CHAPTER 12

More on Performance

IN THIS CHAPTER WE CONTINUE our discussion of performance. We'll begin with a discussion of Amdahl's law, which seems to imply that there is little hope that large-scale parallel systems will be able to achieve large speedups. During the period 1967–1988, Amdahl's law was the bête noire of researchers in parallel computing. In 1988, however, three researchers at Sandia National Labs reported speedups of more than 1000 on a 1024-processor nCUBE. This resulted in a serious reexamination of Amdahl's law. One of the principal outcomes of this reexamination was the idea of **scalability**. Essentially, a parallel program is scalable if it is possible to maintain a given efficiency as the number of processes is increased by simultaneously increasing the problem size.

After our discussion of these somewhat theoretical issues, we turn to some more practical issues. We discuss some potential problems with estimating parallel program performance, and we discuss MPI's profiling interface. We close with a brief discussion of a software tool that has been developed to analyze the performance of parallel programs.

12.1 Amdahl's Law

Suppose that a certain program requires time T_σ to solve an instance of a problem using a single process. By an "instance" of a problem, we mean the problem together with a specific set of input data—e.g., integrating e^{-x^2} over the interval $[0, 1]$ using the trapezoidal rule with 512 trapezoids. Suppose also that this program contains a fraction, r, of statements ($0 \leq r \leq 1$) that are "perfectly parallelizable." That is, regardless of how large p is, this fraction of the program has linear speedup and hence runtime rT_σ/p when it is parallelized

and run with p processes. However, suppose that the remaining fraction, $1 - r$, of the program is inherently serial. That is, regardless of how large p is, the runtime of this fraction is $(1 - r)T_\sigma$ when it is parallelized with p processes. An example of what might be inherently serial code is an input segment, where a single value is needed by all the processes. We'll defer, for the moment, a discussion of how reasonable these assumptions are.

Under these assumptions, the speedup of the parallellized program with p processes is

$$\begin{aligned} S(p) &= \frac{T_\sigma}{(1-r)T_\sigma + rT_\sigma/p} \\ &= \frac{1}{(1-r) + r/p}. \end{aligned}$$

If we differentiate S with respect to p, we get

$$\frac{dS}{dp} = \frac{r}{[(1-r)p + r]^2} \geq 0.$$

So $S(p)$ is an increasing function of p, and as $p \to \infty$, we see that

$$S(p) \to \frac{1}{1-r}.$$

So $S(p)$ is bounded above by $(1 - r)^{-1}$. For example, if $r = 0.5$, the maximum possible speedup is 2. If $r = 0.75$, the maximum possible speedup is 4, and if $r = 0.99$, the maximum possible speedup is 100. In particular, if $r = 0.99$, even if we use 10,000 processes, we can't get a speedup of better than 100, which would correspond to an efficiency of 0.01!

Is Amdahl's law correct? If so, it would seem to imply that it's hopeless for us to try to use massive parallelism (i.e., hundreds or thousands of processes). The mathematics is clearly correct. Hence, insofar as we can trust its hypotheses, it is correct. So is it reasonable then, to accept its hypotheses? Certainly given an instance of a problem and a program for solving the problem, it seems reasonable to assume that a certain fraction of statements in the program cannot be parallelized. Indeed, it seems excessively optimistic to assume that the remainder of the statements are "perfectly" parallelizable.

However, what may not be reasonable is the use of arbitrarily many processes to solve a given instance of the problem. In other words, the parameter n, the problem size, is conspicuously absent from all these equations. As an extreme example, it seems clear that if we try to use more than 500 processes to solve a problem instance that only requires 500 calculations, we'll be wasting our resources. Rather, we should use more processes to solve larger instances of a problem, and perhaps if we take a larger instance of a problem, the serial fraction will get smaller. This was one of the key ideas in the development of the first programs that obtained speedups of more than 1000. For example, in our trapezoidal program, there are essentially two serial statements: reading the input values and printing the solution. So as n, the number of trapezoids, gets larger, the serial fraction of the program will approach zero.

12.2 Work and Overhead

The observation that in many programs, as the problem size increases, the inherently serial fraction decreases, suggests a general view of parallel programming that allows us to make more optimistic predictions than Amdahl's law. An inherently serial segment of a program can be parallelized in essentially one of two ways: one process can execute the statements in the segment while the others remain idle, or all the processes can execute the statements—that is, the statements in the segment are replicated by each process. Both of these approaches are sources of parallel **overhead**: they increase the total amount of work performed by the parallel program over the amount performed by the serial program.

Let's make this idea precise. The amount of **work** done by a serial program is simply the runtime:

$$W_\sigma(n) = T_\sigma(n).$$

The amount of work done by a parallel program is the sum of the amounts of work done by each process:

$$W_\pi(n,p) = \sum_{q=0}^{p-1} W_q(n,p),$$

where $W_q(n,p)$ is the amount of work done by the program executed by process q. Now observe that (unlike in the real world) *work includes idle time.* If we're timing a serial program and it remains idle for some reason (e.g., it's waiting for input), then this idle time is included in the total runtime or the work. Thus, each $W_q(n,p)$ should include the time that process q is idle. But this vastly simplifies our calculations, for it implies that for each q, $W_q(n,p) = T_\pi(n,p)$. That is, $W_q(n,p)$ is simply the parallel runtime because process q is either doing useful work or idle for the duration of the execution of the parallel program. So our formula for $W_\pi(n,p)$ is simply

$$W_\pi(n,p) = pT_\pi(n,p).$$

Thus, an alternative definition of efficiency is

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

So in this setting, linear speedup, or $E(n,p) = 1$, is equivalent to the statement "The amount of work done by the parallel program is the same as the amount of work done by the serial program."

We can formalize our definition of overhead: since it is the amount of work done by the parallel program that is not done by the serial program, it's simply

$$T_O(n,p) = W_\pi(n,p) - W_\sigma(n) = pT_\pi(n,p) - T_\sigma(n).$$

Alternatively, we could define **per-process overhead** as the difference between the parallel runtime and the ideal parallel runtime that would be obtained with linear speedup:

$$T'_O(n,p) = T_\pi(n,p) - T_\sigma(n)/p.$$

In this setting we can think of our definition of overhead as "total overhead":

$$T_O(n,p) = pT'_O(n,p).$$

As an example, let's find $T_O(n,p)$ for the trapezoidal rule program. In our analysis of its runtime, we found that

$$T_\sigma(n) = k_1 n,$$

and

$$T_\pi(n,p) = k_1 n/p + k_2 \log_2(p).$$

Hence, the work done by the parallel program is

$$W_\pi(n,p) = k_1 n + k_2 p \log_2(p)$$

and the overhead function for the parallel trapezoidal rule program is

$$T_O(n,p) = k_2 p \log_2(p).$$

In other words, the total work done by the parallel program will differ from the total work done by the serial program by $k_2 p \log_2(p)$. This is true regardless of the value of n. So if n is large relative to p, we would expect that the amount of work done by the parallel program would be very close to the amount of work done by the serial program.

12.3 Sources of Overhead

Although we haven't formalized this observation, previous comments imply that there are three main sources of overhead:

1. communication
2. idle time
3. extra computation

It's clear that interprocess communication is not an issue in a serial program. Hence, any interprocess communication will contribute to T_O. Idle time may or may not contribute to overhead. Unfortunately, it generally does: one or more processes will remain idle while they wait for information from another process. For example, in the trapezoidal rule program, processes $1, 2, \ldots, p-1$ must remain idle while process 0 reads in the input data. Finally, there are

numerous opportunities for a parallel program to do extra computation. Some of the most obvious are performing calculations not required by the serial program and replicated calculations. An example of the former is the calculation in the trapezoidal rule program of each process's subinterval of integration. Clearly this calculation isn't required in the serial program. An example of the latter occurs when each process calculates the number of trapezoids used in estimating the integral over its subinterval. Since the program assumes that p divides n, this value is the same on all the processes. (Note that this is also an example of a calculation not required by the serial program.)

The formula for the overhead in the parallel trapezoidal program is simply $p \times$ (the time to execute `MPI_Reduce`). That is, if we ignore I/O, the vast majority of the overhead comes from the call to `MPI_Reduce`. The bulk of the time for this function comes from communication. However, there is also idle time and extra computation: recall that using the tree-structured communication, each process other than the root (in our case, process 0) is eventually idled before the computation is complete. Also, each process must do some calculations at each stage in order to determine which process it communicates with.

12.4 Scalability

Amdahl's law says that there is a definite upper bound on the speedup that can be attained in a parallel program. In terms of efficiency, it asserts that if r is the fraction of the program that can be parallelized, then

$$\begin{aligned} E &= \frac{T_\sigma}{p(1-r)T_\sigma + rT_\sigma} \\ &= \frac{1}{p(1-r) + r}. \end{aligned}$$

So an equivalent formulation of Amdahl's law would conclude that the efficiency achievable by a parallel program on a given problem instance will approach zero as the number of processes is increased.

Let's look at this assertion in terms of overhead. Since

$$T_O(n,p) = pT_\pi(n,p) - T_\sigma(n),$$

efficiency can be written as

$$\begin{aligned} E(n,p) &= \frac{T_\sigma(n)}{pT_\pi(n,p)} \\ &= \frac{T_\sigma(n)}{T_O(n,p) + T_\sigma(n)} \\ &= \frac{1}{T_O(n,p)/T_\sigma(n) + 1}. \end{aligned}$$

Thus, we might reply to Amdahl that the behavior of the efficiency as p is increased is completely determined by the overhead function, $T_O(n,p)$. In the trapezoidal rule program, since

$$T_O = k_2 p \log_2(p),$$

the efficiency attained on a given problem instance will approach zero as the number of processes is increased. This, is completely expected: both our theoretical estimates and our actual timings show that for a given value of n, the efficiency decreases as p is increased. However, this formula also shows that for fixed p, as n is increased, we can obtain efficiencies as close to one as we like. What happens if we increase both n and p simultaneously? If we substitute our formulas for T_σ and T_O into the new formula for E, we get

$$E(n,p) = \frac{1}{k_2 p \log_2(p)/k_1 n + 1}.$$

Thus, if we increase n and p at the same rate, our efficiency will decrease since $p \log_2(p)$ will grow slightly faster than $n = p$. However, if n is increased as fast as $p \log_2(p)$, then we can maintain a constant efficiency.

For example, if we examine the speedups in Table 11.2, we see that if $p = 4$ and $n = 512$, the efficiency is $E = 3.6/4 = 0.90$. If we wish to maintain this efficiency with $p = 8$ processes, we shouldn't double n. Rather, we should choose n so that the value of

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

is constant. Substituting our values for n and p, we see that

$$\frac{k_2 4 \log_2(4)}{k_1 512} = \frac{k_2 8 \log_2(8)}{k_1 n},$$

and hence the new value of n should be 1536. Examining the table of speedups again, we see that if $p = 8$ and $n = 1536$, then the efficiency is $E = 7.1/8 = 0.89$. More generally, we see that if n is increased at the same rate as $p \log(p)$, then we can maintain any desired level of efficiency.

This last observation is the key to scalability: a parallel program is **scalable** if, as the number of processes (p) is increased, we can find a rate of increase for the problem size (n) so that efficiency remains constant. Of course, this definition suggests that there are varying degrees of scalability. If we can solve our efficiency formula for n in terms of p, we can obtain an explicit formula for the rate at which n must grow. For example, the efficiency formula for the trapezoidal rule program is

$$E(n,p) = \frac{k_1 n}{k_2 p \log_2(p) + k_1 n}.$$

Solving for n gives

$$n = \frac{E}{1-E}\frac{k_2}{k_1} p \log_2(p).$$

That is, in order to maintain a constant efficiency E, n should grow at the same rate as $p \log_2(p)$ (which we just observed).

A couple of caveats are in order. First, in general it may be very difficult to solve for n in terms of p. For example, if $T_\sigma(n)$ has degree 5 in n, there is no general approach that will allow us to solve for n in terms of E and p. Second, and more important, don't lose sight of practical considerations when manipulating formulas. For example, suppose we have written a parallel program that achieves linear speedup. Then

$$T_O(n,p) = pT_\pi(n,p) - T_\sigma(n) = pT_\sigma(n)/p - T_\sigma(n) = 0.$$

So in theory, $E = 1$, regardless of n and p. However, in a real-world problem this formula will fail very quickly.

For example, suppose we have to add two distributed vectors that are to remain distributed after addition. Then each process simply adds its local components. If n, the order of the vectors, is evenly divisible by p, and each process is assigned n/p components of each vector, then the program achieves linear speedup. The key here is that n is evenly divisible by p. In particular, if $p > n$, then some processors will remain idle and the overhead will no longer be 0. In other words, in order to maintain an efficiency of 1, for arbitrary p, n must grow as p.

12.5 Potential Problems in Estimating Performance

Now that we've developed a theoretical underpinning for performance estimation, let's return to a discussion of practical performance estimation and discuss some problems that we might encounter. In Chapter 11 we have already discussed several difficulties. The most important of these was getting reliable estimates of t_a, t_s, and t_c. Another problem we encountered was underestimating the cost of a collective communication function. Both of these problems have to do with an empirical analysis of a *component* of the program, and as such are relatively easy to solve. For example, as soon as we suspect that our estimate of the runtime of `MPI_Reduce` is inaccurate, we can write a short timing analysis program and determine whether this is, in fact, a problem. However, there are other problems that are not so easily remedied. In this section we'll discuss some of the more common problems.

12.5.1 Networks of Workstations and Resource Contention

Unfortunately, timings on networks of workstations are almost always suspect. If we write a parallel program for a network of workstations, the ideal environment should allow the user exclusive access to processors, peripherals, and network during execution. That is, only the absolute minimum set of system processes and no other user processes should run on each workstation, and there should be no network traffic other than that generated by our

program. Of course, this ideal environment is virtually impossible to achieve. Even if we manage to find a time when no one else is logged on to any of the machines, it's still entirely possible, for example, that a mail message might be received by one of the machines during the execution of our program. However, unless we have a program that runs for a long time, such interruptions will be relatively rare, and if our program does run for a long time, we should expect such random interruptions and, as a consequence, we should be cautious about rejecting predicted timing analyses that differ from actual runtimes.

So the first rule of thumb here is to try to run our programs when there are as few users and as little network traffic as possible. Unfortunately, this tends to be at unpleasant times like 3 AM, but, heck, we're programmers, and programmers like to stay up all night. Right? The second rule of thumb is that if our program takes a long time to run and/or we are using a network with relatively heavy traffic, we should expect to get timings that are at variance with our predicted timings. In most settings (e.g., ethernet-connected workstations), this variance is more often due to network traffic than reduced access to processors. In other words, it's more likely that the network is saturated than that the processors are heavily loaded, but this will depend on your particular installation.

In order to compare predicted times with actual times, it's not unreasonable to run the program repeatedly and see if the runtimes are clustering around a fixed value. Usually this will be the case. If it is, this cluster time is probably more reliable than an average of all runtimes.

Networks of workstations provide an extreme example of a problem that can be encountered on virtually any parallel system: resource contention. For example, if a program generates large amounts of data that are stored on a single disk, the processes may be forced to access the disk sequentially—even on a dedicated parallel system. The situation could be even worse if a parallel system provided virtual memory but very limited I/O capabilities.

12.5.2 Load Balancing and Idle Time

We've mentioned idle time as a source of overhead in a parallel program. The most common cause of idle time is load imbalance: one or more processes must carry out more computation than one or more other processes. There can be other causes. A common source occurs when one process must wait for some system resource to become available before it can proceed with its computations. For example, in a network of workstations, if the network is saturated, a process may be forced to wait to send a message. In any case, when we're carrying out performance analyses, we can identify two basic sources of idle time: predictable or regular, and unpredictable or irregular.

By *predictable* or *regular* idle time, we mean idle time that will occur every time the program is run with more than one process—regardless of the input data. For example, when we use a tree-structured broadcast to communicate a value from one process to the other processes, if there is a point of syn-

chronization before a broadcast, most of the processes will necessarily be idle while they wait for the data to propagate through the broadcast tree.

By *unpredictable* or *irregular* idle time, we mean idle time that does not occur every time the program is run. For example, a program that searches a binary tree might partition the problem by assigning disjoint subtrees to different processes. If the number of nodes in the different subtrees varies, then one or more of the processes may be idle. However, this is clearly data dependent. For example, if the tree is generated as the program is run, there may be no way of knowing before the program is run whether one subtree will contain fewer nodes than another. Another source has been previously alluded to: system limitations. If your program doesn't have exclusive access to the system resources when it's run, then one or more processes may be idled while some system resources are used by another program.

The reason for this distinction is clear: we can anticipate regular idle time in our performance analysis, and, as a consequence, it should not be a source of inaccurate performance prediction. The second type of idle time is more problematic. We've already discussed various options if we don't have exclusive access to the system. If, on the other hand, the source of the imbalance is in the data or algorithm, we need to determine how serious the resulting load imbalances will be.

As an example, consider the search tree problem. If we can make reasonable a priori estimates of the structure of the trees, we can determine how serious the load imbalances will be. If we determine that, in the vast majority of cases, each nonleaf node will have the same number of children, and all the leaves will be in about the same level, then simply assigning a distinct subtree to each process will cause little difficulty with load imbalance and idle time. If, on the other hand, we determine that the structure of the trees is highly variable, we should probably look for another partitioning algorithm.

None of this should be taken to imply that we believe that any idle time is desirable. We are simply observing that, in some cases, we can include predicted idle time in our performance analysis, while in other cases it may be very difficult to do so, and, as a consequence, our analyses will be unreliable.

12.5.3 Overlapping Communication and Computation

If each compute node of your system contains a communication coprocessor, or if you use nonblocking communications (see Chapter 13), it's possible that communications and computation can be overlapped. This can cause difficulties in performance prediction because it may be extremely difficult to make any a priori estimate of the extent to which these are overlapped. For example, we can estimate t_s and t_c by running the following code and using least squares to fit a line to the data:

```
/* two process ping-pong */
if (my_rank == 0) {
```

```
    for (test = 0, size = min_size;
            size <= max_size; size = size + incr, test++) {
        for (pass = 0; pass < MAX; pass++) {
            MPI_Barrier(comm);
            start = MPI_Wtime();
            MPI_Send(x, size, MPI_FLOAT, 1, 0, comm);
            MPI_Recv(x, size, MPI_FLOAT, 1, 0, comm,
                &status);
            finish = MPI_Wtime();
            raw_time = finish - start - wtime_overhead;
            printf("%d %f\n", size, raw_time);
        }
    }
} else { /* my_rank == 1 */
    for (test = 0, size = min_size; size <= max_size;
             size = size + incr, test++) {
        for (pass = 0; pass < MAX; pass++) {
            MPI_Barrier(comm);
            MPI_Recv(x, size, MPI_FLOAT, 0, 0, comm,
                &status);
            MPI_Send(x, size, MPI_FLOAT, 0, 0, comm);
        }
    }
}
```

However, consider the following segment of code:

```
MPI_Barrier(comm);
if (my_rank == src)
    MPI_Send(x, m, MPI_FLOAT, dest, tag, comm);
else if (my_rank == dest)
    MPI_Recv(x, m, MPI_FLOAT, src, tag, comm,
        &status);
for (i = 0; i < k; i++)
    y[i] = w[i] + z[i];
```

We would predict that the runtime for the code following the call to `MPI_Barrier` would be

$$t_s + mt_c + kt_a.$$

However, if we have communication coprocessors (and smart compilers), the send/receive pair may be run simultaneously with the `for` loop, and the actual runtime might be

$$\max\{t_s + mt_c, kt_a\}.$$

In this simple situation, it's not difficult to predict the actual runtime. However, if we repeatedly encounter these situations or fail to synchronize the processes before they occur, the cumulative effects may make it extremely difficult to accurately predict runtime. In such cases, it is necessary to rely almost completely on empirical data.

12.5.4 Collective Communication

The problem of overlapping communication and computation can further complicate the prediction of the runtime of collective communication. However, even greater difficulties can arise if you don't know what algorithm(s) a given collective communication is using. For example, some implementations of MPI simply use a linear `for` loop and `MPI_Send/Recv` to implement a broadcast:

```
if (my_rank == root) {
    for (proc = 0; proc < p; proc++)
        MPI_Send(x, size, datatype, proc, tag, comm);
}
MPI_Recv(x, size, datatype, root, tag, comm, &status);
```

while, as we've already seen, other implementations use a tree structure. This problem isn't very difficult to diagnose. However, an implementation may use several different algorithms depending on, for example, the size of the message. There may also be lower-level optimizations that can be exploited by coding in assembler. So, once again, it may be necessary to simply rely on empirical data for runtime prediction.

12.6 Performance Evaluation Tools

You may already be familiar with some performance evaluation tools that are commonly available for serial programs. Many UNIX systems provide the tool `prof` (an abbreviation for *profile*). This program is used to provide information on how much time is being spent in various parts of the program during execution—such information is called an **execution profile.** Typically `prof` is used by compiling a program with the `-p` flag. Then each time the program is run, a **log file** called `mon.out` is created. The log file contains information on how much time is spent in various parts of the program, and this data can be examined by calling the function `prof`.

The exact contents of `mon.out` and the data displayed by `prof` will vary from system to system. Typically they will have one of two forms: counts of the number of times various parts of the program have been executed or, more often, percentages of execution time spent in various parts of the program. (This latter form of the data is, properly speaking, an execution profile.) In order to generate the counts, one typically links the program with special libraries. These libraries contain copies of all the functions in the libraries you normally link your program with, except that the functions in the special libraries contain counters that record the number of times each function is called. In order to generate the times, the program is interrupted at fixed intervals by the operating system and data is collected on which function (or statement) is being

executed. This data can then be used to get a rough picture of how much time is being spent in various parts of the programs.

Typically both forms of profiling are available to users of MPI. For example, if you are running a program on a network of workstations, you can simply compile your MPI program with the `-p` option, and your system will generate a `mon.out` file on each machine. (Note that this may cause problems if the directory containing your executable is NFS-mounted on multiple machines in your network.)

Unfortunately, simply generating a serial profile of each executable frequently doesn't provide us with enough information to make detailed analyses of where the time is being spent in our programs. For example, the profiling data may tell you that 50% of process 0's execution time was spent in calls to `MPI_Recv`, but this won't tell you whether the time was spent equally in all calls to `MPI_Recv` or whether one call took most of the time, and, if the latter is the case, it won't tell you which call it was and which process called the matching `MPI_Send`. Even worse, the profiling may show that, say, 80% of the runtime was spent in function *A*, while 20% was spent in *B*, but *A* was perfectly parallel, while *B* was executed sequentially by the processes. In this situation, we're liable to ignore *B* and spend all our time trying to improve *A*.

So we need a unified profile of the behavior of a parallel program. There are several difficulties with providing such a profile. In the first place, many distributed-memory systems don't have a single, "universal" clock, and the clocks on the various processors are not synchronized. So it is very difficult to tell how events on one process correspond to events on another. Further, even if we could synchronize the clocks, there's no guarantee that each of the clocks runs at the same rate. Finally, in trying to overcome these difficulties, we may find that we've seriously affected the overall performance of the program, and that it may be difficult to distinguish performance problems that are a result of our code and performance problems that are a result of the profiling code. It should be mentioned that this last problem may be encountered in profiling serial programs. However, on most modern systems, its effect is either very small or nonexistent, while on parallel systems, the effect may be very large. For example, the profiling software may create a single log file recording events. Since the events themselves will, in general, be distributed, the recording of the information may involve communication so that the data can be stored in a single location.

As a consequence of these problems, the development of performance evaluation tools for parallel programs remains an active area of research, and there are a number of tools currently under development. Since a comprehensive discussion of these tools is beyond the scope of this text, we'll limit ourselves to a discussion of MPI's profiling interface and one tool that is commonly available with MPI implementations.

12.6.1 MPI's Profiling Interface

We mentioned earlier that in order to use some serial profiling tools, it's necessary to link in special libraries before your program is executed (or at execution time for dynamically linked libraries). In order to understand the reason for this, suppose, for example, that you wish to determine how many times a function is called during execution of your program. If you wrote the function, you could define a static global counter and make the first executable statement of the function increment this variable. Before the program completes, you could simply print the value of the variable.

If you haven't written the function, however, this basic strategy could be very painful, or even impossible, to use. Since you haven't written the function, the global counter must be incremented outside the function. Thus, each time you call the function, you'll have to add a statement to your source code incrementing the counter. Furthermore, this may actually be impossible: if calls to the function are contained in other system-defined functions, you won't be able to count these calls. A solution to this problem is use a system-defined function with its own built-in counter. Of course, you don't want to waste CPU cycles on incrementing counters or taking timings during a production run. So a simple solution to the problem is to have two distinct libraries of functions: a profiling library in which each function has a built-in counter or timer, and a production library in which each function is optimized and, as a consequence, doesn't contain counters or timers.

This idea is at the heart of MPI's profiling interface. The MPI standard requires that each implementation allow each MPI function to be called by its usual name, and the usual name preceded by a capital "P". For example, a process can send the `float` stored in `x` to process 0 with either

```
MPI_Send(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD)
```

or

```
PMPI_Send(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD)
```

How does this help? Each user can write his or her own profiling implementation—even if he or she doesn't have access to the MPI source code.

For example, suppose we want to find out how much time is being spent by our program in calls to `MPI_Send`—including the calls made by the MPI functions themselves. We can write our own `MPI_Send` function:

```
static double send_time = 0.0;

int MPI_Send(void* buffer, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm) {
    double start_time;
    double finish_time;
    int return_val;
```

```
    start_time = MPI_Wtime();
    return_val = PMPI_Send(buffer, count, datatype,
                     dest, tag, comm);
    finish_time = MPI_Wtime();
    send_time = send_time + finish_time - start_time;
    return return_val;
}
```

Now when we link our program, we first link with the file containing this code. Then we link with the MPI libraries. So when `MPI_Send` is called, our own version will be executed. It, in turn, will call the real `MPI_Send` using the call to `PMPI_Send`.

12.6.2 Upshot

`Upshot` is a parallel program performance analysis tool that comes bundled with the public domain `mpich` implementation of MPI. See the references at the end of the chapter and Appendix B for information on obtaining a copy. It should be stressed that `Upshot` is not a part of MPI. We discuss it here because it has many features in common with other parallel performance analysis tools, and it is readily available for use with MPI.

`Upshot` provides some of the information that is not easily determined if we use data generated by serial tools such as `prof` or simply add counters and/or timers using MPI's profiling interface. It attempts to provide a unified view of the profiling data generated by each process by modifying the time stamps of events on different processes so that all the processes start and end at the same time. It also provides a convenient form for visualizing the profiling data in a **Gantt chart.** A Gantt chart is simply a series of timelines—one per process–running side by side. The timeline corresponding to process q shows the state of process q at each instant of time by color-coded bars. For examples of Gantt charts, see Figures 12.1 and 12.2.

There are basically two methods of using `Upshot`. In the simpler approach, we can link our source code with appropriate libraries and obtain information on the time spent by our program in each MPI function. If we desire information on more general segments or states of our program, we can get it to provide custom profiling data by adding appropriate function calls to our source code.

Let's look at each approach. Using the parallel trapezoidal rule program as an example, in order to use the first approach, we just link our program with the profiling library that comes with the code. When we run the program, it generates a log file containing information on the time spent in the various MPI communication functions. After execution is completed, we simply run `Upshot` on the log file. If the parallel trapezoidal rule is run on four processors of an nCUBE with 2048 trapezoids, the Gantt chart generated from the log file

Figure 12.1 `Upshot` output using automatic profile generation

looks something like Figure 12.1. Note that bars are generated only for MPI communication function calls; other parts of the program are indicated by a horizontal line. Also note the huge amount of time spent by processes 1–3 in the call to `MPI_Bcast`. Most of this clearly represents idle time. When `Get_data` is called, all the processes build the derived type before the broadcast, but process 0 must also read in the limits of integration and the number of trapezoids, which evidently takes a long time. While process 0 is reading in this data, the remaining processes block in the call to `MPI_Bcast`, waiting for process 0 to finish reading in the data and initiate the broadcast.

Alternatively, we can define various parts of the code that we wish to study. For example, we might want information on the following parts of the trapezoidal rule program:

1. Construction of derived datatype
2. Broadcast
3. Local calculations for trapezoidal rule
4. Reduce

In order to get this information, we need to define the states we're interested in. This involves three additional function calls for each state. The first describes the state, and the remaining two mark its beginning and end. In our example, we could describe the states by adding the following code before the call to `Get_data`:

```
if (my_rank == 0) {
    MPE_Describe_state(1, 2, "Derived", "gray");
    MPE_Describe_state(3, 4, "Broadcast", "black");
    MPE_Describe_state(5, 6, "Trapezoid", "white");
    MPE_Describe_state(7, 8, "Reduce", "gray");
}
```

These calls specify a pair of integers, a name, and a color for each state. The integers are used to mark the beginning and the end of code corresponding to the state. The name and the color are used in the display generated by Upshot.

In order to identify the states, we place calls to MPE_Log_event at the beginning and end of each code segment corresponding to one of our states. For example, in order to mark the code segments for building the derived datatype and the broadcast, the body of Get_data would be modified as follows:

```
if (my_rank == 0){
    scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
}

MPE_Log_event(1,0,"start Derived");
Build_derived_type(a_ptr, b_ptr, n_ptr, &mesg_mpi_t);
MPE_Log_event(2,0,"end Derived");

MPE_Log_event(3,0,"start Bcast");
MPI_Bcast(a_ptr, 1, mesg_mpi_t, 0,
    MPI_COMM_WORLD);
MPE_Log_event(4,0,"end Bcast");
```

So we mark the start of the state with the first integer we defined in MPE_Describe_state, and we mark the end of the state with the second. We won't make use of the remaining arguments: it suffices to know that the second is an int and the third is a string.

In order to start and finish logging, we need to include calls to MPE_Init_log and MPE_Finish_log. MPE_Init_log takes no arguments, and it should be called at some point between the call to MPI_Init and the calls to MPE_Describe_state. MPE_Finish_log takes a single argument (a string), the name of the log file. For example, we might use

```
MPE_Finish_log("customprof.log")
```

This would create a log file called customprof.log.

Finally, it should be stressed that none of these MPE functions is part of MPI. Consequently, it's necessary to add the preprocessor directive

```
#include "mpe.h"
```

It's also necessary to link your program with the mpe libraries. Ask your local expert for details.

With our trapezoidal rule function modified as we've described, Upshot would create the Gantt chart in Figure 12.2.

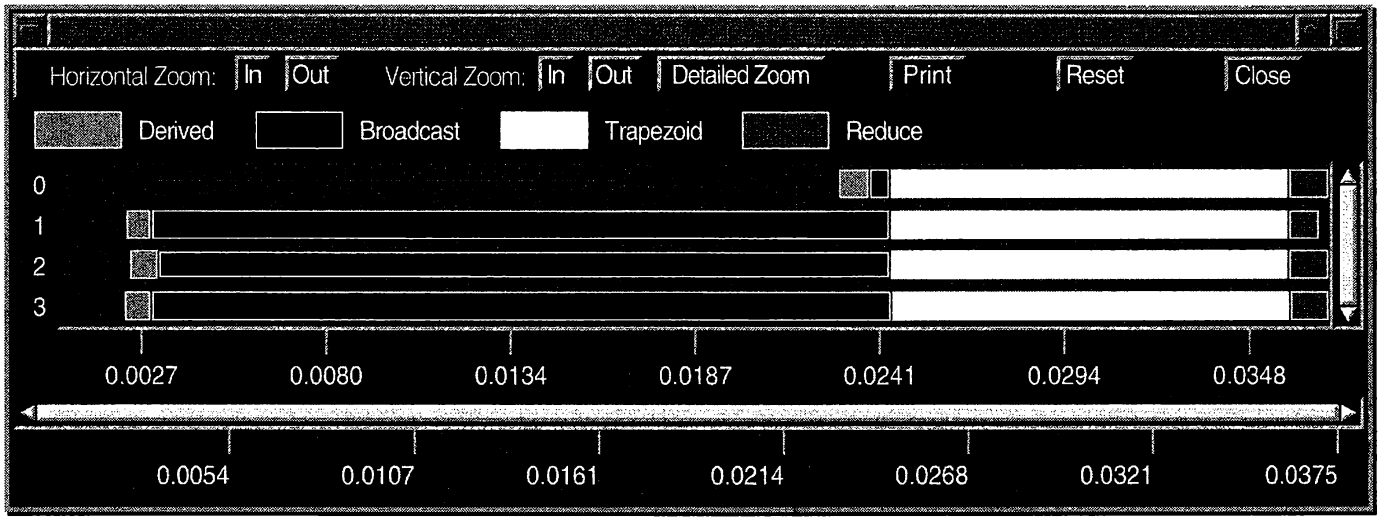

Figure 12.2 `Upshot` output using custom profile generation

12.7 Summary

Amdahl's law asserts that if a fraction s of a program that solves an instance of a problem is inherently serial, then $1/s$ is an upper bound on the speedup that can be achieved by any parallel program—regardless of the number of processes. An inherently serial part of a program is one that cannot be parallelized. For example, if a program needs to read in a value, this will probably be an inherently serial part of the program.

Amdahl's law seems to imply that many problems are not amenable to solution on a parallel system. For example, if 25% of a program is inherently serial, then the maximum possible speedup is $1/(1/4) = 4$. So even if we use 100 processes, we can't do better than a speedup of 4, which would be an efficiency of 0.04.

We saw that Amdahl's law can be unnecessarily pessimistic. Indeed, it is true that, for a fixed problem instance, increasing the number of processes will decrease efficiency. That is, if we fix n (the problem size) and increase p (the number of processes), then it is more than likely that efficiency will diminish. However, if we increase our computational power (p), we will, in all likelihood, also increase the size of the problem we want to solve. As an extreme example, it wouldn't make much sense to use 1000 processes to run the trapezoidal rule with only 500 trapezoids.

This critical observation leads to the idea of *scalability*. A program is *scalable* if we can maintain constant efficiency by increasing the problem size at the same time that we increase the number of processes.

In order to understand scalability, we defined two additional concepts: *work* and *overhead*. The work done by a serial program is just its runtime. The work done by a parallel program is the sum of the work done by each of its processes. Thus, the work done by a parallel program is simply $W_\pi(n,p) = pT_\pi(n,p)$. The overhead incurred by a parallel program is the difference be-

tween the work done by the parallel program and the corresponding serial program:

$$T_O(n,p) = W_\pi(n,p) - W_\sigma(n) = pT_\pi(n,p) - T_\sigma(n).$$

Thus, the efficiency of a parallel program can be written

$$\begin{aligned} E(n,p) &= \frac{T_\sigma(n)}{pT_\pi(n,p)} \\ &= \frac{1}{T_O(n,p)/T_\sigma(n) + 1}. \end{aligned}$$

If, as p is increased, we can increase n so that the fraction T_O/T_σ remains constant, then we can maintain a constant efficiency.

There are three main sources of overhead in parallel programs:

1. interprocess communication
2. process idle time
3. additional computation (computation carried out by the parallel program that is not carried out by the serial program)

There are many situations in which it may be very difficult to make accurate a priori estimates of parallel program performance. Some of the most common problems occur in the following situations:

1. Resource contention. In any parallel system, we can run into problems of resource contention: processes attempting to simultaneously use the same system facility (e.g., a disk). This problem is especially troublesome in networks of workstations, where we almost never have exclusive access to system resources, and, as a consequence, our programs are subject to unpredictable delays.
2. Load imbalance. In some situations, it is possible to predict idle time. For example, if a value is read by, say, process 0 and distributed to the other processes, we can predict that the processes other than 0 will be idle for the duration of the input operation. However, there are many situations where we can't predict idle time with any reliability. For example, if our program distributes a tree by assigning subtrees to different processes, and the tree is unbalanced, the computational load may be unevenly distributed among the processes.
3. Overlapping communication with computation. Our simple formula for communication, $t_s + mt_c$, may not accurately predict the cost of communication if it's possible that communication and computation can be overlapped. This can happen if each node of the parallel system has a separate communication coprocessor, or if we use nonblocking communications.

4. Collective communication. Our system may use complicated, proprietary algorithms for collective communication. An unfortunate consequence of this is that we may be unable to predict the runtime of collective operations.

In order to help users analyze program performance, MPI provides a profiling interface. Essentially, this allows users to create their own profiling library for any MPI functions that they wish to study. The key to this is that any implementation of MPI must allow a program to call each MPI function, `MPI_Xxx`, by both `MPI_Xxx` and `PMPI_Xxx`. Thus, by defining your own version of `MPI_Xxx` that calls `PMPI_Xxx`, you can study such matters as length of time spent in `MPI_Xxx`.

We closed the chapter with a brief discussion of `Upshot`, a program that can be used to study the behavior of parallel programs. Although `Upshot` is not a part of MPI, it is commonly available with MPI implementations. It can be used to generate Gantt charts of parallel programs. It can be used either with or without modification to the source program.

12.8 References

Amdahl's law was originally formulated by Gene Amdahl [2]. The development of programs that achieved efficiencies near 1 on a 1024-processor system was reported in [23]. The resulting reconsideration of Amdahl's law was published in [22].

Our discussion of overhead and scalability is largely based on Kumar et al. [26]. They also provide a very complete discussion of other measures of parallel program performance.

See the MPI Standard [28, 29], Gropp et al. [21], and Snir et al. [34] for discussions of MPI's profiling interface.

The *User's Guide for* `mpich` [20] discusses the features of `Upshot`. `Upshot` can be downloaded from Argonne National Lab at `ftp://info.mcs.anl.gov`. Foster [17] provides an extensive discussion of profiling parallel programs.

12.9 Exercises

1. Determine T_O for the solution to the trapezoidal rule that uses a simple loop of receives on process 0. Compare the scalability of this implementation with the implementation that uses a tree-based reduce.
2. Determine T_O for the basic parallel matrix multiplication algorithm (see section 7.1) and Fox's algorithm (section 7.2). Can you devise simple formulas describing the scalability of these algorithms? If not, how might we go about studying their scalability?

12.10 Programming Assignments

1. Write a program that generates timing statistics on your collective communication functions. Compare your data with predicted runtimes. Do your collective communication functions use the algorithms you thought they used?
2. Use MPI's profiling interface and a profiling tool such as `Upshot` (if you have access to one) in order to study the performance of the basic parallel matrix multiplication algorithm (section 7.1). Does the profile indicate that there is a significant amount of idle time on any process? Do the results of your profiling suggest possible improvements in the algorithm?
3. Repeat assignment 2 for Fox's algorithm (section 7.2).
4. Repeat assignment 2 for Jacobi's method (section 10.3).
5. Repeat assignment 2 for our parallel sorting program (section 10.5).