CHAPTER 10

Design and Coding of Parallel Programs

NOW THAT WE HAVE MASTERED the basics of parallel programming in MPI, it's time to consider the problem of program design and coding.

There are essentially two approaches to designing parallel programs:

1. **The data-parallel approach.** In this approach we partition the data among the processes, and each process executes more or less the same set of commands on its data.
2. **The control-parallel approach.** In this approach we partition the tasks we wish to carry out among the processes, and each process executes commands that are essentially different from some or all of the other processes.

It should be noted that most parallel programs involve both approaches. However, data-parallel programming is more common and it is generally much easier to do. Perhaps most importantly, data-parallel programs tend to **scale** well: loosely, this means that they can be used to solve larger and larger problems with more and more processes.

All of the programs in this text use a mix of both methods, but data parallelism tends to predominate, and, as a consequence, we'll devote most of our attention to the design of data-parallel programs.

10.1 Data-Parallel Programs

We have already seen an example of a data-parallel program: the trapezoidal rule program. In it, the input data is the interval $[a, b]$, which is partitioned equally among the processes, and each process estimates the integral over its subinterval. If we ignore the I/O and estimate the entire integral by using `MPI_Allreduce` rather than `MPI_Reduce`, we see that the program is perfectly data parallel: each process executes exactly the same statements on its data.

So how do we design a data-parallel program? Generally, we start by examining serial solutions to the problem. Then we try partitioning the data in various ways among the processes. If the solution can be obtained by simply running the serial algorithm on each subset of the data, we have the simplest possible candidate for a data-parallel solution. This is exactly what happened with the trapezoidal rule. In general, of course, this won't happen—we'll have to write some code for communicating among the processes.

Also note that ideally the program should be designed to run with arbitrarily many input values and arbitrarily many processes. However, in practice, we usually make some simplifying assumptions. For example, in our dot product function, we made the simplifying assumption that n, the order of the vectors, was evenly divisible by p, the number of processes. If the simplifying assumptions are too restrictive, the program should be designed so that it is fairly straightforward to modify it later. In the dot product example, we might replace the parameter `n_bar` ($= n/p$) by the number of components assigned to the process.

10.2 Jacobi's Method

As a simple example, let's write a program for solving a system of linear equations using Jacobi's method. So suppose $A\mathbf{x} = \mathbf{b}$ is a system of linear equations where $A = (a_{ij})$ is a nonsingular $n \times n$ matrix, $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$, and $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})^T$. Recollect that Jacobi's method is an iterative method; that is, after making an initial guess, $\mathbf{x}^0$, to a solution, the method generates a sequence of approximations $\mathbf{x}^k$, $k = 1, 2, 3, \ldots$, to the solution $\bar{\mathbf{x}}$. The ith component, $x_i^{(k+1)}$ of the $(k+1)$st iterate, $\mathbf{x}^{k+1}$, is computed using the ith row of the system:

$$a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{ii}x_i + \cdots + a_{i,n-1}x_{n-1} = b_i.$$

If we solve this equation for x_i and substitute $x_i^{(k+1)}$ for x_i and $x_j^{(k)}$ for $x_j, j \neq i$, we obtain the iteration formula

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}\right).$$

In general, this iteration may not converge. However, if the system is **strictly diagonally dominant**, it will. That is, if for $i = 0, 1, \ldots, n-1$, we have

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|,$$

then Jacobi's method will converge.

In order to terminate the iteration, we can compute the size of the difference between successive estimates,

$$\|\mathbf{x}^{k+1} - \mathbf{x}^k\|,$$

and when this becomes less than some predefined tolerance, the iteration terminates. Since the iteration may not converge, we should also keep track of the number of iterations, and terminate if there is no convergence after some maximum number of iterations.

Given these considerations, we can write a serial version of Jacobi's method.

```
/* Return 1 if iteration converged, 0 otherwise */
/* MATRIX_T is just a 2-dimensional array       */
int Jacobi(
        MATRIX_T  A          /* in  */,
        float     x[]        /* out */,
        float     b[]        /* in  */,
        int       n          /* in  */,
        float     tol        /* in  */,
        int       max_iter   /* in  */) {
    int    i, j;
    int    iter_num;
    float  x_old[MAX_DIM];

    float Distance(float x[], float y[], int n);

    /* Initialize x */
    for (i = 0; i < n; i++)
        x[i] = b[i];

    iter_num = 0;
    do {
        iter_num++;

        for (i = 0; i < n; i++)
            x_old[i] = x[i];

        for (i = 0; i < n; i++){
            x[i] = b[i];
            for (j = 0; j < i; j++)
                x[i] = x[i] - A[i][j]*x_old[j];
```

```
            for (j = i+1; j < n; j++)
                x[i] = x[i] - A[i][j]*x_old[j];
            x[i] = x[i]/A[i][i];
        }
    } while ((iter_num < max_iter) &&
             (Distance(x,x_old,n) >= tol));

    if (Distance(x,x_old,n) < tol)
        return 1;
    else
        return 0;
} /* Jacobi */

float Distance(float x[], float y[], int n) {
    int   i;
    float sum = 0.0;

    for (i = 0; i < n; i++) {
        sum = sum + (x[i] - y[i])*(x[i] - y[i]);
    }
    return sqrt(sum);
} /* Distance */
```

10.3 Parallel Jacobi's Method

If there are p processes, and we make the assumption that $n \geq p$, a natural approach to parallelizing `Jacobi` is to have each process calculate the entries in a subvector of the solution vector $\mathbf{x}$. In order to do this, we begin by considering how to distribute the data among the processes. In other words, how should `A`, `n`, `b`, `tol`, `max_iter`, `x`, and `x_old` be distributed? As a rule of thumb, we can give each process a copy of all the scalar variables—since the amount of memory used by scalars is generally negligible. In our case, these are `n`, `tol`, and `max_iter`. So after process 0 reads these values, it should broadcast them to all the processes.

There are two obvious possibilities for the vectors `x`, `b`, and `x_old`. They can be partitioned into subvectors of more or less the same size and each process can be assigned a distinct subvector, or each process can be assigned an entire copy of the vector. In order to decide, let's look at our initial idea for parallelizing. The heart of the algorithm is

```
Process q:  Calculate the entries in x that are
    assigned to q.
```

In order to do this we need to calculate

```
Process q:
    for each subscript i assigned to q {
```

Table 10.1 Block partitioning of $\mathbf{x}$

Process	*Components*
0	$x_0, x_1, \ldots, x_{\bar{n}-1}$
1	$x_{\bar{n}}, x_{\bar{n}+1}, \ldots, x_{2\bar{n}-1}$
$\vdots$	$\vdots$
q	$x_{q\bar{n}}, x_{q\bar{n}+1}, \ldots, x_{(q+1)\bar{n}-1}$
$\vdots$	$\vdots$
$p-1$	$x_{(p-1)\bar{n}}, x_{(p-1)\bar{n}+1}, \ldots, x_{n-1}$

```
        x[i] = b[i];
        for (j = 0; j < i; j++)
            x[i] =  x[i] - A[i][j]*x_old[j];
        for (j = i+1; j < n; j++)
            x[i] =  x[i] - A[i][j]*x_old[j];
        x[i] = x[i]/A[i][i];
    }
```

So in order to carry out the basic calculations, each process needs a *complete* copy of `x_old`, but only its own entries in `x` and `b`.

This calculation also suggests a partitioning of `A`: each process is assigned the rows of `A` corresponding to its entries in `x` and `b`.

Note that there is considerable ambiguity in the notation `x[i]`. In the first place, `x` is to be distributed among the processes. So the variable we called `x` in the preceding pseudocode is not the same as the variable `x` in our serial implementation of Jacobi's method. For example, the first `x` will, in general, reference a smaller block of memory than the second. There is further ambiguity in the use of the subscript `i`. The implication in the pseudocode is that `i` is a *global* subscript. That is, the variable `i` in the pseudocode is the same as the variable `i` in the serial implementation. This is also not, in general, the case, since array subscripts in C must begin at 0. It is easy enough to remedy the first problem: simply append the string "`_local`" to arrays that have been distributed. The second problem, however, is not so easily remedied, and it is up to us, the programmers, to make sure that we keep track of the meaning of subscripts.

The only remaining issue is which entries of `x` (and hence entries of `b` and rows of `A`) should be assigned to each process. If there are p processes, in order to balance the computational load, we would like to assign approximately n/p entries to each process. So in order to simplify the assignment, let's assume that p evenly divides n. With this assumption, there are two obvious approaches to partitioning x: a block partitioning and a cyclic partitioning. We've already encountered both. Recall that if $\bar{n} = n/p$, the block partitioning is defined as in Table 10.1. In the cyclic partitioning, each process is assigned the components whose subscripts are equivalent to its rank mod q. That is,

Table 10.2 Cyclic partitioning of $\mathbf{x}$

Process	*Components*
0	$x_0, x_p, x_{2p}, \ldots, x_{(\bar{n}-1)p}$
1	$x_1, x_{p+1}, x_{2p+1}, \ldots, x_{(\bar{n}-1)p+1}$
⋮	⋮
q	$x_q, x_{p+q}, x_{2p+q}, \ldots, x_{(\bar{n}-1)p+q}$
⋮	⋮
p-1	$x_{p-1}, x_{2p-1}, x_{3p-1}, \ldots, x_{n-1}$

assign x_i to process q if $q \equiv i \bmod q$. Thus, if we use the cyclic partitioning, we'll make the assignments shown in Table 10.2.

Is there any reason to prefer one partitioning scheme to the other? For example, if you think about the parallel dot product, you can see that the cyclic partition would work just as well as the block. Indeed, the function we wrote in Chapter 5 can be used without any modification with a cyclic partition of the vectors. Is the same thing true for the parallel Jacobi? Let's look more closely at the solution.

```
Process q:
    Assign entries of b_local to x_local;

    iter_num = 0;
    do {
       iter_num++;
       Copy entries of x_local from each process
           to x_old;

       Calculate entries of x_local;

    } while ((iter_num < max_iter) &&
             (Distance(x_global, x_old, n) >= tol));
```

The only parts of the algorithm that we haven't already looked at are

```
Copy entries of x_local from each process
    to x_old
```

and the calculation of `Distance`. Since the distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ is just

$$\sqrt{(\mathbf{x}-\mathbf{y})\cdot(\mathbf{x}-\mathbf{y})}$$

(i.e., a dot product), there's no advantage to using one partitioning scheme over the other in the calculation of `Distance`.

In order to carry out the calculation

```
Copy entries of x_local from each process
    to x_old
```

we want to execute something like the following:

```
Copy x_local to temp;
for (root = 0; root < p; root++) {
   MPI_Bcast(temp, n_bar, MPI_FLOAT, root,
       MPI_COMM_WORLD);
   Copy temp into appropriate locations in x_old;
}
```

If we have used a block partitioning of x, this has the same overall effect as a single call to the MPI collective communication function, `MPI_Allgather`:

```
MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_old,
    n_bar, MPI_FLOAT, MPI_COMM_WORLD)
```

That is, each process's array `x_local` is sent to every other process, and the received arrays are copied in process rank order into `x_old`. Clearly we would prefer to use this single command, since it can be optimized for the particular system we're using. In order to use it with the block mapping, we can simply call it with the arguments indicated above, since the subvectors are simply concatenated in process rank order. However, in order to use it with the cyclic mapping, we must "interleave" the entries received from the different processes, and, in order to do this, we must build a derived datatype. Thus, it is somewhat simpler to use the block mapping, and since we have no other reason to prefer one mapping to the other, let's use the block mapping.

Now we're ready to write a parallel Jacobi routine.

```
#define Swap(x,y) {float* temp; temp = x; \
                   x = y; y = temp;}

/* Return 1 if iteration converged, 0 otherwise */
/* MATRIX_T is a 2-dimensional array            */
int Parallel_jacobi(
        MATRIX_T  A_local    /* in  */,
        float     x_local[]  /* out */,
        float     b_local[]  /* in  */,
        int       n          /* in  */,
        float     tol        /* in  */,
        int       max_iter   /* in  */,
        int       p          /* in  */,
        int       my_rank    /* in  */) {
    int     i_local, i_global, j;
    int     n_bar;
    int     iter_num;
    float   x_temp1[MAX_DIM];
```

```
    float   x_temp2[MAX_DIM];
    float*  x_old;
    float*  x_new;

    float Distance(float x[], float y[], int n);

    n_bar = n/p;

    /* Initialize x */
    MPI_Allgather(b_local, n_bar, MPI_FLOAT, x_temp1,
        n_bar, MPI_FLOAT, MPI_COMM_WORLD);
    x_new = x_temp1;
    x_old = x_temp2;

    iter_num = 0;
    do {
        iter_num++;

        /* Interchange x_old and x_new */
        Swap(x_old, x_new);
        for (i_local = 0; i_local < n_bar; i_local++){
            i_global = i_local + my_rank*n_bar;
            x_local[i_local] = b_local[i_local];
            for (j = 0; j < i_global; j++)
                x_local[i_local] = x_local[i_local] -
                    A_local[i_local][j]*x_old[j];
            for (j = i_global+1; j < n; j++)
                x_local[i_local] = x_local[i_local] -
                    A_local[i_local][j]*x_old[j];
            x_local[i_local] = x_local[i_local]/
                    A_local[i_local][i_global];
        }

        MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_new,
            n_bar, MPI_FLOAT, MPI_COMM_WORLD);
    } while ((iter_num < max_iter) &&
             (Distance(x_new,x_old,n) >= tol));

    if (Distance(x_new,x_old,n) < tol)
        return 1;
    else
        return 0;
} /* Jacobi */


float Distance(float x[], float y[], int n) {
    int i;
    float sum = 0.0;
```

```
    for (i = 0; i < n; i++) {
        sum = sum + (x[i] - y[i])*(x[i] - y[i]);
    }
    return sqrt(sum);
} /* Distance */
```

Note that we used storage for an extra temporary vector—x_new. We did this so that each pass through the main loop involved only one communcation—the call to MPI_Allgather. If we wanted to avoid using additional storage, we could have gathered into x_old at the beginning of the loop, and used a call to MPI_Allreduce in Distance. That is, the main loop would look like

```
do {
    MPI_Allgather(x_local, . . . , x_old, . . .);
    Calculate new entries in x_local;
} while ((iter_num < max_iter) &&
          (Distance(x_local,x_old,n) >= tol));
```

and the Distance function would first compute the dot product

```
for (i_local = 0; i_local < n_bar; i++) {
    i_global = i_local + my_rank*n_bar;
    sum += (x[i] - y[i_global])*(x[i] - y[i_global]);
}
```

and then call

```
MPI_Allreduce(&sum, . . . );
```

10.4 Coding Parallel Programs

The mechanics of actually coding a parallel program are much the same as they are for coding a serial program. Before any code is written, we decide on the basic outline of the program and identify the main data structures.

When we actually begin to code, we can either proceed "top down" or "bottom up." Generally, a mix of the two approaches is useful: most of the code can be developed top-down. However, as soon as data structures are (provisionally) determined, it is essential that we write I/O routines and basic access functions (or macros) for the structures.

Thus you typically begin by defining the data structures with "dummy" type definitions—e.g., typedef int MAIN_TYPE. Write functions and/or macros for accessing the data in the structures. By doing this, you only need to modify localized pieces of code when you decide that you need to change your data structures. Write real, working code for the main program, defining subprograms with stubs—e.g., void Sub_program(int param1, int

param2){}. When the main program is done, begin defining the subprograms. The definition of the actual data structures should be deferred for as long as possible. The basic rule of thumb is, If it looks hard, procrastinate. If a piece of code seems complicated, defer writing it by assigning it to a subprogram.

As subprograms are completed, the code should be tested. As with serial programs, this involves two phases. First test the subprogram in a **driver**—a complete program whose sole function is to call the subprogram. If it's at all possible, make your driver a serial program. Even if the function being tested involves communication, the driver program can be initially run on a single process.

After testing the subprogram in the driver, test it in the evolving program. For this it will be necessary to add informative output statements to the program skeleton so that you can verify that control is flowing as planned. As we noted in Chapter 8, this can be a problem on some systems. So you may want to use the functions we wrote for output.

Since typical parallel programs are designed to run with varying numbers of processes, at each stage you should test your code on varying numbers of processes. Typically, if your program won't run with just one process, it has no hope of success with two. Of course, even if it runs correctly with one, it may fail with two, etc.

Since I/O tends to be one of the most troublesome aspects of parallel programming, it may be useful to first code a program with hardwired input. That is, rather than write a complete input function, it may be easier to simply assign values to the variables inside the program. For example, if we wish to read in a distributed array of floats, we might first write the following function.

```
void Read_array(float x_local[], int local_order) {
    int i;

    /* Defer writing the code to actually read in the
     * data. */
    for (i = 0; i < local_order; i++)
        x_local[i] = (float) i;
} /* Read_array */
```

Alternatively, you can use the self-initializing code discussed in Chapter 8.

10.5 An Example: Sorting

As an example of parallel program development, let's write a program that sorts a list of keys (e.g., numbers or words) into process-increasing order. That is, the keys on process q are sorted into increasing order, and all the keys on q are greater than all the keys on $q - 1$ and less than all the keys on $q + 1$. If you prefer, you can imagine that we have a linear array distributed in block

fashion across the processes, and we want to sort the elements of the array in increasing order.

Perhaps surprisingly, general sorting on parallel processors is a difficult problem and remains an active area of research. So in order to simplify the problem, we'll assume that we know the distribution of the entire set of keys. That is, we know the probability that a randomly selected key falls into a given range of keys. Knowing the distribution, we can determine which process should be assigned which keys. For example, suppose we have integer keys distributed among four processes, and we know that the keys are *uniformly* distributed in the range 1–100. Then process 0 should receive keys in the range 1–25, process 1 should receive keys in the range 26–50, etc.

With this assumption, it's easy to devise an algorithm:

```
Get local keys.
Use distribution to determine which
    process should get each key.
Send keys to appropriate processes.  Receive
   keys from processes.
Sort local keys.
Print results.
```

We still haven't specified the type, the distribution, and the source of the keys. So let's suppose that the keys are nonnegative integers and that they're uniformly distributed. Since, as usual, I/O is a problem, we'll just generate the keys on the processes, and the input to the program will just be the total number of keys. In order to generate the keys, we can use the C function `rand`. The actual range of values generated by `rand` varies from system to system. However, most allow at least the range 0–32,767, so let's use this as the range of our keys.

10.5.1 Main Program

We know we'll be needing storage for the keys on each process, so let's make a dummy type definition.

```
typedef int LOCAL_LIST_T;
```

In the interests of making the program easier to modify, we can also define a type for the keys:

```
typedef int KEY_T;
```

After adding an input function, we can just convert our basic algorithm into a sequence of function calls.

```
/* First Level Program */
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include "cio.h"
#include "sort.h"

int       p;
int       my_rank;
MPI_Comm  io_comm;

main(int argc, char* argv[]) {
    LOCAL_LIST_T  local_keys;
    int           list_size;
    int           error;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);

    list_size = Get_list_size();

    /* Return negative if Allocate failed */
    error = Allocate_list(list_size, &local_keys);

    Get_local_keys(&local_keys);
    Print_list(&local_keys);
    Redistribute_keys(&local_keys);
    Local_sort(&local_keys);
    Print_list(&local_keys);

    MPI_Finalize();
} /* main */

int Get_list_size(void) {
    Cprintf(io_comm, "", "%s", "In Get_list_size");
    return 0;
} /* Get_list_size */

/* Return value negative indicates failure */
int Allocate_list(int list_size,
    LOCAL_LIST_T* local_keys) {

    Cprintf(io_comm, "", "%s", "In Allocate_key_list");
    return 0;
} /* Allocate_list */
```

```
void Get_local_keys(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm, "", "%s", "In Get_local_keys");
} /* Get_local_keys */

void Redistribute_keys(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm, "", "%s", "In Redistribute_keys");
} /* Redistribute_keys */

void Local_sort(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm, "", "%s", "In Local_sort");
} /* Local_sort */

void Print_list(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm, "", "%s", "In Print_list");
} /* Print_list */
```

Notice that we're using a header file `sort.h`. This is especially convenient when the various definitions haven't been finalized—it gives us a fixed single location for changing `typedef`s and function prototypes as they evolve. (Unfortunately, we still need to change the actual function calls.)

```
/* sort.h */
#ifndef SORT_H
#define SORT_H

#define KEY_MIN 0
#define KEY_MAX 32767
#define KEY_MOD 32768

typedef int KEY_T;
typedef int LOCAL_LIST_T;

#define List_size(list) (0)
#define List_allocated_size(list) (0)

int Get_list_size(void);
int Allocate_list(int list_size,
    LOCAL_LIST_T* local_keys);
void Get_local_keys(LOCAL_LIST_T* local_keys);
void Redistribute_keys(LOCAL_LIST_T* local_keys);
void Local_sort(LOCAL_LIST_T* local_keys) ;
void Print_list(LOCAL_LIST_T* local_keys) ;
#endif
```

Also notice that we always pass `local_keys` by reference—in spite of the fact that not every function will modify it. The reason for this is that we expect this data structure to be large, and we don't want to waste time and space copying it when we call a function. Further, we expect the list data structure to contain

information on the number of keys stored and the space allocated, so we've written two dummy macros for accessing this information.

Since the program is so simple, the only real purpose in testing at this point is to look for typos. If we do compile it and run it with one process, the output is

```
Process 0 > In Get_list_size

Process 0 > In Allocate_key_list

Process 0 > In Get_local_keys

Process 0 > In Print_list

Process 0 > In Redistribute_keys

Process 0 > In Local_sort

Process 0 > In Print_list
```

If we run it with two processes, the output is

```
Process 0 > In Get_list_size
Process 1 > In Get_list_size

Process 0 > In Allocate_key_list
Process 1 > In Allocate_key_list

Process 0 > In Get_local_keys
Process 1 > In Get_local_keys

Process 0 > In Print_list
Process 1 > In Print_list

Process 0 > In Redistribute_keys
Process 1 > In Redistribute_keys

Process 0 > In Local_sort
Process 1 > In Local_sort

Process 0 > In Print_list
Process 1 > In Print_list
```

10.5.2 The "Input" Functions

OK. Now it's just a matter of filling in the subprograms.

Since input can be a major problem, we'll first write our program with hardwired input. During the initial testing, we're probably going to want to

look at the actual contents of the list, so we don't want to be printing thousands of keys. So let's assume that we'll have five keys per process when we begin. That is, `Get_list_size` should return $5p$.

```
int Get_list_size(void) {
    Cprintf(io_comm, "", "%s", "In Get_list_size");
    return 5*p;
} /* Get_list_size */
```

We can't actually allocate the list yet, since we haven't decided on the actual structure, but we do want members of the structure to record the number of keys and the space allocated. So we can provisionally define a struct with three members, one of which is a dummy:

```
/* Goes in sort.h */
typedef struct {
    int allocated_size;
    int local_list_size;
    int keys;  /* dummy member */
} LOCAL_LIST_T;

/* Assume list is a pointer to a struct of type
 * LOCAL_LIST_T */
#define List_size(list) ((list)->local_list_size)
#define List_allocated_size(list) ((list)->allocated_size)
```

With these definitions, we can update `Allocate_list`.

```
 /* Returns negative value if malloc fails.  */
int Allocate_list(
        int           list_size  /* in  */,
        LOCAL_LIST_T* local_keys /* out */) {

    List_allocated_size(local_keys) = list_size/p;
    List_size(local_keys) = list_size/p;
    return 0;
} /* Allocate_list */
```

In order to get the keys, we can use the C functions `srand` and `rand`. The first function seeds the random number generator. We can use it to make sure that each process gets a different set of keys by seeding each process's random number generator with its rank. The second function, `rand`, actually generates the keys. Of course, we need to create storage for the list of keys. But this would imply that we know the data structure `LOCAL_LIST_T`. So we'll avoid this issue by putting the key list accesses into functions, which, for the time being, will be stubs.

```
/* Use random number generator to generate keys */
void Get_local_keys(LOCAL_LIST_T* local_keys) {
    int i;

    /* Seed the generator */
    srand(my_rank);

    for (i = 0; i < List_size(local_keys);  i++)
        Insert_key(rand() % KEY_MOD, i, local_keys);
} /* Get_local_keys */
```

Notice that we are assuming that `list_size` is evenly divisible by the number of processes.

The stub for `Insert_key`:

```
void Insert_key(KEY_T key, int i,
    LOCAL_LIST_T* local_keys) {
} /* Insert_key */
```

We omit the call to `Cprintf` because `Insert_key` will be called five times by each process and we don't want to be overwhelmed by the output.

10.5.3 All-to-all Scatter/Gather

The heart of the redistribution of the keys is each process's sending of its original local keys to the appropriate process. This is an example of an **all-to-all scatter/gather,** also called a **total exchange.** This is a collective communication operation in which each process sends a distinct collection of data to every other process. Recall that a scatter is a collective communication in which a fixed root process sends a distinct collection of data to every other process, and a gather is a collective communication in which a root process receives data from every other process. Thus an all-to-all scatter/gather, or total exchange, can be viewed as a series of scatters—each process carries out a scatter—or as a series of gathers—each process carries out a gather. As you can probably guess, MPI provides a function for all-to-all scatter/gathers.

There are two forms of total exchange in MPI: `MPI_Alltoall` and `MPI_Alltoallv`. The first form is used when we wish to send the same *amount* of data to every process. At first glance it might appear that we have no use for this, since, in general, each process will send a different amount of data to every other process. However, in order to use the second form, we need to know how much data is to be received by each process, and a simple way to communicate this information is to use the first form of `MPI_Alltoall`.

The syntax of `MPI_Alltoall` is

```
int MPI_Alltoall(
        void*         send_buffer  /* in  */,
```

```
int           send_count   /* in  */,
MPI_Datatype  send_type    /* in  */,
void*         recv_buffer  /* out */,
int           recv_count   /* in  */,
MPI_Datatype  recv_type    /* in  */,
MPI_Comm      comm         /* in  */)
```

This is a collective operation, so every process in the communicator `comm` must call the function. Its effect on process q is to send `send_count` elements of type `send_type` to every process (including itself). The first block of `send_count` elements goes to process 0, the second block to process 1, etc. Process q will also receive `recv_count` elements of type `recv_type` from every process. The elements from process 0 are received into the beginning of `recv_buffer`. The elements from process 1 are received immediately following those from 0, etc.

The syntax of `MPI_Alltoallv` is similar:

```
int MPI_Alltoallv(
        void*         send_buffer           /* in  */,
        int           send_counts[]         /* in  */,
        int           send_displacements[]  /* in  */,
        MPI_Datatype  send_type             /* in  */,
        void*         recv_buffer           /* out */,
        int           recv_counts[]         /* in  */,
        int           recv_displacements[]  /* in  */,
        MPI_Datatype  recv_type             /* in  */,
        MPI_Comm      comm                  /* in  */)
```

Its effect is also similar. The difference is that each process can send (receive) a different amount of data to (from) every process. So `send_count` and `recv_count` have been replaced by arrays of counts and arrays of displacements. The displacements are measured in elements of type `send_type` (`recv_type`) from the beginning of `send_buffer` (`recv_buffer`). Notice that the various arrays, `send_counts`, `send_displacements`, `recv_counts`, and `recv_displacements` all have p entries, where p is the number of processes in `comm`.

10.5.4 Redistributing the Keys

In view of our newly acquired knowledge of MPI collective communications, we can construct a fairly simple solution to redistribution:

1. Determine what and how much data is to be sent to each process.
2. Carry out a total exchange on the amount of data to be sent/received by each process.
3. Compute the total amount of space needed for the data to be received and allocate storage.

4. Find the displacements of the data to be received.
5. Carry out a total exchange on the actual keys.
6. Free old storage.

An easy way to carry out the first step is to sort the local keys, then simply traverse the list of local keys, determining where the keys going to processor q end and those going to processor $q + 1$ begin. We'll need two lists: one to keep track of where the keys being sent to each process begin and one to keep track of how many keys go to each process. Since this information will be used directly by `MPI_Alltoallv`, in the form of arrays, we'll allocate arrays for this information.

Note also that `MPI_Alltoallv` uses arrays for the lists of keys that will be distributed. So if we want to avoid recopying the data in the key lists, it should be stored in arrays also. So let's tentatively define

```
typedef struct {
    int allocated_size;
    int local_list_size;
    KEY_T* keys;
} LOCAL_LIST_T;
```

With these definitions we can write the function `Redistribute_keys`.

```
void Redistribute_keys(
         LOCAL_LIST_T* local_keys  /* in/out */) {
    int new_list_size, i;
    int* send_counts;
    int* send_displacements;
    int* recv_counts;
    int* recv_displacements;
    KEY_T* new_keys;

    /* Allocate space for the counts and displacements */
    send_counts = (int*) malloc(p*sizeof(int));
    send_displacements = (int*) malloc(p*sizeof(int));
    recv_counts = (int*) malloc(p*sizeof(int));
    recv_displacements = (int*) malloc(p*sizeof(int));

    Local_sort(local_keys);
    Find_alltoall_send_params(local_keys,
        send_counts, send_displacements);

    /* Distribute the counts */
    MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts,
        1, MPI_INT, MPI_COMM_WORLD);

    /* Allocate space for new list */
```

```
    new_list_size = recv_counts[0];
    for (i = 1; i < p; i++)
        new_list_size += recv_counts[i];
    new_keys = (KEY_T*)
        malloc(new_list_size*sizeof(KEY_T));

    Find_recv_displacements(recv_counts, recv_displacements);

    /* Exchange the keys */
    MPI_Alltoallv(List(local_keys), send_counts,
        send_displacements, key_mpi_t, new_keys,
        recv_counts, recv_displacements, key_mpi_t,
        MPI_COMM_WORLD);

    /* Replace old list with new list */
    List_free(local_keys);
    List_allocated_size(local_keys) = new_list_size;
    List_size(local_keys) = new_list_size;
    List(local_keys) = new_keys;

    free(send_counts);
    free(send_displacements);
    free(recv_counts);
    free(recv_displacements);

} /* Redistribute_keys */
```

The most important thing to notice is that in the process of redistributing the keys, we create an entirely new list of keys—recall that MPI does not allow us to use the same list of keys as both the input argument (first argument) and as the output argument (fifth argument).

Notice also that we access the elements of `*local_keys` using macros. This makes the program much easier to modify if we decide to change the representation of any of the data. With the definition of `LOCAL_LIST_T` that we made previously, the following macros should be added to `sort.h`.

```
#define List_size(list) ((list)->local_list_size)
#define List_allocated_size(list) ((list)->allocated_size)
#define List(list) ((list)->keys)
#define List_free(list) {free List(list);}
```

Also notice that in the call to `MPI_Alltoallv` we use `key_mpi_t`. This is a derived datatype—as opposed to the C datatype `KEY_T`. In general, we would need to write a function defining this type, but since our keys are just ints, we can simply add the definition

```
#define key_mpi_t MPI_INT
```

to `sort.h`.

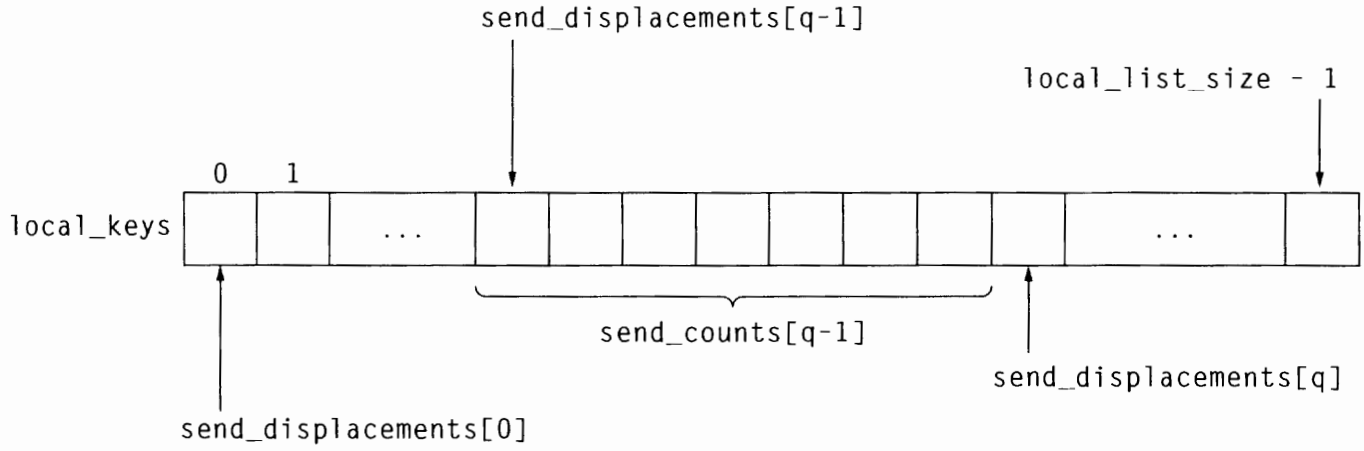

Figure 10.1 Displacements for `MPI_Alltoallv`

10.5.5 Pause to Clean Up

We leave the writing of the stubs for `Find_alltoall_send_params` and `Find_recv_displacements` to you. You should also finish up the incomplete code for `Allocate_list` and write the functions `Local_sort`, `Insert_key`, and `Print_list`. The `Local_sort` function can be simply a call to the standard C function `qsort`, or you can write your own sorting function. It is not very convenient to use the `Cprintf` function as the basis of `Print_List`. The difficulty is that `Cprintf` essentially prints one line of output per process, and if the local lists are long, it can get very tedious filling in the parameters to `Cprintf`. So you should probably use `Cprintf` as a model, but instead of sending a short, formatted list, send the list of keys. Be sure to address the problem of variable length lists. We strongly encourage you to update and test your complete code at this point.

10.5.6 `Find_alltoall_send_params`

In the function `Find_alltoall_send_params` we need to compute the values of the entries in the arrays `send_counts` and `send_displacements`. Since the entries in `List(local_keys)` have been sorted, we simply traverse this list until we encounter a value too large for the current process. The value in `send_counts[q]` is incremented as each key is examined. Since `send_displacements[q]` is the index of the first key to be sent to process q, its value can be computed as

```
send_displacements[q] = send_displacements[q-1] +
                        send_counts[q-1].
```

See Figure 10.1. In view of the value $q - 1$, the calculation for process 0 must be done separately. In order to calculate the cutoff—i.e., the first value that goes to the next higher ranked process—we use our assumption that the keys

are uniformly distributed in the range KEY_MIN–KEY_MAX (0–32,767). Hence the cutoff for process q is just

$$\text{cutoff} = (q + 1) * (\text{KEY\_MAX} + 1)/p$$

We'll put this in a function to allow for the possibility that a future modification of the program may use a different key type or a different distribution. A word of warning here: if your computer uses 16-bit integers and the type of the variable cutoff is int, then the preceding calculation will overflow, since KEY_MAX + 1 = 2^{15}. So you may need to make cutoff a float and cast the values in these calculations to float.

In view of these considerations, we can write the function.

```
void Find_alltoall_send_params(
         LOCAL_LIST_T* local_keys         /* in  */,
         int*          send_counts        /* out */,
         int*          send_displacements /* out */) {
    KEY_T cutoff;
    int i, j;

    /* Take care of process 0 */
    j = 0;
    send_displacements[0] = 0;
    send_counts[0] = 0;
    cutoff = Find_cutoff(0);
    /* Key_compare > 0 if cutoff > key */
    while ((j < List_size(local_keys)) &&
           (Key_compare(&cutoff,&List_key(local_keys,j))
                > 0)) {
        send_counts[0]++;
        j++;
    }

    /* Now deal with the remaining processes */
    for (i = 1; i < p; i++) {
        send_displacements[i] =
            send_displacements[i-1] + send_counts[i-1];
        send_counts[i] = 0;
        cutoff = Find_cutoff(i);
        /* Key_compare > 0 if cutoff > key */
        while ((j < List_size(local_keys)) &&
               (Key_compare(&cutoff,&List_key(local_keys,j))
                    > 0)) {
            send_counts[i]++;
            j++;
        }
    }
} /* Find_alltoall_send_params */
```

Notice that we have used an additional macro—List_key—and another function—Key_compare. List_key simply finds the *j*th key in the list. Key_compare returns positive if the first key is greater than the second, zero if they're equal, and negative if the first is less than the second. If you used the standard C function qsort, you had to write just such a function. We'll leave the writing of these to you.

A *serial* driver for Find_alltoall_send_params should allocate and initialize local_keys, sort the keys, print them, and print send_counts and send_displacements. We can use our routines, Allocate_list and Get_local_keys, to allocate and initialize. We can use the Local_sort function for sorting.

```
#include <stdio.h>
#include "sort.h"

int p;
int my_rank;

main() {
    LOCAL_LIST_T local_keys;
    int size;
    int* send_counts;
    int* send_displacements;
    int i, q;

    /* Ctrl-C to exit */
    while (1) {
        printf("Enter p, my_rank, and list size\n");
        scanf("%d %d %d", &p, &my_rank, &size);

        send_counts = (int*) malloc(p*sizeof(int));
        send_displacements =
            (int*) malloc(p*sizeof(int));

        /* Use p*size, since functions assume first
         * argument = global list size */
        Allocate_list(p*size, &local_keys);
        Get_local_keys(&local_keys);

        Local_sort(&local_keys);
        Find_alltoall_send_params(&local_keys,
            send_counts, send_displacements);

        printf("keys = ");
        for (i = 0; i < size; i++)
            printf("%d ",List_key(&local_keys,i));
        printf("\n");
```

```
            printf("counts = ");
            for (q = 0; q < p; q++)
                printf("%d ", send_counts[q]);
            printf("\n");

            printf("displacements = ");
            for (q = 0; q < p; q++)
                printf("%d ", send_displacements[q]);
            printf("\n\n");

            free(List(&local_keys));
            free(send_counts);
            free(send_displacements);
        }
    }
```

A run of the driver program produced the following output:

```
Enter p, my_rank, and list size
1 0 10
keys = 824 2495 8741 9255 10552 12096 16011 17326 20464 26067
counts = 10
displacements = 0

Enter p, my_rank, and list size
2 0 10
keys = 824 2495 8741 9255 10552 12096 16011 17326 20464 26067
counts = 7 3
displacements = 0 7

Enter p, my_rank, and list size
2 1 10
keys = 263 6436 9850 10838 10997 15681 19269 20264 21361 22597
counts = 6 4
displacements = 0 6

Enter p, my_rank, and list size
4 0 10
keys = 824 2495 8741 9255 10552 12096 16011 17326 20464 26067
counts = 2 5 2 1
displacements = 0 2 7 9
```

10.5.7 Finishing Up

Now there are essentially two functions left to write: `Find_recv_displacements` and `Get_list_size`. If you've understood `Find_alltoall_send_params`, you won't have any trouble writing `Find_recv_displacements`. However, `Get_list_size` brings up the thorny issue of access to `stdin`. If your system provides no access to `stdin`, you're either stuck

with the version we've just written, you can use command line arguments, or you can use a file for input. See Chapter 8 for details. If you do have access to `stdin`, you should be able to simply call the function `Cscanf` (see Chapter 8) as follows:

```
int size;

Cscanf(io_comm,"How big is the list?","%d", &size);
return size;
```

10.6 Summary

In this chapter we briefly looked at the design and coding of parallel programs. Parallel programs are often classified as *data-parallel* or *control-parallel* programs. As the name implies, data-parallel programs obtain parallelism by dividing the data among the processes, and each process executes more or less the same instructions on its subset of the data. Control-parallel programs divide the tasks we wish to carry out among the processes. So in a control-parallel program, the instructions executed by one process may be completely different from those executed by another. Most parallel programs use both approaches. However, since it is usually easier to write a data-parallel program that will perform well on many processes, the data-parallel parts of most programs tend to predominate. Roughly speaking, a program is said to be *scalable* if, by increasing the amount of data, we can continue to obtain good performance as the program uses more and more processes. Thus, data-parallel programs tend to scale better than control-parallel programs.

We illustrated the design of a data-parallel program by writing a function that solved a linear system using Jacobi's method. We started by studying the serial solution. We saw that we could easily parallelize the serial program by giving each process a copy of the scalar parameters (e.g., n and convergence tolerance) and by partitioning the arrays (coefficient matrix, right-hand side, and solution vector) among the processes. The steps in our parallel solution were almost identical to the steps in the serial solution. The two exceptions were that there was a communication phase in the parallel solution, and we slightly modified the organization of the solution so that we could reduce the amount of communication.

We illustrated the coding of a parallel program by writing a program that sorted a distributed list: when the program completed, the keys assigned to each process were sorted in increasing order, and if $A < B$, the keys assigned to process A were all less than or equal to the keys assigned to process B. Our approach to coding was modelled on standard approaches to coding serial programs. We used both top-down and bottom-up design and coding. Initially we proceeded top-down: we defined the main program and used stubs for the subprograms. However, as soon as we had proceeded far enough in our design that we had partially defined a data structure, we changed to bottom-up

coding: we defined output functions for the structure so that we could examine its contents during debugging, and we defined member access functions or macros so that if the design changed, we would only need to change the access functions.

Since input can be a major problem on parallel systems, we avoided the problem of dealing with input functions until the program was nearly complete.

The only new MPI functions we learned about were `MPI_Alltoall` and `MPI_Alltoallv`.

```
int MPI_Alltoall(
        void*         send_buffer  /* in  */,
        int           send_count   /* in  */,
        MPI_Datatype  send_type    /* in  */,
        void*         recv_buffer  /* out */,
        int           recv_count   /* in  */,
        MPI_Datatype  recv_type    /* in  */,
        MPI_Comm      comm         /* in  */)

int MPI_Alltoallv(
        void*         send_buffer           /* in  */,
        int           send_counts[]         /* in  */,
        int           send_displacements[]  /* in  */,
        MPI_Datatype  send_type             /* in  */,
        void*         recv_buffer           /* out */,
        int           recv_counts[]         /* in  */,
        int           recv_displacements[]  /* in  */,
        MPI_Datatype  recv_type             /* in  */,
        MPI_Comm      comm                  /* in  */)
```

Both functions scatter the contents of each process's `send_buffer` among the processes in `comm`. In the first, each process sends the same amount of data to every other process, while with the second, the amount of data sent to process *B* from *A* is specified by `send_counts[A]`, `send_displacements[A]`, and `send_type`.

10.7 References

We've just suggested a few of the issues in parallel software engineering. For a much more comprehensive treatment, see [17].

10.8 Exercises

1. Rewrite the parallel Jacobi function so that it no longer assumes that n is evenly divisible by p. What changes will have to be made in the calling program?

2. The C functions `drand48` and `srand48` can be used to generate random `doubles`. If you were using these functions to generate `doubles` or `floats` that were distributed across a set of processes, how would you try to insure that the numbers generated on distinct processes were independent?

10.9 Programming Assignments

1. Write a calling program for the parallel Jacobi function. Use your random number generator from exercise 2 to generate most of the entries in the matrix. What will you do for the diagonal entries?
2. If your system provides I/O access to process 0, write input and output functions to be used in the sorting program. Process 0 will read in the input list and distribute it by blocks to the processes. When the computation is complete, process 0 will gather the blocks of the list from the processes and print the sorted list.
3. Use `MPI_Alltoall` to write a matrix transpose function. Suppose the $n \times n$ matrix A is distributed by block rows among the processes and p divides n. After the operation is complete, A^T should also be distributed by block rows. Should your program overwrite A with A^T or use additional storage? What modifications need to be made to your program if you use a cyclic row distribution? What modifications need to be made if n isn't evenly divisible by p?
4. A two-dimensional binary cellular automaton consists of a two-dimensional rectangular grid together with an initial assignment of zeroes and ones to the vertices, or cells, of the grid and a "next-state" function for updating the values at the vertices. The updating function is applied simultaneously to all the cells. Typically, the value of the next-state function at an individual cell will depend on the current value at the grid point and the four neighbors immediately above, below, to the right, and to the left of the cell. However, it is also quite common for the next-state function to depend on the four diagonally adjacent neighbors as well—upper-left neighbor, upper-right neighbor, lower-left neighbor, and lower-right neighbor.

   John Conway's game of Life is a well-known example. In it, cells with ones are "alive," while cells with zeroes are dead. Updating corresponds to finding the "next generation," and the new value of a cell depends on all eight immediately adjacent neighbors. The rules are the following:

   a. Living cells with one or zero neighbors will die from loneliness.
   b. Living cells with two or three neighbors will survive into the next generation.
   c. Living cells with four or more neighbors will die from overcrowding.
   d. Empty cells with exactly three neighbors will come to life.

Write a parallel program that simulates a two-dimensional binary cellular automaton. Use Conway's Life as an example to clarify your thinking, but allow for the possibility that the update rule can be changed.

The program should take as input the order of the automaton (number of rows and number of columns), the maximum number of generations, and an initial configuration. After each new generation is computed, the program should print it out.

How should the cells of the automaton be partitioned among the processes? Should you use a block-row partitioning? A cyclic row partitioning? A block-checkerboard partitioning? How should this be decided?