Reproducible Measurements of MPI Performance Characteristics*

William Gropp and Ewing Lusk

Argonne National Laboratory, Argonne, IL, USA

Abstract. In this paper we describe the difficulties inherent in making accurate, reproducible measurements of message-passing performance. We describe some of the mistakes often made in attempting such measurements and the consequences of such mistakes. We describe `mpptest`, a suite of performance measurement programs developed at Argonne National Laboratory, that attempts to avoid such mistakes and obtain reproducible measures of MPI performance that can be useful to both MPI implementors and MPI application writers. We include a number of illustrative examples of its use.

1 Introduction

Everyone wants to measure the performance of their systems, but different groups have different reasons for doing so:

- Application writers need understanding of the performance profiles of MPI implementations in order to choose effective algorithms for target computing environments.
- Evaluators find performance information critical when deciding which machine to acquire for use by their applications.
- Implementors need to be able to understand the behavior of their own MPI implementations in order to plan improvements and measure the effects of improvements made.

All of these communities share a common requirement of their tests: that they be *reproducible*. As obvious as this requirement is, it is difficult to satisfy in practice. Parallelism introduces an element of nondeterminism that must be tightly controlled. The Unix operating system, together with network hardware and software, also introduces sporadic intrusions into the test environment that must be taken into account. The very portability of MPI suggests that the performance of the same operations (MPI function calls) can be meaningfully compared among various parallel machines, even when the calls are implemented in quite different ways. In this paper we review the perils of shortcuts frequently taken in

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

attempting to acquire reproducible results, and the approach we take to avoid such perils.

Over the years, the MPICH group has developed a suite of programs that characterize the performance of a message-passing environment. An example is the program `mpptest` that can be used to quickly characterize the performance of an MPI implementation in a variety of ways. For example, it was used to quickly measure the performance of a variety of pre-MPI message-passing implementations and to identify message sizes where sharp performance transitions occurred; see [4], Figures 1 and 3. An example of the use of `mpptest` in tuning an MPI implementation is shown in [3]. These programs are portable to any MPI implementation.

A number of parallel performance tests already exist. Many of these strive to be benchmarks that produce a "figure of merit" for the system. Our interest is in the details of the behavior, rather than a few numbers representing the system. Perhaps the closest project to our work is SKaMPI [7]. The SKaMPI system provides graphical output detailing the behavior of a wide variety of MPI functions and includes an adaptive message-length refinement algorithm similar to that in `mpptest`. The testing method in SkaMPI is somewhat different from ours and uses different rules when accepting experiments.

A number of well-known benchmarks are accessible through BenchWeb [1]. The ParkBench Organization provides a variety of codes, including "compact application benchmarks". A review of some of the issues in developing a benchmark may be found in [2]. Previous PVMMPI meetings have included papers on performance measurement; see, for example, [5, 6].

We hope that the unusual approach taken in `mpptest` makes it a useful addition to the collection of performance measurement procedures for MPI programs. The paper first reviews (Section 2) some of the difficulties of performance performance measurements and characterizations. Section 3 briefly describes the testing methods and variations, relating the choices to the issues raised in Section 2. Section 4 presents a few examples that illustrate the capabilities of our performance characterization tests.

2 Perils of Performance Measurement

Simple tests can be misleading. As communication software becomes more sophisticated (for example, dynamically allocating resources to busy communication channels), simple tests become less indicative of the performance of a full application. The following list describes some of the pitfalls in measuring communication performance, in the form of "mistakes often made."

1. **Forget to establish initial communication link.** Some systems dynamically create connections. The first communication between two processes can take far longer than subsequent communications.
2. **Ignore contention with unrelated applications or jobs.** A background file system backup may consume much of the available communication bandwidth.

3. **Ignore nonblocking calls.** High-performance kernels often involve nonblocking operations both for the possibility of communication overlap but, more important, for the advantage in allowing the system to schedule communications when many processes are communicating. Nonblocking calls are also important for correct operation of many applications.
4. **Ignore overlap of computation and communication.** High-performance kernels often strive to do this for the advantages both in data transfer and in latency hiding.
5. **Make an apples-to-oranges comparison.** Message-passing accomplishes *two* effects: the transfer of data and a handshake (synchronization) to indicate that the data are available. Some comparisons of message passing with remote-memory or shared-memory operations ignore the synchronization step.
6. **Confuse total bandwidth with point-to-point bandwidth.** Dedicated, switched networks have very different performance than shared network fabrics.
7. **Compare CPU time to elapsed time.** CPU time may not include any time that was spent waiting for data to arrive. Knowing the CPU load caused by a message-passing system is useful information, but only the elapsed time may be used to measure the time it takes to deliver a message.
8. **Ignore correctness.** Systems that fail for long messages may have an unfair advantage for short messages.
9. **Time events that are small relative to the resolution of the clock.** Many timers are not cycle counters; timing a single event may lead to wildly inaccurate times if the resolution of the clock is close to the time the operation takes. A related error is to try to correct the clock overhead by subtracting an estimate of the time to call the clock that is computed by taking the average of the time it takes to call the clock; this will reduce the apparent time and artificially inflate performance.
10. **Ignore cache effects.** Does the data end up in the cache of the receiver? What if data doesn't start in the cache of the sender? Does the transfer of data perturb (e.g., invalidate) the cache?
11. **Use a communication pattern different from the application.** Ensuring that a receive is issued before the matching send can make a significant difference in the performance. Multiple messages between different processes can also affect performance. Measuring ping-pong messages when the application sends head-to-head (as many scientific applications do) can also be misleading.
12. **Measure with just two processors.** Some systems may poll on the number of possible sources of messages; this can lead to a significant degradation in performance for real configurations.
13. **Measure with a single communication pattern.** No system with a large number of processors provides a perfect interconnect. The pattern you want may incur contention. One major system suffers slowdowns when simple butterfly patterns are used.

The programs described in this paper attempt to avoid these problems; for each case, we indicate below how we avoid the related problem.

3 Test Methodology

In this section we discuss some of the details of the testing. These are related to the issues in measuring performance described in Section 2. Our basic assumption is that in any short measurement, the observed time will be perturbed by some positive time Δt and that the distribution of these perturbations is random with an unknown distribution. (There is one possible negative perturbation caused by the finite resolution of the clock; this is addressed by taking times much longer than the clock resolution.)

3.1 Measuring Time: Minimum of Averages

The fundamental rule of testing is that the test should be repeatable; that is, running the test several times should give, within experimental error, the same result. It is well known that running the same program can produce very different results each time it is run.

The only time that is reproducible is the minimum time of a number of tests. This is the choice that we make in our testing. By making a number of tests, we eliminate any misleading results do to initialization of links (issue 1).

Using the minimum is not a perfect choice; in particular, users will see some sort of average time rather than a minimum time. For this reason, we provide an option to record the maximum time observed, as well as the average of the observations. While these values are not reproducible, they are valuable indicators of the variation in the measurements.

The next question concerns what the minimums should be taken over. We use a loop with enough iterations to make the cost of any timer calls inconsequential by comparison. This eliminates errors related to the clock resolution (issue 9). In addition, the length of time that the loop runs can be set; this allows the user to determine the tradeoff between the runtime (cost) of the test and its accuracy.

3.2 Message Lengths

The performance of data movement, whether for message-passing or simple memory copies, is not a simple function of length. Instead, the performance is likely to consist of a number of sudden steps as various thresholds are crossed. Sampling at regular or prespecified data lengths can give misleading results. The `mpptest` program can automatically choose message lengths. The rule that `mpptest` uses is to attempt to eliminate artifacts in a graph of the output. It does this by computing three times: $f(n_0)$, $f(n_1)$, and $f((n_0+n_1)/2)$, where $f(n)$ is the time to send n bytes. Then `mpptest` estimates the error in interpolating between n_0 and n_1 with a straight line by computing the difference between $(f(n_0)+f(n_1))/2$ and $f((n_0+n_1)/2)$. If this value is larger than a specified threshold (in relative terms), the interval $[n_0, n_1]$ is subdivided into two intervals, and the step is repeated. This can continue until a minimum separation between message lengths is reached.

3.3 Scheduling of Tests

The events that cause perturbations in the timing of a program can last many milliseconds or more. Thus, a simple approach that looks for the minimum of averages within a short span of time can be confused by a single, long-running event. As a result, it is important to spread the tests for each message length over the full time of the characterization run. A sketch of the appropriate measurement loop is shown below:

```
for (number of repetitions) {
    for (message lengths) {
        Measure time for this length
        if this is the fastest time yet, accept it
    }
}
```

Note that this approach has the drawback that it doesn't produce a steady stream of results; only at the very end of the test are final results available. However, it comes much closer to what careful researchers already do—run the test several times and take the best results. This helps address contention with other applications or jobs (issue 2), though does not solve this problem.

Tests with anomalously high times, relative to surrounding tests, are automatically rerun to determine if those times reflect a property of the communication system or are the result of a momentary load on the system. This also aids in producing reproducible results.

Note also that it is important to run a number of cycles of this loop before refining the message intervals. Otherwise, noise in the measurements can generate unneeded refinements.

3.4 Test Operations

Rather than test only a single operation, such as the usual "ping pong" or round-trip pattern, our tests provide for a wide variety of tests, selected at run time through command line arguments. The following list summarizes the available options and relates them to the issues in Section 2.

Number of processors. Any number of processors can be used. By default, only two will communicate (this tests for scalability in the message-passing implementation itself; see issue 12). With the `-bisect` option, half of the processors send to the other half, allowing measurement of the bisection bandwidth of the system (issue 6).

Cache effects. The communication tests may be run using successive bytes in a large buffer. By selecting the buffer size to be larger than the cache size, all communication takes place in memory that is not in cache (issue 10).

Communication Patterns. A variety of communication patterns can be specified for the bisection case, addressing issue 13. In addition, both "ping pong" and head-to-head communication is available when testing two communicating processes. More are needed, in particular to make it easier to simulate an application's communication pattern (issue 11).

Correctness. Correctness is tested by a separate program, `stress`. This program sends a variety of bit patterns in messages of various sizes, and checks each bit in the receive message. Running this test, along with the performance characterization tests for large message sizes, addresses issue 8.

Communication and computation overlap. A simple test using a fixed message length and a variable amount of computation provides a simple measurement of communication/computation overlap, addressing issue 4.

Nonblocking Communication. Nonblocking communication is important; in many applications, using nonblocking communication routines is the easiest way to ensure correctness by avoiding problems related to finite buffering. The performance of nonblocking routines can be different from the that of the blocking routines. Our tests include versions for nonblocking routines (issue 3).

4 Examples

This section shows the capabilities of the `mpptest` characterization program in the context of specific examples.

Discontinuous Behavior The need for adaptive message length choice can be seen in Figure 1(a). This illustrates why the simple latency and bandwidth model is inappropriate as a measure of performance of a system.

We see the stair-steps illustrating message packet sizes (128 bytes). We also see the characteristic change in protocol for longer messages. Here it looks like the protocol changes at 1024 bytes, and that it is too late. The implementation is not making an optimal decision for the message length at which to switch methods; slightly better performance could be achieved in the range of 768 to 1024 bytes by using the same method used for longer messages.

Figures 1(c) shows the behavior for nonblocking sends instead of blocking sends. Note the small but measurable overhead compared with Figure 1(a).

Cache Performance Effects Performance tests often use the same, relatively small, data arrays as the source and destination of messages. In many applications, data is moved from main memory, not from cache. Figure 1(b) shows an example where the performance with data in cache is better than when the data is not in cache, both in absolute terms and in the trend (lower slope for in-cache data).

Variation in Performance In an application, the minimum times for an operation may not be as important as the average or maximum times. Figure 1(d) shows

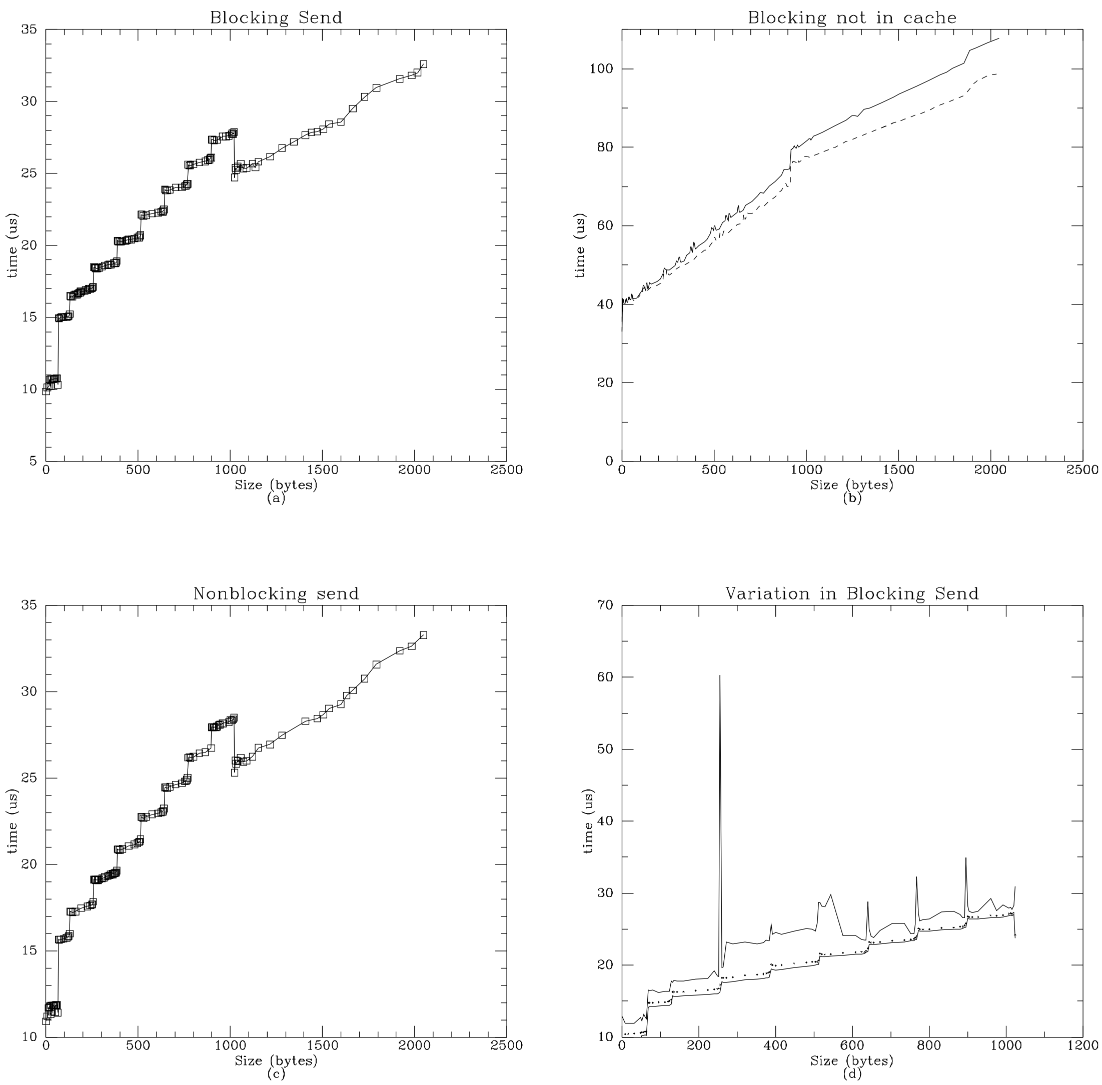

Fig. 1. Example results generated by `mpptest` on several platforms. Graph (a) shows an example of discontinuous performance identified by `mpptest`. Note the sharp drop in latency for zero bytes, the significant steps up to 1024 bytes, and the lower slope beyond 1024 bytes. Graph (b) shows an example of the change in performance between data in cache (dashed line) and not in cache (solid line). Graph (c) shows the extra cost of using nonblocking sends. Graph (d) gives an example of the variation in performance even on a system communicating with shared-memory hardware (see text for details).

the variations in times on a lightly-loaded shared-memory system. This graph is very interesting in that the minimum (bottom line) and average times (dots) are close together in most places, but the maximum observed time (top line) can be quite large. The peaks seem to line up with the transitions; however, since there are more measurements near the transitions, the correlation may be an accident. Further testing, particularly with the evenly spaced message sizes, would be required to determine if those peaks occurred only at the transitions.

5 Conclusion

We have illustrated the difficulty in characterizing performance and have discussed how the MPICH performance characterization programs can be used to discover properties of the parallel environment. The software is freely available from `http://www.mcs.anl.gov/mpi/mpich/perftest` or `ftp://ftp.mcs.anl.gov/pub/mpi/misc/perftest.tar.gz`.

References

1. Benchweb. World Wide Web. http://www.netlib.org/benchweb/.
2. Parkbench Committee. Public international benchmarks for parallel computers. *Scientific Programming*, 3(2):101–146, 1994. Report 1.
3. W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.
4. W. D. Gropp and E. Lusk. Experiences with the IBM SP1. *IBM Systems Journal*, 34(2):249–262, 1995.
5. J. Piernas, A. Flores, and J. M. García. Analyzing the performance of MPI in a cluster of workstations based on fast Ethernet. In Marian Bubak, Jack Dongarra, and Jerzy Waśniewski, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 17–24. Springer, 1997. 4th European PVM/MPI Users' Group Meeting.
6. Michael Resch, Holger Berger, and Thomas Boenisch. A comparision of MPI performance on different MPPs. In Marian Bubak, Jack Dongarra, and Jerzy Waśniewski, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 25–32. Springer, 1997. 4th European PVM/MPI Users' Group Meeting.
7. R. Reussner, P. Sanders, L. Prechelt, and M Müller. SKaMPI: A detailed, accurate MPI benchmark. In Vassuk Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, 1998. 5th European PVM/MPI Users' Group Meeting.