Supercomputer Simulation Design Through Simulation

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

Abstract

Performance modeling of large scale applications and predicting the impact of architectural changes on the behavior of an application is difficult. Traditional approaches to measuring applications often change their behavior. Modeling an application on a new architecture requires in-depth knowledge of the application. Both tasks often involve studying an application to learn where probes have to be inserted.

In this paper we describe a novel approach that is a hybrid between discrete event simulation and measuring the behavior of a running application. We explain how our early prototype works and how it can be used. We mention several experiments that we have already performed with this protype and show its potential for future research areas.

Keywords: MPI, simulation, performance, virtual time

1 Introduction

Large-scale parallel systems grow more complex with each generation. Estimating how a new system or design feature will affect a particular application is a necessary but difficult task. Sandia National Laboratories has begun a project to use discrete event simulation for a complete system containing thousands of nodes. Even using parallel simulators this is a nearly impossible task. This paper describes a very early prototype towards that lofty goal.

Decisions on how to improve a next-generation system are often made by people with a lot of experience and intuition about how these complex systems work. It seems obvious to assume that performance will increase if a more powerful processor is used. What is not so obvious is how much performance will improve. For other decisions, for example a change in the network routing algorithm or implementing collective operations inside the network interface card (NIC), there may be no performance gain at all. Predicting the application impact of even small changes in a complex system is very difficult. Simulation promises a more scientific way of answering such questions beforehand.

In this paper we describe the prototype of a hybrid simulator. The application executes unchanged on the compute nodes while a network simulator runs on one additional node. For each MPI message the application sends, an event is sent to the network simulator. The simulator keeps statistics about the MPI traffic and provides network delay information back to the application. The application uses that information to update a virtual clock on each of its nodes.

As far as the application is concerned, it operates at the same performance level inside the virtual time frame as before when the network simulator was not present. The network simulator has knowledge of all MPI traffic in the system and can change the perceived characteristics of the network. This allows us to measure application performance with a simulated network that is faster, slower, or has different collectives performance than the actual network in use. It also allows us to collect any imaginable statistics about an applications message passing behavior.

Although this simulator only exists as a simple protype so far, it shows extraordinary promise and has already yielded useful information about benchmarks and applications.

In Section 2 we describe in detail how the simulator works. How the simulator is used in practice is described in Section 3. In Section 4 we show some early results of validating our simulator, and in Section 5 we describe some of the experiments we are currently conducting. We then list advantages of our hybrid approach and compare it to other work in Section 6. We close the paper with a brief summary

*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.

in Section 7.

2 Design

In this section we describe the design of the first prototype of our simulator. It consists of a very simple network simulator and a wrapper library for the MPI calls. We will describe the simulator and the wrapper library in the next two sub-sections, and follow with a section on how the simulator handles virtual time, and a final section that describes how the simulator can be used.

2.1 Node (Application)

The MPI application runs as it would normally; it is not simulated. The network simulator runs on an additional node. For example, if an application requires 64 nodes to run, we will need 65 nodes to run it with a simulated network. For each MPI message that is sent, including collective operations, the application sends an event to the network simulator. During each receive the application will block until it receives an event from the network simulator.

The receive event has to match the MPI message the application is currently receiving and contains a delay value Δ. With this value, the network simulator informs the application node how long, in simulated time, the message spent in the network. Based on that information the node may then adjust its virtual time.

Sending, receiving, and matching events to MPI messages is done by a wrapper library that has to be linked with the application that is to be simulated. The wrappers also contain the algorithm to adjust the virtual time. The wrappers make use of the MPI profiling interface and are described is Section 2.2.

No change to the application is needed except that it must be linked with the simulation wrappers. Since we are not controlling or changing the application between MPI calls, a simulated application must run on the same machine that we want to simulate. The early prototype simulator described in this paper can only simulate the network. In order to only change variables associated with message passing but not compute time, the application has to run on the hardware that we are interested in evaluating.

We call it a hybrid simulator because only the message passing is simulated and the application runs, between MPI calls, as it did before. Through the profiling interface the original MPI library is used to transmit the application data across the existing network. This assures that the application runs correctly.

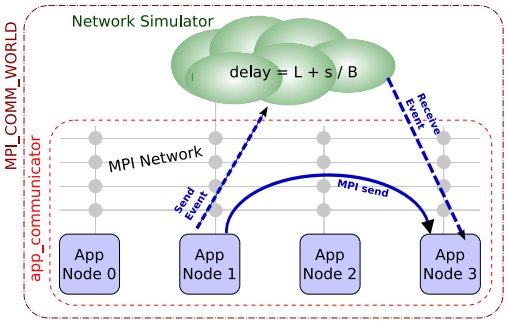

Figure 1: Data and event traffic

2.2 MPI wrapper library

Figure 1 shows a conceptual view of the simulator. The application runs unchanged using the **`MPI_COMM_WORLD`** communicator. The wrapper library changes that communicator for each MPI call to a communicator the simulator has created at startup and only includes the nodes of the application. Therefore, the application is not aware of the existence of the network simulator that runs on an additional node. The real **`MPI_COMM_WORLD`** includes the application nodes plus the network simulator node. It is used to exchange events between the application nodes and the network simulator node.

When an application sends an MPI message, it does so through the regular MPI library and the protocol stack and network that was in place before. This is shown in Figure 1 with the example of node 1 using **MPI_Send()** to transmit data to node 3. The data is sent and received in the same manner as before the application was linked with the network simulator.

After the message has been received, the function **event_wait()** waits for the matching event from the network simulator. The event contains the delay information calculated by the simulator and is used to update the local virtual time.

Event handling for collective operations is a little bit more complex. As with the point-to-point operations, the wrappers call the actual MPI library to carry out the operation. Each node that participates in the operation sends an event to the network simulator indicating what type of collective operation is in progress. The simulator forwards these events to the root node which waits for all the events. In the case of a broadcast operation the root node then sends an event via the network simulator to all the recipients of the collective operation. Figure 4 de-

```
int MPI_Send(void *data, int len,
             MPI_Datatype dt,
             int dest, int tag,
             MPI_Comm comm)
{
    t_x = get_vtime();
    SET_COMM();

    // Send the MPI message
    rc = PMPI_Send(data, len, dt, dest,
                   tag, comm);
    // Send event to simulator
    event_send(t_x, len, dt, dest, tag);
    return rc;
}
```

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

```
int MPI_Recv(void *data, int len,
             MPI_Datatype dt,
             int src, int tag,
             MPI_Comm comm,
             MPI_Status *status )
{
    t_1 = get_vtime();
    SET_COMM();

    // Receive the MPI message
    rc = PMPI_Recv(data, len, dt, src,
                   tag, comm, status);
    t_2 = get_vtime();

    // Wait for the matching event
    event_wait(&t_x, &Δ,
        status ->MPI_TAG,
        status ->MPI_SOURCE);

    // Adjust virtual time
    if (t_x + Δ > t_1) t_3 = t_x + Δ;
    else t_3 = t_1;
    set_vtime(t_3);
    return rc;
}
```

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

picts the traffic flow.

As with point-to-point operations, each node waits for the actual operation to complete and a matching event. The virtual time is adjusted based on the longest delay between the root node and any

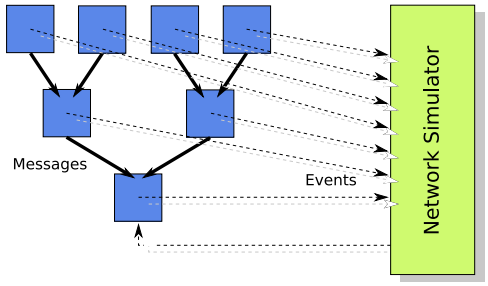

Figure 4: Event flow for collectives (reduction example)

of the sending nodes of a reduce operation.

2.3 MPI Communicators

During initialization the simulator sets up a communicator for the application nodes only. This communicator contains the nodes the application runs on and is used for intra-application communication. The regular **`MPI_COMM_WORLD`** communicator encompasses the application and the simulator, and is used to exchange events between the application and the simulator.

In order for the application to use the new communicator instead of **`MPI_COMM_WORLD`** without actually changing the application, we use a little trick. When an MPI wrapper sees **`MPI_COMM_WORLD`** being passed in, it replaces it with the communicator the simulator initially created for the application only.

If an application creates sub-communicators they are within the communicator the simulator created for the application nodes only.

2.4 Virtual time

Each application node maintains a virtual time frame. For each event it receives it may adjust the local virtual time depending on when the event was sent and how much of a delay the network simulator has assigned to it. Figure 5 shows the algorithm graphically and Figure 6 shows the pseudo code that implements it.

If we receive a messages with t_x (including the delay Δ added by the network simulator) that is less than t_1, then we set the local virtual time to t_1. That is we remove the actual time spent processing and receiving the data. The reason for this is that we cannot say how long before t_1 the message arrived.

If the virtual arrival time is after t_1 then we adjust the local virtual time to the virtual time the message

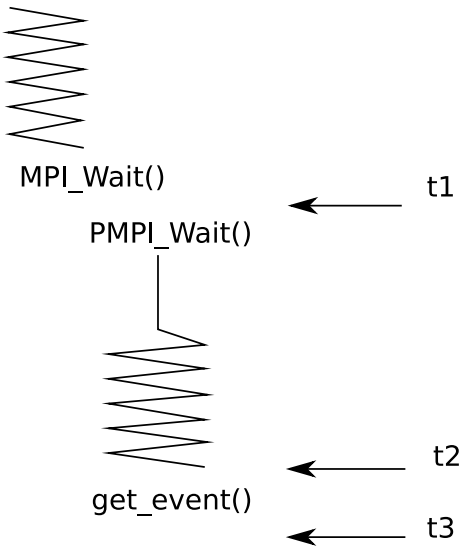

Figure 5: Virtual time

```
if  (t_x + Δ > t_1)
      t_3 = t_x + Δ;
else
      t_3 = t_1;
set_vtime (t_3);
```

Figure 6: Local virtual time adjustment

was sent plus the virtual time it spent in the network. This way the nodes synchronize their virtual times whenever sends and receives occur close enough to each other.

The function **MPI_Wtime()** is a wrapper that returns the virtual time that has passed since the program started.

2.5 Network simulator

The network simulator of the current prototype framework is very simple. It uses a model of the point-to-point performance of a network to calculate a delay based on message length. This delay, possibly adjusted to simulate a faster or slower network, is sent in the event to the receiving node. The receivers takes it into consideration when updating their local virtual time.

For point-to-point messages the simulator basically applies the well know function: $\Delta = \frac{s}{B} + L$, where the delay Δ is calculated based on the message size s, taking into consideration the bandwidth B and latency L of the network.

For collective operations the simulator assumes that a logarithmic fan-out and fan-in algorithm is used by the MPI library.

In addition to calculating network delays, the network simulator also keeps message statistics. Currently it counts all collective operations separately and all point-to-point operations. It also maintains tables to accumulate the number of messages and the amount of data sent between each node pair. It also updates a set of buckets that are used to keep track of the message sizes used. The simulator counts all messages of size ≤ 16 bytes, $\leq 64B$, $\leq 256B$, and so on.

Future version of the network simulator will take the topology of the network into account and will allow evaluation congestion and routing delays.

2.6 Linking

The network simulator and the MPI wrappers form a library that needs to be linked with the application to be simulated. The current prototype requires the renaming of the **main()** function of the application. In Fortran programs the keyword **`program`** must be replaced with **`subroutine`** and the function must be named **main_node()**.

Since simulator already has a stub for **MPI_Init()** and an application has to initialize MPI before it can make any message passing calls, it is easier to put simulator initialization into the **MPI_Init()** wrapper. Future generations of the simulator framework will not require a source code change at all. A simple linking with the simulator will be enough.

3 Usage

There are two steps involved in getting the simulator ready to be linked with an application. First, the point-to-point performance of the target architecture has to be measured (or predicted) using a simple point-to-point latency/bandwidth benchmark. Using the information, the network simulator needs a function that calculates the delay for a given message length.

A second measurement is necessary to accurately model collective operations. We run an all-to-all operation inside a tight loop and measure its performance on the target architecture for various size messages. That allows us to create a simple logarithmic model of the algorithm used by MPI. Figure 7 shows the model compared to the measured performance for our Cray XT3 Red Storm system.

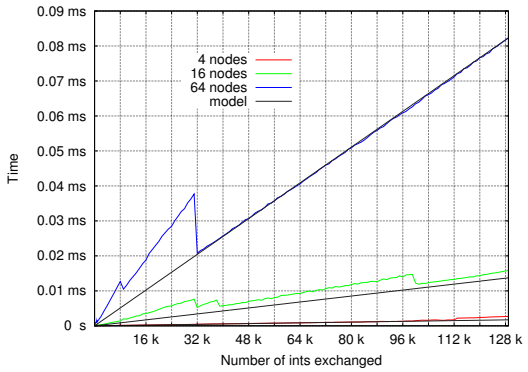

Figure 7: Modeling collective operations

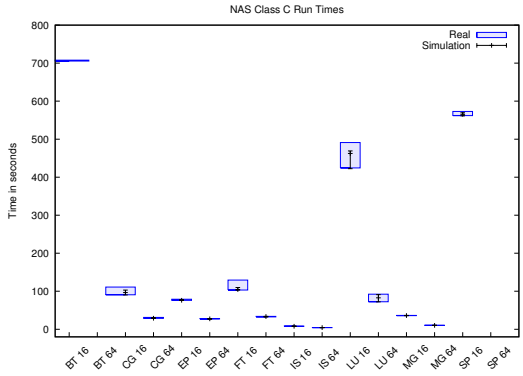

Figure 8: Runtime comparison of class A benchmarks

Once these simple models are in place, the simulator can be linked with the application as described in Section 2.6. Since we run the application natively, we must run it on the same types of nodes that the target system has otherwise the compute timing would be skewed.

In order for the simulator to be useful in the prediction of future systems it is necessary to run it on nodes that are of a different kind than the target architecture. This will be possible once we have actual node simulators and leave the hybrid model behind. It may also be possible to linearly adjust the compute portion of a job to match the target system. We will conduct experiments to evaluate the size of error introduced by such a simplistic model.

Once the application and the simulator are linked together we need a configuration file before we can run it. The file contains the number of nodes (currently always 1) tha the network simulator runs on, and the number of nodes the application is using. The file also contains a line with parameters that are passed to the simulator. Currently supported are factors for bandwidth, latency, and collectives that the simulator uses to adjust the computed delay based on the point-to-point and logarithmic collectives model. This allows us to conduct experiments pretending that collective operation incur zero-cost.

The configuration file also contains the parameters that would ordinarily be passed on the application command line. The simulator places those into an **`argv`** array and passes it to the applications original **`main`** function.

4 Validation

We have begun validating our approach. Figure 8 shows the performance of the class C NAS parallel benchmarks. Each benchmark was run seven times and each simulation was run seven times. The shaded rectangles in the plot show the range of timing values measured by the benchmark. The error bars are the times reported by the benchmark when run under the simulator. All of the simulated runs fall within the range of the run times reported when the benchmark was run in stand-alone mode.

We did not have access to a dedicated machine for our studies. To ensure the greatest accuracy possible, we allocated the required nodes and ran all seven iterations on the same set of nodes, interleaved with the seven runs of the simulation.

5 Experiments

In this section we describe some of the experiments that we have performed using the prototype simulator described in this paper.

5.1 Communication patterns

Since all message traffic causes events to be sent to the network simulator, it is easy to count and accumulate information about the message passing patterns an application exhibits. We have conducted experiments using the NAS parallel benchmarks and published them in [2].

5.2 Varying bandwidth and latency

The current network simulator basically uses the formula $\Delta = \frac{s}{B} + L$ to calculate the virtual time a message took through the network. It is therefore easy to include two additional parameters to modify the model of the network: $\Delta = \frac{s}{\alpha B} + \beta L$. The default for α and β is 1.0, but they can be changed in the configuration file. We are currently working on a paper

describing these experiments.

5.3 Zero-Cost collectives

In addition to the α and β parameters discussed above, the simulator accepts a third parameter that can be used to influence the behavior of collective operations. We use it to model a network that has the usual cost of point-to-point operations but imparts no delay at all for collective operations. Running applications like that gives us the best possible performance an application could achieve, if the performance of collective operations were improved.

We submitted a paper describing our results in [3].

5.4 Intrusion-free MPI traces

Another aspect of our simulator design which we have not explored yet, is its ability to collect huge amount of information without changing the virtual time an application runs in. For example, it would be easy to collect for each individual messages all its envelope information; e.g., source, destination, tag, len, and type of collective operation. We would also store the virtual send and receive time for each message and store that potentially huge amount of information on a disk.

The wall-clock run time of the simulation would dramatically increase, but the virtual time seen by the application would not. If this approach works as predicted, we have the potential to gather very large and accurate timing and communication profiles for applications; even if they span thousands of nodes.

6 Related work

This work combines discrete event simulation with performance evaluation of parallel applications. Each field has a large number of publications. However, combining the two in the manner described in this paper seems to be new.

Instrumenting applications to do performance analysis always introduces timing artifacts that skew the measurements. A large body of work has been devoted to keep that skew as small as possible, but all of these approaches 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 [1], for example, the NAS parallel benchmarks were rewritten in C.

There have been efforts, such as [4] that account for the overhead introduced by the measuring tool. The work in [4] 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 (virtual) timing and behavior. This is because we combine the aspects of a simulator with the properties of a profiling tool.

7 Summary

This paper describes a novel approach combining discrete event simulation with application performance measuring in a hybrid way. Our approach requires no changes to the application being measured and seems to be very accurate. The application runs in virtual time and does not perceive the intrusion caused by the events sent between the network simulator and the application nodes.

The current network simulator provides only a very simplistic network model. Future version will include knowledge about the network topology and provide more accurate information. Especially in the case of congested networks. The current version is already quite capable in collecting message passing behavior data and has been used in several experiments with benchmarks and real applications.

This work seems to be a promising avenue towards a simulation framework that will allow the simulation of large scale massively parallel machines.

8 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. Torsten Hoefler from the Technical University of Chemnitz has been very helpful with his assistance in running real applications with the simulator and suggesting improvements and additional research areas.

References

[1] Sundeep Prakash and Rajive 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, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[2] Rolf Riesen. Communication patterns. In *Workshop on Communication Architecture for Clusters CAC'06*, Rhodes Island, Greece, April 2006. IEEE.

[3] Rolf Riesen and Torsten Hoefler. **In submission:** zero-cost mpi collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI Users' Group Meeting Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[4] Sameer Shende, Allen D. Malony, Alan Morris, and Felix Wolf. Performance profiling overhead compensation for MPI programs. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack 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. Springer Verlag, 2005.