An Introduction to Performance Debugging for Parallel Computers [*]

William Gropp
Mathematics and Computer Science Division
Argonne National Laboratory

Abstract

Programming parallel computers for performance is a difficult task that requires careful attention to both single-node performance and data exchange between processors. This paper discusses some of the sources of poor performance, ways to identify them in an application, and a few ways to address these issues.

1 Introduction

The basic approach to achieving performance on parallel computers is similar to the methods used with more conventional computers. This paper will emphasize the similarities wherever possible. The first step is to identify that there is in fact a problem. The next few sections describe how to analyze a distributed-memory parallel program for performance. Once a problem is identified, it must be located. Various tools, including the use of visualization tools and computer-aided post processing, are discussed. Some sources of performance problems are peculiar to parallel computing. A sampling of these in the context of message passing is discussed. Finally, some techniques for fixing performance problems are discussed.

The examples in this paper use the Message Passing Interface (MPI) standard [1, 2]. The translation to other message-passing systems should be clear (see also the porting guide in [5]).

[*]This work was supported by the Mathematical, Information, and Compoutational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

2 Identifying Performance Problems

The first step in programming for performance is to know what performance one can expect. This is a particularly crucial step because parallel computers do not fit the model of computational complexity one may expect.

Most numerical programmers are taught to estimate the time that a program will take by counting the number of floating-point operations (sometimes only multiplies and divides). This practice dates from a time when floating-point operations took far longer than any other operation and were thus a good estimate of work for a numerical program. In modern systems, however, a load or store from memory may take several times as long as a floating-point multiply. Even for uniprocessors, floating-point operation counts are no longer of much in in estimating performance. The dominant cost for most computations is now the number and kind of memory reference. On a uniprocessor, the memory references can be divided into three categories: register, cache, and (main) memory. (Some systems may have multiple levels of cache and/or main memory; for simplicity this distinction will be ignored.) Only registers provide memory that is as fast as the CPU; cache memory may require a cycle (instruction) or two, and main memory may require tens of cycles to provide data for an operation.

For parallel computers, the situation is much worse because there is an additional memory category: nonlocal memory. Accessing this memory can take hundreds to hundreds of thousands of cycles. Moreover, accessing this data usually requires the effort several processors, thereby taking cycles away from the computation.

Thus, before beginning to tune a program for performance on a parallel (or even sequential) system, it is important to have at least a simple model of the performance that is expected. This model will help identify the two major types of problem: (a) predicted performance is too low and (b) observed performance is lower than predicted performance. In case (a), one must re-think the algorithm and problem. In case (b), one must examine the implementation.

Fortunately, the same techniques may be used to identify both problems. By performing a simple scalability anaysis, one can estimate the performance of a parallel code. Then, by using a combination of tools to observe the per-node and parallel performance, one can identify the parts of the code that are under-performing.

2.1 Scalability Analysis

Scalability analysis is an analytic estimate of expected performance as a function of the number of processes. A simple scalability model may be used to estimate the performance of distributed-memory parallel computers. In this model, we ignore the memory cost for all but nonlocal operations, and model nonlocal operations by

$$s + rn,$$

where s is the *latency* or startup time, r is the time to transfer a single byte, and n is the number of bytes being transferred. In addition, we use f to indicate the time to perform a floating-point operation (inverse flops).

Below are the approximate values for an (thin-node) SP2:

$$\begin{aligned} s &= 50\ \mu\text{sec} \\ r &= 1/8\ \text{MB/sec} = 1.25 \times 10^{-7}\text{sec/B} \\ f &= 1/125\ \text{Mflops} = 8 \times 10^{-9}\text{sec/flop} \end{aligned}$$

Better scalability models may include the effects of loads and stores, communication contention, and other factors. Some examples of scalability analysis for both algorithms and applications may be found in [4, 6, 3].

Using a scalablility model, one can estimate the amount of time a computation will take. For improving performance, however, some metrics can be more informative than just the time:

Speedup T_p/T_1, problem size fixed.

Scaled Speedup T_p/T_1, problem size scaled with p

Efficiency $Speedup/p$

All of these suggest how efficiently the resources of a parallel computer are being utilized by a computation. Speedup (or Scaled Speedup) of p (for p processors) represents perfect utilization. It is important to remember that, although the speedup of any computation can be improved by having each processor do additional work that does not depend on an interaction with other processors, doing so also increases the amount of time that the computation takes. That this is bad may seem obvious, but many algorithms have been proposed for parallel computing that do essentially this (e.g., point-Jacobi relaxation for solving certain sparse linear systems). One should be wary of pursuing perfect speedup at the cost of overall efficiency.

2.2 Shared Memory

Computations on shared-memory parallel computers may be modeled in much the same way as for distributed memory, though with a different expression for the time it takes to access remote data. For example, on a shared-memory multiprocessor where all of the memories share a single bus, the time to access data might be

$$\frac{rn}{\max(k,p)}.$$

Here, r/k represents the time that it takes a single processor to access remote memory; k is the number of processors that can simultaneously access memory before some processor must wait.

2.3 Per-node Performance

The first step to take is to tune a code for per-node performance. That is, each individual process must be made to run as fast as possible. There are two reasons for doing this first. First, as discussed above, any estimate or modeling of parallel scalability can be misleading if each individual process is doing "extra" work. An untuned code is essentially doing such extra work. Second, as will be discussed in the sections on message-passing performance, the performance of the message-passing (or other parallel communication) part of the code depends critically on the exact timing of operations on the local and remote (partner/destination/source) processor. Thus, if the message-passing part is tuned first, the tuning of the code itself may "de-tune" the message-passing components.

There are numerous sources of information on tuning; many vendors offer tuning guides for their systems, and these should be consulted. One question that most tuning guides do not answer, however, is how to determine whether tuning will help. For many numerical codes, an indicator of the need for tuning is the ratio of the Flops rate (floating-point operations per second) compared with the values published for the LINPACK benchmarks. If the ratio is less than 0.75, the code should be examined. The LINPACK benchmarks provide a reasonable value for the achievable peak performance for many systems.

A slightly better indicator uses the number of memory references and compares the observed time to the time that the memory bandwidth of the processor/memory configuration would predict. If the ratio is too low, then the application may be making ineffective use of the memory heirarchies (for example, memory cache).

In any case, just as for speedup, one must not lose sight of the goal: a faster code. For example, replacing a sparse matrix algorithm with a dense matrix algorithm may improve Flops and memory usage, but at the cost of a code that, overall, is slower.

2.4 Comparision with Others

When evaluating the performance (both sequential and parallel) of a code, it is vital that, whenever possible, the performance be compared with the performance of similar codes developed by others. Particularly important is to compare the performance of the *best* algorithm for sequential machines. A literature search may also reveal useful information about the performance of similar applications (it is very much to be hoped that as computer science matures as a discipline, the value in reporting on the performance of carefully crafted implementations will be better recognized and served than it is today).

Another approach that can be useful, particularly when trying to understand the performance of a large and complex (hard to model) parallel code when using large data sets (too large for a single processor), is to run the code with all of the message-passing (or shared-memory) operations disabled. The difference in time between running with and without the message-passing is roughly the amount of time spent coordinating and sharing data rather than computing, and is a rough measure of overhead.

3 Locating Problems

Once a code has been determined to need performance tuning, the problem becomes one of finding the performance problem. For locating problems in the the per-node performance, all of the conventional techniques (e.g., `gprof`, `prof`) may be used (as long as the vendor supports such familar tools). For locating problems in the parallel parts of the code (i.e., message-passing or shared-memory operations), no standard tools exist. Predictive techniques (such as warnings from vectorizing compilers about nonvector code) are usually unavailable for parallel programs. Further, because operations between processors may take far longer than operations within a processor, just because an operation (such as sending data from one processor to another) takes a long time does not mean that there is a way to make it take *less* time. To help identify those parts that take too much time, we introduce the concept of *deficiency analysis* (see Section 4).

Poor parallel performance has five possible causes: (1) the numerical algorithm, (2) the implementation, (3) a mismatch between user model and reality, (4) the communication algorithm, and (5) small variations in timing use of the individual processors.

The rest of this paper will concentrate on cases (4) and (5), but one should remember that the first three sources of poor performance must be tackled first. In particular, the choice of numerical algorithm is the first aspect of a program to consider. The parallel numerical algorithm may perform too much communication for the size of system. An example is a numerical algorithm that relies on fast broadcasts of data from one processor to all others; this communication strategy can be implemented efficiently on some parallel machiness but not others. Other problems (trading scalability for overall performance) have already been mentioned.

In looking at the per-node performance, in addition to considering the speed of floating-point operations and memory references, it is important to consider the cost of subroutine call overhead, various system services (e.g., malloc, which often is both expensive and performs poorly as more elements are allocated [9]), and inappropriate optimization (e.g., long vectors on RISC processors.)

3.1 Communication Performance

Once the single-node performance is tuned, it is time to look at the performance of the comunication algorithm. Poor performance has three major causes. The first is idle time: the time that a processor spends waiting on the arrival of a message. For example, in the following program fragment, processor one spends time waiting for processor zero (assume that $k < 1000$:

```
call MPI_Comm_rank( MPI_COMM_WORLD, myrank, ierr )
if (myrank .eq. 1) then
     do 10 i=1, k
         ... do some computation ...
10   continue
     call MPI_Recv( ... buf ... )
else
     do 20 i=1,1000
        ... do some computation ...
20   continue
     call MPI_Send( ... )
```

In this example, the idle time incurred by one processor is caused by imperfect load balancing.

In many parallel systems, it is possible to overlap computation with communication. Because the speed at which data is sent between processors is usually much less than the speed at which data moves locally, overlapping communication with computation can be important. In the example above, the code can be reordered as follows:

```
call MPI_Comm_rank( MPI_COMM_WORLD, myrank, ierr )
if (myrank .eq. 1) then
     call MPI_Irecv( ... buf ..., request, ierr )
     do 10 i=1, k
         ... do some computation ...
10   continue
     call MPI_Wait( request, ierr )
else
     call MPI_Isend( ... request, ierr )
     do 20 i=1,1000
        ... do some computation ...
20   continue
     call MPI_Wait( request, ierr )
```

In this example, the code initiates a receive (`MPI_Irecv`) or a send (`MPI_Isend`), does some computation, and then waits for the operations to complete (`MPI_Wait`). Using this forumlation requires that the data be available when the send is initiated (`MPI_Isend`) and that it not be modified before the send is completed (`MPI_Wait`). These operations are called *nonblocking*. This strategy allows the data to be transferred while the computation is taking place. Note, however, that not all environments support an efficient implementation of nonblocking

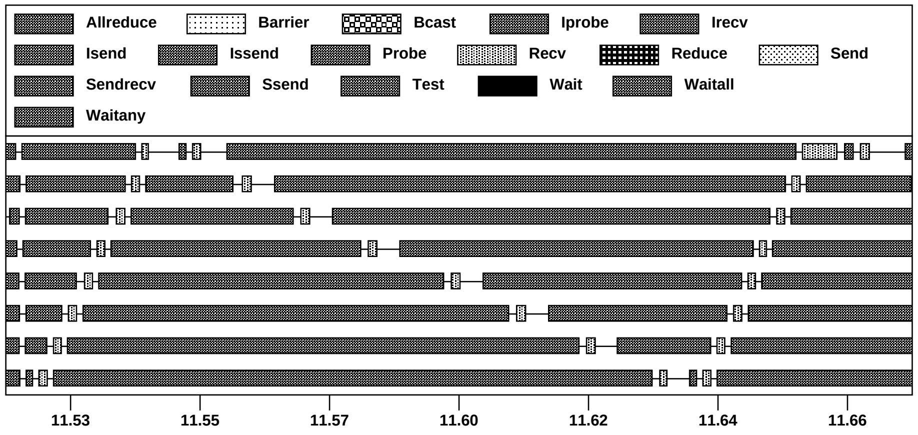


Figure 1: Time lines for a program with sends waiting for matching receives for messages too large to be sent without matching receives

operations. Hence, it may be necessary to look closely at whether the use of nonblocking operations provides improved performance.

Another source of poor performance is caused by reaching past the limits of the underlying system. Consider the following program:

```
if (myrank .eq. 1) then
    call MPI_Send( buffer, size, ... )
    ... do work ...
else
    ... do work ...
    call MPI_Recv( buffer, .... )
endif
```

At first glance, this code appears fine. And the code is correct. It even seems to provide for the overlapping of communication with computation (the `MPI_Send` starts the send and then overlaps with the computation). But there is a subtle performance problem. If `size` is large enough, there will not be enough space on the destination processor to hold the message, thereby forcing the sending processor to wait until the matching receive is issued. Thus, when the program is tested with small data sets, it will work as expected; with larger data sets, however, the performace will suddenly decrease. An example of this effect is shown in Figure 1. (There is a correctness problem here as well; if two processors send at each other and then receive, the program may work until the messages become too large, at which point it will hang.)

3.2 Timing Variations

When examining timings for performance problems, it is immediately obvious that the timings are not precisely reproducible. There are many sources of these variations, including other users on the system, other system activity, network activity, paging and I/O systems, nondeterminism in user program, and tradeoffs between detail and accuracy. Most of these are fairly obvious and affect sequential programs. However, nondeterminism in user programs is (nearly) unique to parallel computing. Consider the following program fragment:

```
if (rank == 0) {
    MPI_Send( rank=1 ... ); MPI_Recv(...);
    }
else {
    MPI_Recv(...); MPI_Send( rank = 0 ... );
    }
```

Does data from the send in process zero arrive before or after the receive is issued by process one?

The answer is that either can happen, and which happens first can significantly change the amount of time that this code takes to execute. Consider the first case: the data from the send arrives before the receive is issued. A possible sequence is

1. send data arrives, causing user process to be interrupted;
2. no destination buffer is available, so space is allocated and data is copied;
3. later, the receive is issued, data is copied from the buffer, and the buffer is deallocated.

Now consider the other case: the receive is issued before the send arrives. Then the sequence of operations may be

1. receive is posted and the application waits for data to arrive;
2. data arrives and is copied directly into buffer designated by the receive.

This second sequence of operations may take much less time the the first sequence, particularly if the receive does not wait long (a similar sequence applies if nonblocking operations are used).

One of the major reasons that the early arrival of data can cause a significant increase in the time taken is the interrupt of the user process for the system to service the incoming message. One could decrease this cost (in some cases) by not taking the interrupt. Instead, the system could wait until the receive is issued, or occasionally poll the interconnection network. Both the IBM MPL and the TMC CMMD provide options to select between these approaches. Note that if polling is used, instead of time being lost to interrupts, it may be lost while a send on one processor stalls waiting for a receive to occur on another processor.

4 Deficiency Analysis

Most performance tuning tools identify those components of the code that take the most time. What they do not identify is whether those components can be made to run faster. For example, a profiling tool might indicate that routine `a` takes 230 seconds and `b` takes 180 seconds. But if `a` is achieving 140 MFlops out of 150 and `b` is achieving 15 MFlops, then clearly any tuning should be carried out in routine `b`, not routine `a` (in this case, tuning may not be worthwhile because less than a factor of two improvement could be realized). The approach of identifying parts of the code that both take significant time and are underperforming is called *deficiency analysis*.

No tools currently analyze both the time and the performance of parallel computers. However, it is possible to instrument critical parts of a program to help identify underperforming sections of code. For example, to look for underperforming message-passing code, one can combine the complexity model in Section 2.1 with runtime calls to measure the elapsed time used by the calls. The following code generates a message whenever a message takes more than `THRESHOLD` times as long as predicted:

```
int MPI_Send( ... )
{
t1 = MPI_Wtime();
err= PMPI_Send( ..., count, datatype, ... );
t2 = MPI_Wtime() - t1;
MPI_Type_size( datatype, &size );
if (t2 > THRESHOLD*(s + r * size * count)) {
    ... log problem at __FILE__, __LINE__
    }
return err;
}
```

(This example uses the MPI profiling interface described below.)

5 Event Log Tools

Many tools help visualize the behavior of a parallel program. These include portable tools created by research groups such as ParaGraph [7], Pablo [10], Upshot [8], and AIMS [11], vendor-provided tools, such as the IBM vt. All these tools create and display logs of events that are generated by instrumented versions of the user's application; each emphasizes a different view of the data. These can be useful in identifying unexpected patterns of behavior that are not apparent when looking only at a single processor. An example of Upshot output is shown in Figure 2.

5.1 MPI's Profiling Interface

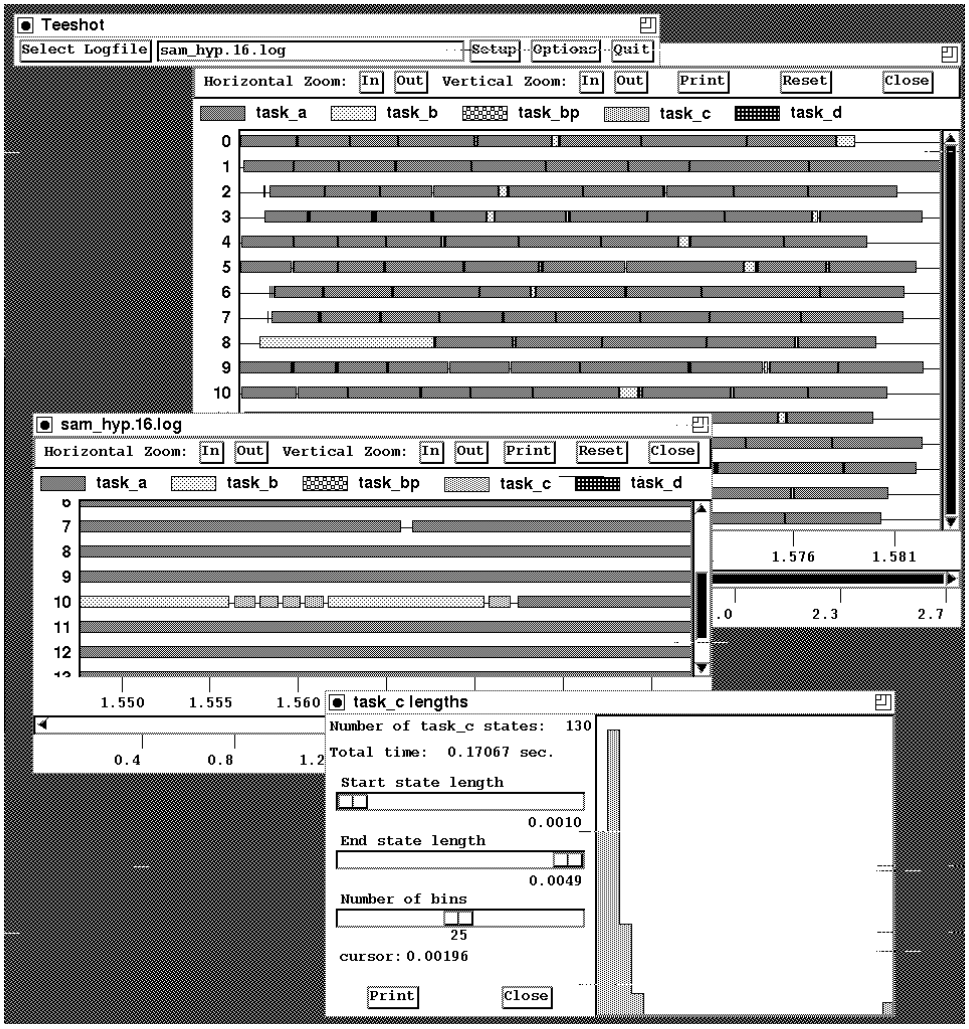
All MPI routines also available as `PMPI_xxx`. This interface allows each (or only some) MPI routine to be replaced with a routine that performs some additional action (such as logging the use of the MPI routine or measuring the time it takes) and then performs the MPI operation. By ensuring that the replacement routine occurs first in the linker list, the user's application gets the replacement routine without needing to be recompiled. Figure 3 shows how link-time replacement of routines happens in MPI. An example using the MPI profiling interface was shown in Section 4.

6 Conclusion

The following steps have been described for performance debugging for parallel computers:

1. Develop an analytical time-complexity model. Measure the achieved performance (perhaps on smaller problem).
2. Determine whether tuning is needed.
3. Tune for single-node performance.
4. Tune for message-passing or remote memory performance.

Techniques for identifying and locating problems include using deficiency analysis to identify deviations between model and actual performance and using profiling and program visualization tools. In the end, however, tuning a code requires understanding the memory reference model (for local and shared-memory operations) and the operations performed by message-passing routines.



Number of task_c states: 130
Total time: 0.17067 sec.
Start state length 0.0010
End state length 0.0049
Number of bins 25
cursor: 0.00196

Figure 2: Sample Upshot output

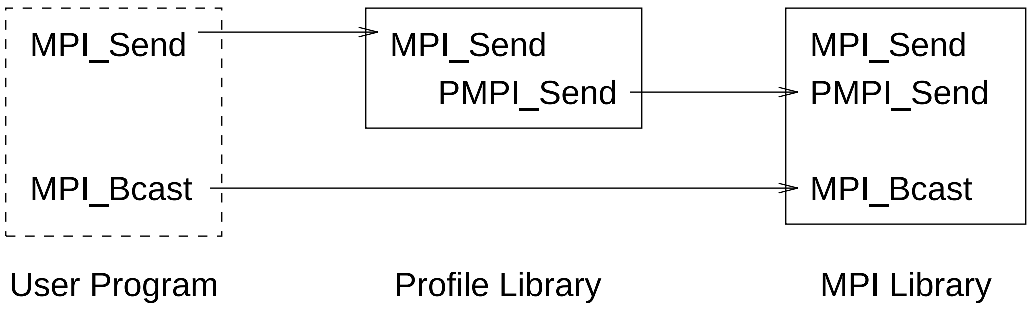


Figure 3: Diagram showing how MPI profiling routines are selected at link time

References

[1] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[2] Message Passing Interface Forum. MPI: A message-passing interface standard. http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html, May 1994.

[3] Ian Foster, William Gropp, and Rick Stevens. The parallel scalability of the spectral transform method. *Monthly Weather Review*, 120:835–850, 1992.

[4] W. Gropp and E. Smith. Computational fluid dynamics on parallel processors. *Computers and Fluids*, 18:289–304, 1990.

[5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* MIT Press, 1994.

[6] William D. Gropp and David E. Keyes. Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations. *SIAM J. Sci. Statist. Comput.*, 9(2):312–326, 1988.

[7] Michael T. Heath and Jennifer Etheridge Finger. Visualizing performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[8] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, Mathematics and Computer Science Division, August 1991.

[9] David M. Nichol. Inflated speedups in parallel simulations via malloc(). Technical Report ICASE-90-63, ICASE, September 1990.

[10] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.

[11] Jerry Yan, Philip Hontalas, and Sherry Listgarten. *The Automated Instrumentation and Monitoring System (AIMS) Reference Manual*, November 1993. NASA-TM-108795.