Communication Patterns

Rolf Riesen

Sandia National Laboratories*
P.O. Box 5800
Albuquerque, NM 87185-1110
rolf@cs.sandia.gov

Abstract

Parallel applications have message-passing patterns that are important to understand. Network topology, routing decisions, and connection and buffer management need to match the communication patterns of an application for it to run efficiently and scale well. These patterns are not easily discerned from the source code of an application, and even when the data is available it is not easy to categorize it appropriately such that meaningful knowledge emerges.

We describe a novel system to gather the information we need to discover an application's communication pattern. We create five categories that help us analyze that data and explain how information from each category can be useful in the design of networking hardware and software. We use the NAS parallel benchmarks as examples on how to apply our techniques.

1 Introduction

A message-passing application uses point-to-point and collective operations to communicate among the nodes it is running on. Which nodes send messages to what other nodes, how often, and containing how much data depends on the algorithm implemented by the application, and sometimes on the input and setup of the application. Knowing these communication patterns is important for application developers and designers of networks and communication software stacks. It is also important for purchasers of parallel machines, so they can make decisions about which topology and network technology will best fit their applications.

In this paper we describe a novel method to gather the kind of information we are interested in. With data gathered from the NAS parallel benchmarks [1] we show how to present that data to make understanding easier, and provide examples of hardware and software design areas where that information can be useful.

The tool we use to collect the data is an early prototype of a network simulator. The application executes directly, the code is not simulated, but events sent between the application and the network simulator let the simulator observe and record all message traffic.

We use the NAS parallel benchmarks to demonstrate the technique and present a sample set of measurements. There are eight benchmarks that make up the NAS parallel benchmarks suite: BT, CG, EP, FT, IS, LU, MG, and SP. Processing power and memory sizes of modern supercomputers have been increasing. The work a benchmark does on each node had to increase for it to remain a true test of parallel computer performance. Therefore, the benchmarks come in classes. Among the parallel versions, class A has the smallest requirements, while class B is a little bit more demanding, and class C requires significant amounts of memory, processing power, and communication capabilities to complete within a reasonable time. There are a few other classes, such as W for work stations, but for this paper we will limit ourselves to classes A, B, and C.

Due to the particular algorithms used, most of the NAS parallel benchmarks require a square or power of two number of nodes to run on. For this paper we ran experiments on 4, 16, and 64 nodes. While these are not large sizes given today's state-of-the-art clusters and parallel machines, it is enough to illustrate the kinds of measurements we are interested in.

Presenting graphs of all measurements for all classes and sizes would fill tens of pages. Therefore, we will select some of the more interesting ones and show them to illustrate our measurements. The complete set is available in [5].

This paper makes two main contributions: it describes a novel method for gathering the necessary data from a running application and provides a method to present and evaluate that data. We use the NAS parallel benchmarks as well

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

1-4244-0054-6/06/$20.00

known parallel applications to demonstrate our methods. Our simulator approach is interesting because instrumentation of the applications is trivial, the simulator does not change the perceived running time of the application, and in the future our tool will be able to gather additional data to what we present in this paper. Examples include intrusion-free MPI traces and the ability to change the performance characteristics of the network.

We also list why the particular measurements we have taken are of interest to hardware and system software designers. While no machine will be built to run the NAS parallel benchmarks efficiently, those benchmarks do represent a certain class of applications, and the method used to gather the results for this paper, can also be used to gather similar data from more important applications.

In the next section we describe the experimental setup that we used to obtain our measurements. We present the data in Section 3 in an easy to understand and compact format. We also provide information why having that information is important. In the related work section (Section 4), we look at earlier methods for collecting data and explain how our approach differs. We close the paper in Section 5 with a summary and some work we are planning to do in the future.

2 Experimental setup

We obtained the measurements presented in Section 3 using an early prototype of a supercomputer simulator. The only thing it simulates so far is a very simplistic network model similar to the formula $L+s/B$, where L is the latency, B is the bandwidth, and s is the size of the message. The parameters L and B were chosen to mimic the zero-length latency and asymptotic bandwidth of our test system.

In addition to calculating a delay based on message length, the simulator also keeps track of the source and destination of each message, whether it was sent point to point or as part of a collective operation, and how many bytes were sent. When the application ends, the simulator prints all that information to the screen.

Figure 1 shows how the network simulator is organized. The application is directly executed. That means it runs as it would without the simulator present and uses the standard network and MPI protocol stack to exchange data. The simulator runs on one additional node. That is, if the application requires 16 nodes, we need 17 nodes to run the experiment.

Using the MPI profiling interface, the application sends an event to the simulator node for each MPI send, telling it about the length, tag, and destination of that send. The event also contains the current virtual time t_x (see Figure 2 and [3]). In each blocking receive or wait function, the application waits for the MPI message and an event from the

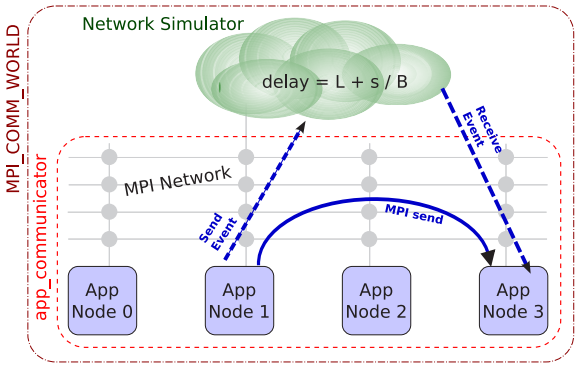

Figure 1. Data and event traffic

simulator. The dark arrow in Figure 1 from node 1 to node 3 shows an MPI message traveling through the standard MPI stack and network. In addition, the sender (node 1) sends an event to the network simulator (dashed arrow). When node 3 receives the MPI message, it will not proceed until it has also received an event from the network simulator (second dashed arrow). The pseudocode in Figure 2 illustrates the send of an MPI message, and Figure 3 shows the reception of an MPI message using `MPI_Recv()`. Similar stubs exist for the other blocking MPI receive and wait functions.

```
int  MPI_Send(...)      {
     t_x = get_vtime();
     PMPI_Send(....);              // Send the data
     send_event (t_x, ...);        // Send event to sim
}
```

Figure 2. Stub code example for `MPI_Send()`

```
int  MPI_Recv(...)      {
     t_1 = get_vtime();
     PMPI_Recv(....);              // Receive the data
     t_2 = get_vtime();
     wait_event (&t_x, &Δ, ...);   // Wait for the sim
     if  (t_x + Δ > t_1)
          t_3 = t_x + Δ;
     else
          t_3 = t_2;
     set_vtime (t_3);              // Adjust virt . time
}
```

Figure 3. Stub code example for `MPI_Recv()`

The sender in Figure 2 records the virtual time t_x at the start of the send. It packs that information, together with the

length of the message, the tag, and the destination, into an event to the simulator.

On the receive side, upon entering a blocking MPI receive or wait function, the application records the current virtual time t_1. It then calls the MPI library to receive the data, and waits for an event from the simulator node. For each message the simulator calculates a delay Δ based on the length of that message, and sends Δ to the receive node. It also sends the corresponding t_x it has received from the sender, to the receiver.

When a node receives an event from the simulator, it does a simple calculation to determine whether that message arrived before it started to wait for it (t_1) or after. If it is the latter, it synchronizes its virtual time to that of the sender, taking the delay the network simulator provided into account: $t_x + \Delta$. If the message was sent before the receiver entered `MPI_Recv()`, $t_x + \Delta < t_1$, then the virtual time is set to t_2, which excludes the wait for the event.

This mechanism synchronizes the virtual clocks on each application node with the other nodes it communicates with. It also allows the simulator to be arbitrarily slow without the application noticing it. The virtual time between entering a blocking MPI receive or wait function and when that function returns, is just the delay the simulator calculated. The benchmarks we ran for this paper report the same execution time when run with or without our simulator. That is because we changed the `MPI_Wtime()` function to return the virtual time.

The only change we had to make to the application source code was to rename the `main()` function of the IS benchmark and replace the keyword `program` with `subroutine` in the Fortran benchmarks. This is necessary so the simulator can setup MPI communicators before it starts the benchmark. The simulator code and the benchmark are then linked together with our library that provides stubs for all of the MPI functions. Send function stubs transmit the data to the destination node, and an event to the simulator (Figure 2). Blocking receive and wait function stubs contain the code that is described in Figure 3. All stubs contain a statement that replaces the `MPI_COMM_WORLD` communicator of the application with one that the simulator created before the application started (`app_communicator` in Figure 1). The application uses it, through the stub library, to send and receive its MPI messages. `MPI_COMM_WORLD` contains all nodes and is used for the transmission of events between the simulator and the application.

The idea of running a network simulator on a separate node that can, through events, control the message passing of an application has two advantages. The stub library takes up only a very small amount of extra code space; all the data is collected on the simulator node. That means even applications and data sets that use all of the physical memory on their node can be instrumented and evaluated. Second, the time required to gather and possibly filter, analyze, and save the data on the simulator node, does not affect the virtual time of the application nodes.

For this paper we use only statistics gathered by the simulator that are not time dependent. Although adjusting and synchronizing the virtual time on the nodes of the application seems to work (the benchmarks report the same execution time as when run natively), we have not done enough testing and validation on this early prototype to report results.

We ran our experiments on a 256-node Myrinet cluster with dual 3 GHz Intel Xeon CPUs. Each node has 4 GB of memory.

3 Measurements

In this section we define five measurements that are of interest when analyzing communication patterns. For each of these measurements we describe what they are, how they have been obtained, and we make some suggestions on why a particular measurement is important in communication software and hardware design or purchase decisions.

3.1 Message density distribution

Knowing which nodes in an application communicate with what other nodes is important for several reasons. In connection oriented networks this knowledge can help with strategies to manage open connections. It also relates to the topology the application runs on. More efficient routing, node allocation, and node numbering algorithms may exist for a given communication pattern. Optimizations to minimize hot spots in the network may also benefit from this analysis.

We gather this information by counting each message and storing that counter in a two-dimensional array indexed by source and destination. Figure 4 shows CG on 4 nodes and Figure 5 shows it on 16 nodes. The darker a rectangle in the graph is, the more messages were sent between the corresponding source and destination node. White indicates that no communication (or a minuscule amount) has taken place between those two nodes.

In the case of CG running on 4 nodes, the light gray intensity at coordinate 0, 0 indicates that node 0 sends some messages to itself (416), while the large majority (1265) go to node 1. Node 0 sends one message each to nodes 2 and 3, but that is not enough to show up as a distinguishable gray level in Figure 4. Running CG on 16 nodes makes it very clear in Figure 5 that most of the messages are sent among nodes that are close to each other in MPI's logical node numbering scheme: most messages are exchanged among nodes 0 through 3, 4 through 7, 8 through 11, and 12 through 15 respectively.

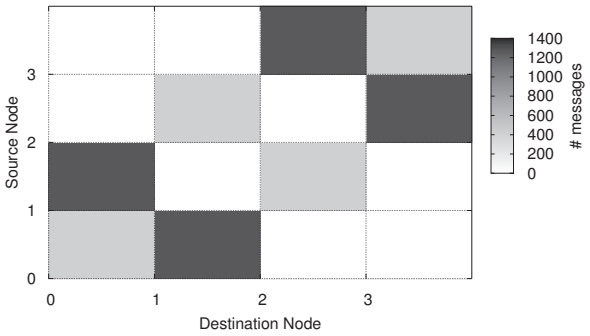

Figure 4. CG message density distribution

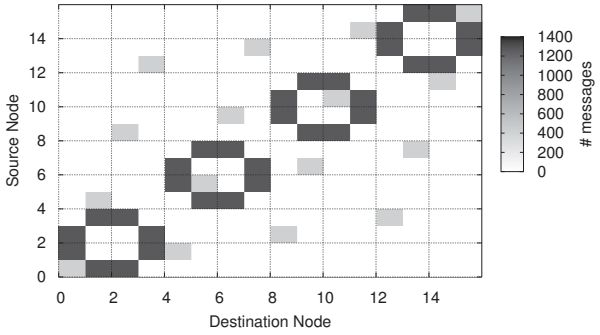

Figure 5. CG message density distribution

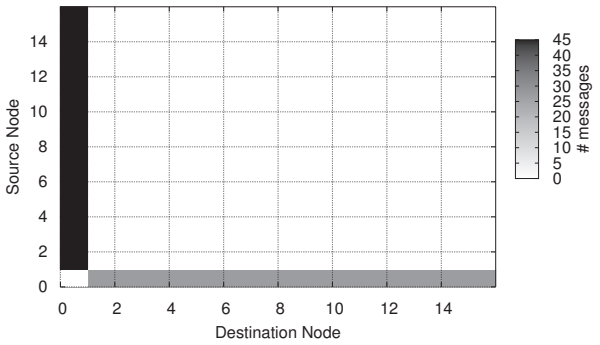

Figure 6. FT message density distribution

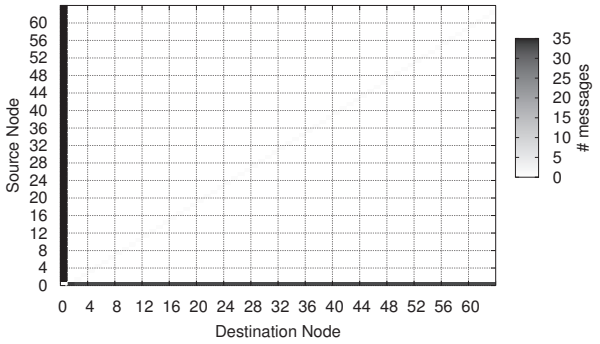

Figure 7. IS message density distribution

The network simulator counts point-to-point messages as well as messages that are part of collective operations. That means that a broadcast and reduce operation originating at node 0 will show up as a band in row 0 and column 0, indicating that node 0 has sent messages to all other nodes and received messages back from all of them. An example of this is shown in Figure 6 for the FT, and in Figure 7 for the IS benchmarks.

In reality, the underlying broadcast is probably using a fan-out tree algorithm in which nonleaf nodes in the tree pass messages further down the tree. This cannot be detected by the network simulator because it happens below the MPI API level and does therefore not show up in the diagrams presented here.

In many cases it is important to know what the actual network traffic looks like, and future version of the network simulator will be able to provide that. However, knowing communication patterns at the algorithmic level is also important. The FT and IS examples show that they use collective operations almost exclusively. We will see that in more detail in Section 3.3. The graphs in Figure 6 and 7 also show that all of these collective operations originate and terminate at node 0.

The MD benchmark has a much more interesting message density distribution pattern. Figures 8 and 9 show a communication pattern similar to CG. Communication is localized within groups of 4 nodes. However, MG exhibits a larger pattern where the groups of 4 are themselves grouped in clusters of 16 nodes. In addition, the long diagonal bands indicate communication from each cluster to nodes in the next lower cluster.

Finding inexpensive network topologies and connection management schemes that best fit applications is difficult. A network that works well for all-to-all communication patterns is expensive.

The FT benchmark is usually considered to have an all-to-all message density distribution. We have seen above that this may be true depending on the implementation of MPI and how it routes collective operations. However, algorithmically, FT performs only broadcast and reduce operations with node 0 as the root. A specific topology and/or dedicated hardware for collective operations rooted at node 0 might benefit FT greatly. Good nearest neighbor communication would also help, if the broadcast and reduce algorithm were implemented in a fan-out/fan-in tree that takes advantage of that.

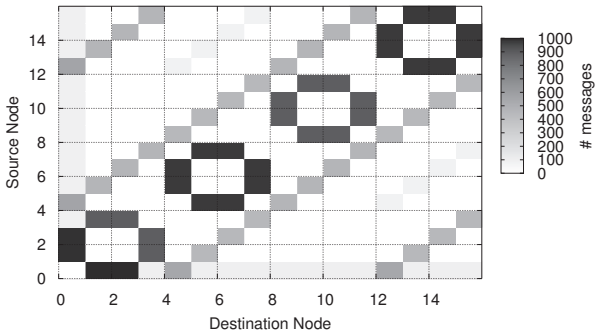

Figure 8. MG message density distribution

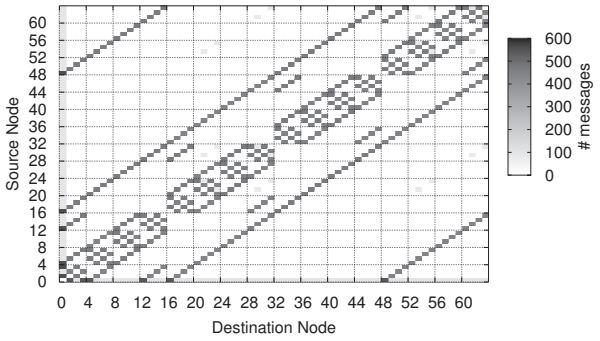

Figure 9. MG message density distribution

Optimizing for nearest neighbor communication would also help CG and MG. However, MG needs more. Although it communicates in a clearly clustered fashion, it does communicate with more than just the nearest neighbors, and the pattern is complex enough that an optimized solution would, most likely, also support the all-to-all case well.

3.2 Data density distribution

The data density distribution is similar to the message density distribution. Instead of counting messages, though, we accumulate the number of bytes sent from one node to another. This can be interesting and different from the message density distribution. Some applications exchange large size messages among some nodes, and large number of messages among other nodes.

A more common case is where one node sends much more data to another node, but only receives short acknowledgments back. The number of messages exchanged is the same, but the data flow is not. We can see an example of this in Figures 10 and 11. They show the message density distribution of the BT benchmark on 16 nodes and the data density distribution of the same benchmark and run.

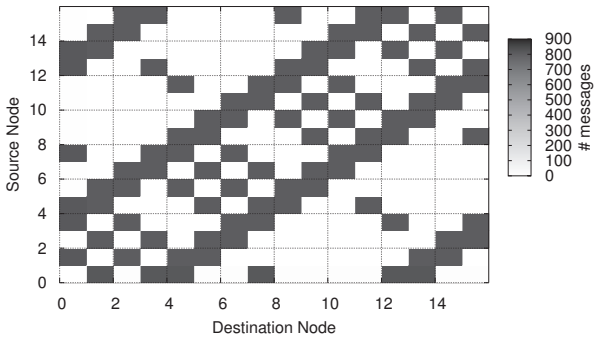

Figure 10. BT message density distribution

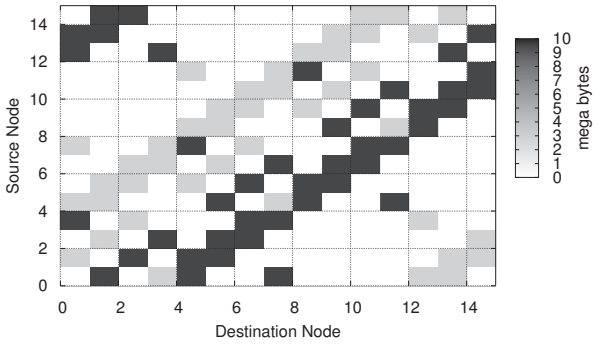

Figure 11. BT data density distribution

3.3 Collectives and point-to-point

The idea of performing collective operations, such as broadcasts or reductions, inside a NIC is appealing. The nodes within the fan-out or fan-in tree usually perform very simple operations, such as an addition or finding a maximum or minimum. Interrupting the host processor to execute such simple functions takes much more time than the actual execution of these functions. Executing the functions on the NIC, even on a fairly feeble NIC CPU, can improve performance [10].

Off-loading collective operations onto the NIC does not always make sense. NIC resources are scarce, and point-to-point operations have to be high performance as well. The ratio of point-to-point versus the number of collective operations is therefore an important characteristic of an application.

For the NAS parallel benchmarks, that ratio is not dependent on the size of the benchmark. However, the ratio does depend on the class of FT and MG. We show the ratio for all eight benchmarks in Figure 12 running on 64 nodes.

The graph shows that the BT, CG, LU, and SP benchmarks use point-to-point almost exclusively, while EP, FT,

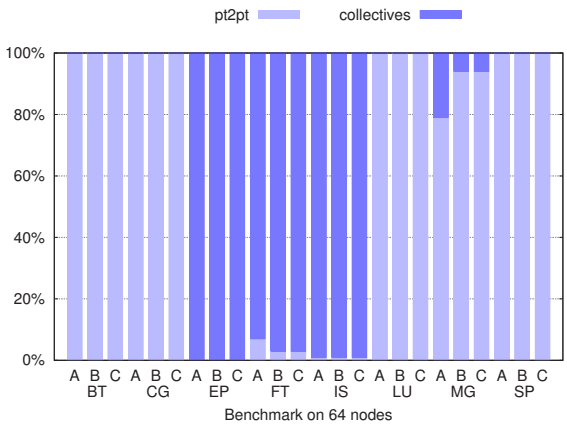

Figure 12. Collectives/point-to-point ratio

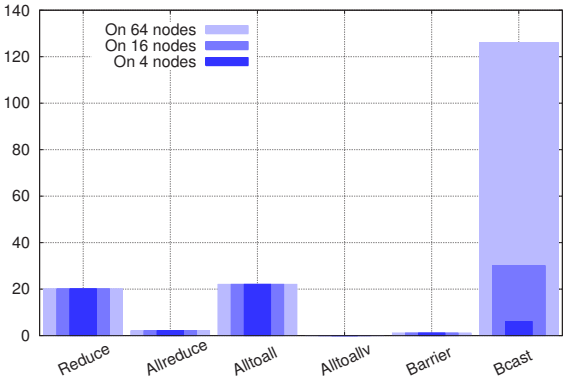

Figure 13. Collectives used by FT

and IS make heavy use of collective operations. The MG benchmark uses a few collectives; fewer with increasing class size.

We count the total number of point-to-point messages and divide by the number of nodes. Each collective operation counts as one operation. We count both an `MPI_Bcast()`, a fan-out, and an `MPI_Allreduce()`, a fan-in and a fan-out, as one operation.

3.4 Number and type of collectives

Once we know that an application uses collective operations, it is also important to know which ones are used. Six of the MPI collective operations are used by the NAS parallel benchmarks. Figure 13 shows that FT uses five of them. `MPI_Alltoallv()` is only used by IS. When we increase the number of nodes used by FT, the use of `MPI_Bcast()` increases, while the use of the other collectives remains constant. That is an indication that they are only used at the beginning of the program to distribute initialization data and at the end to gather results. BT, LU, MG, and SP behave the same way: various collective operations are used a small, fixed number of times, while the use of the broadcast operation increases with the number of nodes. The CG benchmark uses one reduce operation and one barrier operation independent of size or class. EP uses four `MPI_Allreduce()` operations and one `MPI_Barrier()`. IS uses two reduce operations and eleven each of `MPI_Allreduce()`, `MPI_Alltoall()`, and `MPI_Alltoallv()`.

Judging only by the NAS parallel benchmarks, it would make sense to devote effort into providing an efficient and fast broadcast operation, while other collectives are less important. It will be interesting to see whether that conclusion holds for other benchmarks and real applications.

Each call to a collective operation generates an event inside the network simulator. We assign a different event type to each type of collective operation and can easily keep track of which ones are called by the application and how often.

3.5 Message size distribution

The size of messages sent by an application is another important characteristic. Many message-passing systems have latency optimizations for shorter messages and try to reach the highest achievable bandwidth with the smallest message possible. Practical considerations make this a difficult task and designers have to make tradeoffs. Knowing what size messages an application sends, can help with decisions like choosing the packet size and buffer sizes inside switches.

For most of the benchmarks, message sizes get smaller the more nodes are used. Figure 14 shows this for SP, class A. The total number of messages goes up with the number of nodes used (but not the class of the benchmark). At 4 nodes, SP sends about 5,000 messages of length $\leq$ 64 kB and $15,000$ of length $\leq$ 16 kB. On 16 nodes it sends $96,000$ of length $\leq$ 16 kB, and $58,000$ of length $\leq$ 4 kB. The trend continues on 64 nodes. At that size, SP sends $155,000$ of length $\leq$ 16 kB, $540,000 \leq$ 4 kB, and $540,000 \leq$ 1 kB.

An exception to this behavior is EP which does not send messages larger than 16 bytes. Another special case is IS, which sends mostly messages no larger than 16 bytes and some that are $>$ 4 k KB and $\leq$ 16 kB. MG is the only one that sends messages of various sizes; the other benchmarks send either very small messages, or much larger ones. Presumably the smaller messages are synchronization messages, while the larger ones are used to exchange data.

We create a bucket for messages $\leq$ 16 B, another bucket for messages $>$ 16 B and $\leq$ 64 B, another for messages $>$ 64 B but $\leq$ 256 bytes, and so on. In all, there are thirteen

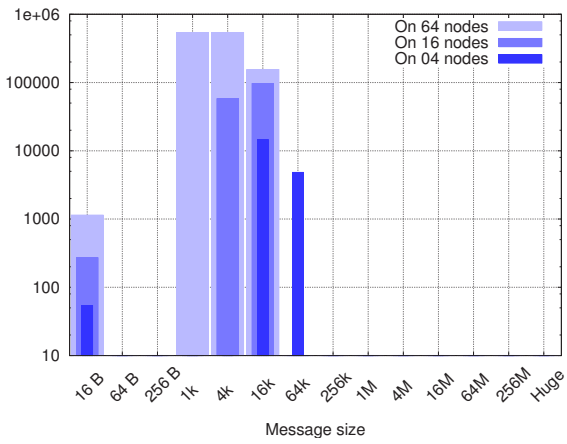

Figure 14. Message sizes used by SP

buckets to count the various message sizes.

4 Related work

The behavior of the NAS parallel benchmarks is fairly well understood. They have been analyzed in the past with different methodologies and emphasis on various aspects of a parallel application. For example [11] looks at their scalability and how the architecture of a ccNUMA machine and a workstation cluster meet the requirements of the NAS parallel benchmarks. That paper does not evaluate the message patterns described in Section 3.

Counting the number of MPI send and receive operations, and adding the number of bytes transmitted is relatively easy, as [7] shows. For the experiments described in that paper, print statements were inserted before each MPI call. Of course, this greatly changes the timing, and possibly the behavior, of the application and prevents any measurements that are timing dependent (for example the compute to communication ratio). The communication patterns mentioned in [7] are the number of times the MPI functions are called by the NAS parallel benchmarks, not the patterns exhibited by the messages traversing the (logical) grid.

The term communication patterns is used in [2] in the same sense as in the work presented here. For the use of those patterns, the paper gives the example of preestablishing connections between only those nodes that actually communicate with each other. The paper also analyzes the message size distribution as we do in Section 3.5.

MPIP is an MPI profiling library that wraps MPI communication calls in timers. Vetter and Yoo use MPIP in [9] to measure how much time is spent communicating. They also use MPIP to gather MPI tracing information. They have to do so judiciously because the trace data has to be written to disk which perturbs the application. The tracing information contains the data to create a message density distribution graph like we present in Section 3.1. A similar approach is used in [8] to look at message data distribution and message size distribution.

All of these experiments have one or more of the following drawbacks:

1. a large and extensive effort to instrument the application,
2. the almost impossible task of keeping the running application as unperturbed as possible,
3. a reduction in the amount of memory available to the application so trace data can be stored, and
4. a language specific measuring tool. In [4], for example, the NAS parallel benchmarks were rewritten in C.

There have been efforts, such as [6] that account for the overhead introduced by the measuring tool. The work in [6] is interesting because it makes use of the MPI profiling interface and attempts to compensate for measurement overhead as it occurs; not simply adjusting the total execution time. We belive that our approach is more precise, although we have not yet shown that.

The work we describe in this paper shows that the same measurements can be done with no real change to the application and no perturbation to its message-passing timing and behavior. This is because we combine the aspects of a simulator with the properties of a profiling tool.

5 Summary and future work

We have introduced a novel tool that lets us collect communication pattern information from an application. The tool, a combination of a network simulator and profiling method, runs on a separate node and does therefore not take away any memory from an application node. It is also independent of the language the application is written in: the NAS parallel benchmarks we used to demonstrate the tool are written in C and Fortran. A one-line change to the source code of the application is needed to link it with our simulator.

A further important aspect of this new way to gather message-passing data, is that the virtual time of the application is not changed. Our benchmarks reported the same execution time whether run together with our simulator or alone.

We grouped the data we collected into five categories: message density distribution, data density distribution, collectives versus point-to-point, and number and type of collectives. We presented interesting examples in each category in an easy to understand format. The complete set is available in [5]

For each category we briefly described why that information gives hardware designers the information they need to create network topologies and network interfaces that are best suited for these benchmarks and the class of applications they represent. The patterns give system software designers the information they need to optimally allocate and number nodes, and optimize for specific message lengths and type of collective operations.

In the future, we will make the simulator itself parallel. The simulation would then run on $n+m$ nodes; n nodes for the application, and m nodes for the network simulator. The current version increases the wall clock time of the application because for each message there are two events that travel across the network. Later, when the simulator takes the network topology and possible congestion into account, more nodes will help running the simulation faster.

Even the current prototype lets us adjust the message delay (Δ in Section 2). For example, we could simulate a network with zero latency. Once we have confirmed that our virtual time algorithm works correctly, we will be able to show application behavior under different network characteristics, do intrusion-free MPI tracing, and conduct studies about what would happen, if collectives were done in hardware.

Acknowledgments

George Riley, Georgia Tech, has helped a lot by teaching me about parallel discrete event simulation and helping shape ideas for the supercomputer simulation project. Many thanks go to Arun Rodriguez for several insightful discussions. I would also like to thank Keith Underwood for suggesting the supercomputer simulation project, and the other team members, Ron Brightwell and Jim Tomkins, for their helpful comments.

References

[1] D. Bailey, T. Harris, W. Saphir, R. V. der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[2] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *Parallel and Distributed Computing and Systems (PDCS)*, pages 724–729, Nov. 2002.

[3] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[4] S. Prakash and R. L. Bagrodia. MPI-SIM: using parallel simulation to evaluate MPI programs. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 467–474, 1998.

[5] R. Riesen. Communication patterns of the NAS parallel benchmarks. `http://www.cs.sandia.gov/~rolf/NAS`, Jan. 2006.

[6] S. Shende, A. D. Malony, A. Morris, and F. Wolf. Performance profiling overhead compensation for MPI programs. In B. D. Martino, D. Kranzlmüller, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18 - 21, 2005. Proceedings*, volume 3666 of *Lecture Notes in Computer Science*, pages 359–367. 2005.

[7] T. B. Tabe and Q. F. Stout. The use of the MPI communication library in the NAS parallel benchmarks. Technical Report CSE-TR-386-99, The University of Michigan, 1999.

[8] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[9] J. S. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.

[10] A. Wagner, H.-W. Jin, D. K. Panda, and R. Riesen. NIC-based offload of dynamic user-defined modules for Myrinet clusters. In *IEEE Cluster Computing 2004*, Sept. 2005.

[11] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing '99 Proceedings*, 1999.