study, we will repeatedly update the values of every interior grid point until the values converge.

13.4.2 Deriving the Sequential Program

We divide spatial dimension x into n pieces and spatial dimension y into m pieces. We define $h = x/n$ and $k = y/m$.

Using the approximation to the second derivative developed in Section 14.2, we determine that

$$u_{xx}(x_i, y_j) \approx \frac{u(x_i + h, y_j) - 2u(x_i, y_j) + u(x_i - h, y_j)}{h^2}$$

$$\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

Similarly,

$$u_{yy}(x_i, y_j) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}$$

If we insert these approximations into the Poisson equation, we get

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = f(x_i, y_j)$$

Assume $\lambda = k/h$. After a series of further approximations (detailed in Plybon [92]), we come up with a new formula for the value of the solution at each grid point

$$w_{i,j} = \frac{\lambda^2(w_{i+1,j} + w_{i-1,j}) + w_{i,j+1} + w_{i,j-1} - k^2 f_{i,j}}{2(1 + \lambda^2)}$$

Now let's look at the particular problem we want to solve. A thin steel plate is surrounded on three sides by a condensing steam bath (temperature 100 degrees Celsius). The fourth side touches an ice bath (temperature 0 degrees Celsius). An insulating blanket covers the top and the bottom of the plate. Our goal is to find the steady-state temperature distribution at evenly spaced points within the plate.

Since the points are evenly spaced, $h = k$ and $\lambda = 1$. Since the plate is insulated, no heat is being introduced inside the plate—only on the edges. That means $f_{i,j} = 0$. Hence our finite difference approximation to the solution of the linear second-order PDE reduces to

$$w_{i,j} = \frac{w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{i,j-1}}{4}$$

Starting with initial estimates for all the $w_{i,j}$ values, we can iteratively compute new estimates from previous estimates until the values converge. Relying on the estimates from iteration i to calculate new estimates for iteration $i+1$ is called the **Jacobi method** [56]. (Note that this is the same algorithm we first encountered in Chapter 12.) A C program implementing a solution to the steady-state heat problem appears in Figure 13.10.

```
/* Sequential Solution to Steady-State Heat Problem */

#include <math.h>
#define N       100
#define EPSILON 0.01

int main (int argc, char *argv[])
{
   double diff;       /* Change in value */
   int    i, j;
   double mean;       /* Average boundary value */
   double u[N][N];    /* Old values */
   double w[N][N];    /* New values */

   /* Set boundary values and compute mean boundary value */
   mean = 0.0;
   for (i = 0; i < N; i++) {
      u[i][0] = u[i][N-1] = u[0][i] = 100.0;
      u[N-1][i] = 0.0;
      mean += u[i][0] + u[i][N-1] + u[0][i] + u[N-1][i];
   }
   mean /= (4.0 * N);

   /* Initialize interior values */
   for (i = 1; i < N-1; i++)
      for (j = 1; j < N-1; j++) u[i][j] = mean;

   /* Compute steady-state solution */
   for (;;) {
      diff = 0.0;
      for (i = 1; i < N-1; i++)
         for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > diff)
               diff = fabs(w[i][j] - u[i][j]);
         }
      if (diff <= EPSILON) break;
      for (i = 1; i < N-1; i++)
         for (j = 1; j < N-1; j++) u[i][j] = w[i][j];
   }

   /* Print solution */
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++)
         printf ("%6.2f ", u[i][j]);
      putchar ('\n');
   }
}
```

Figure 13.10 C program solving the steady-state heat distribution problem using the Jacobi method.

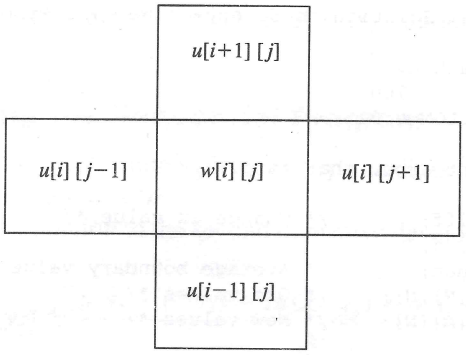

Figure 13.11 The value of `w[i][j]` depends upon the values of `u[i-1][j]`, `u[i][j+1]`, `u[i+1][j]`, and `u[i][j-1]`.

13.4.3 Parallel Program Design

We can associate a primitive task with the computation of each `w[i,j]`. In the Jacobi method the updating step is perfectly parallel. To compute `w[i,j]` each task requires the u values from its neighbors to the north, south, east, and west (Figure 13.11).

We want to agglomerate tasks and assign one agglomerated task to each parallel process. What is the best way to do the agglomeration? If each process is responsible for a rectangular region, then computing elements of `w` on the interior of the rectangle can be performed using locally available values of `u`. Computing elements of `w` on the edge of the rectangle requires values held by other processes.

We can introduce ghost points around each block of values held by a process. After values received from other processes have been stored in the ghost points, then a single doubly nested loop will allow all of the values of `w` to be computed.

One choice is a rowwise block-striped decomposition (Figure 13.12a). With this decomposition each interior process exchanges messages with two other processes. An obvious alternative is a checkerboard block decomposition. In this case each interior process exchanges messages with four other processes.

13.4.4 Isoefficiency Analysis

Suppose we are working with an $n \times n$ mesh. Since the computation time per mesh point is constant, the computational complexity of the sequential algorithm is $\Theta(n^2)$ per iteration.

Let's consider the rowwise block-striped decomposition. Each of the p processes manages a submesh of size approximately $(n/p) \times n$. During each iteration, every interior process must send n values to each of its neighbors and

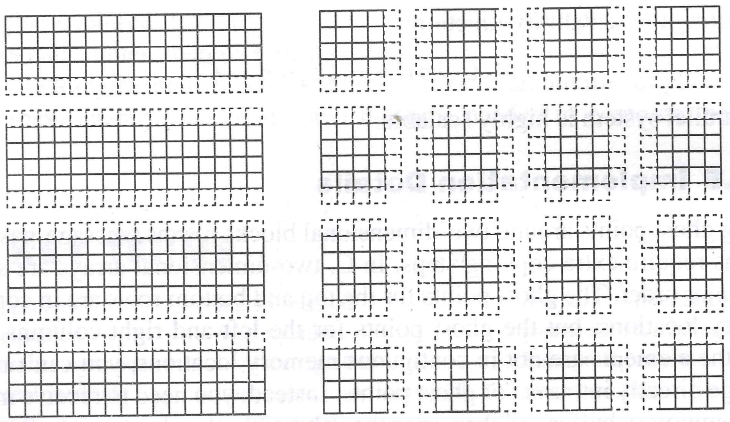

Figure 13.12 Possible data decompositions for solving the two-dimensional steady-state temperature distribution problem. (a) Illustration of a 16 × 16 mesh mapped onto four processes using a rowwise block-striped decomposition. Each process manages an $(n/p) \times n$ region. Ghost points appear as cells edged by dashed lines. (b) Illustration of a 16 × 16 mesh mapped onto 16 processes in a checkerboard block decomposition. Each process manages a region of size $(n/\sqrt{p}) \times (n/\sqrt{p})$. Ghost points are the cells edged by dashed lines.

receive n values from each of them, leading to a communication complexity of $\Theta(n)$. The communication overhead of each iteration of the parallel algorithm is $\Theta(np)$.

The isoefficiency function for the algorithm based on a rowwise block-striped decomposition is

$$n^2 \geq Cnp \Rightarrow n \geq Cp$$

Since $M(n) = n^2$, the scalability function is

$$M(Cp)/p = C^2p^2/p = C^2p$$

The parallel system is not highly scalable.

Now let's look at the checkerboard block-striped decomposition. Each of the p processes manages a submesh of size approximately $(n/\sqrt{p}) \times (n/\sqrt{p})$. During each iteration every interior process must send $n/\sqrt{p}$ values to each of its neighbors and receive $n/\sqrt{p}$ values from them, leading to a communication complexity of $\Theta(n/\sqrt{p})$. The communication overhead of each iteration of the parallel algorithm is $\Theta(n\sqrt{p})$.

The isoefficiency function for the algorithm based on a checkerboard block decomposition is

$$n^2 \geq Cn\sqrt{p} \Rightarrow n \geq C\sqrt{p}$$

Computing the scalability function:

$$M(C\sqrt{p})/p = C^2 p/p = C^2$$

The parallel system is highly scalable.

13.4.5 Implementation Details

Putting ghost points around two-dimensional blocks means message-passing operations require extra copying steps. In C, two-dimensional arrays are stored in row-major order. The ghost points for the top and bottom rows are in contiguous memory locations, but the ghost points for the left and right columns are not. Since the elements are not in contiguous memory locations, you can't receive a message directly into a set of ghost points. Instead, you need to receive messages into a temporary buffer and then copy the values into the ghost points. Similarly, a temporary buffer must be used when assembling values to be sent to a neighboring process's column-oriented ghost points.

13.5 SUMMARY

A partial differential equation is an equation containing derivatives of a function of two or more variables. Scientists and engineers use partial differential equations to model the behavior of a wide variety of physical systems. Realistic problems yield partial differential equations that are too complicated to solve analytically. Instead, scientists and engineers use computers to find approximate solutions to partial differential equations.

The two most common numerical techniques for solving partial differential equations are the finite element method and the finite difference method. Matrix-based implementations of the finite difference method represent the matrix explicitly, using data structures that support efficient access of the nonzero elements. Matrix-free implementations represent the matrix values implicitly. In this chapter we have designed and analyzed parallel programs based on matrix-free implementations of the finite difference method.

Linear second-order partial differential equations can be classified as elliptic, parabolic, and hyperbolic. Different algorithms are used to solve each of these types of PDE, but they do have some similarities. As our case studies, we looked at the solution of the wave equation (an example of a parabolic PDE) and the solution of the heat equation (an example of an elliptical PDE). Hyperbolic PDEs are typically solved by methods not as amenable to parallelization. For each case study, we used our standard parallel algorithm design methodology. We started by identifying primitive tasks and the communication pattern among them. We then chose an agglomeration that represented the best compromise between minimizing communication and maximizing utilization.

In both case studies we used ghost points to store values received from other processes. Once values have been received into the ghost points, all cells can be updated in the same section of code.