Chapter 13

Parallel Processing

The FDTD method is said to be "trivially parallelizable," meaning that there are several simple ways in which the algorithm can be divided into tasks that can be executed simultaneously. For example, in a 3D simulation one might write an FDTD program that simultaneously updates the E_x, E_y, and E_z components of the electric field. These updates depend on the magnetic field and previous values of themselves—they are not a function of each other and hence can be updated in parallel. Then, H_x, H_y, and H_z might be updated simultaneously. Alternatively, one might divide the computational domain into distinct, non-overlapping regions and assign different processors to update the fields in those regions. This way fields in each of the regions could be updated simultaneously.

Here we will example two approaches to parallelizing a program. The threading approach typically use one computer to run a program. A threaded program is designed in such a way as to split the total computation between two or more "threads." If the computer has multiple processors, these threads can be executed simultaneously. Each of the threads can share the same memory space, i.e., the same set of variables. An alternative approach to parallelization uses the Message Passing Interface (MPI) protocol. This protocol allows different computers to run programs that pass information back and forth. MPI is ideally suited to partitioning the computational domain into multiple non-overlapping regions. Different computers are used to update the fields in the different regions. To update the fields on the interfaces with different regions, the computers have to pass information back and forth about the tangential fields along those interfaces.

In this chapter we provide some simple examples illustrating the use of threads and MPI.

13.1 Threads

There are different threading packages available. Perhaps the most common is the POSIX threads (pthreads) package. To use pthreads, you must include the header file `pthread.h` in your program. When linking, you must link to the pthread library (which is accomplished with the compiler flag `-lpthread`).

There are many functions related to pthreads. On a UNIX-based system on which pthreads are installed, a list of these functions can typically be obtained with the command `man -k pthread` and then one can see the individual man-pages to obtain details about a specific function.

Lecture notes by John Schneider. `parallel-processing.tex`

Despite all these functions, it is possible to do a great deal of useful programming using only two functions: `pthread_create()` and `pthread_join()`. As the name implies, a thread is created by the function `pthread_create()`. You can think of a thread as a separate process that happens to share all the variables and memory with the rest of your program. One of the arguments of `pthread_create()` will specify what this thread should do, specifically what function it should run.

The prototype of `pthread_create()` is:

```
int pthread_create(
  pthread_t *thread_id,        // ID number for thread
  const pthread_attr_t *attr, // controls thread attributes
  void *(*function)(void *),  // function to be executed
  void *arg                   // argument of function
);
```

The first argument is a pointer to the thread identifier (which is simply a number but we do not actually care about the details of how the ID is specified). This ID is set by `pthread_create()`, i.e., one would typically be interested in the returned value—it is not something that is set prior to `pthread_create()` being called.

The second argument is a pointer to a variable that controls the attributes of the thread. In this case, the value of this variable is established prior to the call of the function. This pointer can be set to `NULL` in which case the thread is created with the default attributes. Attributes control things like the "joinability" of the thread and the scheduling of threads. Typically one can simply use the default settings. The `pthread_t` and `pthread_attr_t` data types are defined in `pthread.h`.

The third and fourth arguments specify what function the thread should call and what argument should be passed to the function. Notice that the prototype says the function takes a void pointer as an argument and returns a void pointer. Keep in mind that "void" pointers are, in fact, simply generic pointers to memory. We can typecast these pointers to what they actually are. Thus, in practice, it would be perfectly acceptable for the function to take, for example, an argument of a pointer to a structure and return a pointer to a double. One would merely have to do the appropriate typecasting. If the function does not take an argument, the fourth argument of `pthread_create()` is set to `NULL`.

Once a new thread is created using `pthread_create()`, the program continues execution at the next command—the program does not wait for the thread to complete whatever the thread has been assigned to do. The function `pthread_join()` is used to block further execution of commands until the specified thread has completed. `pthread_join()` can also be used to access the return-value of the function that was run in a thread. The prototype of `pthread_join()` is:

```
int pthread_join(
  pthread_t thread_id, // ID of thread to "join"
  void **value_pntr    // address of function's return value
);
```

13.2 Thread Examples

To demonstrate the use of pthreads, let us first consider a standard serial implementation of a program where first one function is called and then another is called. The program is shown in Program 13.1.

Program 13.1 `serial-example.c`: Standard serial implementation of a program where first one function is called and then another. (These function are merely intended to perform a lengthy calculation. They do not do anything particularly useful.)

```
/* serial (i.e., non-threaded) implementation */
#include <stdio.h>

void func1();
void func2();

double a, b;  // global variables

int main() {

  func1();  // call first function
  func2();  // call second function

  printf("a: %f\n", a);
  printf("b: %f\n", b);

  return 0;
}

/* do some lengthy calculation which sets the value of the the global
   variable "a"*/
void func1() {
  int i, j;

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      a = 3.1456*j+i;

  return;
}

/* do another lengthy calculation which happens to be the same as
   done by func1() except here the value of global variable "b" is set
   */
void func2() {
  int i, j;
```

```

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      b = 3.1456*j+i;

  return;
}
```

In this program the functions `func1()` and `func2()` do not take any arguments nor do they explicitly return any values. Instead, the global variables `a` and `b` are used to communicate values back to the main function. Neither `func1()` nor `func2()` are intended to do anything useful. There are merely used to perform some lengthy calculation. Assuming the executable version of this program is named `serial-example`, the execution time can be obtained, on a typical UNIX-based system, by issuing the command "`time serial-example`".

Now, let us consider a threaded implementation of this same program. The appropriate code is shown in Program 13.2.

Program 13.2 `threads-example1.c`: A threaded implementation of the program shown in Program 13.1.

```
/* threaded implementation */
#include <stdio.h>
#include <pthread.h>

void *func1();
void *func2();

double a, b;

int main() {
  pthread_t thread1, thread2;  // ID's for threads

  /* create threads which run in parallel -- one for each function */
  pthread_create(&thread1, NULL, func1, NULL);
  pthread_create(&thread2, NULL, func2, NULL);

  /* wait for first thread to complete */
  pthread_join(thread1,NULL);
  printf("a: %f\n", a);

  /* wait for second thread to complete */
  pthread_join(thread2,NULL);
  printf("b: %f\n", b);

  return 0;
```

```
}

void *func1() {
  int i, j;

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      a = 3.1456*j+i;

  return NULL;
}

void *func2() {
  int i, j;

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      b = 3.1456*j+i;

  return NULL;
}
```

In Program 13.2 `func1()` and `func2()` are slightly different from the functions of the same name used in 13.1. In both these programs these functions perform the same calculations, but in 13.1 these functions returned nothing. However `pthread_create()` assumes the function returns a void pointer (i.e., a generic pointer to memory). Since in this example these functions do not need to return anything, they simply return `NULL` (which is effectively zero).

If Program 13.2 is run on a computer that has two (or more) processors, one should observe that the execution time (as measured by a "wall clock") is about half of what it was for Program 13.2. Again, assuming the executable version of Program 13.2 is named `threads-example1`, the execution time can be obtained by issuing the command "`time threads-example1`". This timing command will typically return three values: the "wall-clock" time (the actual time that elapsed from the start to the completion of the program), the CPU time (the sum of time spent by all processors used to run the program), and system time (time used by the operating system to run things necessary for your program to run, but not directly associated with your program). You should observe that ultimately nearly the same amount of CPU time was used by both the threaded and serial programs but the threaded program required about half the wall-clock time. In the case of the second program two processors were working simultaneously and hence the wall-clock time was half as much, or nearly so. In fact, there is slightly more computation involved in the threaded program than the serial program since there is some computational overhead associated with the threads.

Let us now modify the first function so that it returns a value, specifically a pointer to a double where we simply store an arbitrary number (in this case 10.0). The appropriate code is shown in Program 13.3.

Program 13.3 `threads-example2.c`: Modified version of Program 13.2 where now `func1()` has a return value.

```
/* threaded implementation -- returning a value */
#include <stdio.h>
#include <stdlib.h> // needed for malloc()
#include <pthread.h>

double *func1();  // now returns a pointer to a double
void *func2();

double a, b;

int main() {
  double *c; // used for return value from func1
  pthread_t thread1, thread2; // ID's for threads

  // typecast the return value of func1 to a void pointer
  pthread_create(&thread1, NULL, (void *)func1, NULL);
  pthread_create(&thread2, NULL, func2, NULL);

  // typecast the address of c to a void pointer to a pointer
  pthread_join(thread1,(void **)&c);
  printf("a,c: %f %f\n", a, *c);

  pthread_join(thread2,NULL);
  printf("b: %f\n", b);

  return 0;
}

double *func1() {
  int i, j;

  double *c;  // c is a pointer to a double

  // allocate space to store a double
  c=(double *)malloc(sizeof(double));
  *c = 10.0;

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      a = 3.1456*j+i;

  return c;
}

```

```
45 void *func2() {
46   int i, j;
47
48   for (j=0;j<4000;j++)
49     for (i=0;i<1000000;i++)
50       b = 3.1456*j+i;
51
52   return NULL;
53 }
```

Note that in this new version of `func1()` we declare `c` to be a pointer to a double and then, in line 35, allocate space where the double can be stored and then, finally, store 10.0 at this location. This is rather complicated and it might seem that a simpler approach would be merely to declare `c` to be a double and then return the address of `c`, i.e., end the function with `return &c`. Unfortunately this would not work. The problem with that approach is that declaring `c` to be a double would make it a local variable (one only known to `func1()`) whose memory would disappear when the function returned.

The second argument of `pthread_join()` in line 20 provides the pointer to the return value of the function that was executed by the thread. Since `c` by itself is a pointer to a double, `&c` is a pointer to a pointer to a double, i.e., of type `(double **)`. However, `pthread_join()` assumes the second argument is a void pointer to a pointer and hence a typecast is used to keep the compiler from complaining.

In the next example, shown in Program 13.4, `func1()` and `func2()` are modified so that they each take an argument. These arguments are the double variables `e` and `d` that are set in `main()`.

Program 13.4 `threads-example3.c`: Functions `func1()` and `func2()` have been modified so that they now take arguments.

```
1  /* threaded implementation -- passing arguments and
2     returning a value */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <pthread.h>
6
7  double *func1(double *);
8  void *func2(double *);
9
10 double a, b;
11
12 int main() {
13   double *c; // used for return value from func1
14   double d=3.0, e=2.0; // arguments passed to functions
15   pthread_t thread1, thread2; // ID's for threads
```

```
16
17   pthread_create(&thread1, NULL, (void *)func1, (void *)&d);
18   pthread_create(&thread2, NULL, (void *)func2, (void *)&e);
19
20   pthread_join(thread1,(void **)&c);
21   printf("a,c: %f %f\n", a, *c);
22
23   pthread_join(thread2,NULL);
24   printf("b: %f\n", b);
25
26   return 0;
27 }
28
29 double *func1(double *arg) {
30   int i, j;
31
32   double *c;
33
34   c=(double *)malloc(sizeof(double));
35   *c = 10.0;
36
37   for (j=0;j<4000;j++)
38     for (i=0;i<1000000;i++)
39       a = (*arg)*j+i;
40
41   return c;
42 }
43
44 void *func2(double *arg) {
45   int i, j;
46
47   for (j=0;j<4000;j++)
48     for (i=0;i<1000000;i++)
49       b = (*arg)*j+i;
50
51   return NULL;
52 }
```

In all these examples `func1()` and `func2()` have performed essentially the same computation. The only reason there were two separate functions is that `func1()` set the global variable `a` while `func2()` set the global variable `b`. However, knowing that we can both pass arguments and obtain return values, it is possible to have a single function in our program. It can be called multiple times and simultaneously. Provided the function does not use global variables, the different calls will not interfere with each other.

A program that uses a single function to accomplish what the previous programs used two function for is shown in Program 13.5.

Program 13.5 `threads-example4.c`: The global variables have been removed and a single function `func()` is called twice. The function `func()` and `main()` communicate by passing arguments and checking returns values (instead of via global variables).

```
/* threaded implementation -- passing an argument and checking
   return the value from a single function */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

double *func(double *);

int main() {
  double *a, *b; // used for return values
  double d=3.0, e=2.0;
  pthread_t thread1, thread2; // ID's for threads

  pthread_create(&thread1, NULL, (void *)func, (void *)&d);
  pthread_create(&thread2, NULL, (void *)func, (void *)&e);

  pthread_join(thread1,(void **)&a);
  printf("a: %f\n", *a);

  pthread_join(thread2,(void **)&b);
  printf("b: %f\n", *b);

  return 0;
}

double *func(double *arg) {
  int i, j;

  double *a;

  a=(double *)malloc(sizeof(double));

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      *a = (*arg)*j+i;

  return a;
}
```

Threads provide a simple way to obtain parallelization. However, one may find that in practice they do not provide the benefits one might expect when applied to FDTD programs. FDTD is

both computational expensive and memory-bandwidth intensive. A great deal of data must be passed between memory and the CPU. Often the bottleneck is not the CPU but rather the speed of the "bus" that carries data between memory and the CPU. Multi-processor machines do not have multiple memory busses. Thus, splitting an FDTD computation between multiple CPU's on the same computer will have those CPU's all requesting memory from a bus that is already acting at full capacity. These CPU's will have to wait on the arrival of the requested memory. Therefore, in practice when using threaded code with N threads on a computer with N processors, one is unlikely to see a computation-time reduction that is anywhere close to the hypothetical maximum reduction of $1/N$.

13.3 Message Passing Interface

The message passing interface (MPI) is a standardized protocol, or set of protocols, which have been implemented on a wide range of platforms. MPI facilitates the communication between processes whether they are running on a single host or multiple hosts. As with pthreads, MPI provides a large number of functions. These functions allow the user to control many aspects of the communication or they greatly simplify what would otherwise be quite cumbersome tasks (such as the efficient distribution of data to a large number of hosts). Despite the large number of MPI functions, just six are needed to begin exploiting the benefits of parallelization.

Before considering those six functions, it needs to be said that one must have the supporting MPI framework installed on each of the hosts to be used. Different implementations of the MPI protocol (or the MPI 2 protocol) are available from the Web. For example, LAM MPI is available from `www.lam-mpi.org` but it is no longer being actively developed. Instead, several MPI-developers have joined together to work on OpenMPI which is available from `www.open-mpi.org`. Alternatively, MPICH2 is available from `www.mcs.anl.gov/research/projects/mpich2`. Installation of these packages is relatively trivial (at least that is the case when using Linux or Mac OS X), but some of the details associated with getting jobs to run can be somewhat complicated (for example, ensuring that access is available to remote machines without requiring explicit typing of a password).

There is a great deal of MPI documentation available from the Web (there are also a few books written on the subject). A good resource for getting started is `computing.llnl.gov/tutorials/mpi/`. In fact, Lawrence Livermore has many useful pages related to threads, MPI, and other aspects of high-performance and parallel processing. You can find the material by going to `computing.llnl.gov` and following the link to "Training."

Returning to the six functions needed to use MPI in a meaningful way, two of them concern initializing and closing the MPI set-up, two deal with sending and receiving information, and two deal with determining the number of processes and which particular process number is associated with the given invocation. Programs which use MPI must include the header file `mpi.h`. Before any other MPI functions are called, the function `MPI_Init()` must be called. The last MPI function called should be `MPI_Finalize()`. Thus, a valid, but useless, MPI program is shown in Program 13.6.

Program 13.6 `useless-mpi.c`: A trivial, but valid, MPI program.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  MPI_Init(&argc,&argv);

  printf("I don't do anything useful yet.\n");

  MPI_Finalize();

  return 0;
}
```

When an MPI program is run, each process runs the same program. In this case, there is nothing to distinguish between the processes. They will all generate the same line of output. If there were 100 processes, you would see 100 lines of "`I don't do anything useful yet.`"

13.4 Open MPI Basics

At this point we need to ask the question: What does it mean to run an MPI program? Ultimately many copies of the same program are run. Each copy may reside on a different computer and, in fact, multiple copies may run on the same computer. The details concerning how one gets these copies to run are somewhat dependent on the MPI package one uses. Here we will briefly describe the steps associated with the Open MPI package.

When Open MPI is installed, several executable files will be placed on your system(s), e.g., `mpicc`, `mpiexe`, `mpirun`, etc. On most systems by default these files will be installed in the directory `/usr/local/bin` (but one can specify that the files should be installed elsewhere if so desired). One must ensure that the directory where these executable reside is in the search path.

When using MPI it is usually necessary to compile the source code in a special way. Instead of using the `gcc` compiler on UNIX/Linux machines, one would use `mpicc` (`mpicc` is merely a wrapper that ultimately calls the underlying compiler that one would have used normally). So, for example, to compile the MPI program given above, one would issue a command such as

```
mpicc -Wall -O useless-mpi.c -o useless-mpi
```

One can now run the executable file `useless-mpi`. The command to do this is either `mpirun` or `mpiexec` (these commands are synonymous). There are numerous arguments that can be specified with the most important being the number of processes. The following command says to run four copies of `useless-mpi`:

```
mpiexec -np 4 useless-mpi
```

But where, precisely, is this run? In this case four copies of the program are run on the local host. That is not precisely what we want—we are interested in distributing the job to different machines. There are multiple ways in which one can exercise control of the running of MPI programs and we will explore just a few.

First, let us assume a "multicomputer" consists of five nodes with names `node01`, `node02`, `node03`, `node04`, and `node05`. To make things more interesting, let us further assume that `node01` is one particular brand of computer and the other nodes are a different brand (e.g., perhaps `node01` is an Intel-based machine while the other nodes are PowerPC-based machines). Additionally assume that `node01` has four processors while each of the other nodes has two processors.

We can specify some of this information in a "hostfile." For now, let us exclude `node01` since it is a different architecture. A hostfile that describes a multicomputer consisting of the remaining four nodes might be

```
node05 slots=2
node04 slots=2
node03 slots=2
node02 slots=2
```

Let us assume this information is stored in the file `my_hostfile`. The `slots` are the number of processors on a particular machine. If one does not specify the number of slots, it is assumed to be one.

Let us further assume the executable `useless-mpi` exists in a director called `~/Ompi` (where the tilde is recognized as a shorthand for the user's home directory on a UNIX/Linux machine). If all the computers mount the same file structure, this may actually be the exact same directory that all the machines are sharing. In that case there would only be one copy of `useless-mpi`. Alternatively, each of the computers may have their own local copy of a directory named `~/Ompi`. In that case there would have to be a local copy of the executable file `useless-mpi` present on each of the individual computers.

One could now run eight copies of the program by issuing the following command:

```
mpirun -np 8 -hostfile my_hostfile ~/Ompi/useless-mpi
```

This command could be issued from any of the nodes. Note that the number of processes does not have to match the number of slots. The following command will launch 12 copies of the program

```
mpirun -np 12 -hostfile my_hostfile ~/Ompi/useless-mpi
```

However, it will generally be best if one can match the job to the physical configuration of the multicomputer, i.e., one job per "slot."

In order to incorporate `node01` into the multicomputer, things become slightly more complicated because executables compiled for `node01` will not run on the other nodes and vice versa. Thus one must compile separate versions of the program on the different machines. Let's assume that was done and on each of the nodes a copy of `useless-mpi` was place in the local directory `/tmp` (i.e., there is a copy of this directory and this executable on each of the nodes). The hostfile `my_hostfile` could then be changed to

```
node05 slots=2
node04 slots=2
node03 slots=2
node02 slots=2
node01 slots=4
```

Note that there are four slots specified for `node01` instead of two. The command to run 12 copies of the program would now be

```
mpirun -np 12 -hostfile my_hostfile /tmp/useless-mpi
```

By introducing more arguments to the command line, one can exercise more fine-grained control of the execution of the program. Let us again assume that there is one common directory `~/Ompi` that all the machines share. Let us further assume two version of `useless-mpi` have been compiled: one for PowerPC-based machines called `useless-mpi-ppc` and one for Intel-based machines called `useless-mpi-intel`. We can use a command that does away with the hostfile and instead provide all the details explicitly:

```
mpirun -host node05,node04,node03,node02 \
       -np 8 ~/Ompi/useless-mpi-ppc \
       -host node01 -np 4 ~/Ompi/useless-mpi-intel
```

Note that the backslashes here are quoting the end of the line. This command can be given on a single line or can be given on multiple lines, as shown here, if one "quotes" the carriage return.

13.5 Rank and Size

To do more meaningful tasks, it is typically necessary for each processor to know how many total processes there are and which process number is assigned to a particular invocation. In this way, each processor can do something different based on its process number. In MPI the process number is known as the rank. The number of processes can be determined with the function `MPI_Comm_size()` and the rank can be determined with `MPI_Comm_rank()`. The code shown in Program 13.7 is a slight modification of the previous program that now incorporates these functions.

Program 13.7 `find-rank.c`: An MPI program where each process can determine the total number of processes and its individual rank (i.e., process number).

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5   int rank, size;
6
7   MPI_Init(&argc,&argv);
```

```

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("I have rank %d out of %d total.\n",rank,size);

  MPI_Finalize();

  return 0;
}
```

Assume this is run with four total processes. The output will be similar to this:

```
I have rank 1 out of 4 total.
I have rank 0 out of 4 total.
I have rank 2 out of 4 total.
I have rank 3 out of 4 total.
```

Note that the `size` is 4, but `rank` ranges between 0 and 3 (i.e., `size` $- 1$). Also note that there is no guarantee that the processes will report in rank order.

The argument `MPI_COMM_WORLD` is known as an MPI communicator. A communicator essentially specifies the processes which are grouped together. One can create different communicators, i.e., group different sets of processes together, and this can simplify handling certain tasks for a particular problem. However, we will simply use `MPI_COMM_WORLD` which specifies all the processes.

13.6 Communicating Between Processes

To communicate between processes we can use the commands `MPI_Send()` and `MPI_Recv()`. `MPI_Send()` has arguments of the form:

```
MPI_Send(&buffer, // address where data stored
         count,   // number of items to send
         type,    // type of data to send
         dest,    // rank of destination process
         tag,     // programmer-specified ID
         comm);   // MPI communicator
```

where `buffer` is an address where the data to be sent is stored (for example, the address of the start of an array), `count` is the number of elements or items to be sent, `type` is the type of data to be sent, `dest` is the rank of the process to which this information is being sent, `tag` is a programmer-specified number to identify this data, and `comm` is an MPI communicator (which we will leave as `MPI_COMM_WORLD`). The `type` is similar to the standard C data types, but it is specified using MPI designations. Some of those are: `MPI_INT`, `MPI_FLOAT`, and `MPI_DOUBLE`, corresponding to the C data types of int, float, and double (other types, some of which are specific to MPI, such as `MPI_BYTE` and `MPI_PACKED`, exist too).

`MPI_Recv()` has arguments of the form:

```
MPI_Recv(&buffer,count,type,source,tag,comm,&status);
```

In this case `buffer` is the address where the received data is to be stored. The meaning of `count`, `type`, `tag`, and `comm` are unchanged from before. `source` is the rank of the process sending the data. The `status` is a pointer to a structure, specifically an `MPI_status` structure which is specified in `mpi.h`. This structure contains the rank of the source and the `tag` number.

Program 13.8 demonstrates the use of `MPI_Send()` and `MPI_Recv()`. Here the process with rank 0 serves as the master process. It collects input from the user which will subsequently be sent to the other processes. Specifically, the parent process prompts the user for as many values (doubles) as there are number of processes minus one. The master process then sends one number to each of the other processes. These processes do a calculation based on the number they receive and then send the result back to the master. The master prints this received data and then the program terminates.

Program 13.8 `sendrecv.c`: An MPI program that sends information back and forth between a master process and slave processes.

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i, rank, size, tag_out=10, tag_in=11;
  MPI_Status status;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0) {
    /* "master" process collects and distributes input */
    double *a, *b;

    /* allocate space for input and result */
    a=malloc((size-1)*sizeof(double));
    b=malloc((size-1)*sizeof(double));

    /* prompt user for input */
    printf("Enter %d numbers: ",size-1);
    for (i=0; i<size-1; i++)
      scanf("%lf",a+i);

    /* send values to other processes */
    for (i=0; i<size-1; i++)
      MPI_Send(a+i,1,MPI_DOUBLE,i+1,tag_out,MPI_COMM_WORLD);
```

```
30
31      /* receive results calculated by other process */
32      for (i=0; i<size-1; i++)
33        MPI_Recv(b+i,1,MPI_DOUBLE,i+1,tag_in,MPI_COMM_WORLD,&status);
34
35      for (i=0; i<size-1; i++)
36        printf("%f\n",b[i]);
37
38    } else {
39      /* "slave" process */
40      int j;
41      double c, d;
42
43      /* receive input from the master process */
44      MPI_Recv(&c,1,MPI_DOUBLE,0,tag_out,MPI_COMM_WORLD,&status);
45
46      /* do some silly number crunching */
47      for (j=0;j<4000;j++)
48        for (i=0;i<100000;i++)
49          d = c*j+i;
50
51      /* send the result back to master */
52      MPI_Send(&d,1,MPI_DOUBLE,0,tag_in,MPI_COMM_WORLD);
53    }
54
55    MPI_Finalize();
56
57    return 0;
58  }
```

The six commands covered so far are sufficient to parallelize any number of problems. However, there is some computational overhead associated with parallelizing the code. Additionally, there is often a significant cost associated with communication between processes, especially if those processes are running on different hosts and the network linking those hosts is slow.

The functions `MPI_Send()` and `MPI_Recv()` are blocking commands. They do not return until they have accomplished the requested send or receive. In some cases, especially if there is a large amount of data to transmit, this can be costly. There are also nonblocking or "immediate" versions of these functions. For these functions control is returned to the calling function without a guarantee of the send or receive having been accomplished. In this way the program can continue some other useful task while the communication is taking place. When one must ensure that the communication is finished, the function `MPI_Wait()` provides a blocking mechanism that suspends execution until the specified communication is completed. The immediate send and receive functions are of the form:

```
MPI_Isend(&buffer,count,type,dest,tag,comm,&request);
MPI_Irecv(&buffer,count,type,source,tag,comm,&request);
```

The arguments to these functions are the same as the blocking version except the final argument is now a pointer to an `MPI_Request` structure instead of an `MPI_Status`. The wait command has the following form:

```
MPI_Wait(&request,&status);
```

Note that the communication for which the waiting is being done is specified by the "`request`," not the "`status`." So, if there are multiple transmissions which are being done asynchronously, one may have to create an array of `MPI_Request`'s. If one is not concerned with the status of the transmissions, one does not have to define a separate status for each transmission.

The code shown in Program 13.9 illustrates the use of non-blocking send and receive. In this case the master process sends the numbers to the other processes via `MPI_Isend()`. However, the master does not bother to ensure that the send was performed. Instead, the master will ultimately wait for the other process to communicate the result back. The fact that the other processes are sending information back serves as confirmation that the data was sent from the master. After sending the data, the master process then calls `MPI_Irecv()`. There is one call for each of the "slave" processes. After calling these functions, `MPI_Wait()` is used to ensure the data has been received before printing the results. The code associated with the slave processes is unchanged from before.

Program 13.9 `nonblocking.c`: An MPI program that uses non-blocking sends and receives.

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i, rank, size, tag_out=10, tag_in=11;
  MPI_Status status;
  MPI_Request *request_snd, *request_rcv;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0) {
    /* "master" process collects and distributes input */
    double *a, *b;

    /* allocate space for input and result */
    a=malloc((size-1)*sizeof(double));
    b=malloc((size-1)*sizeof(double));

    /* allocate space for the send and receive requests */
    request_snd=malloc((size-1)*sizeof(MPI_Request));
```

```
    request_rcv=malloc((size-1)*sizeof(MPI_Request));

    /* prompt user for input */
    printf("Enter %d numbers: ",size-1);
    for (i=0; i<size-1; i++)
      scanf("%lf",a+i);

    /* non-blocking send of values to other processes */
    for (i=0; i<size-1; i++)
      MPI_Isend(a+i,1,MPI_DOUBLE,i+1,tag_out,MPI_COMM_WORLD,request_snd+i);

    /* non-blocking reception of results calculated by other process */
    for (i=0; i<size-1; i++)
      MPI_Irecv(b+i,1,MPI_DOUBLE,i+1,tag_in,MPI_COMM_WORLD,request_rcv+i);

    /* wait until we have received all the results */
    for (i=0; i<size-1; i++)
      MPI_Wait(request_rcv+i,&status);

    for (i=0; i<size-1; i++)
      printf("%f\n",b[i]);

  } else {
    /* "slave" process */
    int j;
    double c, d;

    /* receive input from the master process */
    MPI_Recv(&c,1,MPI_DOUBLE,0,tag_out,MPI_COMM_WORLD,&status);

    /* do some silly number crunching */
    for (j=0;j<4000;j++)
      for (i=0;i<100000;i++)
        d = c*j+i;

    /* send the result back to master */
    MPI_Send(&d,1,MPI_DOUBLE,0,tag_in,MPI_COMM_WORLD);
  }

  MPI_Finalize();

  return 0;
}
```

Compared to the previous version of this program, this version runs over 30 percent faster on a dual-processor G5 when using five processes. (By “over 30 percent faster” is meant that if

the execution time for the previous code is normalized to 1.0, the execution time using the non-blocking calls is approximately 0.68.)