy,
            int isign)
   { ... }
```

LISTING 15.19

Half of a two-dimensional FFT

This routine is *half a fft2* routine: FFT in the x direction followed by transposition. Calling it twice, it is equivalent to fft2. This is the routine to be used in the parallel version of this example: each one of the two worker threads will execute half of the two-dimensional FFT computation.

Fourier transform of real-valued functions

We have just stated that the initial array H_n and its Fourier transform h_k are in general complex valued arrays. We can, of course, use these Fourier transforms for real-valued functions H_n, and obtain a set of complex-valued Fourier coefficients h_k, from Equation (15.3). When transforming back to the initial configuration space using Equation (15.2) the initial values are reproduced, namely, complex numbers whose imaginary part is strictly zero within numerical errors. Spending time in computing an already known result equal to 0 is wasteful, and for real-valued periodic functions the Fourier transform computation can be arranged to avoid it. This is discussed in detail in Ref [16]. We will keep in what follows the complex version of the Fourier transform described above because the numerical routines are simpler. Redundant computations will be avoided by using the complex inverse Fourier transform Equation (15.2) *to recover two real functions in a unique computation*, as was discussed in the text.