CHAPTER 11

Performance

OF COURSE, WE WANT TO WRITE PARALLEL PROGRAMS so that we can solve bigger problems in less time. If our serial programs were fast enough to solve all the problems we were interested in, and if they could store all our data, parallel programming would just be an intellectual exercise. So in our discussion of writing parallel programs in Chapter 10, we omitted a critical part of the design process: performance estimation. Before going to the trouble to write a nontrivial program, we should attempt to determine whether its performance will satisfy our needs. In this chapter we'll discuss methods for estimating parallel program performance.

Note that we do not use the more conventional asymptotic methods for analyzing program performance. Although these methods are very useful, especially in the analysis of serial program performance, we have found that they don't provide sufficient detail in the analysis of parallel programs. Rather, we have found that an empirical approach is the most useful. We should also note that, although we will focus on speed, there are many aspects to the evaluation of a parallel program. In particular, you should always keep in mind the cost of developing a parallel program. Many parallel programs are designed in order to obtain the maximum possible performance and, as a consequence, take years to develop. Clearly, you should always ask how much more it will cost to develop a faster, more complex program than a slower, simpler program.

We'll begin by discussing serial program performance.

11.1 Serial Program Performance

We view program performance analysis as an ongoing, integral part of program development. Thus, there are varying degrees of precision implicit in

the expression "performance analysis." Before any code has been written, we can only provide performance estimates that involve arbitrary symbolic constants obtained by estimating the number of statements executed. As the development proceeds, we can replace the symbolic constants with numerical constants that are valid for a particular system and compiler. Thus, in our a priori analysis we will specify runtime by counting statements; as details of the performance emerge, we may specify runtime in milliseconds or microseconds.

When discussing the runtime of a program, we would like to be able to say something like, "The running time of this program is $T(n)$ units if the input has size n." Of course the actual time a program takes to solve a problem—the time from the beginning of execution to the completion of execution—will depend on factors other than the input size. For example, it will depend on

1. the hardware being used
2. the programming language and compiler
3. details of the input other than its size

In order to avoid dealing with the first factor, we will, as we already mentioned, count "statements executed" in our initial analyses. This, brings up the second factor: if we're counting statements executed, are these assembler statements? If so, what type of assembler, RISC or CISC? Or are the statements high-level language statements, and if so, which high-level language? Also, is counting statements reasonable, since, in general, different statements will require different execution times?

Finally, it's easy to come up with examples of programs that behave very differently with different inputs, even if the inputs have the same size. For example, an insertion sort that sorts integers into increasing order and that uses linear search will run much faster if the input is $1, 2, \ldots, n$ than if the input is $n, n-1, \ldots, 2, 1$. We can avoid this problem by discussing worst-case runtime or possibly average-case runtime; in general, when we discuss runtime, we will mean worst-case runtime.

However, when we're counting statements, we can't entirely avoid the first two problems without introducing some imprecision into our measurements. Most authors deal with this imprecision by using **asymptotic analysis.** In asymptotic analysis, we specify bounds on the performance of a program. As an example, people often say that the sorting algorithm commonly known as "bubble sort" is an n^2 algorithm. What is meant by this is that if you apply the bubble sort algorithm to a list consisting of n items, the number of statements executed will be less than some constant multiple of n^2, provided that n is sufficiently large. If you're not accustomed to seeing this type of analysis, this statement may sound very vague. However, for estimating serial program performance, it has proved to be very useful.

Since it is not as useful for parallel program performance, we won't pursue this avenue. Rather, we will count statements executed and include all "con-

stant" multiples explicitly in our formulas—at least until we've determined that they're not necessary. Let's illustrate the ideas by looking at a simple example.

11.2 An Example: The Serial Trapezoidal Rule

Let's estimate the runtime of the serial trapezoidal rule. (If you are skipping around in your reading, take a look at Chapter 4.) Recollect that the serial program reads in the left and right endpoints (a and b), the number of trapezoids (n), and then computes a running sum: it computes the area of the ith trapezoidal and adds it into the sum of the areas of the previous $i - 1$ trapezoids. When this is completed, it prints the result. For ease of reference, the heart of the program is

```
h = (b-a)/n;
integral = (f(a) + f(b))/2.0;
x = a;
for (i = 1; i <= n-1; i++) {
    x = x + h;
    integral = integral + f(x);
}
integral = integral*h;
```

The number of statements executed before and after the `for` loop doesn't depend on the input size, n. Say there are c_1 of these statements. If we assume that evaluation of $f(x)$ requires a constant number of statements, then the number of statements executed in each pass through the `for` loop is also a constant: call it c_2. So the total number of statements executed is

$$T(n) = c_1 + c_2(n - 1).$$

That is, $T(n)$ is a linear polynomial in n; after regrouping and renaming the constants, we have

$$T(n) = k_1 n + k_2.$$

For this simple program, it is not unreasonable to assume that the execution time of the various statements isn't too different. Furthermore, we would expect the total number of statements executed inside the `for` loop to be much greater than the number of statements executed outside the loop. That is, $k_1 n \gg k_2$. Hence, T(n) can be approximated by

$$T(n) \approx k_1 n.$$

So we would predict that if we increase n by a factor of r, $T(n)$ should increase by the same factor. That is, if $T(n) = t$, then we would predict that the runtime for input size rn, $T(rn)$, would be about rt.

OK. How do our predictions compare to actual performance? We'll discuss the problem of taking program timings later. For now, it suffices to say

that if our system isn't running any other processes, we can simply check the system clock at the beginning and the end of the code segments, and take the difference to get the elapsed time.

The actual execution times on a single processor of an nCUBE 2 correspond very well with our estimates. (Here $f(x) = e^{-x^2}$, $a = -2$, and $b = 2$.)

n	512	1024	1536	2048
Time in ms	11.4	22.8	34.2	45.6

Observe, for example, that if we double n, we double the total execution time. Observe also that we can actually estimate the constant k_1 to be 11.4/512 milliseconds/trapezoidal, which is about 22.3 microseconds/trapezoid. We'll make use of this later, when we try to analyze the performance of the parallel trapezoidal rule.

11.3 What about the I/O?

A couple of things should be bothering you about our analysis. We omitted the `scanf` and `printf` statements. Why? They're certainly an important and expensive part of the computation. There are two reasons for leaving them out. The first has to do with how the program might be used in a practical application. In an application, the trapezoidal rule program would probably not be of interest in its own right. It would be part of a larger problem that happens to need to do some integration. So the input values, a, b, and n, would be supplied by the program—not by the user. Similarly, the result of the integration would, in all likelihood, not be of interest—it would be used by the rest of the program to complete its "bigger" calculation.

The second reason has to do with the fundamental difference between I/O statements and statements that simply use the computer's CPU and RAM: I/O statements are *much* slower—typically several orders of magnitude. For example, in a 33 MHz 486 PC running Linux and the `gcc` compiler, a multiplication takes about a microsecond, while a `printf` of a single float takes about 300 microseconds. On a single processor of an nCUBE 2, a multiplication takes about 2.5 microseconds, and a `printf` takes about 500 microseconds.[1] Furthermore these are relatively slow processors. On faster systems, the ratios may be worse, since the arithmetic times will decrease, but the I/O times may remain about the same. In any case, it definitely does not make sense to count I/O statements with statements that only involve the CPU and RAM. Thus, when we are analyzing programs that include I/O statements, our initial

1 These times are for unbuffered output. If the output is buffered, average times can be reduced by about 50%.

analyses will include two terms: one for the calculation and one for I/O:

$$T(n) = T_{calc}(n) + T_{i/o}(n).$$

In our serial trapezoidal rule, we would estimate $T_{i/o} = k$ for some constant k, since we will always read three values and print a single float. (Since we're interested in performance, we'll omit such niceties as self-documenting I/O.) We can estimate k by taking the preceding timings and subtracting them from the time it takes to run the program with the I/O.

The runtimes with the two I/O statements on an nCUBE 2 are

n	512	1024	1536	2048
Time in ms	12.6	24.0	35.4	46.8

So the two I/O statements add about 1.2 milliseconds to the overall runtime, and our new performance estimate is

$$T(n) = 22.3n + 1200.$$

11.4 Parallel Program Performance Analysis

One clear difference between serial and parallel program performance estimation is that the runtime of a parallel program should depend on two variables: input size and number of processes. Thus, instead of using $T(n)$ to denote performance, we'll use a function of two variables, $T(n, p)$. It is the time that has elapsed from the moment when the first process to start actually begins execution of the program to the moment when the last process to complete execution executes its last statement. In many of our programs, $T(n, p)$ will simply be the number of statements executed by process 0 (or whichever process is responsible for I/O), since typically execution will begin with process 0 gathering and distributing input data, and it will end with process 0 printing results.

Note that this definition implies that if multiple processes are running on a single physical processor, the runtime will in all likelihood be substantially greater than if processes are running on separate physical processors. In general, we won't worry about this issue—we'll assume that each process is running on a separate physical processor.

Generally, when parallel program performance is discussed, the subject of how the parallel program compares to the serial program arises. The most commonly used measures are speedup and efficiency. Loosely, **speedup** is the ratio of the runtime of a serial solution to a problem to the parallel runtime. That is, if $T_\sigma(n)$ denotes the runtime of the serial solution and $T_\pi(n, p)$ denotes the runtime of the parallel solution with p processes,[2] then the speedup of the

2 We use the subscripts σ and π so that they can't be confused with S (the speedup) and p (the number of processes).

parallel program is

$$S(n,p) = \frac{T_\sigma(n)}{T_\pi(n,p)}.$$

There is some ambiguity in this definition. For example, is the serial program just the parallel program running with one process, or is it the fastest known serial program that solves the problem? Is the serial program to run on the fastest possible serial machine, or is it to run on a single processor of the parallel system? Most authors define the speedup to be the ratio of the runtime of the fastest known serial program on one processor of the parallel system to that of the parallel program running on p processors of the parallel system.

For a fixed value of p, it will usually be the case that $0 < S(n,p) \leq p$. If $S(n,p) = p$, a program is said to have **linear speedup.** This is, of course, a rare occurrence since most parallel solutions will add some overhead because of communication among the processes. Unfortunately, a far more common occurrence is **slowdown.** That is, the parallel program running on more than one process is actually slower than the serial program. This unfortunate occurrence usually results from an excessive amount of overhead, and this overhead is usually due to communication among the processes.

An alternative to speedup is **efficiency.** Efficiency is a measure of process utilitization in a parallel program, relative to the serial program. It is defined as

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_\sigma(n)}{pT_\pi(n,p)}$$

Since $0 < S(n,p) \leq p$, $0 < E(n,p) \leq 1$. If $E(n,p) = 1$, the program is exhibiting linear speedup, while if $E(n,p) < 1/p$, the program is exhibiting slowdown.

11.5 The Cost of Communication

In several places in the text we've already noted that communication, like I/O, is significantly more expensive than local calculation. Thus, in order to make reasonable estimates of parallel program performance, we should count the cost of communication separately from the cost of calculation and I/O. That is,

$$T(n,p) = T_{calc}(n,p) + T_{i/o}(n,p) + T_{comm}(n,p).$$

In view of this, we need to get a better understanding of what happens when two processes communicate. So suppose we are running a parallel program and process q is sending a message to process r. When process q executes the statement

```
MPI_Send(message, count, datatype, r, tag, comm)
```

and process r executes the statement

```
MPI_Recv(message, count, datatype, q, tag, comm,
    &status)
```

the details of what happens at the hardware level will vary. Different systems use different communication protocols, and a single system will execute two different sets of machine-level commands depending on whether the processes reside on the same or distinct physical processors.

We are mainly interested in the case where the processes reside on distinct processors, and for this case we can make some general observations about what happens on most systems. In this situation, the execution of the send/receive pair can be divided into two phases: a start-up phase and a communication phase. Once again, details of what the system does during these phases will vary. Typically, however, during the start-up phase the message may be copied into a system-controlled message buffering area, and the envelope data, consisting of the source rank (q), the destination rank (r), the tag, the communicator, and possibly other information, may be appended to the message. During the communication phase, the actual data is transmitted between the physical processors. There may be another phase, analogous to the start-up phase, on the receiving process, during which the message may be copied from a system-controlled buffering area into user-controlled memory.

The costs of these phases will vary from system to system, so we'll use symbolic constants to denote the costs. We'll use t_s to denote the runtime of the start-up phase, including any time spent on the receiving process copying the message into user-controlled memory, and t_c will denote the time it takes to transmit a single unit of data from one processor to another. The unit of data can be either a byte, a word, or some larger unit of data. If it isn't clear from the context, we'll specify which. Using this notation, the cost of sending a single message containing k units of data will be

$$t_s + kt_c.$$

The time t_s is sometimes called message **latency,** and the reciprocal of t_c is sometimes called the **bandwidth.**

It is surprisingly difficult to obtain reliable estimates of t_s and t_c. For a particular system, the best thing to do is to actually write programs that send messages and take timings. However, we can make some broad generalizations. On most systems, t_c is within one order of magnitude of the cost of an arithmetic operation, t_a, and t_s is from one to three orders of magnitude greater than t_c. For example, on the nCUBE 2, t_c is about 2.5 microseconds/float, and t_s is about 170 microseconds. Table 11.1 contains data on several systems.[3]

Such values should not be taken too seriously for a variety of reasons:

---

3 Most of the data in this table is taken from [12]. The t_a figures are times for a double precision addition and a multiplication on the 1000 × 1000 Linpack benchmark. The t_c figures were computed for a message size of 1 megabyte.

Table 11.1 Estimates of t_s, t_c, and t_a on several systems (all times are in microseconds; SM denotes use of shared-memory functions; t_c is time per double)

Machine	*Operating System*	t_a	t_s	t_c
Cray T3D (PVM)	MAX 1.2.0.2	0.011	21	0.30
Cray T3D (SM)	MAX 1.2.0.2	0.011	3	0.063
Intel Paragon	OSF 1.0.4	0.030	29	0.052
Intel iPSC/860	NX 3.3.2	0.030	65	2.7
Intel iPSC/2	NX 3.3.2	-	370	2.9
IBM SP-1	MPL	0.0096	270	1.1
IBM SP-2	MPI	0.0042	35	0.23
Meiko CS2 (SM)	Solaris 2.3	0.010	11	0.20
Meiko CS2	Solaris 2.3	0.010	83	0.19
nCUBE 2	Vertex 3.2	0.50	170	4.7
TMC CM-5	CMMD 2.0	-	95	0.89
Ethernet	TCP/IP	-	500	8.9

1. The communication figures tend to become dated very quickly, since computer manufacturers are always tweaking their hardware and software in order to improve the figures.
2. On many systems, MPI is layered on top of proprietary message-passing software, and, as a consequence, figures for MPI may not be as good as those reported in the table.
3. Floating point performance tends to depend highly on the application, and figures for a given application may vary widely from those in the table.

11.6 An Example: The Parallel Trapezoidal Rule

As an example of the use of these ideas, let's estimate the runtime of the parallel trapezoidal rule. We'll also try to estimate its speedup.

Once again, for ease of reference, we include the part of our parallel program that corresponds to the serial program we analyzed earlier. (As before, we'll omit the I/O.)

```
h = (b-a)/n;     /* h is the same for all processes */
local_n = n/p;   /* So is the number of trapezoids */

/* Length of each process's interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;

/* Call the serial trapezoidal function */
integral = Trap(local_a, local_b, local_n, h);
```

```
/* Add up the integrals calculated by each process */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);
```

Except for the function calls, `Trap` and `MPI_Reduce`, all of the statements take constant time and hence add a constant, c_1, to the total runtime. The `Trap` function just executes the preceding serial trapezoidal rule code on each process's interval of integration. Since each process sums the areas of n/p trapezoids, this part of the code has runtime $c_2(n/p - 1) + c_3$. If the parallel implementation of `MPI_Reduce` uses the tree-structured communication pattern we discussed in Chapter 5, then it will have runtime

$$c_4 \log_2(p)(t_s + t_c + t_a) + c_5,$$

for some constants c_4 and c_5. Here, t_a is the cost of an arithmetic operation (the sum computed after each communication phase). So the total runtime for the parallel program is

$$T(n,p) = k_1 \frac{n}{p} + k_2 \log_2(p) + k_3,$$

for some constants k_1, k_2, and k_3.

From our previous discussion, we expect k_3 to be negligible, and we've already estimated $k_1 = 22.3$ microseconds/trapezoid on an nCUBE 2. Furthermore, from our previous estimates of t_a, t_s, and t_c, we can estimate that

$$k_2 \approx 170 + 2.5 + 1.5 \approx 175\,\mu\text{sec}.$$

Thus, our overall first estimate of $T(n,p)$ on an nCUBE 2 is

$$T(n,p) = 22.3\frac{n}{p} + 175\log_2(p),$$

and our predicted speedup is

$$S(n,p) = \frac{22.3n}{22.3n/p + 175\log_2(p)}.$$

How does our prediction compare to the actual performance? Table 11.2 contains the predicted speedups and the actual speedups on an nCUBE 2 running a somewhat optimized version of the `mpich` implementation of MPI. (The limits of integration are $a = -2$ and $b = 2$. The function being integrated is e^{-x^2}.) Figure 11.1 illustrates some of these numbers graphically. As you can see, our speedup estimates are quite good if p is small. However, as p increases, our estimates deteriorate—especially if p is large and n is small. When p is small and n is large, the term $22.3n/p$ will dominate the overall runtime; when p is large and n small, its relative contribution to the parallel runtime will shrink. This suggests that our estimate of the time spent in calculating the

Table 11.2 Predicted and actual speedups of trapezoidal rule

	Number of trapezoids							
	512		*1024*		*1536*		*2048*	
Processes	*Pred.*	*Act.*	*Pred.*	*Act.*	*Pred.*	*Act.*	*Pred.*	*Act.*
2	1.9	2.0	2.0	2.0	2.0	2.0	2.0	2.0
4	3.6	3.6	3.8	3.8	3.8	3.9	3.9	3.9
8	5.8	5.7	6.8	6.7	7.1	7.1	7.3	7.4
16	8.1	7.6	10.7	10.4	12.1	11.8	12.8	12.7
32	9.3	8.1	14.4	13.4	17.6	16.3	19.8	19.0

local integrals is fairly accurate, but we have underestimated the cost of the call to `MPI_Reduce`.

If we try fitting the formula

$$k_1 \frac{n}{p} + k_2 \log_2(p) + k_3$$

to the actual runtimes, we find that $k_1 = 22.2, k_2 = 190$, and $k_3 = 0$. Thus, our a posteriori analysis seems to be correct. That is, we have underestimated the runtime of the global sum function. In fact, if we take timings of the global sum, we find that there is an additional 15 μsec overhead that we didn't consider in our earlier estimate. This additional overhead is probably due to the cost of the function call and the cost of calculating the addresses (source and destination) for the message passing.

11.7 Taking Timings

If you have exclusive access to the processors on which you will be running your program, it's fairly easy to time your program: synchronize the processes at the beginning of the code you wish to time, start a clock on each process, synchronize the processes at the end of the code you wish to time, stop the clock, and compute the elapsed time. If, as in the trapezoidal rule program, the runtime of one process (process 0 in this case) dominates the runtime—i.e., starts first and finishes last—the synchronization steps can be omitted.

In order to synchronize the processes, you can call `MPI_Barrier`:

```
int MPI_Barrier(
        MPI_Comm  comm  /* in */)
```

This function causes each process in `comm` to block until every process in `comm` has called it.

In order to actually take the timings, there are a number of possibilities. MPI provides a timer:

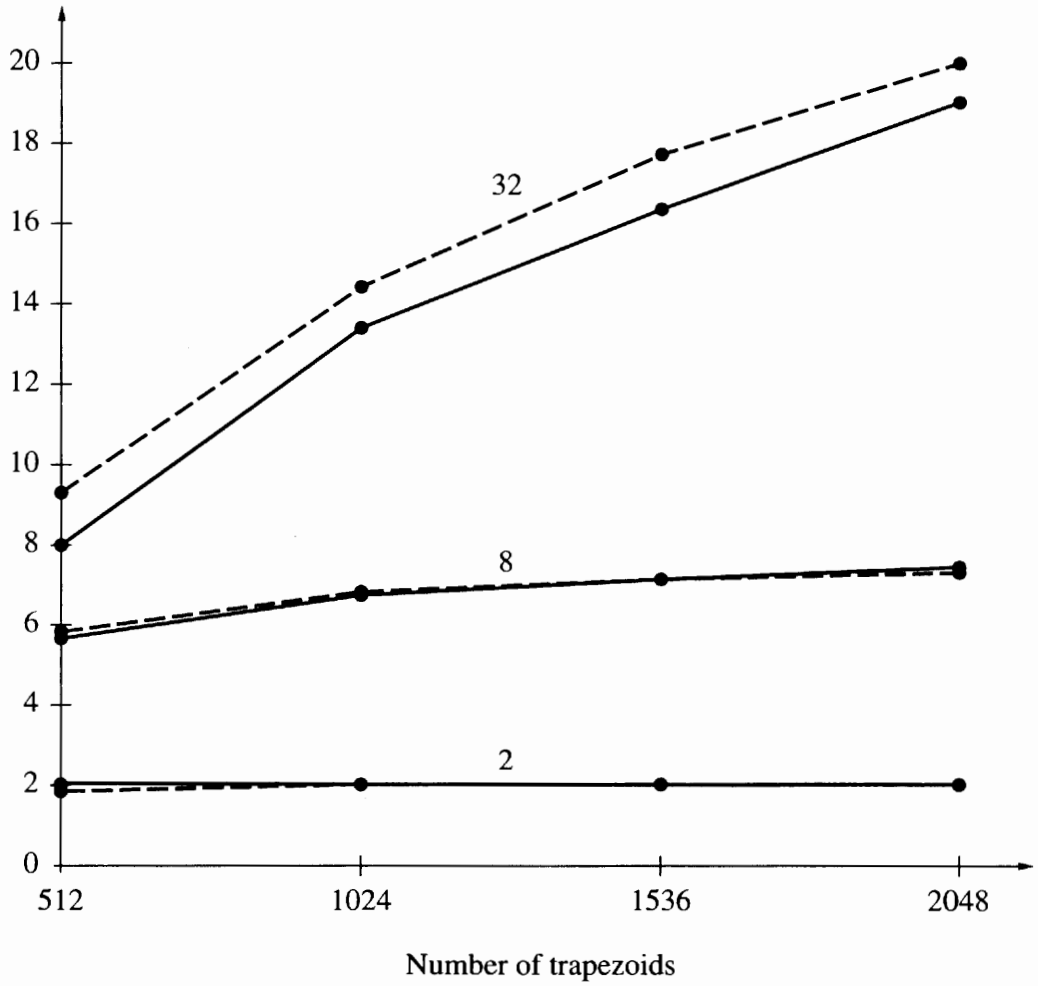

Figure 11.1 Plot of predicted (dashed curves) and actual (solid curves) speedups of trapezoidal rule for $p = 2, 8, 32$

```
double MPI_Wtime(void)
```

It returns a double precision value that represents the number of seconds that have elapsed since some point in the past. Its precision can be found by calling `MPI_Wtick`:

```
double MPI_Wtick(void)
```

For example if `MPI_Wtime` is incremented every microsecond, then `MPI_Wtick` will return 10^{-6}.

Thus, an MPI program can determine elapsed time as follows:

```
double start, finish;

MPI_Barrier(comm);
start = MPI_Wtime();
        .
        .
        .
/* Code being timed */
```

```
        ⋮
MPI_Barrier(comm);
finish = MPI_Wtime();
if (my_rank == 0)
    printf("Elapsed time = %e seconds\n",
        finish - start);
```

If the code you're timing is very short, you can estimate the cost of MPI_Barrier and MPI_Wtime by writing a program that estimates their cost and subtract this cost from the elapsed time.

It should be stressed that MPI_Wtime returns *wall-clock* time. That is, it makes no allowance for such things as system time. So if a process is interrupted by the system, the time it spends idle will be added into the elapsed time.

11.8 Summary

To summarize, in order to estimate the performance of a parallel program, we should carry out the following steps:

1. Develop a formula for the runtime of the serial program that contains symbolic constants.
2. Estimate the size of the symbolic constants in the formula for the serial runtime. Some of these can be estimated from your knowledge of the algorithm and your knowledge of the performance characteristics of your system. In our example, we were able to determine that in our linear formula, the constant term was close to zero. In order to determine the slope, however, we had to actually take timings.
3. Develop a formula for the runtime of the parallel program that contains symbolic constants.
4. Estimate the value of the symbolic constants on the basis of the performance characteristics of your system and the results of the serial program analysis.
5. Estimate the speedup.

In order to actually take the timings, you can use the function MPI_Barrier to synchronize the processes and MPI_Wtime to measure elapsed time. Their syntax is

```
int MPI_Barrier(
        MPI_Comm  comm  /* in */)

double MPI_Wtime(void)
```

11.9 References

Foster [17] provides an excellent discussion of the performance analysis of parallel programs.

Most of our machine-specific performance figures are based on the work of Dongarra and Dunigan. See [12] for a more complete discussion of these figures.

Kumar et al. [26] derive a number of asymptotic parallel program performance estimates.

Bailey [4] has a nice discussion of tricks that have been used to obscure poor performance of parallel programs.

11.10 Exercises

You may want to write programming assignment 1 before working on the exercises.

1. There are some anomalous examples where $S(n,p) > p$. Such programs are said to exhibit **superlinear** speedup. Can you think of a program that might show superlinear speedup?
2. Estimate the runtime of the trapezoidal rule program that used a simple loop of receives on process 0. Compare your predicted speedups with the actual speedups.
3. a. Estimate the runtime of a simple serial matrix multiplication program (see section 7.1). Write a serial matrix multiplication program and compare your prediction to the actual times.
   b. Estimate the performance of the basic parallel matrix multiplication algorithm that partitions the matrix by block rows. Compare your prediction of the performance with the actual performance.
   c. Estimate the performance of Fox's algorithm. (See [26] if you need some help with the formulas.) Compare your prediction of the performance with the actual performance.
   d. Use least squares to determine the size of the constants that will make your predicted runtimes best fit the parallel runtimes.
4. In order to estimate the runtime of Jacobi's method, we need to include a constant representing the number of iterations. Estimate the cost of a single iteration of the serial and parallel Jacobi's method programs. Compare your estimates to the actual performance.
5. Recall programming assignment 4 from Chapter 10, involving a cellular automaton. In it you were asked to try to determine which distribution of the grid was preferred. In light of your recently acquired knowledge of the cost of communication and computation, which distribution of the grid do you think will minimize the runtime?

11.11 Programming Assignments

1. Write a short program that estimates t_s and t_c for your system. This can be done by repeatedly sending fixed-size messages from one process to another. When the receiving process receives its message, it should reply to the sending process. This process should then be repeated with messages of different sizes. You can use least squares to fit a line to the resulting (message size, time) pairs. The intercept will approximate t_s and the slope t_c.
2. Modify your solution to programming assignment 1 so that it uses different derived datatypes (contiguous, vector, indexed, and struct) to communicate various amounts of data. Compare the costs of sending the different derived datatypes.
3. Modify your solution to programming assignment 1 so that the data is first packed on the sending process and then unpacked on the receiving process. Compare the costs to the costs you determined in the preceding two exercises.
4. Study the performance of the cellular automaton program (programming assignment 4 from Chapter 10) using different distributions of the grid. Don't include the printing of the grid in your timings. How does the actual performance compare to the performance you predicted in exercise 5?