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:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \text{ in the interior} \tag{2.1}$$

$$u(x,y) = g(x,y) \text{ on the boundary} \tag{2.2}$$

To compute an approximation solution to this problem, we define a discrete mesh of points (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

$$\frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2} + \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h^2} = f(x_i, y_j) \tag{2.3}$$

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 iterative approach for solving Equation 2.3 that can be written as

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

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.

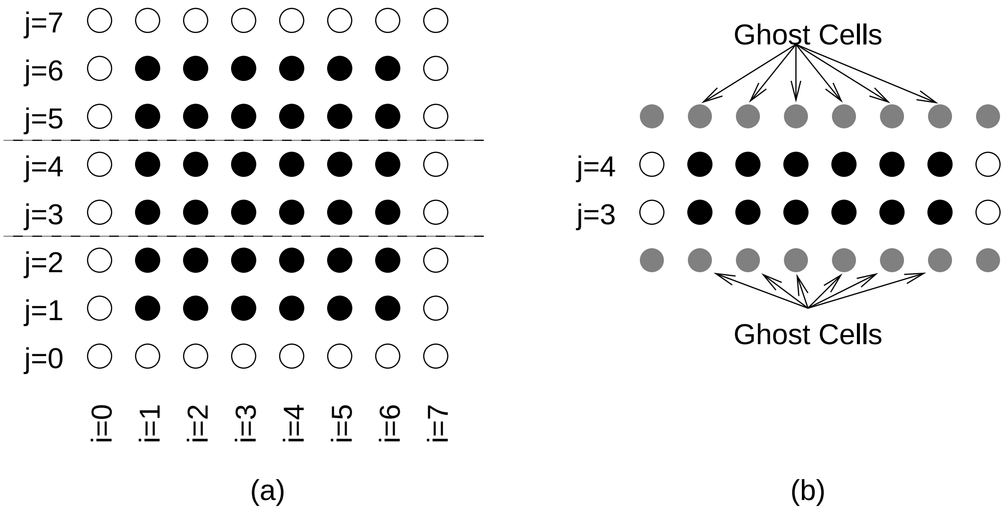

Figure 2.2: Simple decomposition of the mesh across processes. Part (a) shows the entire mesh, divided among three processes. Open circles correspond to points on the boundary. Part (b) shows the part of this array owned by the second process; the grey circles represent the ghost or halo cells.

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 that 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-1, 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

```
    real u(0:n,0:n), unew(0:n,0:n), f(0:n, 0:n), h
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew WITH u
!HPF$ ALIGN f WITH u

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

    h = 1.0 / n
    do k=1, maxiter
      unew(1:n-1,1:n-1) = 0.25 * ( u(2:n,1:n-1) + u(0:n-2,1:n-1) + &
                                   u(1:n-1,2:n) + u(1:n-1,0:n-2) - &
                                   h * h * f(1:n-1,1:n-1) )
      ! code to check for convergence of unew to u.

      ! Make the new value the old value for the next iteration
      u = unew
    enddo
```

Figure 2.4: HPF version of the Jacobi algorithm **CHECK THIS EXAMPLE**

algorithm and algorithms like it. See [?, Chapter 4] for more details.

2.1.4 The Single Name-Space Distributed-Memory Model

High Performance Fortran (HPF) [?] provides an extension of Fortran (Fortran 90) to distributed-memory parallel environments. Unlike the message-passing model, a single variable may be declared as distributed across all processes. For example, rather than declaring the part of the `u` variable owned by each process, in HPF, the program simply declares `u` in the same way as for the sequential program, and adds an HPF *directive* that describes how the variable should be distributed across the processes. All communication required to access neighbor values is handled for the programmer by the HPF compiler. The HPF version of the Jacobi iteration is shown in Figure 2.4.

Variables that are not specifically distributed by the programmer with an HPF directive behave just like variables in the message-passing program: each process has a separate version of the variable. For example, the variable `h` is in a different memory location on each process (even though we give it the same value).

Note also that the details of the distribution are controlled by HPF: the `BLOCK` distribution is specifically defined by HPF and does not exactly match the decomposition shown in Figure 2.2. For values of `n` that are much greater than the number of processes (the only case where parallelism makes any sense), however, the HPF choice is as good as any.

An advantage of HPF is that by changing the single line

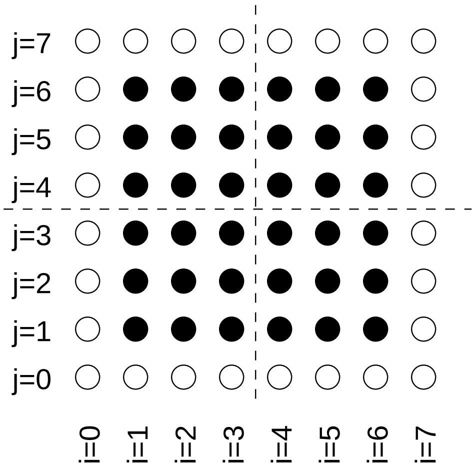

Figure 2.5: Decomposition of the mesh across a two-dimensional array of four processes, corresponding to an HPF BLOCK,BLOCK distribution.

```
!HPF$ DISTRIBUTE u(:,BLOCK)
```

to

```
!HPF$ DISTRIBUTE u(BLOCK,BLOCK)
```

we can change the distribution of the arrays to that shown in Figure 2.5.

We call this the single name-space, distributed memory model because all communication between processes is handled with variables (like `u`) that are declared globally, that is, they are declared as if they were accessible to all processes. This allows many programs to be written so that they are very similar to the sequential version of the same program. In fact, the program in Figure 2.4 is nearly identical to Figure 2.1, particularly if the `i` and `j` loops in Figure 2.1 are replaced with the Fortran 90 array expression used in Figure 2.4.

2.1.5 The Shared Memory Model

The shared memory model, in contrast to the distributed memory model, has 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 the generation of 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 2.6.

[1] Well, nearly all.

```
    real u(0:n,0:n), unew(0:n,0:n), f(1:n-1, 1:n-1), 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 2.6: OpenMP (shared memory) version of the Jacobi algorithm **CHECK THIS EXAMPLE**

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 described 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.

Because of the simplicity of the algorithm and the data-structures in this example, these examples are very simple and do not address the many issues that can arise in more complex situations, such as unstructured grids, dynamic (run-time) allocation and management of data structures, and more complex data dependencies between shared data-structures (either between processes or threads). Some of these issues are discussed in more detail in Sections ??. Even the convergence test, a necessary part of this algorithm that we have left out for simplicity, requires care, since the result is a single value that all processes/threads contribute to and that must be available to all processes. Computing this scalably and correctly requires care; each of the programming models illustrated above provides special features to handle this and similar problems. These are discussed in the next section.

Another discussion that focuses on some of the more subtle issues, particularly for the shared memory case is given in [?]. Suggestions for choosing between different approaches to expressing parallel programs are given in Section ??.

2.2 Adding Global Operations

In the examples above, the code to check for convergence was left out. This allowed us to concentrate on how to compute with an array distributed across many processes or processors. For computations such as a convergence test, a single value is needed by all processes or threads. In this section, we discuss how each approach to parallel computing provides this operation.

A simple convergence test is to compute the two-norm of the difference between two successive iterations. In the serial case, this can be accomplished with the code shown in Figure 2.7.

2.2.1 Collective operations in MPI

In the MPI case, computing the two norm of the difference of `unew` and `u` requires two steps. First, the sum of the squares of the differences of the local part of `unew` and `u` are computed. These are then combined with the contributions from all of the other processes and summed together. Because the operation of combining values from many processes is common and important, and because efficient implementations of this operation can require very system-specific code and algorithms, MPI provides a special routine, `MPI_Allreduce`, to combine a

```
real u(0:n,0:n), unew(0:n,0:n), twonorm

! ...
  twonorm = 0.0
  do j=1, n-1
    do i=1, n-1
      twonorm = twonorm + (unew(i,j) - u(i,j))**2
    enddo
  enddo
  twonorm = sqrt(twonorm)
  if (twonorm .le. tol) ! ... declare convergence
```

Figure 2.7: Sequential code to compute the two-norm of the difference between two iterations of the Jacobi algorithm

value from each process and return to all processes the result. This is shown in Figure 2.8.

This operation is called a *reduction* because it combines values from many sources into a single value. MPI provides many routines for communication and computation on a collection of processes; these are called *collective operations*.

2.2.2 Reductions in HPF

Fortran 90 and hence HPF contain built-in functions for computing the sum of all of the values in an array. In HPF, these functions work with distributed arrays, so the code is very simple, as shown in Figure 2.9.

2.2.3 Reductions in OpenMP

The approach taken in OpenMP is somewhat different from that in HPF. Just like MPI, OpenMP recognizes that reductions are a common operation. In OpenMP, you can indicate that the result of a variable is to be formed by a reduction with a particular operator. This is shown in Figure 2.10.

The effect of the `reduction(+:twonorm)` statement is to cause the OpenMP compiler to create a separate, private version of `twonorm` in each thread. When the enclosing scope ends, OpenMP combines the contributions in each thread using the specified operation to form the final value.

This code also illustrates the directive `private` to create a variable that is private to each thread (i.e., not shared). Without this directive, the value of `ldiff` added to the thread-private value of `twonorm` could come from the "wrong" thread.

2.2.4 Final Comments

All of these approaches to finding the two-norm exploit the associativity of real arithmetic. Unfortunately, computers don't use real numbers, they use an approximation called floating-point numbers. Operations with floating-point

```
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1), twonorm
integer ierr

! ...

  twonorm_local = 0.0
  do j=js, je
    do i=1, n-1
      twonorm_local = twonorm_local + (unew(i,j) - u(i,j))**2
    enddo
  enddo
  call MPI_Allreduce( twonorm_local, twonorm, 1, &
              MPI_REAL, MPI_SUMM, MPI_COMM_WORLD, ierr )
  twonorm = sqrt(twonorm)
  if (twonorm .le. tol) ! ... declare convergence
```

Figure 2.8: Message-passing version of Figure 2.7

```
    real u(0:n,0:n), unew(0:n,0:n), twonorm
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew with u
!HPF$ ALIGN f with u

    ! ...
      twonorm = sqrt ( sum ( (unew(1:n-1,1:n-1) - u(1:n-1,1:n-1))**2) )
      if (twonorm .le. tol) ! ... declare convergence
    enddo
```

Figure 2.9: HPF version of the convergence test for the Jacobi algorithm **CHECK THIS EXAMPLE**

```
    real u(0:n,0:n), unew(0:n,0:n), twonorm

    ! ..
      twonorm = 0.0
!$omp parallel
!$omp do private(ldiff) reduction(+:twonorm)
      do j=1, n-1
        do i=1, n-1
          ldiff = (unew(i,j) - u(i,j))**2
          twonorm = twonorm + ldiff
        enddo
      enddo
!$omp enddo
!$omp end parallel
      twonorm = sqrt(twonorm)
    enddo
```

Figure 2.10: OpenMP (shared memory) version of the convergence test for the Jacobi algorithm **CHECK THIS EXAMPLE**

number are nearly but not exactly associative. (See any introductory book on Numerical Analysis.) Because of this lack of associativity, the value computed by these methods may be different. In a well-designed algorithm, the difference will be small (in relative terms). However, this difference can sometimes be unexpected and hence confusing.

2.3 Unstructured Meshes

The preceeding sections have focused on regular meshes because these provide the simplist code examples. Many computations, however, rely on unstructured meshes, such as that in Figure 2.11.

Parallelizing a code that uses an unstructured mesh follows a similar path to parallelizing a structured-mesh code. For MPI, the first step is to partition the grid. For parallel finite element calculations, it is necessary to partition the mesh across the processors in such a way that each processor's work load is balanced and the communication between processors is minimized. There are many different ways to partition meshes, and if done naively, the result can be an inefficient parallel implementation. Consider a simple example using linear finite elements on an unstructured, triangular mesh. In this case, the amount of work associated with each element is the same and communication is required to transfer information to nearest neighbor elements that have been assigned to a different processor. Thus to meet our partitioning objective of assigning equal work to all processors while simultaneously minimizing communication costs, we must assign an equal number of elements to the processors and minimize the number of off-processor neighboring elements. In Figure 2.12, we show

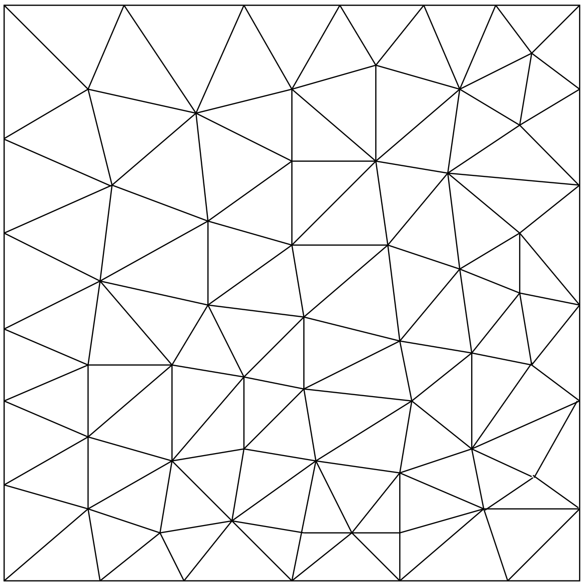

Figure 2.11: A simple unstructured grid

the results of two partitioning strategies for a triangular mesh. In the left figure, we sort the elements by the y-coordinant of their centroid and assign an equal number of elements to each of four processors. In the right figure, we sort the elements in the y-direction and make one cut that divides the set of elements in half. Each subset of elements is then sorted in the x-direction and divided so that they have again been equally distributed to each of the four processors. Although both partitioning strategies balance the work load, their communication patterns are quite different. For example, consider processor P3; the communication required for this processor is indicated by the shaded elements in each figure. There are roughly twice the number of off-processor neighbors in the first partitioning which will result in larger communication costs and a less efficient parallel implementation.

Many techniques have been developed for partitioning meshes; see Chapter **??** for more information.

Once the mesh has been partitioned, neighbor data must be moved between processes just as it was in the structure-mesh case. With MPI, this requires roughly the same routines, though the appropriate data must be gathered from the unstructured-mesh data structure, communicated to the neighboring process, and scattered to the appropriate ghostcells. MPI also provides a way to combine the scatter and gather operations with the communication through the use of MPI datatypes, though few MPI implementations have made these efficient.

An HPF expert should review and change the following

In HPF, similar steps must be used, since it is no longer possible to use HPF

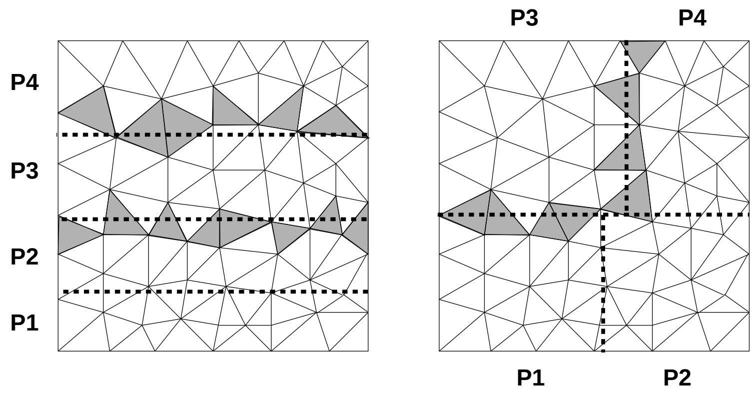

Figure 2.12: The results of partitioning an unstructured mesh using two different strategies

```
!HPF$ DISTRIBUTE ugather(*,BLOCK)
!HPF$ ALIGN uscatter WITH ugather
      real ugather(n,2), uscatter(n,2)
      ! ... gather data into ugather(:,myprocess)
      uscatter(:,myprocess) = ugather(:,neighbor)
```

Figure 2.13: Using HPF arrays to communicate data from process `neighbor` to the calling process.

partitioning directives to partition the unstructured mesh. Communication of neighbor data between processes can be managed by using a communication array as shown in Figure 2.13.

In this example, each process sends `n` data items. In an unstructured mesh computation, the number of neighbor data values needed will probably be different for each neighboring process. With a little more work, each process can arrange to communicate exactly the correct amount of data.

An OpenMP expert should check and change the following

Since OpenMP is a fully shared-memory model, it is unnecessary to explicitly communicate any data. An unstructured mesh often has a single loop (over all mesh cells), rather than nested loops over each coordinate direction; further, the mesh data is often accessed through indirect addressing (e.g., `A(IADD(K))` rather than `A(K)`). Partitioning the mesh and introducing an outer loop over the partitions can help the OpenMP compiler generate efficient code. Partitioning the mesh also helps in maintaining memory locality, which is critical for performance. To reduce the performance consequences of *false sharing*, it may also be necessary to make copies of the neighboring data, similar to the gather/scatter steps that are required for MPI and HPF.

Acknowledgement

Thanks to Lori Freitag for the unstructured mesh example and the discussion of partitioning.