Chapter 2

Applications

2.1 The 2-D Poisson Problem

In this section we briefly describe how an approximate solution to a simple partial differential equation can be found when using parallel computing. This section will allow us to illustrate the issues of parallelizing an application and contrast the two major approaches.

2.1.1 The Mathematical Model

The Poisson problem is a simple, elliptic partial differential equation. The Poisson problem occurs in many physical problems, including fluid flow, electrostatics, and equilibrium heat flow. In two dimensions, the Poisson problem is given by the following equations:

$$
\begin{aligned}
\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} &= f(x,y) && (2.1)\\
& \text{in the interior} && \\
u(x,y) &= f(x,y) \text{ on the boundary} && (2.2)
\end{aligned}
$$

The approximation consists of defining a discrete mesh of point (x_i, y_j) on which we will approximate u. To keep things simple, we will assume that the mesh is uniformly spaced in both the x and y directions, and that the distance between adjancent mesh points is h. That is, $x_{i+1} - x_i = h$ and $y_{j+1} - y_j = h$. We can then use a simple centered-difference approximation to the derivatives in Equation 2.2 [?] to get

$$
\begin{aligned}
&\frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1,j})}{h^2} \quad + \\
&\frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_{i,j-1})}{h^2} \qquad = f(x_i, y_j) \qquad (2.3)
\end{aligned}
$$

at each point (x_i, y_j) of the mesh. To simplify rest of the discussion, we will replace $u(x_i, y_j)$ by $u_{i,j}$.

```
real u(0:n,0:n), unew(0:n,0:n), f(1:n, 1:n), h

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  do j=1, n-1
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                           u(i,j+1) + u(i,j-1) - &
                           h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
enddo
```

Figure 2.1: Sequential version of the Jacobi algorithm

2.1.2 A Simple Algorithm

Many numerical methods have been developed for approximating the solution of the partial differential equation in Equation 2.2 and for solving the approximation in Equation 2.3. In this section we will describe a very simple algorithm so that we can concentrate on the issues related to the parallel version of the algorithm. In practice, the algorithm we describe here should not be used. However, many of the more modern algorithms use the same approach to achieve parallelism.

The algorithm that we will use is called the *Jacobi Method*. This method is an interative approach for solving Equation 2.3 that can be written as

$$u_{i,j}^{k+1} = \frac{1}{4}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - h^2 f_{i,j}\right). \tag{2.4}$$

This equation defines the value of $u(x_i, y_j)$ at the $k+1$st step in terms of u at the kth step (it also ignores the boundary conditions).

We can translate this into a simple Fortran program by defining the array `u(0:n,0:n)` to hold u^k and `unew(0:n,0:n)` to hold u^{k+1}. This is shown in Figure 2.1; details of initialization and convergence testing have been left out.

In the next two sections we will look at two different approaches to making this a parallel program.

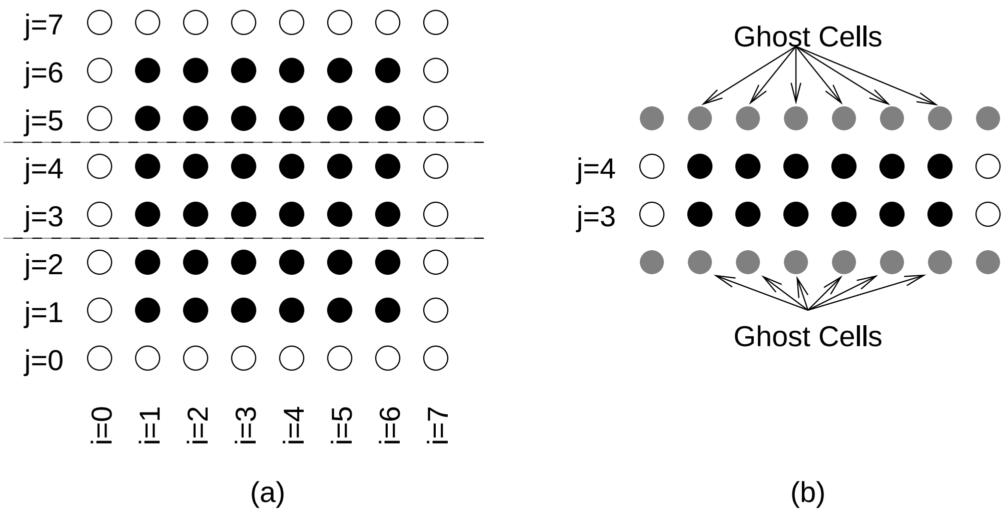

Figure 2.2: Simple decomposition of mesh across processors. Part (a) shows the entire mesh, divided among three processes. Open circles correspond to points on the boundary. Part (b) shows the part of this array owned by the second process; the grey circles represent the ghost or halo cells.

2.1.3 Message-Passing and the Distributed Memory Model

One of the two major classes of parallel programming models is the distrbuted memory model, as discussed in Section ??. In this model, a parallel program is made up of many processes, each of which has its own address space and (usually) variables. Because each process has its own address space, special steps must be taken to communication information between processes. One of the most widely used approaches is *message passing*. In message passing, information is communicated between processes by sending messages using a cooperative approach where both the sender and the receiver make subroutine calls to arrange for the transfer of data between them. Variables in one process are not directly accessible by any other process.

In creating a parallel program for this programming model, the first question to ask is: what data structures in my program must be *distributed* or *partitioned* among these processes? In our example, in order to achieve any parallelism, each process must do part of the computation of `unew`. This suggests that we should distribute `u`, `unew`, and `f`. One such partition is shown in Figure 2.2(a). The part of the distributed data structure the is held by a particular process is said to be *owned* by that process.

Note that the code to compute `unew(i,j)` requires `u(i,j+1)` and `u(i,j-1)`. This means that in addition to the part of `u` and `unew` that each process has (as part of the decomposition), it also needs a small amount of data from its neighboring processes. This data is usually copied into a slightly expanded array that holds both the part of the distributed array managed (or *owned*) by

```
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1), f(1:n, js:je), h
integer nbr_down, nbr_up, status(MPI_STATUS_SIZE), ierr

! Code to initialize f, u(0,*), u(n:*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  ! Send down
  call MPI_Sendrecv( u(1,js), n-1, MPI_REAL, nbr_down, k &
                     u(1,je+1), n-1, MPI_REAL, nbr_up, k, &
                     MPI_COMM_WORLD, status, ierr )
  ! Send up
  call MPI_Sendrecv( u(1,je), n-1, MPI_REAL), nbr_up, k+1, &
                     u(1,js-1), n-1, MPI_REAL, nbr_down, k+1, &
                     MPI_COMM_WORLD, status, ierr )
  do j=js, je
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                           u(i,j+1) + u(i,j-1) - &
                           h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
enddo
```

Figure 2.3: Message-passing version of Figure 2.1

a process with *ghost* or *halo* points that hold the values of these neighbors. This is shown in Figure 2.2(b). A process gets these values by communicating with its neighbors.

The code in Figure 2.3 shows the distributed memory, message-passing version of our original code in Figure 2.1.

The values of `js` and `je` are the values of `j` for the bottom and top of the part of `u` owned by a process. The routine `MPI_Sendrecv` is part of the MPI message-passing standard [?], and both sends and receives data. In this case, the first call sends the values `u(1:n-1,js)` to the process below or down, where it is received into `u(1:n-1,je+1)`.

Note that though each process has variables `js`, `je`, `u`, and so on, these are all *different* variables (precisely, they are different memory locations).

There are many other ways to describe the communication needed for this algorithm and algorithms like it. See [?, Chapter 4] for more details.

2.1.4 The Single Name-Space Distributed-Memory Model

Should this have the HPF version?

2.1.5 The Shared Memory Model

In the shared memory model, in contrast to the distributed memory model, there is only one process but multiple threads. All threads can access all[1] of the memory of the process. This means that there is only single version of each variable. This is very convenient; in some cases, a parallel, shared memory version of Figure 2.1 looks exactly the same: the compiler may be able to create a parallel version directly from the sequential code.

However, it can be helpful, both in terms of code clarity and generating efficient parallel code to include some code that describes the desired parallelism. One method that was designed for this kind of code is OpenMP [?]. The OpenMP version is shown in Figure 2.4.

See Section **??** for a more detailed discussion of OpenMP. A complete Open-MPI code for the Jacobi example is available at the OpenMP web site [?]

2.1.6 Comments

This section has deescribed very briefly the steps required when parallelizing code to approximate the solution of a partial differential equation. While the algorithm used in this discussion is inefficient by modern standards, the approach to parallelism is very similar to what is needed by state-of-the-art approaches for both implicit and explicit solution methods. Sections **??** and **??** in this book discuss more modern techniques.

Another discussion that focuses on some of the more subtle issues, particularly for the shared memory case is given in [?].

[1] Well, nearly all

```
    real u(0:n,0:n), unew(0:n,0:n), f(1:n, 1:n), h

    ! Code to initialize f, u(0,*), u(n:*), u(*,0),
    ! and u(*,n) with g

    h = 1.0 / n
    do k=1, maxiter
!$omp parallel
!$omp do
      do j=1, n-1
        do i=1, n-1
          unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                               u(i,j+1) + u(i,j-1) - &
                               h * h * f(i,j) )
        enddo
      enddo
!$omp enddo
      ! code to check for convergence of unew to u.

      ! Make the new value the old value for the next iteration
      u = unew
!$omp end parallel
    enddo
```

Figure 2.4: OpenMP (shared memory) version of the Jacobi algorithm **CHECK THIS EXAMPLE**