Scalable Collection of Large MPI Traces on Red Storm

Rolf Riesen
Sandia National Laboratories*
Albuquerque, NM 87185-1110
rolf@sandia.gov

Abstract

Gathering large MPI traces and statistics is important for performance analysis and trouble shooting of applications. Traces, with detailed information about each single message an application has sent, are crucial to characterize the message passing behavior of an application. On massively parallel systems like Red Storm the amount of data collected impacts the performance and behavior of the application and is therefore not feasible. We present a new tool to enable the scalable collection of large amounts of data on Red Storm class systems[1].

Keywords: MPI, simulation, traces, virtual time

1 Introduction

Collecting accurate message passing traces of running applications at the MPI level is necessary for several reasons. MPI traces can help understand the behavior of applications by using the traces to visualize the communication patterns of an application. The traces can also be used for debugging and as input to system simulators. These trace driven simulators can help with learning how an application makes use of the communication fabric and how an application will perform on a next-generation machine.

Many tools for collecting MPI traces exist, but all of them have drawbacks. Some tools generate detailed traces, but the amount of data collected is large and requires time to send to storage. This influences the run time, and sometimes the behavior, of the application under measurement. The timestamps in the trace data are influenced as well.

Other tools compress the amount of generated trace data and try to perturb the run time behavior of the application as little as possible. Unfortunately, pre-analyzing and compressing the trace data still takes CPU cycles away from the application. Furthermore, the compression is usually not lossless and timing data, event ordering information, or message envelope information gets lost with these methods.

Some users of trace data are interested in the message data itself. For example, a trace driven simulator that simulates one node of a parallel application needs to feed the process on that node valid data. Otherwise the process might not behave in the same way as it would outside the simulator, when it is running as part of a parallel application. Collecting the application data of every MPI message during an application run generates enormous trace files and greatly influences the timing of an application.

In this paper we describe a tool named Seshat[2] [1] which we have extended to allow tracing of MPI applications. Seshat is execution driven and has a feedback channel into the ap-

*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]This paper presents data that was collected after a bug was fixed that skewed the results presented at CUG 2007.

[2]Seshat was the Egyptian goddess of measurement and recording.

plication that it uses to update the virtual time the application is running in. This makes it possible to collect and store arbitrary amounts of data while the application is running. While the wall-clock time increases, the application or benchmark is not aware of that and reports the same run and iteration time whether it runs natively, with the simulator attached, or writing trace data.

In Section 2 we briefly describe Seshat and its capabilities. We then describe, in Section.3, the experiments we conducted to test the tracing capabilities of Seshat, and analyze the data we have gathered. We conclude the paper with the related work Section 4, and our plans for future work and a summary in Section 5.

2 Seshat

In this section we briefly describe what Seshat [1] is and how it can collect trace data.

Seshat is an execution-driven network simulator. It is written as a library that is linked with an MPI application. No instrumentation of the application code is necessary; relinking it with Seshat is enough. We have tested this with C and Fortran applications using MPI-1.1 and MPI-2. Seshat makes use of the profiling interface that is part of the MPI standard (PMPI). With hooks into most of the MPI calls, Seshat is able to initialize itself, collect information about the running application, and adjust the application's virtual time frame through the profiling interface. For example, during MPI_Init Seshat reads a configuration file, sets up the a communicator for the nodes of the application, and starts the network simulator.

Figure 1 depicts a conceptual view of Seshat. The application runs as before, albeit within a communicator that Seshat has setup for the application nodes only. `MPI_COMM_WORLD` encompasses the application and the nodes occupied by the network simulator. In the example of Figure 1, the application runs on four nodes, while the job was started on five nodes; giving the extra node to the network simulator. Whenever the application makes use of `MPI_COMM_WORLD`, Seshat, through its hooks at the profiling interface level, replaces it with the communicator it created for the application only. Therefore, the application never learns of Seshat's presence and assumes it is running on only four nodes.

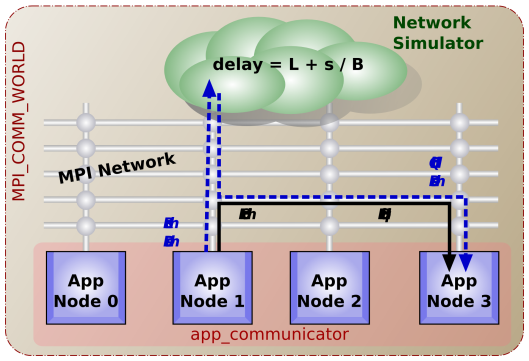

Figure 1: Conceptual View of Seshat

The network simulator that is currently part of Seshat is very simple. It models the point-to-point and collective performance characteristics of MPI on the Cray XT3^{TM} Red Storm computer [2, 3] installed at Sandia National Laboratories. It does so using a C function that calculates the time a message spends in the network based on the message length. The current simulator is not aware of Red Storm's topology and cannot model congestion in the network. We have plans to turn the network simulator itself into a parallel program and simulate the actual network to gain more detailed results in the future.

Whenever the application sends a message, it travels through the network as before and arrives at its destination. In addition, the Seshat wrappers send an event with the message envelope information to the network simulator. The simulator calculates a delay time and sends an event with that information to the node that received the MPI message. A receiving node accepts the MPI message and then waits for the corresponding event from the network simulator. Based on the information contained in the event, the receiving node adjusts its virtual time. The adjustment is usually backwards, since transport and processing of the events takes longer than the transport of the application MPI message alone. However, in cases where Seshat simulates a slower network, the adjustment can be forward.

The application or benchmark uses MPI_Wtime to report run time. Seshat controls MPI_Wtime and returns the local virtual time instead of wall-clock time. The local virtual clocks are synchronized through the messages and events sent among the application processes as described by Lamport in [4]. That means that the application reports the same run time as it does without Seshat attached to it. The time stamps for each MPI message are accurate (within the virtual time frame) and virtual time progresses based on application behavior and the parameters of the network Seshat is simulating. Time spent inside the network simulator extends the wall-clock run time, but does not influence the virtual time of the application.

This independence of the time system the application runs in, plus the knowledge about every single message of the application under test, allows the network simulator to generate MPI traces.

2.1 MPI Traces

The trace format for this initial prototype is very simple. A single ASCII text line is written for every message event that arrives at the network simulator. For a start we collect the information described in Table 1. Each line consists of nine white-space separated fields. Each line in the trace file consumes about 90 bytes.

Currently, no buffering or attempt at compression takes place. The data presented in the next section is from test runs where the trace file was written to the user's NFS-mounted home directory. Obviously writing to a high-performance storage system, buffering the data, and maybe compacting it, would reduce the wall-clock time for the application under test.

Table 1: Seshat Trace File Format

Field	Description
0	Event type
1	Time of event at network simulator
2	Source (or root) of message (collective)
3	Destination
4	Virtual send time
5	Simulated time in network
6	MPI tag
7	Type of collective (or point-to-point)
8	Length of message in bytes

3 Experiments

In this section we describe the experiments we have performed to test our assumption that arbitrarily large trace files can be written by Seshat's network simulator without impacting the the run time the application observes. We do this by running several benchmarks and compare their reported run times to the times they report after a large trace file has been written.

3.1 Experimental Setup

All experiments described in this section were conducted on Sandia's Cray XT3^{TM} Red Storm machine. It was running version 1.5.39 of the Cray system software. The performance characteristics of that software release are also the ones Seshat simulates.

We use some of the NAS parallel benchmarks version 3.2.1 to verify our claims. These benchmarks are simple compared to real applications. However, we are only interested in a proof of concept. Any code that sends and receives a large number of MPI messages will do. In some tests we write so much information that we slow down the benchmark so that it takes several hours of wall-clock time to complete. Yet, it reports the same few seconds or minutes of (virtual) run time as it would when run natively.

We ran all of our benchmarks several times for this study; either in native mode (Seshat

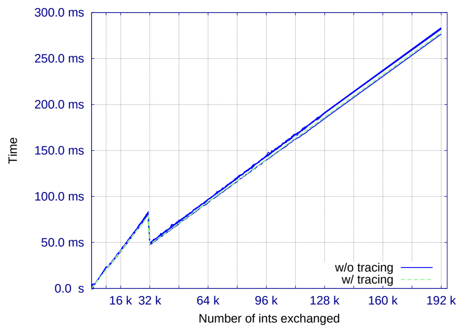

Figure 2: All-to-all Benchmark on 128 Nodes with, and without Tracing

not linked to the application) or trace-enabled mode. We ran each series of tests from within the same PBS script to guarantee that all tests run on the same set of nodes. For each series we interleaved traced and non-traced runs. Red Storm was in production use during these tests, but through its design, the machine dedicates sets of nodes to a single application.

For our experiments we make no attempt to reduce the wall-clock run time of our benchmarks. We write data to the user's NFS-mounted home directory and our trace files are simple ASCII text; one line per event as described in Section 2.1.

3.2 Time Perceived by the Application

We conducted our first test with a benchmark that performs all-to-all operations in a tight loop and reports the average time of an MPI_Alltoall for increasing message sizes. This is the benchmark we use to calibrate Seshat's simulation of collective operations. Therefore, we expect to see a very close match between native and trace-enabled reported run time. Figure 2 shows the measured results.

The graph represents ten interleaved runs; five native and five traced. The reported run times match each other so well that the ten plot lines almost completely overlap each other.

For our next test we ran the NAS LU class A benchmark. Figure 3 shows the results on four nodes, and Figure 4 on 64 nodes. We ran each one five times in native mode and fives times with tracing enabled. The runs are labeled "Run A" through "Run E", and the *y*-axis shows the runtime as reported by the benchmark.

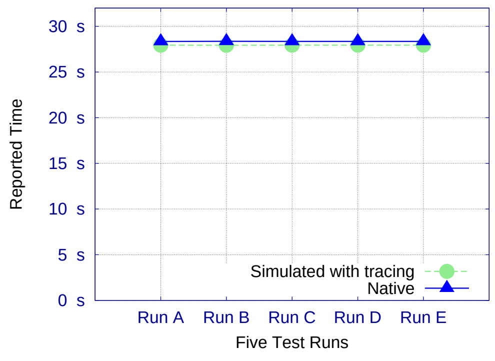

Figure 3: Five Runs of the Class A LU Benchmark on 4 Nodes

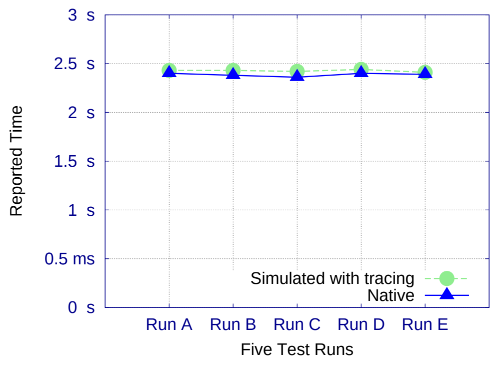

Figure 4: Five Runs of the Class A LU Benchmark on 64 Nodes

These initial results are encouraging, since the reported native run times and with tracing turned on are nearly identical, despite the fact that the traced versions spend use much more wall-clock time.

3.3 Trace File Size and Wall-clock Time

Table 2 lists the wall-clock times for each individual run The first time is the wall-clock time of one run without tracing (Seshat is not attached to the application), while the second time shown includes writing the trace file. The table also lists the number of events in each

Table 2: Trace File Collection Statistics

Code	Nodes	Events	Wall-clock time		Trace size	Simulation	Proportional
			native	w/ trace		error	error
All-to-all	128	4,826,000	1,300s	15,671s	397 MB	-1.2%	-0.099%
CG, A	16	47,149	2s	145s	4 MB	-20%	-0.138%
CG, A	64	269,501	2s	839s	22 MB	-15%	-0.036%
CG, B	64	1,279,421	8s	3,874s	102 MB	-29%	-0.060%
CG, C	64	1,279,421	15s	3,874s	103 MB	-13%	-0.050%
LU, A	4	126,635	30s	391s	11 MB	-1.5%	-0.115%
LU, A	16	759,699	10s	2,288s	61 MB	-6.3%	-0.028%
LU, A	64	3,545,003	4s	10,581s	285 MB	1.6%	0.001%
SP, A	16	154,260	12s	467s	13 MB	-6.5%	-0.167%
SP, A	64	1,233,012	5s	3,734s	101 MB	-18%	-0.024%

trace file (there is one file per run) and the size of the file.

The second last column of the table shows the increase or decrease in reported run time when tracing is enabled compared to the reported run time of the native run. Some of the reported times with tracing enabled are further away from the native run times than we had anticipated. In proportion to the wall-clock time ratio between native and traced runs, these errors are minimal, as the last column of Table 2 shows. We divide the traced wall-clock time by the native wall-clock time and use the result to divide the simulation error rate to arrive at the numbers in the last column.

Nevertheless, it seems there is a timing problem with the current version of the network simulator. Investigating further, we ran CG on 64 nodes using the class A, B, and C versions. We ran each class seven times in native mode interleaved with seven runs under the simulator. This time however, we left tracing turned off. Table 3 shows the results.

Table 3 shows that even without possible tracing interference, the network simulator does not match the reported wall-clock time of a native CG run. This is a problem that needs to be investigated and fixed. However, it shows that tracing itself is not the culprit and the observed variance with tracing enabled is due to a problem in the simulator itself, not the tracing capability.

3.4 Updated Results

The results presented at CUG 2007 were early and based on a version of Seshat that had a serious bug in it. In This section we show updated results after the bug has been fixed.

Figures 5 and 6 show SP, class A on 16 and 64 nodes. With the virtual time bug present, these results were much worse originally.

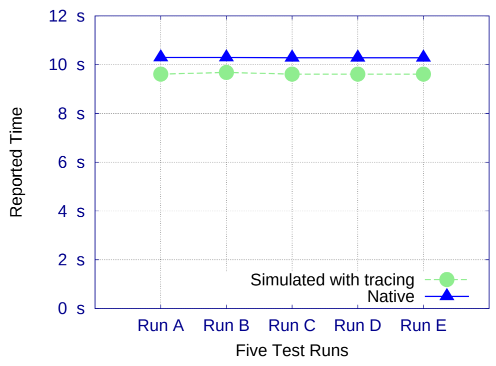

Figure 5: NAS SP Class A on 16 Nodes

Figures 7 and 8 show CG, class A on 16 and 256 nodes. Originally the reported times under tracing were 48% and 385% higher than native.

Table 3: Native CG and Simulated but no Tracing Runs

Code	Nodes	Wall-clock time		Simulation	Proportional
		native	simulated	error	error
CG A	64	1s	32s	75%	2.344%
CG B	64	7s	44s	-22%	3.460%
CG C	64	14s	49s	-12%	3.429%

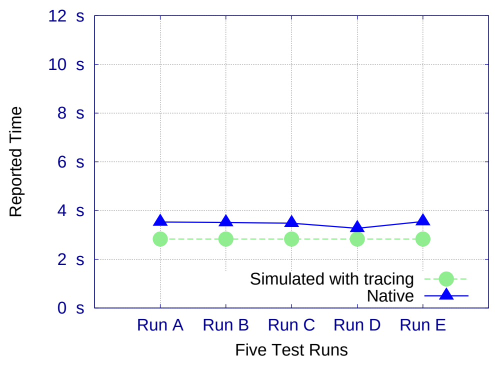

Figure 6: NAS SP Class A on 64 Nodes

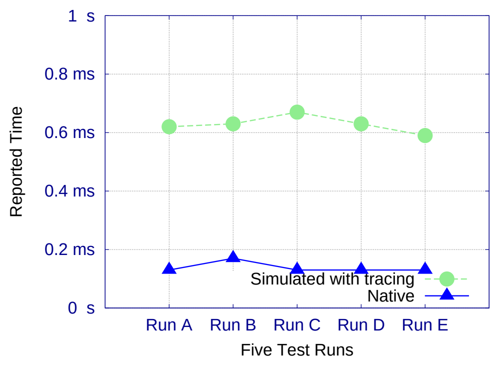

Figure 8: NAS CG Class A on 256 Nodes

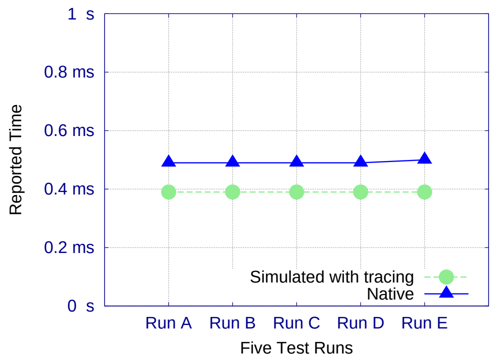

Figure 7: NAS CG Class A on 16 Nodes

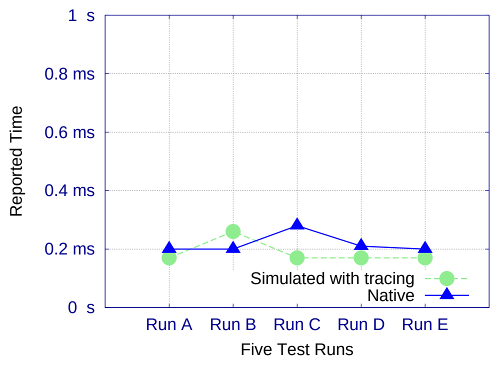

Figure 9: NAS CG Class A on 64 Nodes

To see whether the original problem was related to the class of benchmark, we ran CG on 64 nodes as a Class A (Figure 9), Class B (Figure 10),and Class C (Figure 11). With the error present before, the errors were 110% for class A, 25% for class B, and 3,557% for class C.

Finally we looked at CG runs without tracing to evaluate the impact of simulation alone.

4 Related Work

Many tools to collect MPI message information exist. The difficulty of collecting large, accurate traces has been well documented; e.g., in [5]. The MPI profiling interface provides an easy way to link into an existing application. Most of the existing tools fall into one of two categories. In the first category are tools that collect detailed information, but by doing so, influence the timing behavior of the application

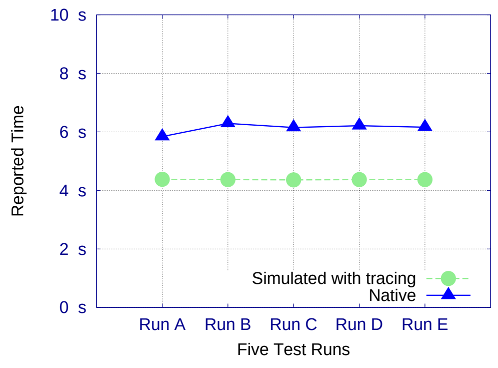

Figure 10: NAS CG Class B on 64 Nodes

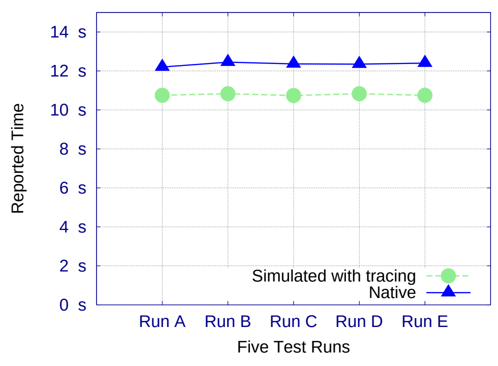

Figure 11: NAS CG Class C on 64 Nodes

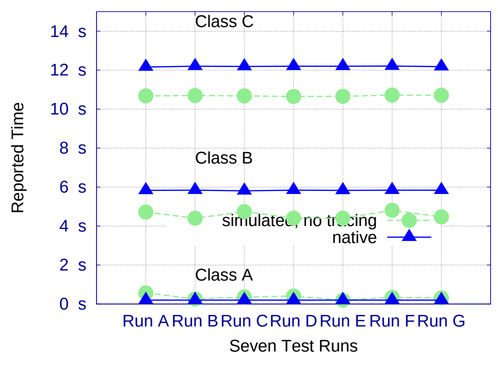

Figure 12: Simulated NAS CG on 64 Nodes, no tracing

under test.

Tools in the second category only collect general information about the message passing behavior of an application. Examples include how many message are sent, the average message size, the ratio of collectives versus point-to-point messages and so on. These tools do not alter the behavior of an application, but at the same time, often do not provide enough detailed information for debugging and traced-based simulation. Examples in this category include pMPI [6].

Recent work presented in [7] uses a clever scheme to collect summary information about collections of messages, for example the ones sent repeatedly from an inner loop. The bundling of that information allows for very high compression ratios. Some loss in detailed message timing information has to be expected, though. Another idea for trace data compression is presented in [8].

Another approach is to estimate the overhead introduced by the debugging/tracing tool, and subtract it from the reported run times. An example of such work was presented in [9].

Work similar to what we have presented in this paper is documented in [10]. However, that work concentrates on performance prediction of MPI programs.

5 Future Work and Summary

The work presented here is in an early prototype stage. More work is needed to make the basic simulator accurate enough for use. However, the method presented here to gather arbitrarily large trace files without unduly perturbing the virtual run time of an application works. Even with runs that last for hours of wall-clock time the reported virtual run time of an application is not made worse than the current, buggy, simulator allows.

Once the above problem is solved several avenues for improvements present themselves. The Seshat configuration file should allow the trace format to be adapted to specific needs, and it should allow the filtering of trace events. In many situations only a subset of events are of interest.

To reduce the wall-clock time of collecting large traces, they should be written to a high-performance storage system instead of the NFS-mounted home directory as we did for this study. Buffering events on the simulator node would also be easy and allow larger writes to the trace file, which will further improve performance. As a matter of fact, buffering on the simulator node and additional nodes allocated to the simulator can be used to compress the trace data and write it to a parallel file system.

For tracing application that require the contents of each message, it would be simple to add the capability for collecting them to Seshat. Compression, buffering, and efficient file writes will be a must under these circumstances.

Finally, Seshat should be made compliant with other libraries that make use to the MPI profiling interface so they can be stacked on top of each other.

References

[1] Riesen, R.: A hybrid MPI simulator. In: IEEE International Conference on Cluster Computing (CLUSTER'06). (2006)

[2] Brightwell, R., Camp, W., Cole, B., DeBenedictis, E., Leland, R., Tomkins, J., Maccabe, A.B.: Architectural specification for massively parallel computers: an experience and measurement-based approach. Concurrency and Computation: Practice and Experience **17**(10) (March 2005) 1271–1316

[3] Camp, W.J., Tomkins, J.L.: The red storm computer architecture and its implementation. In: The Conference on High-Speed Computing: LANL/LLNL/SNL, Salishan Lodge, Glenedon Beach, Oregon (April 2003)

[4] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565

[5] Chung, I.H., Walkup, R.E., Wen, H.F., You, H.: MPI performance analysis tool on Blue Gene/L. In: Proc. IEEE/ACM SuperComputing, Tampa, FL (November 2006)

[6] Vetter, J.S., Yoo, A.: An empirical performance evaluation of scalable scientific applications. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002)

[7] Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalable compression and replay of communication traces in massively parallel environments. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). (2007)

[8] Knüpfer, A., Nagel, W.E.: Compressible memory data structures for event-based trace analysis. Future Generation Computer System **22**(3) (2006) 359–368

[9] Shende, S., Malony, A.D., Morris, A., Wolf, F.: Performance profiling overhead compensation for MPI programs. In: 9th European PVM/MPI Users' Group Meeting Proceedings. Volume 3666 of Lecture Notes in Computer Science., Springer Verlag (September 2005) 359–367

[10] Prakash, S., Bagrodia, R.L.: MPI-SIM: using parallel simulation to evaluate MPI programs. In: WSC '98: Proceedings of the 30th conference on Winter simulation, Los Alamitos, CA, USA, IEEE Computer Society Press (1998) 467–474