Chapter 4 Message-Passing Programming

"Perilous to us all are the devices of an art deeper than we possess ourselves. " Gandalf, in Lord of the Rings, Part II: The Two Towers [1]

4.1 Introduction

This chapter begins our study of parallel programming using a message-passing model. The advantage of using a message-passing model, rather than a shared memory model, as a starting point, is that the message-passing model can be used on any model of multicomputer, whether it is a shared memory multiprocessor or a private memory multicomputer. The next decision we have to make is which parallel programming language we choose for implementing the algorithms we will develop. The ***Message Passing Interface*** (***MPI***) standard is very widely adopted, and this is our choice. It is not a language, but a library of functions that add a message-passing model of parallel programming to ordinary sequential languages like *C*, *C*++, and Fortran. ***OpenMPI*** is a free version of MPI. MPI is available on most commercial parallel computers, making programs that use the standard very portable.

In this chapter, we will introduce the following MPI functions:

MPI Function	**Purpose**
`MPI_Init`	initializes MPI
`MPI_Comm_rank`	determines a process's ID number (called its rank)
`MPI_Comm_size`	determines the number of processes
`MPI_Reduce`	performs a reduction operation
`MPI_Finalize`	shuts down MPI and releases resources
`MPI_Bcast`	performs a broadcast operation
`MPI_Barrier`	performs a barrier synchronization operation
`MPI_Wtime`	determines the "wall" time
`MPI_Wtick`	determines the length of a clock tick

4.2 About *C* and *C*++

We will almost exclusively describe only the *C* syntax of MPI in these notes; sometimes we will show the *C*++ syntax as well. Although you can use *C*++ with MPI Versions up to 2.2, it became deprecated as of Version 2.2 and is not supported in Version 3.0. What this means is that you can write *C*++ programs that use MPI, but they cannot use the *C*++ binding; they must call functions using their *C* syntax. If you intend to program in *C*++, you can visit the MPI online documentation, `http://www.open-mpi.org/doc/v1.6/`, for the *C*++ syntax. However, you should be aware that there are more than just syntactic differences between the *C* and *C*++ library bindings, and it is advised that you spend some time familiarizing yourself with those differences. The *C*++ binding is an object-oriented one, and the error handling is also different. For example, almost all MPI routines return an error value; *C* routines do so as the return value of the MPI function but *C*++ functions do not return errors. If the default error handler is set to `MPI::ERRORS_THROW_EXCEPTIONS`, then on error the *C*++ exception mechanism will be used to throw an `MPI:Exception` object.

As the textbook is based on *C*, all of the code in these notes is likewise in *C*.

4.3 The Message-Passing Model

The message-passing parallel programming model is based on the idea that processes communicate with each other through messages, rather than by accessing shared variables. The underlying assumption is that each process runs on a processor that has a local, private memory, and that the processors on which the processes run are connected by an interconnection network that supports the exchange of messages between every pair of processes. In this sense there is a natural correspondence between the task/channel model from Chapter 3 and the message-passing model: a task is a process in the model, and the interconnection network provides an implicit channel between every pair of tasks. In short, it is as if there is a fully-connected graph topology connecting all tasks to each other, even if our programs do not use all of the channels in that graph.

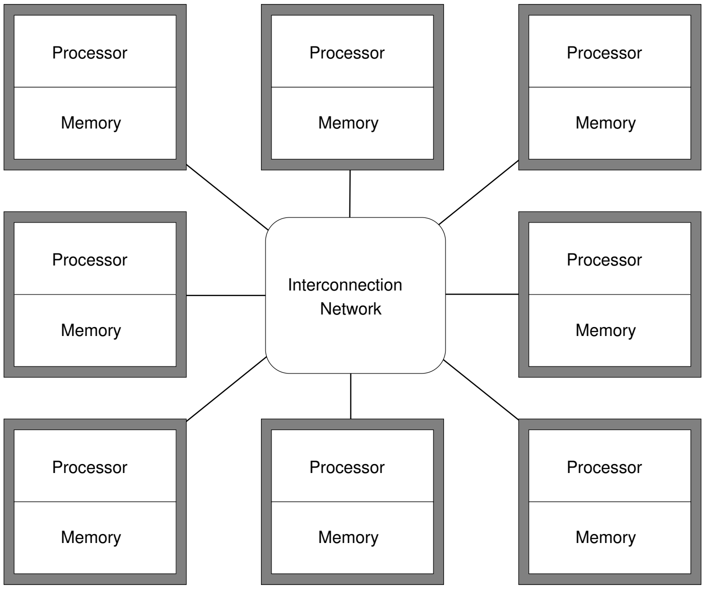


Figure 4.1: The message-passing model's hardware model. It treats it as if there are independent processors with their own local memories, connected by an interconnection network.

MPI-1 imposes restrictions on the style of programming:

- We cannot create new processes on the fly; all processes have to be created at the start of the program. MPI-2 allows dynamic process creation.
- *All processes execute the same program, but they each have a unique identification number, so if we really want some of the them to execute different code, they can branch to different code conditionally based on their ID.* This is sometimes called a ***Single Program Multiple Data (SPMD)*** paradigm.

Messages are used not just to exchange data between processes, but to synchronize as well. The receive operation is a synchronous primitive; when a process calls it, it does not return until some other process has sent a message to the calling process. Therefore, it can be used to synchronize the execution of processes with each other. As Quinn puts it, " even a message with no content has a meaning."[2]

The message-passing model has two benefits over the shared memory model. One is that, the way that message-passing programs are designed, there are two methods of data access: accesses to the process's local address space, and communication through messages. The local data accesses tend to result in high cache hit rates because of spatial locality, which improves performance. Secondly, because the message-passing programmer knows that communication costs are a drag on performance, this also leads to a style of programming that can improve performance, as compared to programs that use shared variables indiscriminately.

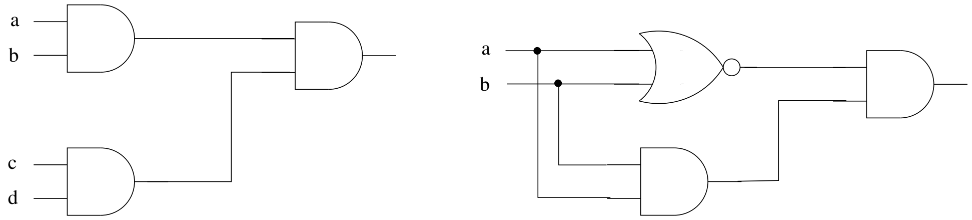
The second benefit is that debugging message-passing programs is easier than debugging shared memory programs. This is because there are almost no ***race conditions*** in message-passing programs. A race condition exists in a system of two or more processes when the state of the system is dependent on the order in which these processes execute a sequence of instructions. Because the message-passing library provides a form of synchronization, the non-deterministic behavior associated with shared memory parallel programs can generally be avoided. Of course a programmer that works hard enough can create them anyway!

4.4 The MPI Story

Before we begin, we should know a bit about the where and why of MPI. Before MPI, there were no standards. Different parallel computer vendors were providing different libraries for message-passing on their systems. By the early 1990's, there were several different and incompatible interfaces such as *Intel NX*, *IBM-EUI/CCL*, *PVM*, *P4*, and *OCCAM*. This made it impossible to port from one system to another without rewriting significant amounts of code. In April 1992, the *Center for Research on Parallel Computing* sponsored the *Workshop on Standards for Message Passing in a Distributed Memory Environment* in Williamsburg, Virginia. This workshop drew representatives from academia, industry, and the government. A working group called the *Message Passing Interface Forum* was formed to develop a draft standard, and in November 1992, the first draft standard, dubbed *MPI*, was produced. This draft *MPI* standard was presented at the *Supercomputing '93* conference in November 1993. It subsequently underwent a revision, and *MPI-2* was adopted in 1997. A third version, *MPI-3,* was completed and MPI-3.0 was adopted in September 2012.

4.5 Circuit Satisfiability

A ***decision problem*** is a problem that has a yes/no answer, such as, given a graph, does it have a cycle. A solution to a decision problem is an algorithm that, for all possible instances of the problem, outputs yes or no correctly. The ***circuit satisfiability problem*** is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true. The circuit may contain any number of gates. If there exists an assignment of values to the inputs that makes the output of the circuit true, it is called a ***satisfiable*** circuit; if not it is called ***unsatisfiable***. This problem has been proven to be ***NP-complete***. For those unfamiliar with the meaning of NP-completeness, it means roughly that there are no known deterministic polynomial-time algorithms that can solve all instances of this problem (that is the NP part of the term), and that if one were found, then all other problems that are in this class called NP would also have deterministic polynomial time algorithms (that is the "complete" part of the term.) [1]



(a) A satisfiable circuit.

(b) An unsatisfiable circuit.

Figure 4.2: Two boolean circuits. Circuit (a) can be satisfied by the assignment a=b=c=d=1, but no assignment can make the output of (b) 1.

Some circuits are satisfiable and others are not. Figure 4.2 demonstrates this fact with two very small circuits. For a circuit with N independent inputs, there are a total of 2^N possible combinations of these inputs. A brute force way of solving this problem is to try every possible combination of these 2^N inputs, equivalent to building a truth table for the circuit and searching for a row of the truth table that has a true output value. For small values of N, this is feasible, but not for large values. For example, the circuit in Figure 4.3 has 32 inputs and therefore $2^{32} = 4,294,967,296$ possible input combinations.

[1] This is not a technically correct definition, but it is how you should think of the meaning of the term.

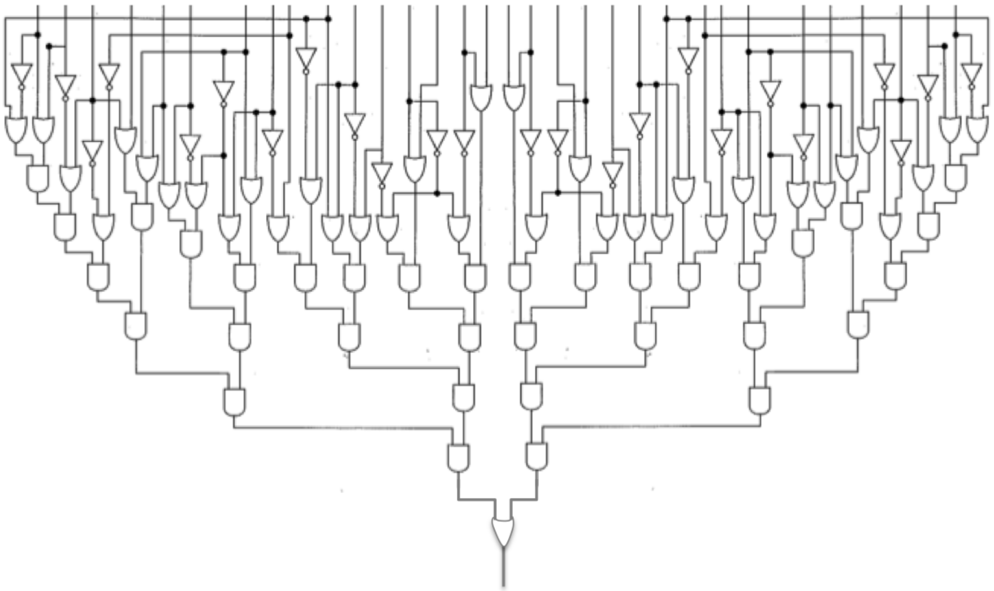


Figure 4.3: A 32-bit circuit.

Although there are no polynomial-time algorithms that solve this problem, there are intelligent algorithms that solve it efficiently, meaning in time $O(c^N)$, where $1 < c < 2$. These are still exponential running time of course. Because our goal in this chapter is to develop a parallel algorithm, we base it upon the brute force sequential algorithm, as it is a very simple one and is therefore a good starting point for us. Let us assume that the input size is 16, as Quinn does [2].

The brute force sequential algorithm is of the form

```
// for each 16-bit non-negative integer n
for ( int n = 0; n < 65536; n++)
    if the bits of n satisfy the circuit
        output ‘‘satisfiable’’ and stop
```

This algorithm solves the decision problem, but we will modify it slightly so that, instead of just reporting whether it is a satisfiable circuit or not, it prints out every integer that satisfies the circuit. Therefore, the starting point will be the following sequential algorithm:

```
// for each 16-bit non-negative integer n
for ( int n = 0; n < 65536; n++)
    if the bits of n satisfy the circuit
        output n
```

4.5.1 Partitioning

It should be fairly obvious that there is data parallelism in this computation because each input is independent and checking may be performed in parallel. Thus, for the partitioning step of Foster's design methodology, we should associate a primitive task with each input integer. Each such task would check whether that integer satisfies the circuit, and if it does, it will print the integer as a binary number.

The task/channel graph is thus a sequence of adjacent nodes with no channels between them, as shown in Figure 4.4. This is sometimes called an ***embarrassingly parallel problem***. Each task may produce output, so it will be connected to the output device by an I/O channel.

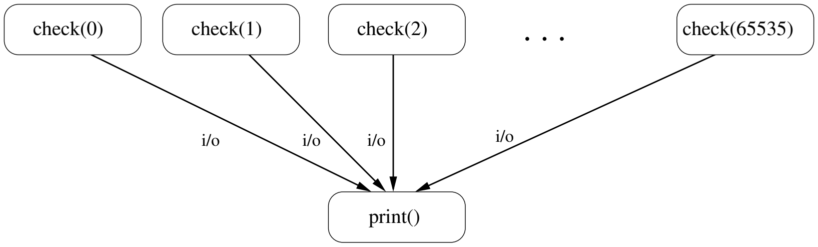


Figure 4.4: Task/channel graph after partitioning for circuit satisfiability.

4.5.2 Agglomeration and Mapping

There is a fixed number of tasks and no communication among them, so we follow the leftmost branches of Foster's decision tree (Chapter 3). The next question is whether all tasks spend the same amount of time in their computations. In general, they can spend vastly different amounts of time computing. It depends on how quickly they find an and-gate that has a 0 output. For example, if you look at the leftmost two input bits in the circuit in Figure 4.3, you see that the pattern 00 will drive the circuit to a 0 output very quickly, so all tasks that have those two leading 0's might finish quickly, whereas other may have to check all gate outputs. Because tasks do not have an even computational load, we follow the right branch of the last decision node and decide to cyclically map tasks to processors to balance the computation load.

To reduce overhead, we assign one task per processor. Thus we are cyclically mapping numbers to processors. Assume there are p processors and n numbers. A cyclic interleaving of the numbers to the processors is like a round-robin scheduling: the first p numbers are given to tasks 0, 1, 2, ..., $p-1$ and the next p numbers are given to those same tasks again, so that task k gets numbers k, $k+p$, $k+2p$, $k+3p$, ... up to the largest number $k+mp$ that is less than $n-1^2$. If n is a multiple of p, then all tasks have the same number of computations to perform. If not, then some will do one more round than others. You may wonder why this distribution of data items to tasks balances the load. The simple explanation is that the amount of computation required for two adjacent bit combinations will tend to be the same, so assigning them to different tasks tends to even out the load.

4.5.3 Coding the Solution in *C* Using *MPI*

Every program that uses MPI has to start by including the *MPI header file* `<mpi.h>`. This particular program also writes to standard output using the *C* library's `printf()` function, so we start the program with

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

Because MPI must be initialized by calling `MPI_Init`, which must be passed the program's command line parameters, the main program must declare them in its header:

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

Every process will execute its own copy of this main program. All variables declared in this program, including the automatic variables declared within any function block, or global variables declared outside all function scopes, will be private in each process. The complete program appears in Listing 4.1. It implements checking of the circuit shown in Figure 4.5. We explain the important components of this program after the listing.

Listing 4.1: circuitsat1.c

```
/*
```

[2] Another way to see this is that number j is assigned to task $j \, mod \, p$.

```
  Title          : circuitsat1.c
  Author         : Michael J. Quinn, modified by Stewart Weiss
  Created on     : January  6, 2014
  Description    : MPI program to check satisfiability of a boolean circuit

  Notes:
  This is version 1 of the circuit satifiability program.
  This is an MPI program that checks whether a boolean circuit is satisfiable.
  The boolean circuit is hard-wired into the program for simplicity. It is
  a 16-input circuit requiring the texting of 2^16=65536 possible boolean
  combinations of 0's and 1's. This is equivalent to supplying it all possible
  non-negative integers with 16 bits.
*/

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

// The following macro is the value of the ith bit of n. It does a bitwise-and
// of n and the number 0 everywhere except for the ith bit (defining the 0 bit
// as the least signficiant bit.
#define  EXTRACT_BIT(n,i)   ( ((n)&(1<<(i)) )?1:0)


// check_circuit(id,n) checks whether inputval encodes a 16-bit input for
// the circuit, and if it does it prints the 16 bits along with process id
// of the process that executed it.
void check_circuit (int proc_id, int inputval);

int main ( int argc, char* argv[])
{
    int i;
    int id;
    int p;

    MPI_Init      ( &argc, &argv);
    MPI_Comm_rank ( MPI_COMM_WORLD, &id);
    MPI_Comm_size ( MPI_COMM_WORLD, &p );

    for ( i = id; i < 65536; i += p )
        check_circuit( id, i);

    printf("Process %d has finished\n", id);
    fflush(stdout);
    MPI_Finalize();
    return 0;
}

// circuitvalue(v,len) evaluates the set of 16 bits stoed in array v[0..15]
// and returns 1 if the circuit is true and 0 if it is false.
int circuitvalue ( short v[] )
{
    return (
        ((v[0]  ||  v[1])  && (!v[1]   ||  v[3]) )
    &&  ((v[2]  ||  v[3])  && (!v[3]   || !v[4]) )
    &&  ((v[4]  || !v[5])  && ( v[5]   ||  v[6]) )
    &&  ((v[5]  || !v[6])  && ( v[7]   || !v[8]) )
    &&  ((v[8]  ||  v[9])  && ( v[8]   || !v[9]) )
    &&  ((!v[9] || !v[10]) && ( v[10]  || v[11]) )
    &&  ((v[11] ||  v[9])  && ( v[12]  || v[13]) )
    &&  ((!v[8]  || !v[13]) && ( v[13]  || !v[14]) )
    &&  ((v[14]  ||  v[15]) && ( !v[15] || v[6]) )
    );
}

void  check_circuit (int proc_id, int inputval)
{
    short v[16];
    int i;

    for ( i = 0; i < 16; i++ )
        v[i] = EXTRACT_BIT(inputval,i);

    if ( circuitvalue(v) ) {
        printf ( "%d: %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", proc_id,
          v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],v[10],v[11],
          v[12],v[13],v[14],v[15] );
        fflush(stdout);
    }
}
```

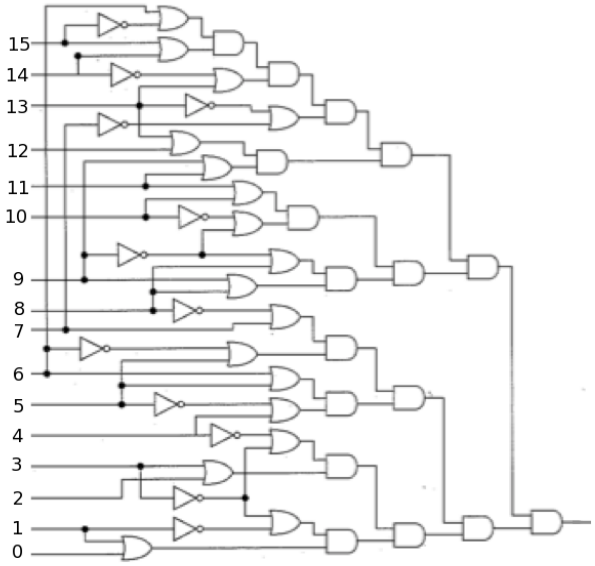


Figure 4.5: The 16-bit circuit that program `circuitsat.c` checks.

4.5.4 The `MPI_Init` Function

The `MPI_Init` function must be called before any other MPI function. Its *C* syntax is

```
int MPI_Init(int *argc, char ***argv)
```

The call may be placed anywhere in the program as long as this condition is met. It is best, from a software engineering perspective, if it appears in the main program, in a place that is easy to see, but this is not required. Its purpose is to initialize the MPI library for this process, i.e., to perform a setup. It must be passed the command line word count and the array of command line words, both passed by address. Thus, the typical form of the main program in *C* should be[3]

[3] In *C++*, there is a form of `MPI_Init` that does not require the arguments. See the man page.

```
int main( int argc; char* argv[] )
{
    /* declare variables */
    MPI_Init(&argc, &argv);
    /* parse argc and argv arguments   */
    /* main program body */
    MPI_Finalize();
}
```

This function, like all functions in the MPI library, starts with the prefix MPI_. In fact all MPI identifiers start with MPI_, followed by an uppercase letter, followed by a string of lowercase letters and underscores. MPI constants are always in uppercase.

4.5.5 The MPI_Comm_rank and MPI_Comm_size Functions

A ***communicator*** is an object that makes it possible for processes to communicate with each other using MPI message-passing primitives. We do not care how it works as long as it does this for us. Although it is possible to create customized communicators, we do not need to do this now, because when MPI is initialized by calling MPI_Init, it creates a default communicator named MPI_COMM_WORLD. Our first program will use this communicator.

Processes within a communicator are ordered. Each has a unique position in this ordering, called its ***rank***. If a program has p processes, they will have ranks between 0 and $p-1$. In previous chapters we saw how a data set could be partitioned in such a way that each process was responsible for its own part of it. A process's rank allows it to determine which part of the data set is its responsibility. To obtain its rank, a process requests it from the communicator using the function MPI_Comm_rank, whose syntax is

```
int MPI_Comm_rank(
    MPI_Comm  comm,   /* Communicator handle */
    int       *rank   /* Rank of the calling process in group of comm */
    )
```

The first argument is the name of the communicator and the second is the address of an integer variable to be given the process's rank. We call the communicator argument a ***handle***; this is a commonly used term that typically means a pointer to a usually large structure (which is what an object of type MPI_Comm is.) Our program will call

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

The communicator also allows a process to obtain the communicator's size, which is how many processes are in it, using MPI_Comm_size, whose syntax is

```
int MPI_Comm_size(
    MPI_Comm  comm,   /* Communicator handle */
    int       *size   /* Number of processes in the group of comm */
    )
```

The first argument is again the name of the communicator and the second is the address of an integer variable to be given the number of processes. Our program will call

```
MPI_Comm_size (MPI_COMM_WORLD, &p);
```

In the program, the first three executable instructions initialize MPI and then get the process's rank and the total number of processes, in id and p respectively. These variables are used in the subsequent for-loop as a means of picking the numbers it will input to the circuit:

```
for ( i = id; i < 65536; i += p )
    check_circuit( id, i);
```

Notice that the first value that each process checks is its rank; it sets `i` to `id`, `id+p`, and so on, so you see that in general it checks all inputs of the form $id + jp$, $j = 0, 1, 2, ..., \left\lfloor \frac{65536}{p} \right\rfloor - 1$. The function call `check_circuit(id,i)` will check whether the number `i` satisfies the circuit. It is passed `id` so that it can print it out if `i` satisfies it. It is an easy but tedious function to understand, because all it really does is evaluate the boolean expression that is equivalent to the circuit. If you are unfamiliar with bit operations in *C*, the *C* macro

```
#define  EXTRACT_BIT(n,i)   ( ((n)&(1<<(i)) )?1:0)
```

evaluates the bitwise-and of the first argument, `n`, and the number `000...0100..000`, consisting of zeros in all but positions except position `i`, and if it evaluates to non-zero, it "returns" 1 and otherwise 0. It is a very efficient way to extract the i^{th} bit of an integer.

4.5.6 Output

When the separate processes in a parallel program make calls to library routines such as `printf` to perform output to shared output streams such as the standard output device (e.g., the terminal), or shared files, a race condition may exist. This is because the *C* and *C++* libraries by themselves cannot prevent the output of these function calls from being intermingled. The subject of how the standard library actually performs output, and how that output appears on the terminal, is beyond the scope of the material of this course. (You can read about it in my *Unix Lecture Notes* series, at `http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes.php`.) If there are enough processes simultaneously trying to write their output to the standard output device, and that device is a terminal, and the output is large enough in size, meaning they are all busy writing lots of output, and the number of characters being written by each call to `printf` is also large, then there is a good chance that the lines from different processes will appear mixed on terminal lines. We cannot prevent this unless we do something in our code to remove this race condition. This program does not attempt to do that.

It is not likely that the lines written by two different processes will appear on the same line of the terminal, at least not on most Unix systems, because the strings that `printf` is printing are all terminated by newline characters. However, this program includes the following function call after each call to `printf`:

```
fflush(stdout);
```

The `fflush` function flushes the internal buffer for the given stream used by the *C* standard I/O library for the process. However, the newline character forces the flush as well, so this call will have no effect. The only reason to include this call is in case the output of this program is redirected to a file. If so, and for some reason, one or more of the processes crash during execution, this call will increase the chance that the output will be written to the file.

4.5.7 Cleaning Up: `MPI_Finalize`

Like many other library API's, the MPI standard includes a function whose purpose is to release resources used by MPI for the program. It's named `MPI_Finalize`, and should be called after all calls to the MPI library have been made. Its syntax is[4]

```
int MPI_Finalize()
```

In general, your program should call this function just before it returns.

[4] In *C++* there is no return value.

Note. All processes must call this routine before exiting. All processes will still exist but may not make any further MPI calls. Once this function is called, no MPI routine (not even `MPI_Init`) may be called, except for the following three: `MPI_Get_version`, `MPI_Initialized`, and `MPI_Finalized`. If there has been any communication among processes, your program has to ensure that all communications have been completed before making this call. We will have more to say about this in a later chapter.

4.5.8 Compiling and Running MPI Programs

To compile a *C* program that uses the MPI API, the easiest method is to invoke the `mpicc` command. This is actually a thin wrapper that calls the underlying compiler, such as `gcc`, with the flags that it needs to compile the code and link it to the library. It will also pass most of the compiler options to it. For example, to compile our first program, we can use the command

```
$ mpicc -Wall -o circuitsat1 circuitsat1.c
```

The `-Wall` option turns on all warning messages, something you should always do, and it will be passed to `gcc`. The `-o` option tells `gcc` to put the output into the file `circuitsat1`. You can read more about `mpicc` in its manpage.

To run an MPI program, you need to use the `mpirun` command. This command, in its simplest form, specifies just the program to run:

```
$ mpirun circuitsat1
```

It will attempt to run a process on each core.

You can also specify the number of processes you wish to run, provided that it is at most the number of cores on the host, with the `-np <num-procs>` option. For example, if we want to run our `circuitsat1` program with 6 processes, we would enter the command

```
$ mpirun -np 6 circuitsat1
```

The `-np` option must be followed by an integer specifying the number of processes, and the name of the executable should follow this. To force MPI to create more processes than cores, use `--oversubscribe`, as in

```
$ mpirun --oversubscribe  -np 200 circuitsat1
```

If you are running on a multiprocessor with multiple cores and possibly more hardware threads, you can tell it to create a process for each hardware thread:

```
$ mpirun --use-hwthread-cpus circuitsat1
```

Later we will learn about other options that we can give to `mpirun`.

When we run our program using 4 processes, we get the following output:

```
$ mpirun -np 4 circuitsat1
1: 1010111110011001
1: 1010111111011001
1: 1010111110111001
Process 0 has finished
Process 2 has finished
Process 1 has finished
Process 3 has finished
```

Here is another run with 5 processes:

```
$ mpirun -np 5 circuitsat1
0: 1010111111011001
2: 1010111110111001
Process 0 has finished
Process 4 has finished
Process 2 has finished
3: 1010111110011001
Process 3 has finished
Process 1 has finished
```

In the first run, process 1 found all three inputs that satisfied the circuit. This is because they only differ in bit positions 5 and 6, and because they have least significant bits `001`, and are thus are all $1 \, mod \, 4$, and process 1 has all numbers that are $1 \, mod \, 4$. The three inputs all differ by small multiples of 32. When we run it with 5 processes, because 5 is relatively prime to 32, no two of these inputs will be assigned to the same process. (Why?)

Finally, notice that the order in which they write to the screen is unpredictable. In fact if you repeatedly run it, you will see that it can be different every time. This is one reason it is hard to debug parallel programs.

4.6 Collective Communication: Reduction

As a way to introduce a new concept, we will modify the circuit satisfiability program so that, instead of outputting the different input combinations that satisfy the circuit, it just outputs *the number of combinations* that do. With what we know right now, we cannot do this, because in order to compute the total, the processes would need to communicate with each other.

In Chapter 3, we introduced the concept of a reduction. Recall that a reduction is the process of computing $a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}$ for some associative, binary operator $\oplus$. If each of our processes counts how many of its combinations satisfy the circuit, then, to get the total, our program needs to perform a reduction, using the addition operator, i.e., it needs to compute $c_0 \oplus c_1 \oplus c_2 \oplus \cdots \oplus c_{p-1}$, where c_j is the count computed by process j. We will now see how to request a reduction operation from the MPI library.

A ***collective communication*** in MPI is a communication operation in which a group of processes cooperate to perform a global operation across all the members of the group. Some collective communications move data around, either gathering it or distributing it. Some gather and compute with it simultaneously . Some neither move data nor compute, but simply synchronize. The reduction operation is one that gathers and computes. Its syntax is

```
int MPI_Reduce(
    void         *operand, /* address of first operand to send            */
    void         *result,  /* address where first result will be stored   */
    int          count,    /* number of operands in the operand send buffer */
    MPI_Datatype datatype, /* data type of operands to reduce             */
    MPI_Op       op,       /* reduction operator to apply                 */
    int          root,     /* rank of process that gets the result(s)     */
    MPI_Comm     comm      /* communicator  handle                        */
    )
```

In short, the `MPI_Reduce` function combines the elements provided in the input buffer given by its first parameter, `operand`, of each process in the group, using the operation `op`, and returns the combined value in the output buffer, which is the second parameter, `result`, of the process whose rank is `root`. The number of elements in the input and output buffers is given by `count` – it must be the same size – and the type of data that they store is given by `datatype`. (The list of allowed data types is below.) ***The routine must be called by all group members, i.e., processes in the communicator, using the same arguments for `count`, `datatype`, `op`, `root`, and `comm`, otherwise it will fail.***

Name	*C* **Data Type**
`MPI_CHAR`	`signed char`
`MPI_WCHAR`	`wchar_t` - wide character
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_LONG_LONG_INT`	`signed long int`
`MPI_LONG_LONG`	`signed long long int`
`MPI_SIGNED_CHAR`	`signed char`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_UNSIGNED_LONG_LONG`	`unsigned long long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_C_COMPLEX`	`float _Complex`
`MPI_C_FLOAT_COMPLEX`	`float _Complex`
`MPI_C_DOUBLE_COMPLEX`	`double _Complex`
`MPI_C_LONG_DOUBLE_COMPLEX`	`long double _Complex`
`MPI_C_BOOL`	`_Bool`
`MPI_C_LONG_DOUBLE_COMPLEX`	`long double _Complex`
`MPI_INT8_T`	`int8_t`
`MPI_INT16_T`	`int16_t`
`MPI_INT32_T`	`int32_t`
`MPI_INT64_T`	`int64_t`
`MPI_UINT8_T`	`uint8_t`
`MPI_UINT16_T`	`uint16_t`
`MPI_UINT32_T`	`uint32_t`
`MPI_UINT64_T`	`uint64_t`
`MPI_BYTE`	8 binary digits (not a *C* type)
`MPI_PACKED`	data packed or unpacked with `MPI_Pack()`/ `MPI_Unpack()` (not a *C* type)

Table 4.1: Partial list of MPI predefined constants of type `MPI_Datatype`.

- The first parameter, `operand`, is the address of an input buffer. This is where the process stores the value to be contributed to the reduce operator. If `count` is greater than 1, then this is a sequence of adjacent elements in memory, such as an array.

- The second parameter, `result`, is the address of a receiving buffer. This is where the result of applying the reduction to all operands contributed by all processes will be stored. If `count` is greater than 1, then this is a sequence of adjacent elements in memory, such as an array.

- The third parameter, `count`, is the number of operands in the input buffer, and the number of results in the output buffer. For example, if the operation is addition, and the input buffer contains two elements that are summable, such as integers, then `result[0]` will be sum of all `operand[0]` values in each process and `result[1]` will be the sum of all `operand[1]` values in each process.

- The fourth parameter, `datatype`, indicates the type of the elements that will be reduced. It must be of type `MPI_Datatype` and of a type for which a reduction is possible. A list of allowable constants that meet the requirements is in Tables 4.1 and 4.2 below.

- The fifth parameter, `op`, indicates the operation to be performed, and must be of type `MPI_Op`. A list

Name	*C* Data Type
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_LONG_INT	long and int
MPI_SHORT_INT	short and int
MPI_2INT	int and int
MPI_2COMPLEX	_Complex and _Complex
MPI_2DOUBLE_COMPLEX	double _Complex and double _Complex

Table 4.2: MPI constants for reduction operators `MPI_MAXLOC` and `MPI_MINLOC`. These types have two values, such as `MPI_2INT`.

of the built-in reduction operators is in Table 4.3. MPI also has a provision for user-defined reduction operators, but we will not discus that here.

- The sixth parameter, `root`, is the rank of the process that will receive the result. After all processes participating in the reduction have returned from the call, the `root` process, and the `root` alone, will have the correct values in its `result` argument.
- The seventh and last parameter is a handle to the communicator for this group of processes.

Our program needs to add up the integer subtotals from each process and have that result available for printing. Only one process will do the printing, and it does not matter which, so we arbitrarily pick the one whose rank is 0. Therefore, all processes must send their counts to process 0 using the `MPI_Reduce` function. Therefore the fourth argument should be `MPI_INT` since the counts are integers, and the fifth should be `MPI_SUM` as we are adding. We make the sixth argument 0. In the main program we introduce two variables named `subtotal` and `grand_total`. Each process has a copy of these. The former will store the number of inputs found by the process to satisfy the circuit; the latter will be the sum of all of these. Based on this description, the call that our program needs to make is

```
MPI_Reduce(&subtotal, &grand_total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Once again, you must remember that every process must call `MPI_Reduce`. Furthermore, each must have the exact same parameters except for the first[5]; they are allowed to have different first parameters. In our program, which appears in Listing 4.2, they all make the exact same call. This is what you should always try to arrange.

Because our initial design of the `check_circuit` function returned a 1 if a solution was found and a 0 if not, all we need to do is to suppress its output and modify the main program as follows:

```
subtotal = 0;
for ( i = id; i < 65536; i += p )
    subtotal += check_circuit( id, i);
```

This will store the total number of solutions found by each process into its own private variable, `local_total`. Lastly, we modify the output at the end. Only process 0 prints, and all it does is print a single number:

```
if ( 0 == id )
    printf("%d\n", grand_total);
```

[5] The second parameter is allowed to be different, but this will only cause problems. You should never try to arrange for the processes to send their results to different locations, as errors will result.

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bit-wise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Table 4.3: MPI predefined global reduction operators.

Listing 4.2: circuitsat2.c

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

// Same macro as appeared in circuitsat1.c
#define  EXTRACT_BIT(n,i)   ( ((n)&(1<<(i)) )?1:0)

// check_circuit(id,n) is same as in circuitsat1.c
int check_circuit (int proc_id, int inputval);

int main ( int argc, char* argv[])
{
    int i;
    int id;
    int p;

    int subtotal;
    int grand_total;

    MPI_Init      ( &argc, &argv);
    MPI_Comm_rank ( MPI_COMM_WORLD, &id);
    MPI_Comm_size ( MPI_COMM_WORLD, &p );

    subtotal = 0;
    for ( i = id; i < 65536; i += p )
        subtotal += check_circuit( id, i);

    MPI_Reduce(&subtotal, &grand_total, 1, MPI_INT,
               MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Finalize();

    if ( 0 == id )
        printf("%d\n", grand_total);
    return 0;
}

// circuitvalue(v,len) evaluates the set of 16 bits stored in array v[0..15]
// and returns 1 if the circuit is true and 0 if it is false.
int circuitvalue ( short v[] )
{
    return (
        ((v[0]  ||  v[1])  && (!v[1]   ||  v[3]) )
    &&  ((v[2]  ||  v[3])  && (!v[3]   || !v[4]) )
    &&  ((v[4]  || !v[5])  && ( v[5]   ||  v[6]) )
    &&  ((v[5]  || !v[6])  && ( v[7]   || !v[8]) )
    &&  ((v[8]  ||  v[9])  && ( v[8]   || !v[9]) )
    &&  ((!v[9] || !v[10]) && ( v[10]  || v[11]) )
    &&  ((v[11] ||  v[9])  && ( v[12]  || v[13]) )
    &&  ((!v[8] || !v[13]) && ( v[13]  || !v[14]) )
    &&  ((v[14] ||  v[15]) && ( !v[15] || v[6]) )
    );
}

int  check_circuit (int proc_id, int inputval)
{
    short v[16];
    int i;

    for ( i = 0; i < 16; i++ )
        v[i] = EXTRACT_BIT(inputval,i);

    if ( circuitvalue(v) )
        return 1;
    else
        return 0;
}
```

4.7 Collective Communication: Broadcasting

4.7.1 MPI_Bcast

A second important type of collective communication is a ***broadcast***. A broadcast is a collective communication that distributes data. The MPI function that performs a broadcast is MPI_Bcast. Its syntax is

```
int MPI_Bcast(
    void *buffer,           /* address of first element to send     */
    int count,              /* number of elements to send           */
    MPI_Datatype datatype,  /* data type of elements to send        */
    int root,               /* rank of process that sends the data  */
    MPI_Comm comm           /* communicator  handle                 */
)
```

MPI_Bcast broadcasts a message from the process with rank root to all processes in the communicator's group, itself included. It must be called by all processes in the group using the same arguments for comm and root. When it returns, the contents of root's communication buffer, buffer, have been copied to all processes. The count is the number of elements to send, and datatype is the type of the elements. For example, a code snippet that would have process 0 send the contents of an entire array to all processes in the MPI_COMM_WORLD communication group would be

```
int Num_items;
int list[Num_items];
/* code here to fill list and initialize Num_items */

MPI_Init( &argc, &argv );
MPI_Bcast(list, Num_items, MPI_INT, 0, MPI_COMM_WORLD);
```

4.7.2 Example: Calculating PI

The input to the circuit satisfiability problem was hard-coded into the program, so the number of bit combinations was known at compile-time. In many problems, the size of the input is unknown at compile-time and is determined at run-time, either as a command-line argument to the program, a value read from a file stream, or a value entered interactively by the user. Suppose that we want the value either to be entered interactively via standard input, or read from a file. Remember that only one process can perform input. Suppose, without any loss of generality, that it is process 0. Then, whether the input comes from a file or from standard input, process 0 will read it, but no other process will know it. The problem then is how process 0 can share it with the remaining processes, which is why broadcasts are an important tool.

Broadcasts are needed in many parallel programs. It is often the case that one process acquires a piece of information that will be needed by all other processes. This could be the size of an array, or some computed value they will all need, or a number that controls how long they should all iteratively solve some problem. We give an example of just such a problem. We will illustrate the use of the broadcast function `MPI_Bcast` in a program that interactively estimates the value of the mathematical constant π. There are dozens of methods of approximating the value of π. We will use a method based on the mathematical identity

$$\tan(\frac{\pi}{4}) = 1.0 \tag{4.1}$$

In other words, the tangent of a 45° angle is 1.0. This implies that

$$4 \cdot \arctan(1.0) = \pi \tag{4.2}$$

We can therefore approximate π by finding the value of arctan(1.0) and multiplying it by 4. There is no analytical formula that gives us the value of the arctangent of a number; we have to approximate it. We can do this by a standard technique of finding the area under a suitable curve.

Suppose that we do not know how to compute the value of $f(x)$ for some particular x, with any formula, but that we know that f is differentiable and we know what the first derivative of $f(x)$ is. Let $f'(x)$ be the first derivative of $f(x)$. The second fundamental theorem of calculus tells us that

$$\int_0^x f'(t)dt = f(x) - f(0) \tag{4.3}$$

Another way to say this is that the area under the curve of f' on the interval $[0, x]$ is $f(x) - f(0)$. If it is known that $f(0) = 0$, then $f(x)$ is just the area under the curve. In particular, to approximate π, we can use Equations 4.2 and 4.3. The derivative of $\arctan(x)$ is $1/(1 + x^2)$ and $\arctan(0) = 0$, so we can compute π from the formula

$$\pi = 4 \cdot \arctan(1.0) = 4 \cdot \int_0^1 \frac{1}{1+t^2} dt$$

Figure 4.6 illustrates the idea. To find the area under the graph of the function $1/(1 + x^2)$ on the interval $[0, 1]$ we can use approximate integration. We can divide the interval between 0 and 1 into n equal size segments, $s_1, s_2, ..., s_n$, find the center x_k of each segment s_k, sum the values $1/(1 + x_k^2)$, and multiply by the width of each segment $1/n$ to get the area under the curve. Then four times this is an approximation of π.

We begin by looking at the sequential code that can compute π in this way. A function to compute π sequentially, given the number of segments to use, is in Listing 4.3 below.

Listing 4.3: approximate_pi sequentially

```
double approximate_pi ( int num_segments)
{
    double dx, sum, x;
    int i;
```

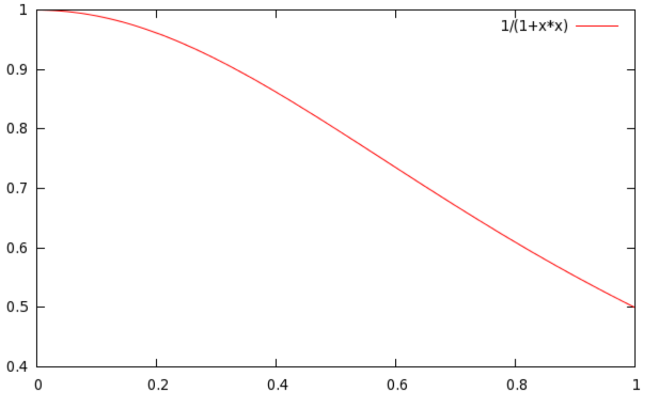


Figure 4.6: Using approximate integration to find the value of π. The area under the curve is $\pi/4$.

```

    if ( 0 == num_segments ) return 0;  /* handle bad argument */

    dx = 1.0 / (double) num_segments; /* Set dx to the width of a segment */
    sum = 0.0;
    for (i = 1; i <= num_segments; i++ ) {
        x = dx * ((double)i - 0.5); /* x is midpoint of segment i */
        sum += 1.0 / (1.0 + x*x);   /* add new area to sum */
    }
    return  4.0 * dx * sum;  /* we multiply sum by dx because we are computing an integral and dx is the differential */
}
```

This problem has a great deal in common with circuit satisfiability, so we will not elaborate on the stages of Foster's design methodology in detail.

As with circuit satisfiability, the approximation of π is highly data parallel. The primitive tasks are essentially computing each iteration of the body of the loop, and the loop iterations are independent. We could create a task for every segment, but because there is no communication between primitive tasks and we want to avoid process creation overhead, we should agglomerate by creating one task for each processor. The next question is how to map the primitive tasks to the processors. Although each segment requires the same amount of computing effort, because the number of segments may not be a multiple of the number of processors, we will use an interleaved mapping of the segments to each task, just as we did for the circuit satisfiability problem. Thus, task k will compute the areas of segments k, $k+p$, $k+2p$, ... up to the largest number $k+mp$ that is less than or equal to the number of segments.

A pseudocode description of the program is

```
MPI is initialized and the rank and size are obtained;
Repeat
    Root process prompts user to enter number of segments and reads this number;
    If user enters 0, loop breaks and program exits;
    Root broadcasts the number of segments, N, to every process;
    Each process calls a function to compute a partial sum of the segments
        for which it is responsible;
    Every process participates in a reduction to compute the total of
        the partial sums;
    Root process prints the sum as well as the difference between the sum and the
       math library's stored value of pi.
```

Thus, the root process (process 0) is in charge of all input and output. The number of terms that is entered by the user is broadcast by the root process to all processes. This is where `MPI_Bcast` is used. After all processes compute their partial sums, a reduction is performed. The complete program appears in Listing 4.4 below.

Listing 4.4: estimate_pi.c

```c
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "mpi.h"

#define ROOT 0

/** approximate_pi()
 *  This returns an approximation of pi based on the mathematical identity
 *      tan(pi/4) = 1.0
 *  which is equivalent to
 *      4.0 * arctan(1.0) = pi
 *  To compute pi, we can approximate arctan(1.0). We use the fact that
 *      d/dx(arctan(x) = 1/(1 + x*x)
 *  and compute the area under the curve of 1/(1 = x*x) in the interval
 *  from 0 to 1.0.  The area under this curve is arctan(1.0) because the
 *  the integral from 0 to 1 of 1/(1 = x*x) is arctan(1.0) - arctan(0) = pi/4.
 *  We multiply this by 4 to get pi.
 *  We use the rectangle method to approximate the area under the curve.
 *  We divide [0,1.0] into n segments of length 1/n each, and compute the
 *  value of 1/(1 + x*x) at the midpoint of each segment. By summing these
 *  values and multiplying by 4 times 1/n, we have an approximation.
*/
double approximate_pi ( int num_segments, int id, int p)
{
    double dx, sum, x;
    int i;

    if ( 0 == num_segments ) return 0;

    /* Set dx to the width of each segments */
    dx = 1.0 / (double) num_segments;

    /* Initialize sum */
    sum = 0.0;

    /* Each process will compute its share of the segments. If the segments are
        numbered 1, 2, 3, ...,n, from left to right, then process id computes
        segment k if id = (k-1) % p, or equivalently it computes segments id+1, id+
        p+1, id+2p+1, ... up to id+mp+1,  where m is the largest number such that
        id+mp+1 <= num_segments. */
    for (i = id + 1; i <= num_segments; i += p) {
        x = dx * ((double)i - 0.5); /* x is midpoint of segment i */
        sum += 4.0 / (1.0 + x*x);   /* add new area to sum */
    }
    return  dx * sum;  /* we multiply sum by dx because we are computing an
        integral and dx is the differential */
}

int main( int argc, char *argv[] )
{
    int id;             /* rank of executing process   */
    int p;                  /* number of processes          */
    double pi_estimate;     /* estimated value of pi        */
    double local_pi;        /* each process's contribution */
    int num_intervals;      /* number of terms in series    */

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &id );
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    /* repeat until user enters a 0 */
    while ( 1 ) {
        if (ROOT == id ) {
            printf("Enter the number of intervals: [0 to quit] ");
            fflush(stdout);
            scanf("%d",&num_intervals);
        }

        /* The root proces will broadcast the number of intervals entered to all
            other processes. Each process must call MPI_Bcast though. The data to
            be broadcast is num_intervals; the count = 1, it is of type MPI_INT;
            ROOT is the sender */
        MPI_Bcast(&num_intervals, 1, MPI_INT, ROOT, MPI_COMM_WORLD);

        if (0 == num_intervals )
             break;

        local_pi = approximate_pi(num_intervals, id, p);

        /* MPI_Reduce collects the local estimate from each process into a global
            value, pi_estimate. The ROOT is the process that receives  all values
            in the reduction. The reduce operator is MPI_SUM. */
        MPI_Reduce(&local_pi,
                   &pi_estimate, 1, MPI_DOUBLE,
                   MPI_SUM, 0, MPI_COMM_WORLD);

        /* ROOT does the printing. The error is caculated by comparing to the math
             library's value for PI, M_PI. */
        if (ROOT == id ) {
            printf("pi is approximated to be %.16f. The error is %.16f\n",
            pi_estimate, fabs(pi_estimate - M_PI));
            fflush(stdout);
        }
    }
    MPI_Finalize();
    return 0;
}
```

4.8 Benchmarking Parallel Program Performance

The reason that we are interested in writing parallel programs is to improve their running time. Therefore, we should not go too far into the subject without first describing how to measure their running time. After all, we want to know whether or not the effort has paid off.

4.8.1 Timing Programs Using MPI

The MPI library provides a portable function named MPI_Wtime that we can use to measure the total time used by the program. MPI_Wtime returns a floating-point number, which is the number of seconds, representing *elapsed wall-clock time* since some time in the past. The expression "wall-clock time" means the time on some imaginary clock like the clock on a wall. This imaginary clock does not provide the actual time of day; we do not really need the actual time. All we care about is that it is some "time in the past" that is guaranteed not to change during the life of the process, and this is exactly what it returns. In other words, it picks a number and uses that number as the time. All future calls to the function will return time relative to the same reference time while this process is running. Its signature is

```
double MPI_Wtime()
```

A second function in the library can tell us the granularity, or precision, of MPI_Wtime. In other words, if the wall-clock time is only being updated every millisecond, it is less precise than if it is updated every microsecond. It is important to know the size of a "clock tick", i.e, the length of the interval between successive updates to its internal counter. The function MPI_Wtick returns the number of seconds between ticks as a double precision floating point number. Its signature is

```
double MPI_Wtick()
```

The typical way that MPI_Wtime is used is

```
double starttime, endtime;
starttime = MPI_Wtime();
....  stuff to be timed  ...
endtime   = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

which measures the elapsed time between when the "stuff to be timed" started and when it ended. This is different from the actual time it used on the processor. It includes time it spent waiting for I/O operations, waiting because it might have been removed from the processor by the operating system kernel to allow some other process to run, and time spent by the operating system doing things on its behalf. This is why it is called elapsed wall-clock time, because it is measuring the time that would pass on a clock on the wall, not the time that the computer used.

When a process sends output to a terminal device, this can take a long time, relative to the time it spends computing. Terminals are slow devices, relatively speaking. We are only interested in comparing the running time of our parallel program with that of the serial program. If the serial program takes χ seconds actually computing results and β seconds writing them to the terminal, then the parallel program will also spend β seconds writing to the terminal, because they both write the same information to the same device. If the parallel program, however, spends χ/s seconds computing results, where s is some number bigger than 1, then we want to say that the parallel program is s times faster than the serial one. If we measure the elapsed time including all I/O as the fraction $(\chi+\beta)/((\chi/s)+\beta)$, we will not get s, but something much smaller, depending on the value of β. For example, if $\chi = 0.01$ and $\beta = 2.0$ and $s = 4$, then if we include the I/O time we get $2.01/2.0025 = 1.0037$, which is barely an improvement. On the other hand, if we exclude the I/O from the measurement, we get $0.01/0.0025 = 4$, which is a factor of 4.

Exercise. For a fixed χ and β, to what value will $(\chi+\beta)/((\chi/s)+\beta)$ converge as the speed-up increases? For fixed speed-up s, as the ratio β/χ increases, to what value will it converge?

While the overhead of the parallel program is important and cannot be overlooked when comparing the running time of the parallel program to its sequential version, we will ignore it for now, for two reasons. One is that, in general, the time that is spent creating the processes, establishing communication links (such as sockets) between them, and initializing the MPI library should be very small in comparison to the time spent computing, for realistic programs. Second is a practical matter: we cannot call MPI_Wtime until we have already called MPI_Init, and we cannot call it after we have called MPI_Finalize. Therefore it must be called in between these.

4.8.2 Collective Communication and Barrier Synchronization: `MPI_Barrier`

Another restriction is that we need to measure the time from the moment that all processes have been created and are ready to start executing, until the last one has finished. In Chapter 2 we introduced the concept of barrier synchronization. Recall that a ***barrier synchronization*** instruction is one that, when it is executed by a process, causes that process to wait until all other processes have reached that same instruction in their code. This is exactly what we need; if we had a means of starting the clock at a point in the code when we knew all processes were ready to start executing the real stuff, and stopping it when they all return from the `MPI_Reduce` call, then we could measure the amount of time they spent doing the actual computation.

The function `MPI_Barrier` is an example of a collective communication operation that performs synchronization, in particular, barrier instruction. Its signature is

```
int MPI_Barrier(MPI_Comm comm)
```

Its only parameter is a handle to the communicator to which the processes belong. It blocks the calling process until all group members have called it, and returns only after all group members have entered the call.

We illustrate its use in a third version of the circuit satisfiability program and in a timed version of `estimate_pi.c`. In `circuitsat3.c`, we remove all output except the time. We do not need to know how many inputs satisfy the circuit now, so that code is removed. We have the process with rank 0 print out the time. The program is displayed in Listing 4.5. The calculation of elapsed time uses a single variable instead of two. It initializes it to `-MPI_Wtime()` and then adds the time at which the processes terminate to this time, effectively forming the sum (newtime - oldtime).

Listing 4.5: circuitsat3.c

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

// Same macro as appeared in circuitsat1.c
#define  EXTRACT_BIT(n,i)   ( ((n)&(1<<(i)) )?1:0)

// check_circuit(id,n) is same as in circuitsat1.c
int check_circuit (int proc_id, int inputval);

int main ( int argc, char* argv[])
{
    int i;
    int id;
    int p;
    double elapsed_time;  /* Time to find, count solutions */

    int subtotal;
    int grand_total;

    MPI_Init      ( &argc, &argv);

    /* Start timer */
    MPI_Barrier (MPI_COMM_WORLD);

    MPI_Comm_rank ( MPI_COMM_WORLD, &id);
    MPI_Comm_size ( MPI_COMM_WORLD, &p );

    elapsed_time = - MPI_Wtime();

    subtotal = 0;
    for ( i = id; i < 65536; i += p )
        subtotal += check_circuit( id, i);

    MPI_Reduce(&subtotal, &grand_total, 1, MPI_INT,
                MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Barrier (MPI_COMM_WORLD);

    /* Stop timer */
    elapsed_time += MPI_Wtime(); /* elapsed time = current time - start time */

    if (0 == id) {
       printf ("Execution time %8.6f\n", elapsed_time);
       fflush (stdout);
    }
    MPI_Finalize();
    return 0;
}

int circuitvalue ( short v[] )
{
    return (
        ((v[0]  ||  v[1])  && (!v[1]   ||  v[3]) )
    &&  ((v[2]  ||  v[3])  && (!v[3]   || !v[4]) )
    &&  ((v[4]  || !v[5])  && ( v[5]   ||  v[6]) )
    &&  ((v[5]  || !v[6])  && ( v[7]   || !v[8]) )
    &&  ((v[8]  ||  v[9])  && ( v[8]   || !v[9]) )
    &&  ((!v[9] || !v[10]) && ( v[10]  || v[11]) )
    &&  ((v[11] ||  v[9])  && ( v[12]  || v[13]) )
    &&  ((!v[8] || !v[13]) && ( v[13]  || !v[14]) )
    &&  ((v[14] ||  v[15]) && ( !v[15] || v[6]) )
    );
}

int  check_circuit (int proc_id, int inputval)
{
    short v[16];
    int i;

    for ( i = 0; i < 16; i++ )
        v[i] = EXTRACT_BIT(inputval,i);

    if ( circuitvalue(v) ) {
        return 1;
    }
    else
        return 0;
}
```

The program as designed makes it easy to collect data to tabulate. On a 4 core computer, we ran 10 trials with 4 processes, one per core, 10 with 2 processes, putting one process on its own core, and 10 trials with a single process. The results are plotted in Figure 4.7. The actual times are displayed as the solid line in the figure. The dashed line shows what the execution times would be if doubling the number of processors halved the execution time.

What accounts for the difference? The reduction collective communication adds overhead to the running time, proportionately to the logarithm of the number of processes. The sequential program has no such overhead.

Last, here is a listing for a timed version of the `estimate_pi.c` program. Comments are omitted to save

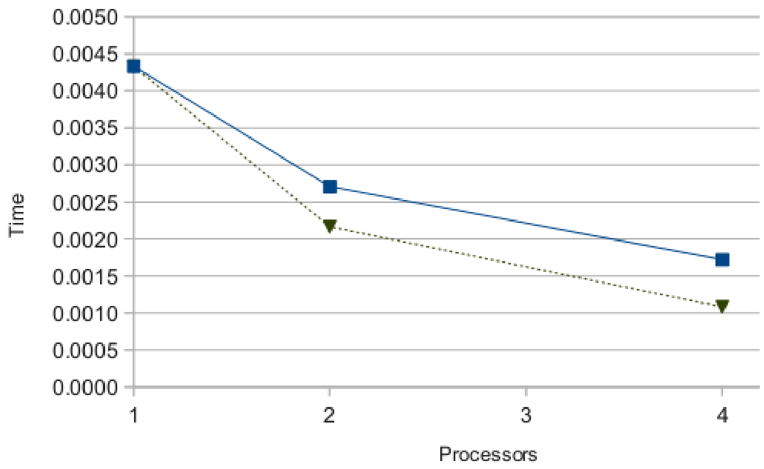


Figure 4.7: Graph of execution time.

space.

Listing 4.6: estimate_pi_timed.c

```c
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "mpi.h"

#define ROOT 0

double approximate_pi ( int num_segments, int id, int p)
{
    double dx, sum, x;
    int i;

    dx = 1.0 / (double) num_segments;
    sum = 0.0;
    for (i = id + 1; i <= num_segments; i += p) {
        x = dx * ((double)i - 0.5); /* x is midpoint of segment i */
        sum += 4.0 / (1.0 + x*x);   /* add new area to sum */
    }
    return  dx * sum;
}

int main( int argc, char *argv[] )
{
    int id;               /* rank of executing process   */
    int p;                /* number of processes         */
    double pi_estimate;   /* estimated value of pi       */
    double local_pi;      /* each process's contribution */
    int num_intervals;    /* number of terms in series   */
    double elapsed_time;/* Time to compute pi          */

    MPI_Init      ( &argc, &argv);
    MPI_Comm_rank ( MPI_COMM_WORLD, &id);
    MPI_Comm_size ( MPI_COMM_WORLD, &p );

    while ( 1 ) {
        if (ROOT == id ) {
            printf("Enter the number of intervals: [0 to quit] ");
            fflush(stdout);
            scanf("%d",&num_intervals);
        }
        MPI_Barrier (MPI_COMM_WORLD);
        elapsed_time = - MPI_Wtime();

        MPI_Bcast(&num_intervals , 1, MPI_INT , ROOT , MPI_COMM_WORLD);

        if (0 == num_intervals )
             break;

        local_pi = approximate_pi(num_intervals , id, p);

        MPI_Reduce(&local_pi ,
                   &pi_estimate , 1, MPI_DOUBLE ,
                   MPI_SUM , 0, MPI_COMM_WORLD);

        MPI_Barrier (MPI_COMM_WORLD);

        /* Stop timer */
        elapsed_time += MPI_Wtime();

        if (ROOT == id ) {
            printf("pi is approximated to be %.16f. The error is %.16f\n",
            pi_estimate , fabs(pi_estimate - M_PI));
            printf("Computation took %8.6f seconds.\n", elapsed_time);
            fflush(stdout);
        }
    }
    MPI_Finalize();
    return 0;
}
```

References

[1] J. R. R. Tolkien. *Lord of the Rings: Book II The Two Towers*. George Allen & Unwin, England, 1954.

[2] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

Subject Index