CHAPTER

14 FURTHER DATA PARALLEL EXAMPLES

14.1 INTRODUCTION

This chapter focuses on *stencil codes*, that is, parallel applications in which a map operation is performed on data values defined on a multidimensional grid. Data values updates are in general independent, and they can be performed in parallel, but they are updated in a way that depends on the near-neighbor values. A typical use case arises in image-filtering algorithms: pixel values defined on a two-dimensional grid are smoothed out by taking into account the neighbor pixel values. The set of near-neighbor data values that participate in the pixel update is called a stencil, and the whole image update process is a map operation in which each pixel is convoluted with its stencil. To the extent that pixel updates are totally independent, there is a huge amount of potential parallelism in this context.

Two examples concerning the solution of two-dimensional partial differential equations are developed. The first one is a straightforward map parallel pattern that can immediately benefit from multithreading and vectorization. The second example involves *data dependencies* that prevent a straightforward work-sharing pattern, and the way this issue can be handled is discussed in detail.

14.1.1 EXAMPLES IN THIS CHAPTER

The two examples proposed in this chapter involve solving the Laplace or Poisson equation in two dimensions. In the first case, the purpose is to determine the stationary temperature distribution $T(x, y)$ inside a rectangular metallic plate when the borders of the plate are maintained at some fixed temperature values. Under these conditions, the stationary temperature distribution is a solution of Laplace's equation:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

and the problem is to find the solution inside the plate with the given boundary conditions on the border.

The second example involves a two-dimensional rectangular domain bounded by metallic walls. Inside this rectangular domain, there is an electric charge with density $\rho(x, y)$ mainly concentrated in the center of the domain, and vanishing at the borders. The problem to be solved is the computation of the electric potential $U(x, y)$ inside the domain, which is a solution of Poisson's equation:

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00014-9

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = \rho(x, y)$$

with the boundary condition $U = 0$ on the four borders. Indeed, the electric potential must be constant on the conducting walls, and without loss of generality this constant can be set equal to zero.

14.1.2 FINITE-DIFFERENCE DISCRETIZATION

To obtain the numerical solutions of the above equations, the *finite-difference method* is adopted. 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:

$$x_n = x_0 + n\Delta, \quad n = 0, \ldots, (N-1)$$
$$y_n = y_0 + m\Delta, \quad m = 0, \ldots, (M-1)$$

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 14.1, the whole rectangular domain, *including the borders*, is covered when the (m, n) indices swap the domains indicated in the equation above.

From now on, we will write $U_{m,n}$ for $U(x_n, y_m)$ and $\rho_{m,n}$ for $\rho(x_n, y_m)$. Now, $U_{m,n}$ is just a two-dimensional matrix with m as row index and n as column index, as shown in Figure 14.1. Using the standard finite-difference representation for the second derivative, the Poisson equation becomes

$$\frac{U_{m+1,n} + U_{m-1,n} - 2U_{m,n}}{\Delta^2} + \frac{U_{m,n+1} + U_{m,n-1} - 2U_{m,n}}{\Delta^2} = \rho_{m,n}$$

or equivalently

$$U_{m+1,n} + U_{m-1,n} + U_{m,n+1} + U_{m,n-1} - 4U_{m,n} = F_{m,n} \tag{14.1}$$

where $F_{m,n} = \Delta^2 \rho_{m,n}$. This is the discretized form of the original differential equations that will be used in what follows.

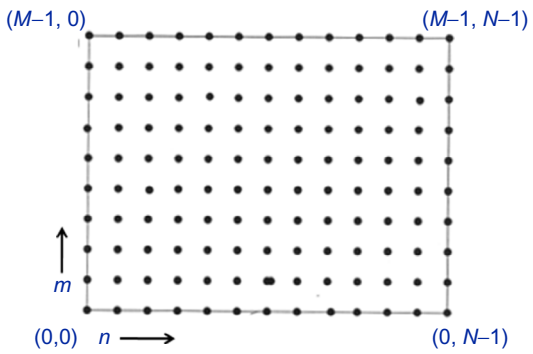

FIGURE 14.1

Discretization of the Laplace and Poisson problems.

14.2 RELAXATION METHODS

The different relaxation methods for solving Equation (14.1) start from an initial guess for $U_{m,n}$, a kind of approximate solution that fits the boundary condition. Then, an iterative procedure is applied, by successively updating all the values of this matrix, until convergence is achieved. The iteration procedure adopted by the different relaxation methods to be used next is easily described by rewriting Equation (14.1) as:

$$U_{m,n} = 0.25(U_{m+1,n} + U_{m-1,n} + U_{m,n+1} + U_{m,n-1}) - 0.25F_{m,n} \tag{14.2}$$

The different iteration procedures swap all the grid points—except the boundaries where the $U_{m,n}$ values are fixed from the start—with a double loop, first over rows (loop on m) and next an inner loop on columns (loop on n), and replace the old value of $U_{m,n}$ by a new value given by the right-hand side of Equation (14.2). For the homogeneous Laplace equation, the source term is set to $F_{m,n} = 0$. The numerical solution we are searching for is a fixed point of this iteration procedure: old values are equal to new values. There are three iteration procedures to be considered:

- *Jacobi method*: The update Equation (14.1) uses *old values* in all sites during the grid swap. New, updated values of $U_{m,n}$ are copied to another matrix, but they are not used as they become available in the right-hand side of Equation (14.1). This means the updates of different grid points are totally independent and can be performed in parallel. This method can be easily parallelized, but its convergence is poor. The number of iterations needed is proportional to the problem size NM.
- *Gauss-Seidel method*: In this case, the new, updated values of $U_{m,n}$ *are used in the right-hand side of Equation (14.1) as soon as they become available*. Updated values are not copied to another matrix, because in this case $U_{m,n}$ is updated *in place*. However, there are now data dependencies because the update of a given grid site depends on the values of the previous updates, and parallel treatment is not straightforward. The convergence is better than in the Jacobi case, but still the number of iterations required for convergence is proportional to the problem size NM.
- *Over-relaxed Gauss-Seidel method*: Like Gauss-Seidel, except that the average of the four near-neighbor values in the right-hand side of Equation (14.1) is multiplied by an *over-relaxation parameter* ω in the range $1 \leq \omega \leq 2$. For $\omega = 1$, Gauss-Seidel is recovered. The case $\omega > 1$ corresponds to an overcorrection of the old result that accelerates the convergence. Indeed, for some optimal values of ω the number of iteration only grows like the square root of the problem size, $\sqrt{NM}$.

A good discussion of these relaxation methods applied to the problems discussed here can be found in Chapter 19 of *Numerical recipes in C* [16], where convergence issues are analyzed in detail. In the next examples, the grid swaps will be performed as follows. First, the difference $E_{m,n}$ between the old and new values of $U_{m,n}$ is computed at each site:

$$E_{m,n} = 4U_{m,n} - (U_{m+1,n} + U_{m-1,n} + U_{m,n+1} + U_{m,n-1}) - F_{m,n}$$

This quantity is a measure of the local error of the current approximate solution. Absolute values of $E_{m,n}$ are accumulated in a global variable E, which, at the end of the swap, measures the global error of the current approximate solution. The new values of $U_{m,n}$ are:

$$U_{m,n}^{\text{new}} = U_{m,n} + 0.25\,\omega\,E_{m,n}$$

14.3 FIRST EXAMPLE: HEAT.C

The stationary temperature distribution in a rectangular plate, shown in Figure 14.1, is computed. The borders are maintained at a fixed temperature, as follows:

- The front border ($n = 0$) is a temperature $T = 1$.
- The back border ($n = N - 1$) is a temperature $T = 0$.
- On the upper and lower borders, there is a linear decrease of temperature from 1 to 0.

The obvious solution to this problem is a homogeneous temperature distribution in the vertical direction, with the same linear decrease in the horizontal direction imposed on the upper and lower rows in all the other rows of the two-dimensional domain:

$$T_{m,n} = -\frac{n}{(N-1)} + 1 \tag{14.3}$$

An approximate initial solution is adopted in this example, equal to the exact solution given above, plus a random value in each one of the inner grid points. Then, the Jacobi relaxation method is used to watch how this initial configuration relaxes to the correct stationary temperature distribution (Equation 14.3).

14.3.1 AUXILIARY FUNCTIONS IN HeatAux.C

As was done in the molecular dynamics example, a number of our auxiliary functions not directly related to multithreading, needed for pre and postprocessing, are factored out in the HeatAux.C file that must be linked with the different versions of the code discussed below. It is not necessary, for our purposes, to understand in detail how they operate. They are commented on in the code sources. These auxiliary functions are:

- void InputData(). Reads from an input file heat.dat the global variables needed to define the problem: the problem sizes N and M, and the values of three integer parameters: maxIts, the maximum number of iterations tolerated, nThreads, the number of threads (ignored by the sequential code), and stepReport, the number of iteration steps between two control messages sent to stdout.
- void InitJob(M, N). Allocates the two matrices U and V needed for the computation, following the procedure described in Section 13.8. Two matrices are needed because the matrix U is not updated *in place*: the result of its update is stored in matrix V. This function also sets the initial values of the temperature distribution $T_{m,n}$ in the way discussed above, and initializes the initial global error.
- PrintResult(). Prints the temperature distribution along x at mid-height to check that the correct solution has been obtained.

14.3.2 SEQUENTIAL HEAT.C CODE

The listing below shows the complete sequential code in source file Heat.C.

```
// Global data
/ - - - - - -
double **U, **V;
```

```
const double EPS = 1.0e-5;
double initial_error, curr_error;
int nIter;

// Data read from file "heat.dat"
// ----------------
int N, M;
int maxIts, nThreads, stepReport;

// Auxiliary function: prepares next iteration, and decides
// if it must take place
// ------------------------------------------
bool NextIteration(double error)
   {
   double **swap = U;      // swap U and V matrices
   U = V;
   V = swap;

   nIter++;                // increase counter, and print message
   if(nIter%stepCount==0)
      cout << "\n Iteration " << nIter << endl;
   // Decide if we keep iterating
   // ----------------
   if( (error > EPS*initial_error) &&
             (nIter <= maxIts) ) return true;
   else return false;
   }


int main(int argc, char **argv)
   {
   int m, n;
   double global_error, error;
   bool isnext;

   // Initialize the run
   // ---------
   InputData();
   InitJob(M, N);
   TimeReport TR;

   TR.StartTiming();
   do
      {
      global_error = 0.0;
      // Stencil operation for the update of the solution
```

Continued

```
    // – – – – – – – – – – – – – – – – – – – – – – – –
    for(m=1; m<(M–1); m++)
       {
       for(n=1; n<(N–1); n++)
          {
          error = U[m+1][n] + U[m–1][n] + U[m][n–1]
                  + U[m][n+1] – 4 * U[m][n];
          global_error += fabs(error);
          V[m][n] = U[m][n] + 0.25 * error;
          }
       }
    moreCycles = NextIteration(global_error);
    }while(more Cycles)
 TR.StopTiming();

 // Output results, and exit job
 }
```

LISTING 14.1

Sequential Heat.C code

There are a number of observations to be made about this code:

- The initial error, stored in the global variable initial_error, is initialized by InitJob().
- The computation is driven by a do loop that swaps the inner grid points and keeps updating the solution as long as the current error is bigger than a small factor EPS times the initial error, and as long as the number of iterations does not exceed the maximal accepted value provided as input to the program.
- Since, in the Jacobi method, $U_{m,n}$ is not updated in place, the current solution after an iteration is V, not U. Therefore, to always have U as the current solution the matrices must be swapped. Because of the way the matrices have been allocated, *it is sufficient to swap the pointers* U and V.
- The local error at each site is accumulated in a variable global_error initialized to 0 for each new swap of the grid. Then, at the end of the swap, this global error is copied to a global variable curr_error. This is redundant in this case, but will be needed in the parallel versions of this code.
- An auxiliary function NextIteration(double error) is defined, which prepares the next iteration and returns a bool that informs if it must take place, or if the iterative process must be stopped. This function swaps the matrices, increases the iteration counter, and performs the convergence test. A similar function will be used in all the parallel versions of the code.

Example 1: Heat.C

To compile, run make heat. The input data is read from file heat.dat. With the default parameters provided in this file, convergence is achieved after 56000 iterations.

14.4 PARALLEL IMPLEMENTATION OF HEAT.C

The natural and efficient way of distributing work among threads in this problem is to assign to each thread a set of contiguous matrix rows to update. It was shown in Section 13.7 in the previous chapter that the matrix rows are successively allocated in a large one-dimensional array of size NM. A subset of successive rows is therefore a compact one-dimensional vector array of contiguous elements in memory, on which a thread can efficiently operate. Once the external loop over rows has been distributed across threads, the inner loop over columns executed by each thread is an excellent candidate for vectorization. This standard approach—parallelize first, vectorize next—is natural because, as discussed in Chapter 2, vectorization is an internal, single-core affair. This procedure is adopted in all the parallel versions of the code.

As in the example in the previous chapter, there are several parallel versions of Heat.C:

- *HeatF.C*: OpenMP microtasking version using parallel for, with the collapse(2) clause to enforce loop fusion, possible in this case.
- *HeatOmp.C*: OpenMP macrotasking version, where the work distribution across tasks is done by hand.
- *HeatNP.C*: vath macrotasking version using the NPool thread pool.
- *HeatTbb.C*: TBB mictotasking version using the parallel_reduce algorithm.
- *HeatOmpV.C*: Previous HeatOmpV.C version, with a vectorized inner loop.

14.4.1 OPENMP MICROTASKING: HeatF.C

A first OpenMP parallel version of this code can be constructed simply by inserting the parallel for directive *before the outer loop on m* in the way indicated in the listing below.

```
...
global_error = 0.0;
#pragma omp parallel for collapse(2) reduction(+:global_error)
for(m=1; m<M-1; m++)
   {
   for(n=1; n<(N-1); n++)
      {
      error = U[m+1][n] + U[m-1][n] + U[m][n-1]
              + U[m][n+1] - 4 * U[m][n];
      global_error += fabs(error);
      V[m][n] = U[m][n] + 0.25 * error;
      }
   }
...
```

LISTING 14.2

Adding a directive in Heat.C.

The reduction clause is needed to accumulate the errors computed by each worker thread. When the outer for loop exits, global_error contains the full error of the Jacobi iteration. Besides the inclusion of

omp.h, *the parallel for directive is the only modification introduced in the sequential version Heat.C*. The collapse(2) clause enforcing loop fusion is critical. If it is omitted, parallel performance is very poor, as we will discuss in the next section.

Example 2: HeatF.C

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

14.4.2 OPENMP MACROTASKING: HeatOmp.C

Let us look at the synchronization patterns that must be implemented in a macrotasking version. The listing below shows the task function that is called inside an OpenMP parallel region.

```
// Task function for OpenMP
// – – – – – – – – – – – –
void TaskFunction()
   {
   int m, n;
   int beg, end;
   double global_error, error;

   // First, fix the thread loop range
   // – – – – – – – – – – – – – – – –
   beg = 1;
   end = M–1;
   ThreadRangeOmp(beg, end);   // [beg, end) becomes thread range
   do
      {
      global_error = 0.0;
      for(m=beg; m<end; m++)  // thread loop range
            {
            for(n=1; n<(N–1); n++)
               {
               error = U[m+1][n] + U[m–1][n] + U[m][n–1] + U[m][n+1]
                       – 4 * U[m][n];
               global_error += fabs(error);
               V[m][n] = U[m][n] + 0.25 * error;
               }
            }
      #pragma omp critical
        { error_cumul += global_error; }

      #pragma omp barrier
      #pragma omp master
         { moreCycles = NextIteration(error_cumul); }
      #pragma omp barrier
```

```
      }while(moreCycles);
   }
```

LISTING 14.3

OpenMP task function.

This task function operates as follows:

- After determining its target subdomain (the external loop subrange), each thread enters the do loop that performs the successive updates of the matrix U.
- After the update, the thread error is accumulated in the global variable error_cumul, protected by a critical section.
- The code enters next a sequential region executed by the master thread. A barrier is needed to make sure all threads have completed the update before the sequential region is executed.
- In the sequential region, the possible next iteration is prepared. The function that returns the Boolean variable that decides if a new iteration is needed is the same as the one used in the sequential code except that, in addition, it re-initializes the global curr_error variable to 0, so that this variable can be used again at the next iteration—if any—to accumulate new partial error value results.
- Finally, all threads use the shared value moreCycles to decide if iterations must be pursued. Notice that, again, an explicit barrier is needed before the test, because there is no implicit barrier at the end of the master code block, and we need to make sure all threads will read the same value of moreCycles computed in the sequential region.
- The structure of the main() function is always the same: a parallel directive encapsulating a call to the task function listed above.

Example 3: HeatOmp.C

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

14.4.3 NPOOL MACROTASKING: HeatNP.C

The source file HeatNP.C contains the parallel version based on the NPool thread management utility. As discussed in Chapter 11, parallel job submission requires the definition of a class defining the parallel tasks. The listing below shows the global variables used in this code, as well as the task class.

```
NPool             *TP;     // global variables
Barrier           *B;
Reduction<double> R;
bool              moreCycles;
...
class HeatTask : public Task
   {
   private:
    int rank;
```

Continued

```
public:
 HeatTask(int R) : rank(R) {}

 void ExecuteTask()
    {
    // - - - - - - - - - - - - - - - - - - - - - -
    // Determine the range of rows allocated to
    // this task, from the number of threads and
    // the rank of this task
    // - - - - - - - - - - - - - - - - - - - - - -
    do
       {
       double error;
       // - - - - - - - - - - - - - - - - - - - - - -
       // Update matrix rows allocated to this task
       // rank, and compute the thread error value
       // - - - - - - - - - - - - - - - - - - - - - -
       R.Accumulate(error);
       B->Wait();
       if(rank==1)
          {
          moreCycles = NextIteration(R.Data());
          R.Reset();
          }
       B->Wait();
       }while(moreCycles);
    }
 };
```

LISTING 14.4

NPool task class.

Parallel task instances of this class receive as argument an integer rank. The global references to NPool and Barrier objects are initialized by the main function once the number of threads is known. The Reduction<double> object is used to accumulate partial errors. In the listing above, the task function follows exactly the same synchronization logic applied in the OpenMP code.

The main function gets the required number of threads from the input data file, and initializes the thread pool and barrier objects. Then, it follows the standard protocol for job submission: creation of a TaskGroup object, creation of the parallel tasks and insertion in the TaskGroup, submission of the TaskGroup, and wait for the job termination.

Example 4: HeatNP.C

To compile, run make heatnp. The input data is read from file heat.dat. This is the NPool parallel version.

These two macrotasking parallel versions are very close to each other, and they end up having comparable performance after the Barrier class listed above is replaced by a more efficient implementation

of this synchronization utility, provided by the atomic-based ABarrier class. This interesting issue is discussed in more detail later on.

14.4.4 TBB MICROTASKING: HeatTBB.C

In the TBB environment, the parallel_reduce algorithm replaces the OpenMP parallel_for directive. As discussed in Chapter 11, two classes are needed: one class to describe the splittable integer range on which the algorithm operates—we use the blocked_range<int> class provided by the TBB library—and a "Body" class to describe the operation of the task function. This class is listed below.

```
// – – – – – – – – – – – – – – – – – – – – – – – –
// The "Body" class used for the parallel_reduce
// TBB algorithm.
// – – – – – – – – – – – – – – – – – – – – – – – –
bool moreCycles;
...
class DomainSwap
   {
   public:
   double error_norm;
   DomainSwap() : error_norm(0.0) {}
   DomainSwap(DomainSwap& dom, split toto) : error_norm(0.0) {}

   void operator() (const blocked_range<size_t>& rg)
      {
      for(size_t m=rg.begin(); m!=rg.end(); m++)
            {
            for(size_t n=1; n<(N–1); n++)
               {
               double resid = U[m+1][n] + U[m–1][n] + U[m][n–1]
                              + U[m][n+1] – 4 * U[m][n];
               error_norm += fabs(resid);
               V[m][n] = U[m][n] + 0.25 * resid;
               }
            }
      }

   void join(const DomainSwap& dom)
      { error_norm += dom.error_norm; }
   };

int main(int argc, char **argv)
   {
   // initialization
   ...
   task_scheduler_init init(nThreads);
```

Continued

```
tbb::tick_count t0 = tbb::tick_count::now();
do
   {
   DomainSwap DS;
   parallel_reduce(blocked_range<size_t>(1, M-1, M/nThreads), DS);
   moreCycles = NextIteration(DS.error_norm);
   }while( moreCycles );
tbb::tick_count t1 = tbb::tick_count::now();
// Output results and exit
}
```

LISTING 14.5

TBB Body class.

The constructor of the TBB blocked_range class representing a splittable integer range requires a granularity argument that controls how deep the range splitting is carried out. The ideal situation is our case is to end up having as many final subranges as threads, and for this reason the granularity in the listing above has been chosen as M/nThreads. However, the number of final subranges is in all cases a power of 2, and some mild performance issues arise when the number of threads is not a power of 2. We will come back to this point in the next section.

Example 5: HeatTBB.C

To compile, run make heatbb. The input data is read from file heat.dat. This is the TBB parallel version.

14.5 HEAT PERFORMANCE ISSUES

Next, the performance of the different parallel implementations of the code are compared. However, the same word of caution given in the previous chapter applies here. Performance is strongly dependent on the code profiles and no general conclusion on the overall relative merits of the different programming environments should be formulated. The OpenMP and TBB task schedulers have lots of sophisticated features that are not tested in these relatively simple data parallel examples.

The first issue we have to face is the very poor scaling behavior of the NPool version of the code. This can easily be seen in a quick run performed in a four-core laptop, on a 400×400 matrix, comparing the macrotasking OpenMP and NPool versions. The results are shown in Table 14.1.

While the OpenMP scaling behavior is quite good, the NPool code heatnp has very poor scaling behavior for two threads, and it is imperative to understand why. The first suspicion is some unacceptable overhead coming from the NPool environment itself, but this is discarded when the same poor scaling is observed in a pure Pthreads version of the code, not depending on the NPool thread pool environment, and using in addition the native Pthreads barrier utility. At this point, we are led to explore alternative barrier algorithms, which end up with the adoption of the more efficient ABarrier atomic-based barrier utility discussed in Chapter 9, implementing a barrier algorithm taken from Chapter 8 in [25]. In practice, the equivalent TBarrier class is used, since Pthreads does not have an native atomic utility. Keeping the NPool environment, and just changing the standard barrier by the TBarrier, the results displayed in the heatnpa column are obtained, restoring an acceptable scaling behavior and matching the OpenMP performance.

Table 14.1 Wall Execution Times in Seconds, for M = 400 Rows and N = 400 Columns, With GNU 4.8 Linux Compiler

Heat Barrier Comparison			
Threads	**heatomp**	**heatnp**	**heatnpa**
1	60	64	57
2	29.7	40	28.4
4	14.7	29	15

Notes: The heatnp *code employs the standard* vath *barrier utility and* heatnp_1 *employs the TBB barrier.*

Table 14.2 Wall Execution Times in Seconds, for M=320 Rows and N=1200 Columns, With Intel C-C++ Compiler, Version 13.0.1

Heat Performance						
Threads	**Heat**	**npool**	**omp**	**tbb**	**omp_f**	**omp_v**
1	373					233
4		91.4	91.9	121.8	100.6	57.9
8		43.7	47.9	72	53.6	24.4
12		30.4	34.7	73	40.3	18.6
16		21.8	26.9	98	35.2	14.7

Note: The npool *version uses the TBB barrier.*

Incidentally, this is an example of the importance of keeping an eye on options provided by the basic libraries. They can, in some cases, provide excellent optimization improvements, as is the case here.

More significant performance tests can be performed on the same platform discussed in the previous chapter, an SMP node incorporating two Sandy Bridge sockets with eight cores each. The performance of the different implementations is given in Table 14.2.

A few comments are in order:

- A satisfactory scaling of the macrotasking codes, OpenMP and NPool with the TBB barrier, is observed.
- The microtasking versions (tbb and omp_f show slightly lower performance). The equality of the TBB wall times for 8 and 12 threads is due to the reasons discussed before: there are not 12 subranges that match the 12 TBB threads.
- The last column is the performance of the vectorized omp version. The only difference is just adding the Intel compiler #pragma simd directive before the inner loops. A performance increase close to 2 is obtained.

14.6 SECOND EXAMPLE: SOR.C

The next example deals with a two-dimensional rectangular domain bounded by metallic walls, in the presence of an internal electric charge density $\rho(x, y)$ mainly concentrated at the center of the domain, and vanishing at the borders. Our purpose is to solve Poisson's equation for computing the electric potential, using the Gauss-Seidel successive over-relaxation method. The interest of this example is the adoption of a method of solution having a much faster convergence rate, but at the price of introducing data dependencies that require more sophisticated parallelization strategies.

In the over-relaxed Gauss-Seidel method, updated values of the matrix U are used as soon as they become available, and in addition we have to use a value of the over-relaxation parameter ω in the range $0 \leq \omega \leq 2$. The implications of the immediate use of the updated values of U are as follows:

- As successive rows are swapped in the two-dimensional grid, the matrix U must be updated *in place*.
- The straightforward parallelism of the previous example is lost. The new value of $U_{m,n}$ depends on the updated values of $U_{m,n-1}$—the previous neighbor in the same row—and $U_{m-1,n}$—the nearest neighbor in the previous row. These data dependencies imply that:
  - A thread updating successive matrix elements in a row needs the updated values of the previous row. Row updates are no longer independent, and the thread work-sharing strategy used in the previous example does not apply here.
  - A thread updating a matrix elements in a row needs also the updated value of the predecessor in the same row. This kills the possibility of vectorizing the inner loops.

Data dependencies are very common in stencil codes, and there is a general methodology to cope with them, very well described in Chapter 7 of [35], entirely devoted to the optimization of stencil codes. One way to handle data dependencies is to find a plane—a line in our two-dimensional case—that cuts through the grid of intermediate results so that all references to previously computed values are on one side of this plane. In our case, this separating hyperplane would be descending diagonals on the two-dimensional grid. Then, operations on this separating hyperplane can be performed in parallel. We will say more on this approach at the end of the chapter.

An alternative way of handling this data dependency problem uses control parallelism instead of data parallelism by pipelining threads. This approach is discussed in detail in the next chapter. In the rest of this chapter, two other possible ways of handling the data dependencies in a data parallel approach are explored: the Gauss-Seidel method with white-black ordering—which is not generic and only works in our particular geometry— and the diagonal swapping of the grid, which partly reduces the data dependencies and enables a limited amount of potential parallelism.

14.6.1 GAUSS-SEIDEL WITH WHITE-BLACK ORDERING

One possible way of handling our data dependency problem is to consider the original rectangular two-dimensional grid of lattice spacing Δ, as two interpenetrating grids of lattice spacing 2Δ. The original grid points are labeled with a color attribute (the white or black grid points) to distinguish the two grids. This is shown in Figure 14.2.

It can be immediately observed that the updated U value on a white site depends only on the updated values of U at the neighboring black sites, and vice versa. In updating white (black) sites, only U values

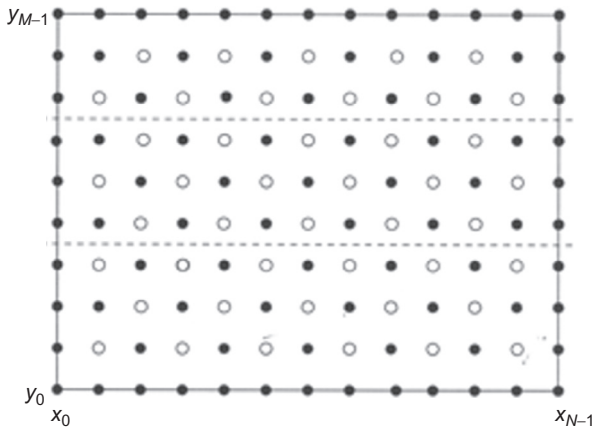

FIGURE 14.2

Gauss-Seidel with white-black ordering.

at black (white) sites are needed. Therefore, data dependencies are avoided if, starting with an initial configuration, with an approximate solution, white (or black) sites are updated first, and black (or white) sites are updated next. The matrix U can be then updated in place, and the updates of the white or black sites show the same amount of potential parallelism that was available in the previous example.

14.6.2 SEQUENTIAL CODE Sor.C

The sequential code for this example is similar to Heat.C, with a few differences:

- We are now solving the inhomogeneous Poisson equation. In updating $U_{m,n}$, the contribution of the source term $F_{m,n}$ must be included.
- The initial double loop that swaps the grid sites has to be changed to two successive double loops swapping the white and black grid sites.
- The over-relaxed Gauss-Seidel method is adopted, by introducing the over-relaxation parameter ω in the $U_{m,n}$ update, whose value is discussed below.

The organization of the code is the same as in the previous example. The listing below shows the part of the main() function that has been modified to include the two passes over white and black grid sited:

```
int main(int argc, char **argv)
   {
   int m, n;
   int nSh, mSh, pass;
   double error;
   ...
```

Continued

```
do
   {
   error = 0.0;
   for(pass=1; pass<=2; pass++)     // passes loop
      {
      mSh = pass/2;
      for(m=1; m<M-1; m++)          // rows loop
         {
         nSh = (m+mSh)%2;
         for(n=nSh+1; n<N-1; n+=2)  // inner loop
            {
            error = U[m+1][n] + U[m-1][n] + U[m][n-1]
                    + U[m][n+1] - 4 * U[m][n];
            error += fabs(error);
            U[m][n] = U[m][n] + 0.25 * w * error;
            }
         }
      }

   nIter++;
   if(nIter%1000==0)
      cout << "\n Iteration " << nIter << endl;
   }while( (error > EPS*initial_error) &&
           (nIter <= maxIts) );
...
}
```

LISTING 14.6

Gauss-Seidel with white-black ordering.

The external loop over the two passes is not entirely obvious, and it took me some thinking to get it right. The problem is in principle very simple. For each pass, the external loop over rows m is executed in the standard way, and only the inner loop over n is modified. This inner loop has stride two and a starting offset that toggles between 0 and 1 as the index m moves from one row to the next one:

- For the first pass, the n initial offsets are 0, 1, 0, 1, ... as m runs over 1, 2, 3, ...
- For the second pass, the n initial offsets are 1, 0, 1, 0, ... as m runs over 1, 2, 3, ...

There is still, however, a mild data dependency among rows, because the offsets for each pass are determined by the starting value of the offset for the first row $m = 1$, and this is not known to threads other than the one in charge of the first subdomain. A way of determining the n loop offsets in an *absolute* way is needed, with no reference to the first row offset. This problem is solved by introducing the auxiliary variables mSh and nSh in the way indicated in the listing above. If the loop over m is restricted to a subdomain of consecutive rows, the two passes operate correctly and they are in fact swapping first the white grid and next the black grid in each subdomain.

Values of ω: What about the over-relaxation parameter ω? It can be shown—Chapter 19 in [16]—that the method is convergent only for $0 < \omega < 2$, and that, under certain mathematical restrictions

satisfied by matrices arising from finite-difference discretizations, only the over-relaxation range ($1 < \omega < 2$) gives better convergence than the Gauss-Seidel method, which corresponds to $\omega = 1$. The question is: how better can the convergence be? We know that the number of iterations in Gauss-Seidel grows like the problem size NM. It can be shown that the over-relaxation method can converge after a number of iterations proportional to N, but only in a fairly narrow window of ω values around an optimal value. This optimal value can only be computed in some simple cases [16]; otherwise, it must be obtained empirically.

It turns out that, for the rectangular grid used in this problem, if $M \simeq N$, this optimal value is:

$$\omega \simeq \frac{2}{1 + \pi/N}$$

and this is therefore the value we use in the codes discussed here.

Example 6: Sor.C

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

14.7 PARALLEL IMPLEMENTATIONS OF Sor.C

We very quickly show next the different parallel implementations. Apart from the presence of the two passes, the code organization is the same as in the Heat example.

14.7.1 OPENMP MICROTASKING: SorF.C

A first, quick OpenMP parallel version can be obtained by inserting the parallel for directive in each pass, just before the outer loop over rows. In this case, the compiler rejects the collapse(2) clause. Indeed, because of the nSh=(m+mSh)/2 statement between the outer and the inner loop, the two loops are not perfectly nested. Just inserting the parallel for directive produces correct parallel code, but with a very poor performance (an example will follow).

Trying to help the compiler to produce better code, the external loop over two passes was eliminated, and the two double loops for each pass were explicitly written. In this case, the performance is acceptable (again, an example will follow).

Example 7: SorF.C

To compile, run make sorf. Parallel version using parallel for. The input data is read from file sor.dat.

14.7.2 OpenMP AND NPool MACROTASKING

The macrotasking versions SorOmp.C and SorNP.C are, again, very close to the HeatOmp.C and HeatNP.C codes. The synchronization patterns are very close. The only difference to be underlined is the presence

of an additional barrier, *because a barrier is required after each pass*. Indeed, when a thread is updating the boundary rows of its subdomain, it needs to use opposite color data coming from the neighboring subdomains, updated by other threads in a previous pass. Therefore, this additional barrier is needed to guarantee that all threads have finished the white updates before the black updates start, and vice versa.

Examples 8 and 9: SorOmp.C and SorNP.C

To compile, run make soromp or make sornp. The input data is read from file sor.dat.

14.7.3 TBB MICROTASKING: SorTBB

Again, the parallel_reduce algorithm is used for the matrix update. The Body class—called DomainSwap—is very similar to the one of the previous example. The only difference is that an additional private data item pass is added, corresponding to the pass value, passed as argument in the constructor. It is therefore clear that a different DomainSwap object must be allocated for each pass.

The listing below shows how these objects are used in the main() function.

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

   InputData();
   InitJob(M, N);

   if(argc==2) nTh = atoi(argv[1]);  // initialization
   task_scheduler_init init(nTh);
   G = M/nTh;                        // granularity
   omega = 1.8;

   tbb::tick_count t0 = tbb::tick_count::now();
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   do
      {
      DomainSwap DS1(1);          // pass 1
      parallel_reduce(blocked_range<size_t>(1, M-1, G), DS1);
      anorm = DS1.error_norm;
      DomainSwap DS2(2);          // pass 2
      parallel_reduce(blocked_range<size_t>(1, M-1, G), DS2);
      anorm += DS2.error_norm;

      nIter++;
      if(nIter%stepReport==0) printf("\n Iteration %d done", nIter);
      }while( (anorm > EPS*anormf) && (nIter <= maxIts) );
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
tbb::tick_count t1 = tbb::tick_count::now();
// print results
}
```

LISTING 14.7

Partial listing of SorTBB

After each pass, the partial error is stored, as in the Heat example, in the error_norm field of the DomainSwap objects. They are read and accumulated in anorm, before the convergence test is performed.

Example 10: SorTBB.C

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

14.8 SOR PERFORMANCE ISSUES

As in the previous example, a quick look at the code performance can be taken on a four-core Linux laptop. It should kept in mind that, because of the two passes, there are in this example more barriers than in the Heat code case. They are explicit (or implicit) in the macrotasking (or microtasking) versions. Table 14.3 shows the wall execution times for a 500 × 500 grid.

Scaling behaviors are satisfactory, except for the OpenMP microtasking sorf code and, again, the NPool code with the standard Pthreads-based barriers. This last issue is corrected as was done before, by using the ABarrier barrier class (last column in Table 14.3). The sorf issue has to do with the fact mentioned before, namely, inserting the parallel for directive inside the pass loop. The compiler does a better job if the pass loop is unrolled, as we will see below.

Next, performance was examined for a 800 × 800 configuration in a Linux SMP node incorporating two Sandy Bridge sockets with eight cores each. Wall execution times are listed in Table 14.4. The omp_f results are much better now, they correspond to unrolled pass loops. The npool scaling behavior—

Table 14.3 Wall Execution Times in Seconds, for M=500 Rows and N=500 Columns, With Linux GNU Compiler 4.8

Sor Barrier Comparison						
Threads	**sor**	**soromp**	**sorf**	**sortbb**	**sornp_1**	**sornp_2**
1	25.8					
2		11.6	30	13.5	11.3	11.7
4		6.2	45	7.1	5.9	11

Notes: The sornp_1 *code employs the TBB barrier, and* sornp_2 *employs the standard* vath *barrier utility.*

Table 14.4 Wall Execution Times in Seconds, for M=800 Rows and N=800 Columns, With Intel C-C++ Compiler, Version 13.0.1

Sor Performances							
Threads	**sor**	**omp**	**omp_f**	**tbb**	**np**	**np_tbb**	**omp_v**
1	280						
4		71.7	79.3	76	69.3	62.9	48.1
8		37.3	45.8	44.3	85.5	32.8	25.2
12		26.1	33.6	45.6	78.9	24.2	18.8
16		28.8	38.3	41	104	27.7	20.4

column np—is very poor, but, as before, this is beautifully corrected by using the ABarrier utility—column np_tbb. The general trend that seems to emerge from the examples in this chapter is that the macrotasking versions are slightly more efficient.

Finally, the last column omp_v corresponds to the SorOmpV.C code, which is simply SorOmp.C with the vectorization pragma simd directive added before the inner loops in each pass. We can observe that, in the region where the code scales correctly, roughly below 12 threads, vectorization provides an additional speedup close to 2. Vectorization seems to be somewhat less efficient here than in the case of the Heat code, where the speedup was close to 3.

An educated guess to explain the reduced vector performance is to blame stride 2 in the vectorizable loops. Indeed, consider the case of the Sandy Bridge socked with four SIMD lanes for doubles. Vector registers are directly loaded with consecutive vector elements from the L2 cache, but the compiler masks the two elements that do not participate in the vector operation, and acts on the other two. There are therefore only two active SIMD lanes, and consequently only two operations per cycle, instead of four.

Example 11: SorOmpV.C

To compile, run make sorompv. This is simply SorOmp.C with vectorization directives added before the inner loops. The input data is read from file sor.dat.

14.9 ALTERNATIVE APPROACH TO DATA DEPENDENCIES

The red-black Gauss-Seidel method discussed in the previous sections is an efficient way of restoring potential parallelism in the presence of data dependencies. It is, however, obvious that this method is not generic: it works with our specific geometry and our simple near-neighbor stencil. Before closing the discussion of this chapter, it is interesting to look at more generic ways of restoring some amount of parallelism in the presence of data dependencies.

If the site update involves data dependencies from neighbor sites within a limited neighbor range, a general result [37] states that it is always possible to find a hyperplane slicing the grid in such a way that all the dependencies are on one side of the hyperplane. Under these conditions, the sites on

the hyperplane can be updated in parallel. In the case of the Sor code, it is obvious that the slicing hyperplanes are the diagonals shown in Figure 14.3. Sites involved in the updating of diagonal sites, are all below the diagonal. If a lattice swap along successive diagonals is organized, sites on a diagonal can be updated in parallel.

Note that this approach is really generic, but the amount of parallelism generated is rather limited. On one hand, successive diagonals are not independent, and they have to be updated sequentially. On the other hand, this is really an irregular problem because the diagonal data set size changes from 1 to (N-2) and then back to 1 as the grid is swapped along the diagonals. Distributing a workload across several worker threads if the data set is too small is highly inefficient, as it only generates synchronization overhead.

At this point, we observe that the built-in domain decomposition strategy in the TBB parallel_reduce algorithm is very well suited for this context. A well-selected granularity can be used to control the number of parallel tasks generated by the algorithm, starting at 1 for the smaller diagonal sizes and growing to the number of worker threads for the largest diagonal sizes. This is probably the best possible parallel strategy for this generic approach.

Two versions of the Sor code based on the diagonal swap are available, generically called DSor:

- *DSor.C*: Sequential implementation of the diagonal swap. This code compares the diagonal swap to the direct, one-pass row swap of the grid.
- *DSorTbb.C*: Parallel version, using the TBB parallel_reduce algorithm as explained above.

Examples 12 and 13: DSor.C and DSorTbb.C

To compile, run make dsor//dsortbb. The input data is read from file sor.dat.

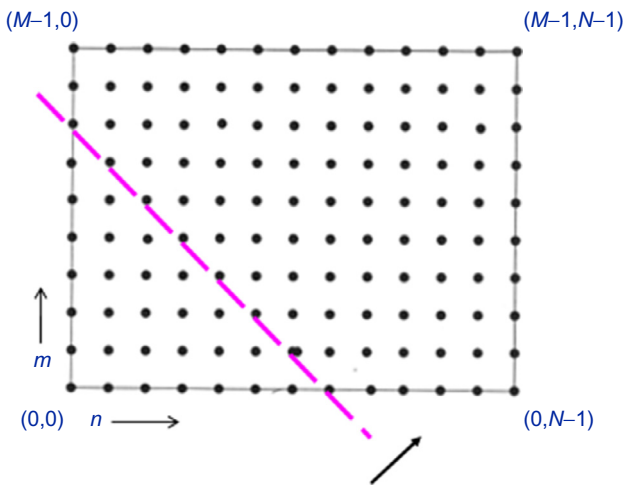

FIGURE 14.3

Diagonal swap to control data dependencies.

Running these examples, it is observed that they operate correctly, but that performance is not as good as in other cases. There are two reasons for that. First of all, it is clear that the amount of real parallelism is very limited: only the swapping of large diagonals provides significant parallel processing. The other point is that diagonal swapping systematically accesses nonconsecutive memory addresses, and that the locality advantages of the L2 caches is not very efficient here.