

MPI-NeTSim: A network simulation module for MPI

Brad Penoff and Alan Wagner
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
Email: {penoff,wagner}@cs.ubc.ca

Michael Tüxen and Irene Rüngeler
Department of Electrical Engineering and Computer Science
Münster University of Applied Sciences
Steinfurt, Germany
Email: {tuexen,i.ruengeler}@fh-muenster.de

Abstract—Programs that execute in parallel across a network often use the Message Passing Interface (MPI) library for communication. The network requirements of an MPI program are often unclear because of the difficulty in exploring alternative network configurations as well as obtaining packet level information about the communication. MPI-NeTSim is an execution environment to emulate MPI programs on simulated networks to allow users to better explore the impact of the network on the performance of MPI programs.

We describe the design of MPI-NeTSim and the integration of OMNeT++’s INET framework into MPICH2’s MPI middleware. We introduce a novel technique for uniformly slowing down the execution of the system to allow the discrete event network simulator to keep up with the execution and provide a consistent view of the communication. We validate our technique with synthetic programs as well as the standard NAS benchmarks. We demonstrate MPI-NeTSim’s usefulness in analyzing the effect of the network on communication by using our environment to study the impact of a slow-link on the NAS benchmarks.

I. INTRODUCTION

The purpose of our work is to explore the communication characteristics of MPI (Message Passing Interface [1]) programs at the network level. It is difficult to understand the impact of communication on the performance of an MPI program. The latency and bandwidth of a connection can vary dynamically over time, with the network load making it difficult to determine the communication overhead for an MPI application. Users typically have no control over the network and, unlike computational resources, users cannot vary network parameters to experimentally discover the limiting effect of communication on computation. Users also often do not have permissions to access data from the network interface adapter and even those with machine permissions often do not have access to data from network middleboxes. The solution we investigate is the integration of a network simulator into MPI.

We focus on programs that execute on small to medium sized clusters with commodity hardware over a TCP-like standard IP transport protocol. The latency associated with IP transport protocols is high and can easily affect performance. There is also a trade-off between bandwidth and protocol processing overhead, which on Gigabit Ethernet, can consume a significant amount of processing cycles and thus slow-down the execution of the MPI program. Yet, the ubiquity of IP

and the ease of deploying MPI programs in an IP network make this a popular alternative even with the performance challenges.

MPI-NeTSim is an emulation environment for MPI programs where the MPI program executes on the hardware but uses a network simulator to communicate between processes. We use the packet-level network discrete event simulator OMNeT++ [2] and its INET framework for simulating IP-based networks. In order for OMNeT++ to act as a communication device that could interface with an MPI implementation such as MPICH2, we created a socket-like API that enabled our own MPICH2 module to interact with the simulator. This allows MPI programs to compile and execute as before with the exception that now all communication is directed into the simulator.

The major challenges in interfacing to OMNeT++ were the simulator design itself as well as the mechanisms necessary for the MPI middleware to ensure the simulation accurately reflects an MPI execution. OMNeT++ was originally constructed as a stand-alone simulator where users could incorporate their own network modules for the simulation of protocols and network architectures. Tüxen et al. [3] added an external interface module to OMNeT++ to allow external network devices to interact with the simulated network devices inside OMNeT++. We extended the external interface design to allow for interprocess communication to an MPICH2 [4] channel device inside the MPI middleware using our socket-like API. In Section III, we describe the changes that were necessary to integrate OMNeT++ into MPI.

The second challenge was to ensure that the communication behavior of the MPI program remained the same in the simulated network as in this same network in reality. Maintaining the communication behavior of the program is possible as long as the simulator can keep up with the communication and match the real-time communication times of the original program. However, as a discrete event simulator, the amount of real-time needed to simulate the activities measured in virtual time depends on the workload of the simulator. Whenever the simulator is running slower than real-time, there are unintended delays which modify the communication behavior of the MPI program. Our solution to this problem is to uniformly slow-down the execution of the entire system to

ensure that the simulator is always able to keep up with a slower execution of the MPI program. In Section IV, we describe in further detail the communication behavior we are attempting to preserve and also the mechanisms added to MPI-NeTSim to implement slow-down. Section V describes how to calibrate and extract data from our tool. In Section VI, we experimentally measure and verify the accuracy of our approach. Finally, in Section VII, we describe limitations to our approach and possible future work to further extend MPI-NeTSim.

II. RELATED WORK

Our design incorporates elements of both emulation and simulation. On the one hand, with respect to the MPI program, the system mimics the execution of the real program except that communication is done via the simulator. On the other hand, with respect to the network, the network activities are simulated according to the models of the devices and protocols in the simulated network.

The simulator tool at the heart of MPI-NeTSim is OMNeT++, an open-source C++ discrete event simulator [2]. OMNeT++ is a modular, component-based simulator designed to be used to simulate large networks. Important for our purposes, OMNeT++’s INET framework has modules for TCP, UDP, SCTP, IP and Ethernet along with modules for routers, switches and hubs.

OMNeT++ has an external interface connection for incorporating live real-time traffic into the simulator. Developed by Rüngeler, Tüxen and Rathgeb [3], the external interface uses the packet capture library (libpcap) and raw sockets to receive and send packets, effectively functioning as a network interface adapter to nodes like routers and hosts in the simulator. We modified this external interface to provide a higher level API as described in Section III as well as changes to the module to support our slow-down mechanism described in Section IV.

Riesen [5] describes a hybrid approach which combines both emulation and simulation. Similar to MPI-NeTSim, Riesen’s system executes the MPI program as before and has the middleware send communication events to a discrete event simulator. The discrete event simulator only processes events, and unlike MPI-NeTSim, is not used as a communication device. The simulator is a custom one and does not simulate at the packet-level of IP transport protocols. The advantage to Riesen’s approach was the ability to scale to a larger number of MPI processes, but that is partially due to not performing a more detailed simulation of the network. Unlike our approach, Riesen’s event driven approach is based on a globally synchronized logical clock and thus reports execution times in terms of simulated time rather than real-time. Riesen’s approach more closely corresponds to a program-driven simulation rather than emulation.

There are a variety of tools and systems available for emulation. One common approach is to use dummynet [6] to add delay or reduce bandwidth between machines in the system. Because only the underlying system changes, this

approach requires no modifications to MPI, and for latency and bandwidth in particular, one can obtain accurate results. However configuring dummynet requires superuser privileges and, depending on the type of devices to be simulated, additional machines to emulate the network devices may be required. When dummynet is used, one has to use an appropriate kernel configuration to provide enough memory to store all the packets and to use an appropriate system clock granularity. This required building a custom kernel. Systems like Emulab [7] configure dummynet for you but this requires having access to a fully configured Emulab cluster. Our approach can be done completely at the user-level and does not require any special access or additional machines.

As a simulator, OMNeT++ also provides a rich environment for experimenting with different modules and network parameters. Although we have focused on OMNeT++’s INET framework, the interface module can be easily modified for use with other non-IP type networks such as InfiniBand or simpler fully connected point-to-point networks. The component-based design of the simulator simplifies the addition of new protocols and devices for use with the simulator.

There have been several projects over the years for the simulation of MPI programs. In [8] they describe a network simulator with similar goals to our own. The major difference is that they simulate only the network adapters and not the internals of the network. As well, the simulator does not use standard protocols but assumes a simple transport mechanism. There is work by [9] which exemplifies the approach of building a dedicated simulation environment for MPI. Again, this does not model standard transports and is a closed system with a fixed number of parameters used to set the properties of the underlying network.

There is work by Nunez et al. [10] that uses OMNeT++ to simulate MPI programs where the focus was on storage I/O requirements. They first gather trace files from an MPI program and use these traces to perform a trace-driven simulation of the execution. Unlike a trace or program-driven simulation, our intent is to emulate the execution and to provide a tool to experiment with the network and thus provide alternative executions of the MPI program with respect to varying network conditions and configurations.

III. INTEGRATION OF OMNET++ WITH MPI

The overall architecture of MPI-NeTSim is shown in Figure 1. The system consists of two parts: the MPI application and the simulator.

In the MPI part of Figure 1, there are one or more processes that are initiated and terminated by a process manager. Once each process has started, they invoke MPI routines that are executed by the MPI middleware, which is responsible for managing the communication between processes to ensure that messages are eventually sent and received. In MPICH2, the open source version of MPI used for MPI-NeTSim, the middleware separates out the communication-specific code into a “channel” module. We used our previous SCTP-based

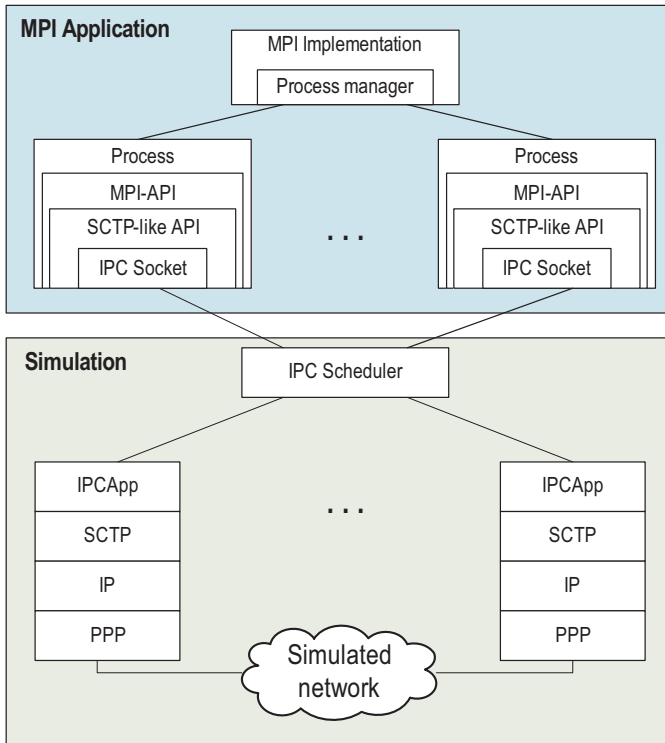


Fig. 1. Architecture of MPI-NeTSim

channel module as the basis of the MPI-NeTSim channel module [11]. Since the SCTP-based module already did message fragmenting and framing, the module was easy to adapt for use with IPC sockets to communicate with the simulator instead of with the kernel SCTP stack. In order to make use of the simulated SCTP stack, we created an SCTP-like API that interfaced with the OMNeT++ `IPCScheduler`. This is a minimal API that has similar semantics to the SCTP socket API.

One simplification in the design of the API is that we only need to specify our MPI rank in messages. All network information about the connection is part of the simulator and not the middleware. All information concerning IP addresses, ports and socket options for a connection are specified in the simulator configuration file¹. Because the network properties are specified in the configuration file, it is easy to change these properties for a program and to experiment with the different options. For example, one can experiment with Nagle’s algorithm to determine whether or not it affects the execution time of the program. In the end, because setting up and managing the connections were done by the simulator, the channel module for MPICH2 was relatively simple.

As shown in Figure 1, the MPI part communicates with the simulator via IPC sockets. Although there are many possible approaches, e.g. message passing, shared memory, we chose IPC sockets because they provide the standard socket API. The `IPCScheduler` module inside the simulator is the endpoint of the IPC from all the MPI processes. An advantage

¹A consequence of this decision is that we do not handle dynamic processes or dynamic changes to socket options.

of using IPC sockets is that the `IPCScheduler` can handle the sockets as normal file descriptors and use `select()`. By using `select()`, the `IPCScheduler` can bind to different kinds of sockets, e.g. IPC and network sockets to handle the call whenever an outside message arrives. This gives the flexibility for allowing our MPI processes locally on the same machine or remotely on another machine. In this paper, we focus on network simulations using one host machine, but in future work, both our design and model will be extended to simulate more distributively.

As shown in Figure 1, for every MPI process there is a corresponding `IPCAApp` module inside the simulator, which acts as an instance of a host. As messages from all MPI processes pass through the `IPCScheduler`, they are analyzed and converted into the appropriate network format. In our case, since we are using SCTP, the messages are converted into the simulator’s internal representation of an SCTP packet. Control messages, e.g. the closing of a socket, have to be converted into the corresponding SCTP commands. Data messages have to be encapsulated into SCTP DATA-chunks, and the source and destination address are adjusted as the messages are routed. Finally, before messages are inserted into the simulation queue, they are time-stamped with the current virtual time, which determines the message’s processing time within the simulation. On egress, the `IPCScheduler` collects all messages from the `IPCAApp`s, finds the corresponding MPI process, transforms the message into the right format and sends it to the middleware of the identified MPI process.

IV. MAINTAINING MESSAGE SYNCHRONY

The previous section described the overall design of the system and the integration of the simulator into the MPI middleware. As a result, all MPI communication is now directed into the simulator which then completes the transport level packet transfer from source to destination. However this does not address the question of when MPI messages are scheduled and the problem that real-time simulation delays can alter the communication behavior of the MPI program. For example, Figure 2 shows timelines for two processes A and B where A first sends a message to B and then B sends a message back to A. The middle bar is a timeline for the simulator, which receives the message from the MPI middleware, simulates the transfer of the message, along with all other network traffic, from ingress to egress, and then finally sending the message to the MPI middleware at the destination. We are using blocking MPI communications, and the grey boxes represent process idle time. Figure 2(a) shows the idealized execution we are trying to reproduce, where the simulator takes exactly 1 time unit of real-time to complete the transfer of the message. In Figure 2(b), when the simulator needs more than one unit of real-time, 2 units at $t = 2$ and 4 units at $t = 5$, then Process A must wait for the message at time $t = 7$. With respect to the MPI middleware, the added delay has changed an expected/posted message in 2(a) to an unexpected message as shown in 2(b). We want to ensure consistent behavior where every execution of the

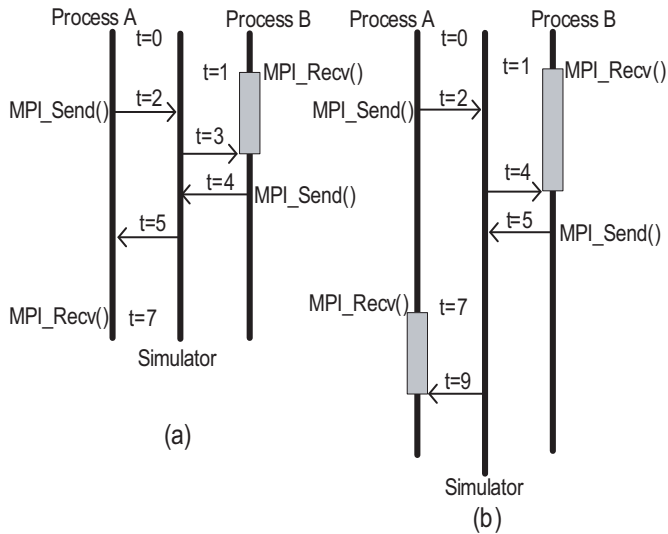


Fig. 2. The effect of the simulator on program execution time; (a) ideal case where each communication is one unit of real-time, (b) case when the simulator needs more than one time-unit

program exhibits the same communication behavior and is not dependent on the real-time execution speed of the simulator.

A. Concept of Slow-down

We have designed and implemented a novel technique that ensures the synchrony between messages entering and leaving the simulator is maintained. Our solution is to introduce a *slow-down time factor*, which we simply call the *time factor*, to uniformly slow down the execution of the MPI processes and the simulator, in order to give the simulator more time to complete its work.

Figure 3 shows the result of executing the program from Figure 2, except now process A and B execute four times

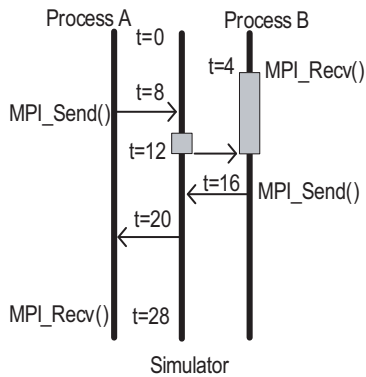


Fig. 3. The effect of the simulator on program execution time; slow-down solution

slower than before. The MPI program now takes 4 times longer to execute but is now identical to Figure 2 with respect to the slowed execution. Prior to $t = 12$, the simulator needs to sleep for two time units to keep to the slower time schedule; for the second communication, the simulation has sufficient time with respect to the slower time schedule. There is just

a factor of 4 difference between the two schedules of events. The result is that the behavior of Figure 3 is identical to that of Figure 2(a) as long as our time factor provides sufficient time for the simulator to keep up in the slower execution.

Given a program, we define *scaled real-time* with respect to a time factor s at time t , $SRT_s(t)$, to equal t divided by s . Now given two executions of the program with time factors $s_2 \geq s_1 \geq 1$ with execution times t_1 and t_2 , then $SRT_{s_1}(t_1) = SRT_{s_2}(t_2)$. In the previous example $s = 4$ and we have that $SRT_4(28) = 7$, the runtime of the original program. To maintain message synchrony we want to also ensure that communication events occur at the same scaled real-time. For all communication events c_i , $SRT_{s_1}(t_1) = SRT_{s_2}(t_2)$, where now t_1 is the real-time c_i occurs in the first execution and t_2 is the real-time the event occurs in the second execution.²

In practice, it will almost always be the case that the simulator cannot keep up to the real-time execution of the program and we are unable to measure $SRT_1(t)$ directly. However, for sufficiently large slow-down factor S , for all $s \geq S$, we have $SRT_s(t_1) = SRT_S(t_2)$ where t_1 and t_2 are the executions times of the two runs of the program. As well, in practice, there will be some error in the mechanism and $SRT_s(t_1)$ will not exactly equal $SRT_S(t_2)$. In Section VI we test for the existence of S as well as measuring the amount of error with respect to message synchrony.

All that remains is to implement mechanisms to produce the desired slowdown. We describe these extensions for appropriately slowing down the execution in the remainder of this section.

B. Implementation of Slow-down

In-between communication calls, MPI programs typically do some computation. In our implementation, all communication uses our SCTP-like API so in order to uniformly slow down the computation of an MPI program, we use a timer to obtain timestamps for every communication event. The timer is started when the SCTP-like socket is created in `MPI_Init()`. The last timestamp is obtained from `MPI_Finalize()` when the socket is closed. For a `send()` or `recv()`, one timestamp is obtained at the beginning of the communication call and one at the end. When a communication call is entered, after the timestamp is obtained, the difference between this timestamp and the previous one is the computation time that occurred between communications. When the time factor value is greater than one, our slow-down mechanism is enabled. We uniformly increase the computation time by calling `usleep()` to sleep t microseconds where

$$t = (\text{timeFactor} - 1) \times \text{computationTime}.$$

The previous mechanism slows down the computational parts of the program by the time factor value, but we still need to

²Our slow-down mechanism cannot maintain message synchrony for external sources of non-determinism such as the program reading the value of the real-time clock and reacting to it.

uniformly slow down the communication inside the simulator by the same amount.

The simulator uses the following method to slow-down the simulated communication. When the simulation starts, it stores the current real time and the current simulation time as a base for the computation. The simulation then computes the simulation time of the next queued event. It calculates the time difference to the stored simulation time base, multiplies this by the time factor and adds this to the stored real time base. Then it executes the action required by the event at that point of real time. However, if there are any IPC messages received before this point of time, they are executed as soon as they are received and additional events are scheduled. After that, the time of the next scheduled event is recalculated as described.

In the current system, a constant time factor values is used by the middleware and the simulator. The value of the slow-down time factor is set as an external parameter to the system. By appropriately setting the value for different executions we experimentally determine threshold S . At that point, we know we have sufficiently slowed-down the the execution of the system to ensure that we have a consistent view of the MPI communication.

V. CALIBRATION AND DATA EXTRACTION

As discussed in Section IV, to ensure that the non-deterministic behavior of the MPI program is not altered by the amount of work performed by the simulator, we need to find the time factor that gives the simulator sufficient time to complete all of its operations. We determine the slow-down experimentally by executing the MPI program with the simulator for various slow-down time factors in order to determine threshold S . We determine S by examining the `maxBehind`, a variable kept in the simulator that reports the maximum number of time units the simulator is behind. The simulator is always behind by a small amount due to potential OS scheduling delays and the time it takes to execute the current event. If the `maxBehind` value becomes larger than the system specific threshold S , then this is an indication of the amount of excessive delay caused by the simulator, in which case the slow-down factor is increased and the MPI program is executed again. This may have to be repeated a few more times since it may be the case that changes to the slow-down factor affects the schedule of MPI messages to the simulator thereby affecting the `maxBehind`. However, for sufficiently larger slow-down, the `maxBehind` is close to constant, and the behavior of the simulated communication converges. The objective is to find the smallest slow-down factor to ensure that `maxBehind` is constant so that we take the least amount of real time to accurately execute our slowed MPI program in our simulated network. In the next Section VI, we validate our emulation by experimenting with a variety of tests to show that we reach a constant state after which communication does not change.

A. Configuration

A great advantage of using a simulator is the possibility to configure a variety of different parameters and to gather information for statistical analysis. The parameters configurable in the simulator are related to modules that are combined to form a network. Routers, queues, protocols, applications, and channels have their own parameters. In general, a default value is set, so that only the changes relevant for the special case have to be performed. The most important parameter which can be controlled are the delays of the links, the available bandwidth, packet error rate, queue sizes of routers and hosts, queuing disciplines, and also several SCTP-specific parameters, controlling the timeouts in the protocol and receive buffers.

B. Gathering and Analyzing Data

To gather statistical data, two types of files are provided by OMNeT++. The scalar files store accumulated data, like the number of packet drops, retransmissions, or acknowledged data. In the vector files, specified values are written with a timestamp. So, for instance, the update of the congestion window can be followed over time. Both types of files are generated for each run, i.e. if a series of runs is performed, the results are stored separately.

Another possibility to follow the course of the communication are pcap network trace files, that are usually captured from the network adapter. We integrated dump modules in the host and the router modules and placed them between the link layer and the network layer. When a packet passes through the dumper, it is analyzed by a so-called serialiser and transformed from the simulation format to the network format. Then a pcap header is added, and the packet is stored in a pcap file, which can afterwards be analyzed by any tool understanding the pcap format. Thus, not only the traffic on any host, but also on routers can be captured. These capture files can be used by packet analyzers like Wireshark. Obtaining complete traces is difficult in real environments especially for high speed interfaces, but using our set-up, we are able to obtain these traces and statistics (without administrator privileges) from the networking of compute nodes as well as routers or even middleboxes.

VI. VALIDATION

In this section, we validate the behavior of MPI-NeTSim using controlled MPI programs of our own design as well as the NAS parallel benchmarks (NPBs). In order to validate the system for all of the programs we tested, we experimentally determine if convergence of all measurements occurred when increasing our time factor towards some threshold, as defined in Section IV. Measurements that should converge include the number of unexpected messages, the simulated runtimes, as well as the maximum time our simulation falls behind (`maxBehind`) when simulating a given event; this determines the simulation's accuracy with respect to message synchrony. After introducing the validation process in Section VI-A, in Section VI-B we begin system validation with several

programs we created where it is easy to control communication parameters such as the message intensity or the number of unexpected messages. In Section VI-C, we calibrate and measure the accuracy of MPI-NeTSim for the NPBs, real programs with varying communications. Finally, as an example of using MPI-NeTSim to vary the network, in Section VI-D we measure the performance sensitivity of the NPBs to a heterogeneous network where there is a slow link.

All experiments were performed on an 8-core Mac Pro with the simulator connected to four or less MPI processes, so as not to over-subscribe any core with more than one process and thereby minimizing scheduling conflicts. As the initial reference network, we choose the network configuration shown in Figure 4, a single 1Gbps switch with 1 millisecond latency. Later in Section VI-C, we modify this network and compare the performance of the NPBs in the two networks.

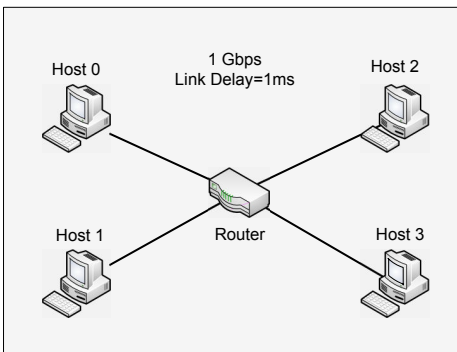


Fig. 4. Simulated Reference Network

A. The Validation Process

In order to validate our slow-down mechanism for any program, we executed it with a variety of time factors. In order to ensure that the runs accurately converge to a consistent behavior, we measure the maximum time our simulation falls behind when simulating a given event. This `maxBehind` value is measured in scaled real-time because that is the clock at which the MPI processes execute. The idea is that as the time factor increases, this gives the simulator enough real-time to process all of the events. As the time factor increases, the `maxBehind` value approaches zero. Ideally, when the `maxBehind` value is zero, then the communication behavior of the program remains fixed for all larger time factors. In practice however, `maxBehind` converges to a fixed value which is dependent on the real-time it takes to execute a single event in the simulator for that program.

We investigated the convergence of other measurements: the simulated runtime and the number of unexpected messages. The time factor for `maxBehind` needed to be scaled the most compared to the other measurements. In this sense, ensuring message synchrony is the most stringent condition, requiring the largest time factor value.

B. Synthetic Programs

Initially we tested the system with a couple of simple MPI programs with parameters to allow us to control their

communications.

First, we implemented a ping-pong program where two processes pass messages back and forth. We implemented this program giving users a set of parameters to specify the number of message exchanges, a compute time to introduce delay between interactions, and the message size. These parameters made it possible to experiment over a range of runtimes and different message rates. In order to stress the system, we ran ping-pong for 10,000 iterations with no compute time between iterations using 32 bytes messages (short MPI messages).

With a low time factor, our simulator’s `maxBehind` was as high as 0.4 seconds for this communication intensive application, indicating that we needed to increase the time factor so the simulator could keep up. We ran the same test up until the time factor was 128, a simulation that took 86 minutes in real-time. The number of seconds behind continued to decrease and eventually converged to a fixed value slightly above zero.

Second, we implemented a program with non-blocking receives and sends to introduce as much non-deterministic behavior as possible with respect to message delivery. We measured the number of expected versus unexpected messages in the MPI middleware to test for the type of inconsistent behavior previously shown in Figure 2. For a sufficiently large time factor the number of expected and unexpected messages should remain the same.

For all of the parameter settings that we tested, we identified the threshold after which larger slow-downs did not change the communication behavior of our test program. We tested the program with a network where the latency of a network was less than the time between receive calls. For this network all messages should be unexpected. In this case, the system converged to have all unexpected messages (i.e., no idle time on the receive side). As we increased the time between receive calls the system converged more quickly. This is because by increasing the time between receive calls, the simulator is better able to keep up with the communication. In similar fashion, the opposite test converged to have all expected messages when the network latency was greater than the time between receive calls. Overall, this test shows that at the MPI level, the non-deterministic behavior of the communication converges to a consistent behavior for slow-downs above a given threshold.

C. NAS Benchmark Programs

We now use MPI-NeTSim to investigate the NAS Parallel Benchmarks. The NPBs are commonly chosen as benchmarks to test the capabilities of clusters and interconnects in the parallel and distributed computing community. In this respect, the ability to execute the NPBs unchanged demonstrated the robustness of our complete system. We used the NPB programs for our tests since they use realistic MPI test programs and use a variety of both point-to-point and collective operations. For 6 of the NPBs, we have chosen to run the second data class W due to time constraints. There are 9 NPBs and we chose these 6 because they could compile and run

within a comparable amount of time with 4 processes for the W class. We use the validation process that was outlined in VI-A.

The accuracy results of our slow-down mechanism for the NPBs are shown in Figure 5 for several time factors. Figure

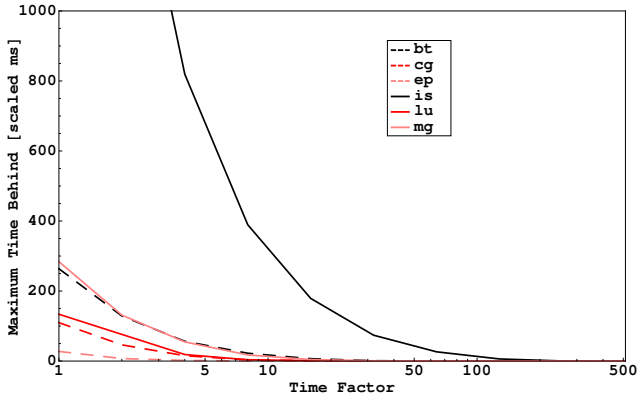


Fig. 5. Accuracy of MPI-NeTSim

5 shows that as the time factor increases, the `maxBehind` value approaches zero, indicating an accurate and consistent result.

The runtime measurements for the NPBs are shown in Figure 6. It should be noted that each of the 6 NPBs we ran converged. The threshold is shown graphically as the leftmost/smallest time factor value at which a given runtime first arrives near to an asymptote. As mentioned before, the convergence of `maxBehind` is a much stronger requirement than may be necessary for another measurement; this is exactly the case for the runtime measurements of the NPBs. This is shown in Figure 6 by noting that the convergence occurs for runtime at smaller time factors than the same program converges for `maxBehind` in Figure 5.

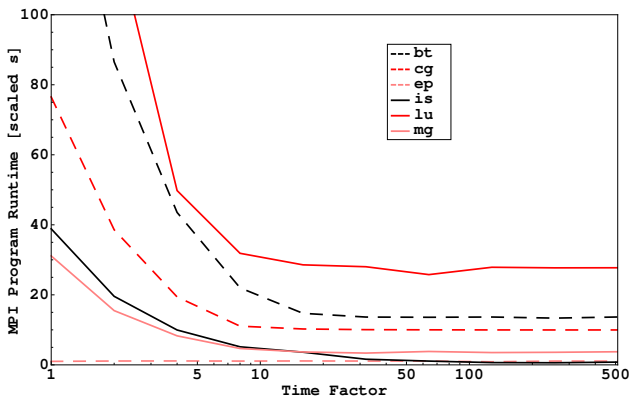


Fig. 6. NAS Runtime Results

Runs with larger time factors produced consistent results but naturally takes more real-time to run since our mechanism is slowing them down more. In order to obtain an accurate result for a given application, the slow-down factor required by the simulator depends on the most communication-intensive part

of the application. However, applications tend to have phases of communication and computation, so the simulator will have relatively little work to do in communication phases. Future work will investigate techniques to enable more dynamic control of the slow-down factor, in order to minimize the real-time of a simulation run, yet keeping the results accurate.

D. Varying the Network

We wanted to demonstrate the ease to which MPI-NeTSim could be used in order to test the effects of network changes on a set of applications. To do this, we modified our reference network used above to have the host executing rank 0 to have less latency (100Mbps) than the other nodes and also have additional latency (an extra 1ms hop). Our modified network is shown in Figure 7. Although this is a simple illustration,

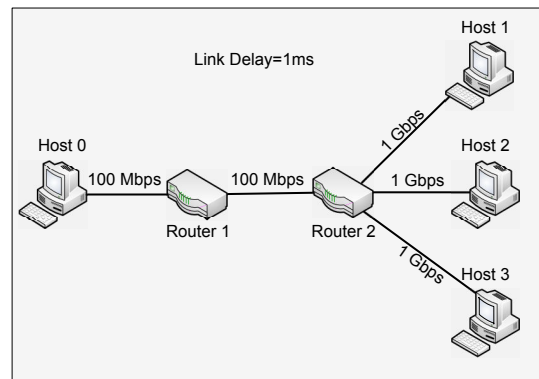


Fig. 7. Placing a Bottleneck on Rank 0

MPI-NeTSim allows one to toggle the network topologies and parameters to see the effects they have on an MPI program.

Similar to our reference network, the runtimes of the NPBs converged as the time factor increased towards the threshold. Figure 8 shows the percent slow-down caused in this new network setup compared to the reference network.

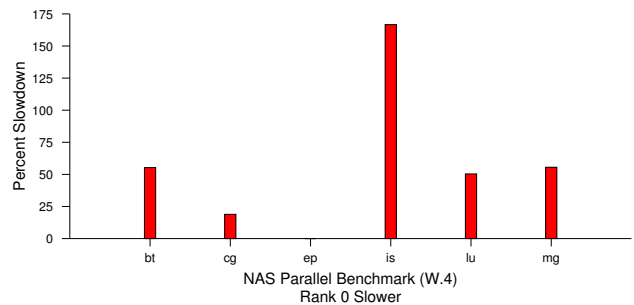


Fig. 8. Slow-down (Reference Network versus Slow Rank 0)

The runtimes of the six NAS benchmarks studied each were affected differently by the slow communication link of rank 0. As Figure 8 shows, for the `ep` benchmark, modifying the network had little effect. This is intuitive because the `ep` test has an initial work distribution phase, a substantially

larger work computation phase, and ends with a quick answer distribution phase. Very little communication occurs so rank 0 having less bandwidth and higher latency is irrelevant to the total runtime.

For the other five NAS benchmarks evaluated, they fall into one of three ranges. The “is” benchmark was affected the most, slowing down its execution by 166%. The second range slowed the benchmark by about 50%; bt, lu, and mg fell into this category. In the third range was cg, which was slowed down by 19%.

The top three ranges pictured in Figure 8 directly correspond to the type and intensity of the collective operations performed by the programs[12]. The collectives can also be categorized into three categories with respect to network requirements. Not coincidentally, “is” uses the most intensive, the second most intensive is used by the middle-range benchmarks (bt, lu, and mg) and cg uses the least intensive collective.

At the upper extreme, there is the all-to-all operation, where all of the ranks pass a piece of data to each of the other ranks. The next most intense category of collectives are those like broadcast or reduce; these are less intense as they are implemented by passing the data from one root node to all nodes by using a tree communication pattern. For the least intensive collective category, which is a barrier, the “data” is all header and no payload and it is only used for synchronization between processes. Barriers are also implemented using a tree but has less payload so it is the least intense with respect to the network. Each collective operation depends on the performance of all of the ranks, and in this case, would be mostly be affected by the slow link of rank 0.

At the extreme, the “is” benchmark was affected the most because it has the highest volume of collective communication of all of the benchmarks. It is the only benchmark that uses all-to-all communication, the most network intensive; it also uses other types of collectives as well as to point-to-point communications.

As Figure 8 indicates, bt, lu, and mg all behave similarly. This is because they all use collectives like broadcast and reduce to pass a payload around to all ranks including rank 0. The final range consists of cg, which is different because with the exception of one barrier call (the least network-intensive collective), it does point-to-point communications mostly, so its communication pattern is least affected by the slow link of rank 0, with the exception of ep, which does almost no communication at all.

VII. CONCLUSIONS

We have introduced MPI-NeTSim, an emulation environment for MPI programs that integrates a packet-level network simulator as a communication device into the MPI middleware. The simulator allows us to extract low-level information about the network and its impact on communication. As a network simulator, we are able to extract information about network load to understand the interaction the network and lower level network protocols on performance. As part of our work, we designed and implemented an external interface to

the OMNeT++ simulator and an MPICH2 module to allow connections between running MPI processes via the OMNeT++ simulator. To maintain the communication behavior of the program, we introduced a novel slow-down technique to uniformly slow down the execution to ensure that the simulator can keep up with the program.

Future work will include extending our current interface to the simulator to scale to larger number of processes by allowing the MPI processes to execute on different processors. There is also the possibility of using existing work to execute OMNeT++ in parallel to reduce the simulation times. We would also need to extend our slow-down technique to account for the network overheads in communicating with the simulator. Another optimization with regard to our slow-down mechanism is to dynamically adjust the slow-down factor depending on the network load. The communication requirements of the MPI program may vary during the phases of its execution where there may be the opportunity to dynamically adjust the slow-down. By dynamically adjusting the time-factor for slowing down the execution, it may be possible to more quickly emulate the program.

REFERENCES

- [1] W. Gropp, E. Lusk, N. Doss and A. Skjellum, “High-performance, portable implementation of the MPI message passing interface standard,” *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sept. 1996.
- [2] A. Varga, “The OMNeT++ Discrete Event Simulation System,” in *In the Proceedings of the European Simulation Multiconference*, 2001.
- [3] M. Tüxen, I. Rüngeler, and E. P. Rathgeb, “Interface connecting the INET simulation framework with the real world,” in *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 1–6.
- [4] “MPICH2 homepage.” [Online]. Available: <http://www-unix.mcs.anl.gov/mpi/mpich/>
- [5] R. Riesen, “A hybrid MPI simulator,” in *Cluster Computing, 2006 IEEE International Conference on*, Sept. 2006, pp. 1–9.
- [6] L. Rizzo, “Dummynet: a simple approach to the evaluation of network protocols,” *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.
- [7] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An integrated experimental environment for distributed systems and networks,” in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*. Boston, MA: USENIX Association, Dec. 2002, pp. 255–270.
- [8] M. Uysal, A. Acharya, R. Bennett, and J. Saltz, “A customizable simulator for workstation networks,” in *Proceedings of the International Parallel Processing Symposium*, 1996, pp. 249–254.
- [9] S. Prakash and R. Bagrodia, “MPI-SIM: using parallel simulation to evaluate MPI programs,” vol. 1, Dec 1998, pp. 467–474 vol.1.
- [10] A. Nunez, J. Fernandez, J. Garcia, and J. Carretero, “New techniques for simulating high performance mpi applications on large storage networks,” 29 2008-Oct. 1 2008, pp. 444–452.
- [11] H. Kamal, B. Penoff, and A. Wagner, “SCTP versus TCP for MPI,” in *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005.
- [12] A. Faraj and X. Yuan, “Communication characteristics in the NAS parallel benchmarks,” in *PDCS*, 2002, pp. 724–729.