A New Fast Algorithm for Connecting the INET Simulation Framework to Applications in Real-time

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

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

ABSTRACT

The coupling of real implementations with simulations helps the developer to better understand the behavior of the real system by shifting to the simulation the parts of the overall system that he cannot influence. This was done for MPI programs in MPI-NeTSim, where the network and transport protocol are simulated in OMNeT++ but the original MPI code executes unchanged. We previously developed a static time factoring algorithm to accurately handle the discrepancies between the wall clock time of the application and the virtual simulation time. However, the original algorithm proved to be too slow for simulations with a higher number of hosts and a greater amount of data. In this paper, we introduce a faster adaptive algorithm that no longer uses time factors nor requires a full system-wide synchronization but still meets the conditions of the ordering of events in a distributed system. The new algorithm scales to significantly more hosts and data.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modelling**]: Model Validation and Analysis; I.6.6 [**Simulation and Modelling**]: Simulation Output Analysis; D.4.4 [**Operating Systems**]: Communications Management

General Terms

Network simulation, Adaptive time synchronization

Keywords

OMNeT++, MPI, Real-time scheduler, software-in-the-loop

1. INTRODUCTION

Simulations have the great advantage that experiments can be performed without the complexity and unpredictable behavior of real implementations. However, simulating the entire system is not always practical or feasible because of the difficulty in accurately modelling the system or because of the time needed to simulate large systems. There are also many examples of hardware, software, or human-in-the-loop type simulations where a simulator is but one component of the system. These cases require crossing the border between the simulated part of the system and the real world, in order to connect both of them together.

One area that can take advantage of the ability to connect a simulator with software executing in real-time is networking. Distributed and parallel computations are commonplace today and the performance of these systems often depends on the underlying network. This makes it difficult to understand the effect the network has on the performance of a particular system. This is important in cloud and cluster applications where decisions about the technology or scale of the network need to be made. Real networks are very cost intensive, especially when the number of hosts grows. Therefore, before buying a new cluster, it is worthwhile finding out whether the new setup will meet the performance constraints. Is it, for instance, better to invest in a technology with a very high bandwidth, or should the emphasis lie on the latency? How should the hosts be connected? Is a Tree topology with fairly cheap switches possible, or is a fully connected Mesh network necessary? Inside a cloud, what type of network resources are best to run the system?

We have studied the use of a network simulator in combination with the real-time execution of the software in the system. The network simulator makes it possible to alter the network parameters and experiment with different network topologies to evaluate possible trade-offs in the network. By simulating only the network, we are able to make minimal changes to the application and allow it to execute as before. Also, since in many systems users have no control over the network, our approach allows users to experiment with the network without the need for any special user privileges.

Our MPI-NeTSim system was introduced in [6]; it uses the INET framework [11] of the open-source OMNeT++ discrete-event simulator [12]. It was developed for the transport of real MPI (Message Passing Interface) messages over a network stack. As one of our research areas is transport protocols, especially SCTP (Stream Control Transmission Protocol) [8], we also examined the influence of the transport protocol parameters on the performance of MPI. MPI processes in MPI-NeTSim use not only simulated networks but also make use of the SCTP implementation inside of the simulator. Although MPI-NeTSim was designed for MPI over SCTP, it can easily be adapted to handle arbitrary applications over other transport protocols like TCP or UDP.

During our subsequent work with MPI-NeTSim, we increased the number of hosts and the amount of data transported. Although the time synchronization algorithm described in [6] worked perfectly, as we experimented with larger systems it became apparent that a faster algorithm was needed to perform the simulation in less time. It was the time to perform the simulation, not the simulated time, that was the problem; we needed to significantly reduce the simulation time to obtain results in a timely manner.

In this paper, we introduce our new adaptive algorithm. This approach uses a local synchronization strategy together with Lamport's time-stamping technique in order to synchronize the outside real-time with the simulated time and maintain the correct ordering of messages. In Section 2, related work is discussed followed in Section 3 by a short overview of the architecture of our application interface. In Section 4, the original synchronization algorithm is outlined describing the general problem of synchronizing with different time sources. Section 5 presents our new algorithm. Necessary mechanisms for sending and receiving messages are described together with the important notion of the base times and the ordering of events. In Section 6, some simulation results are shown to validate our new approach on real executions. In Section 7, the performance of the new algorithm is compared with our former algorithm. Conclusions and future work are given in Section 8.

2. RELATED SIMULATION WORK

There have been several projects over the years for the simulation of MPI programs[10, 6, 7]; these and others are described in more detail in our original paper[5]. In general, while these projects simulate MPI executions, they present custom solutions and do not model standard transport protocols. To contrast, our system is available for others to use as it extends the open-source OMNeT++ simulator[12]; it also provides packet level simulation of the full SCTP transport protocol in addition to the intermediate network.

There have been several related projects that use time dilation techniques together with virtual machines (VMs). Similar to our original algorithm in [5], Gupta's DieCast system [2] statically scales back virtual time in order to simulate large scale executions of VMs. In [1], an adaptive approach for VMs modifies the time factor throughout a bursty execution in order to avoid idle time, as is the case with the new algorithm presented here in this paper. The major difference between time dilation approaches and our new algorithm is that our algorithm is simpler because it is not a full system-wide synchronization but instead it leverages local synchronizations with Lamport's time-stamping technique [3]. Another difference is that these VM-based approaches simulate only the network with custom network solutions and the transport is run within the VM, whereas our MPI-NeTSim system simulates the network as well as the transport protocol inside the OMNeT++ simulator[12].

3. APPLICATION INTERFACE OVERVIEW

There are several aspects that need to be addressed in connecting a simulator to the real-world. The first one is the interface, i.e. the mechanism to send data from the real world to the simulation and vice versa. The second one is the

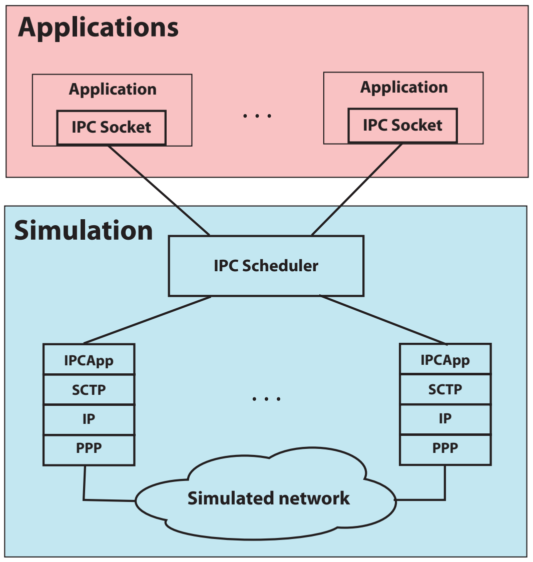


Figure 1: Architecture of the application interface

problem of synchronizing the different time measurements in the simulation and the real world. We chose IPC (Interprocess Communication) sockets to handle the data exchange between the two realms. Figure 1 shows the architecture of our approach. The system consists of two parts: the application and the simulator. From the application's point of view, the simulator is simply a communication device for transporting messages to other applications, whereas from the simulator's point of view, the applications are a collection of independent external processes. Each application is equipped with an IPC socket; these sockets correspond to the equivalent sockets inside the IPC Scheduler. The IPC Scheduler is the central unit of the architecture. The scheduler terminates the simulation, works as an interface, distributes the messages to the hosts and the simulation, and controls the processing of the simulation event queue. The API between the application and the simulation is kept very simple in that the only functions necessary are sendToSim() and receiveFromSim() calls. With the sending of the first message, a connection to the remote host is set up, and details like ports, addresses, and MPI ranks are stored in a table to make the look-up of subsequent transmission parameters easier. MPI calls can be blocking or non-blocking. In either case a return code is received informing about success or failure of the operation. In MPI, both types of calls are used, blocking calls for messages that are so important that the program cannot go on without them, and non-blocking calls when the program can postpone the call and come back to it later. Underlying, we decided to use only blocking IPC sockets for our new algorithm which we could tailor to model the time synchronization of both blocking and non-blocking calls; this is described in detail in Section 5.

4. SYNCHRONIZING EVENTS USING A TIME FACTOR

OMNeT++ is a discrete event simulator where events are executed in time order with respect to their start time. The time inside the simulator is independent from the real-time it takes to simulate the actions associated with the event. In the case of a communication model like INET, the simulation time is only advanced when messages are moved from one module to another according to the configured link parameters (bandwidth, latency). The time spent in the modules that correspond to the processing of data in real systems and, thus, the CPU time, is not taken into account. The consequence is, that the real time advances much faster than the simulation time.

Our first approach was based on the idea to slow down the application to allow sufficient time for the simulator to perform its work.

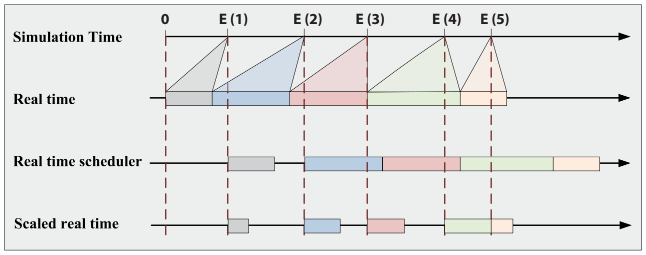


Figure 2: Time Factor

The first timeline in Figure 2 shows a sequence of five events executed in simulation time order. The colored shapes extending from each event on the first line to the second line symbolize the real-time computation needed to simulate the actions for each event. Other than the order of the events, there is no relationship between the duration of events in simulated time and the real-time needed to simulate the events. Thus, depending on the number and execution time of events, the ratio of the simulated time to the real-time can be less or greater than 1.

If the simulation time has to be adjusted to the real-time, things change. The third timeline in Figure 2 depicts the situation where the ”standard” event scheduler is substituted by a real-time scheduler. The real-time scheduler tries to synchronize the virtual and real-time clocks and now, with both clocks starting from 0, events are executed according to their virtual clock start-time. This is necessary when real network devices are connected to the simulation like in [9], and real-time events have to be included in the virtual time simulation queue. As shown in the third timeline of Figure 2, E (1) starts on time and finishes before the start of E (2), and thus E (2) can begin on time as indicated by the dashed vertical lines. However, E (2) takes too long and delays the start of E (3), which in turn delays the start of E (4) and E (5). These delays may alter the behavior of the external processes connected to the simulator because their behavior may be time sensitive.

The third timeline shows the situation when we started running tests. The time that the simulation was “behind” accumulated with the increased event density. Our solution to this problem was to introduce a time factor that resulted in the scaling of the time. The fourth timeline of Figure 2 illustrates the idea to scale both the times of the simulator and the application processes such that one action had finished before the next event occurred. In the fourth timeline, the time factor is two, indicating that real-time has been slowed down by a factor of two where the duration of an event in the fourth timeline is now half the duration of the event in the third timeline. By slowing down the real-time by a factor of two, all events are now executed on time.

Although this approach is accurate, it has the great disadvantage that its execution in real-time is very slow. With a higher number of hosts or a greater amount of traffic, the real execution times of the events increase, such that higher time factors are needed to prevent the simulation from being “behind”. Often, time factors of 500 or more were applied until the run time of the simulation converged, which means that an experiment that would need 1 minute in real-time, now took more than eight hours.

5. NEW FAST ALGORITHM

The time factor was applied to all events whether they needed it or not as is shown by the constant slopes in the left part of Figure 3. We wanted our new algorithm to include a time management mechanism that was not universally applied to all events, but adapted to the duration of its real processing times. That meant that when the density of the events (i.e., number of events in a given time unit) was high, the application had to be slowed down - equivalent to a high time factor - whereas at other times a time factor of one could be applied, meaning that applications and the simulation were synchronized. This stepwise linear behavior is shown in the right part of Figure 3. If the application has to wait, only the real time advances as is shown by a step, while at the sloped sections both times advance simultaneously. To achieve this event dependent behavior, the sendToSim() and receiveFromSim() calls were divided into two parts. A sendToSim() call consists of the actual sending of the data and the reception of a notification which mimics the return code. In the receiveFromSim() call, the application sends a notification and waits for the data in a subsequent receive() call. The time between those two system calls is comparable to the ‘sleep time’ of the former algorithm. Looking at the start and stop times, we could still calculate a time factor greater than one, but this locally calculated dynamic time factor should at any given time be less than or equal to the static time factor required in our old algorithm.

5.1 Introducing base times

To synchronize the times at the step sections, we introduced two base times, one for the simulation and one for the real-time. One task of the simulator is to compute the time that a message arriving from the application has to be executed and insert this message into the simulation event queue. The insertion time depends on the real-time at which the message arrives and the corresponding simulation time.

In Figure 4, the first timelines of Figure 2 are repeated to show the events and their duration in real-time. The last two timelines describe the adjustment of the simulation base time T_{Sim}^{Base} and the real base time T_{Real}^{Base} depending on the processing time of the events. At the beginning of the simulation, $T_{Sim}^{Base}(0)$ and $T_{Real}^{Base}(0)$ are set to the then current simulation and real-time, respectively. As the experiment proceeds, the simulation cannot keep up with the real-time, i.e. it gets “behind”. To prevent the events from piling up, we adjust the base times. $T_{Sim}^{Base}(1)$ is set to the simulation time at event E (2) and $T_{Real}^{Base}(1)$ is set to the real-time at the end of the execution of E (2), since this is the time when we realize that we are “behind”. As a consequence, the following event can be processed without disturbance, until the density of the events requires another adjustment of the base times.

Sending a large amount of data over links with a high

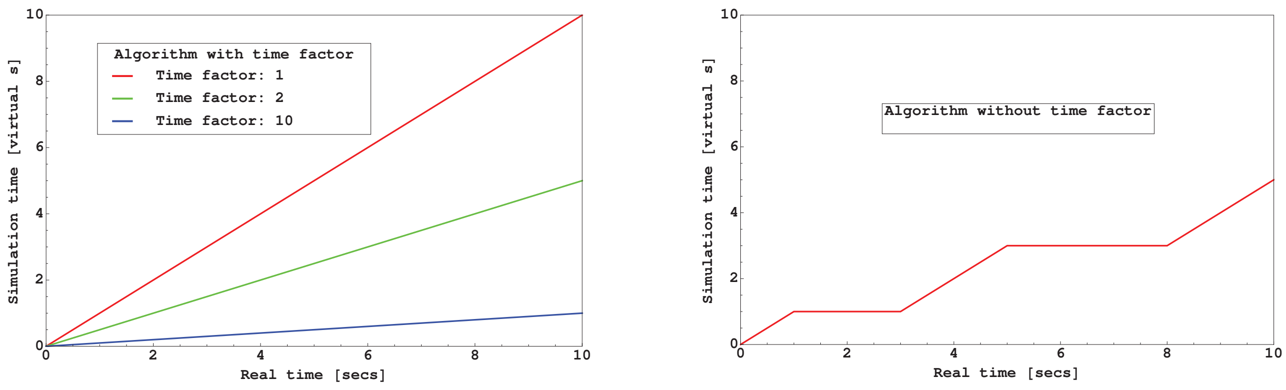


Figure 3: Runtime with and without time factor

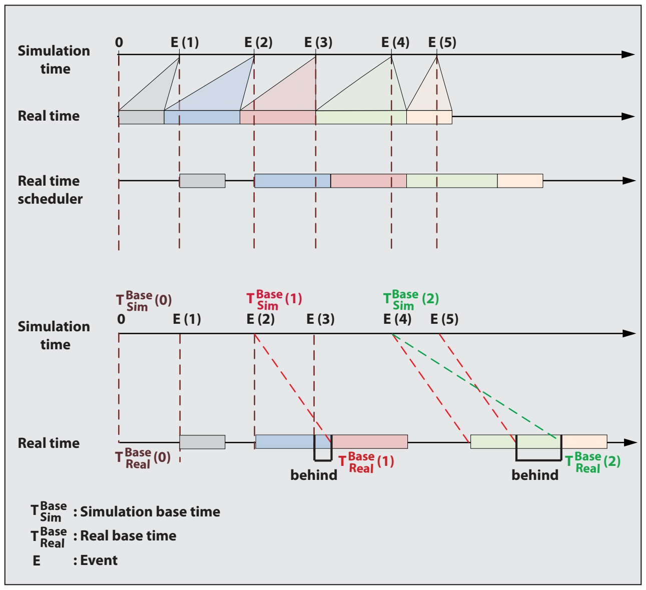
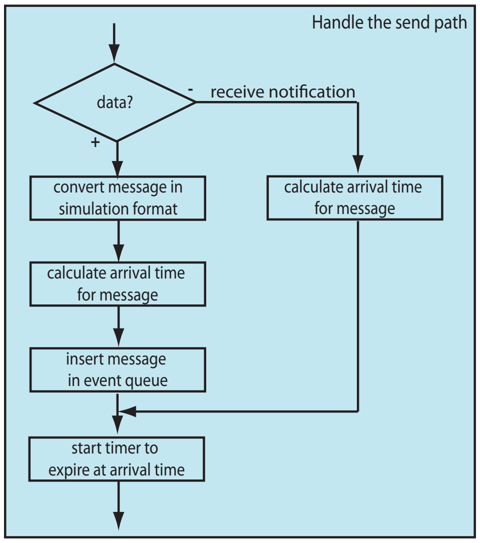


Figure 4: Synchronizing time

bandwidth and a low latency leads to a very high density of events. As a result, the base times have to be adjusted frequently. To adjust the real base time, the function *gettimeofday()* is used to ask for the actual time. Its granularity is microseconds, while OMNeT++ works with a higher resolution. To take advantage of this possibility, we introduced high resolution timers that helped us to achieve the finer granularity of nanoseconds, and thus, a more accurate behavior.

5.2 Handling messages from the application

Besides the correct adjustment of the base times, the handling of the messages also made new demands on the algorithm. Without any time factor, the application was too fast, so that the resulting simulation time was much higher than before. The solution was to let the application wait until the simulation was ready to process the data that it had received from the application. Therefore, in our approach, the application sends a message with a timestamp and subsequently waits for the arrival of a notification indicating that the simulation has injected the message to the simulated network. The message is handed to the IPCApp, the middle-ware between the transport layer and the IPC Scheduler, which functions as the application layer for the transport layer protocol inside the simulation (see Figure 1). After the receipt of this notification, the application can proceed with its tasks.



Handle the send path
data?
-
receive notification
+
convert message in simulation format
calculate arrival time for message
calculate arrival time for message
insert message in event queue
start timer to expire at arrival time

Figure 5: Application to Simulation

Figure 5 shows the flow chart of the send path as implemented in the IPC Scheduler. The send path has to distinguish between messages carrying a payload or receive notifications. The importance of the receive notifications will be explained in Subsection 5.3. The data messages are converted from the real application format to one known by the simulation. In the case of complex formats, this is done by a serializer, like for instance the IPSerializer or SCTPSerializer, which was introduced as an external interface to connect real-world computers to INET [9].

$$T^{Arrival}_{Sim} = T^{Base}_{Sim} + (T_{Real} - T^{Base}_{Real}) \cdot 10^{-9} \quad (1)$$

The arrival time $T^{Arrival}_{Sim}$ is the time when this message should be processed by the IPCApp, the application layer of the simulated network stack. $T^{Arrival}_{Sim}$ is calculated according to Equation 1 by adding the difference between the current real-time T_{Real}, which is equal to the timestamp enclosed in the message, and the actual real base time T^{Base}_{Real} in nanoseconds to the simulation base time T_{Sim}^{Base}. This message is then inserted into the simulation event queue. To be able to send the aforementioned notification to the application, a timer has to be started to expire at $T_{Sim}^{Arrival}$. A separate module handles the timer events started in the IPC Scheduler.

5.3 Handling messages sent to the application

As mentioned in Section 3, the communication is done via sockets. They can be blocking, i.e. waiting until the peer answers, or non-blocking, i.e. the socket call returns immediately either with the desired data or an error. In either case the program can proceed. The decision, whether the sockets should be blocking or not, is taken by the application. We do not block or unblock the sockets as specified by the application, but rather block all sockets and keep a variable, indicating the status of the socket. This has the advantage that we can handle all send and receive calls in the same way.

The messages that arrive from the network, i.e. the ones destined for the application, often arrive faster than they can be processed. Therefore, they have to be temporarily stored in a message queue (see Figure 6). If the application is ready

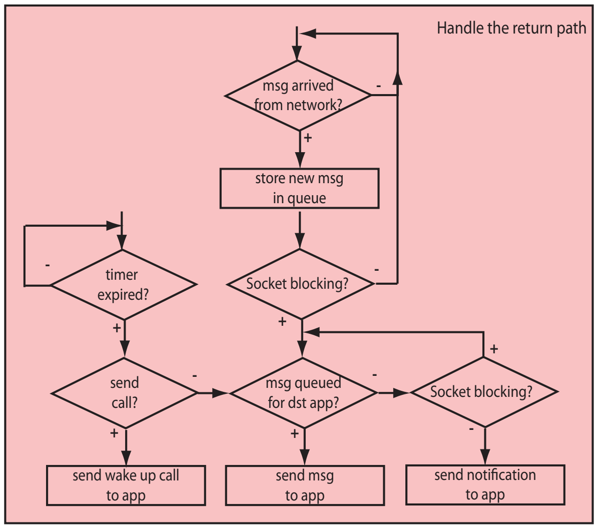


Figure 6: Simulation to Application

to receive data, it sends a notification with a timestamp to the IPC Scheduler. It is handled in the send path of Figure 5. The arrival time is calculated according to Equation 1 and set as the expiration time of the timer. When the timer expires (see the left hand path in Figure 6), it is either the one for the notification of the send call or the receive call. For a send call, a 'wake-up' call is sent to the application to indicate that the message that was just delivered is now being processed. For a receive call, the message queue is searched for data bound for the calling process. If data is present, it is sent to the application. If no suitable message can be found, the action taken depends on the status of the socket. If it is non-blocking, i.e. the application expects a fast answer to be able to go on, a notification is sent indicating it would block thereby telling the application that no data is present. For a blocking socket, the application must wait until the message arrives from the network.

		Latency			
		1 ms	100 μs	10 μs	1 μs
Bandwidth	10 Mbps	363.19	358.91	358.54	358.38
	100 Mbps	48.63	43.92	43.74	43.45
	1 Gbps	19.61	14.26	14.09	13.79
	10 Gbps	14.63	10.76	10.59	9.94

Table 1: Network Effects of LU Runtimes on Mesh

5.4 Ordering of events

The problem of synchronizing events in a distributed system has been dealt with for more than thirty years. Leslie Lamport published his groundbreaking paper [3] about the ordering of events in a distributed system. He states that all processes, although running individual clocks, have to keep an order of the events by meeting two Clock Conditions. The correct order is measured according to a virtual clock that is adjusted, i.e. advanced, by the processes if necessary.

In our system, the processes are not distributed in the sense of running on different computers, but still they are independent processes whose events have to be synchronized with the simulator. We have two clocks, the wall clock time, which is common to all processes, and the virtual simulation time, which is only known to the simulation. All outside events have to be mapped to simulation events in the right order. The simulation time corresponds to Lamport's virtual clock.

The first Clock Condition states that, if one event comes before another one of the same process, the virtual time of the first one has to be smaller than that of the second one. This condition is met by our algorithm because all messages arriving at the IPC Scheduler are ordered according to their timestamps, which is equivalent to the wall clock time common to all processes, and mapped to the virtual time in the same sequence.

The second Clock Condition demands that the sending of a message by one process and its arrival at another process have to be ordered according to the virtual time. The concept of sending notifications indicating the ending of a send call or the beginning of a receive call, combined with the adjustment of the base times when the density of the events has increased, guarantees the reception of the message happens after it has been sent with respect to the virtual time.

6. PLAUSIBILITY CHECKS

We developed our model for the sending of MPI messages over SCTP. The MPI programs we used were the NAS benchmark tests developed by NASA [4]. This set of parallel applications is meant to typify the sorts of application that can be run on a given cluster. Our focus was dependency of run times on the network properties. Run time is the time reported by MPI. It is not equal to the real-time, i.e. the wall clock time, between start and end of the run. The wall clock time is always greater than the reported run time, since the time the application waits for notifications has to be included.

Table 1 shows the run times for the LU (Lower-Upper Symmetric Gauss-Seidel) benchmark over a fully connected Mesh network of eight nodes with varying values for the bandwidth and the latency. It is clear that the run times decrease for a higher bandwidth and a lower latency. It also shows that the influence of latency is not significant.

While the first test was performed without including any

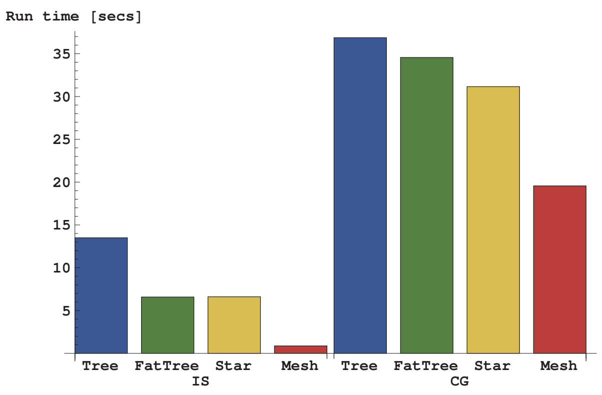


Figure 7: Run time for various topologies

routers or switches in the network, the second one was to show the influence of different topologies. The Tree and the FatTree have the same topology. While the links of the Tree all have the same bandwidth, the links connected to the root switch of the FatTree have a bandwidth an order of magnitude higher than the lower ones. The Star connects all hosts via one central switch.

Figure 7 shows the run times for IS (Integer sort) and CG (Conjugate gradient), again with an eight node network. The results are not astonishing, since we expected the run times to be in the sequence of Mesh $<$ Star $\approx$ FatTree $<$ Tree, but they were an indication to us that our algorithm worked as expected.

7. COMPARING RESULTS

Our main motive for introducing a new algorithm was the necessity to shorten the run time, i.e. the time it takes in real time to perform the experiments. Using time factors had the disadvantage that at least two runs had to be made in order to test for the convergence of the virtual time, i.e. the run time recorded by MPI and the simulator. The new algorithm only needs one pass, which is already a great advantage. In addition, the real run time decreased considerably.

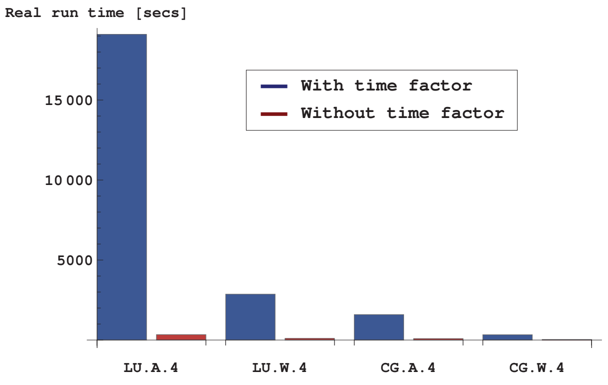


Figure 8: Comparing the real run time for the two algorithms with and without time factor

Figure 8 shows the results for the real run times for the two algorithms running on a Star topology with four nodes. The two benchmarks LU and CG were tested for two different problem sizes 'A' and 'W'. The NAS benchmarks can be compiled for different sizes, whereas 'S' is the smallest, followed by 'W', 'A', 'B', and so on until 'E'. As Figure 8 shows, the new algorithm is significantly faster than our previous algorithm, even for the smallest problem sizes. We have also not included the time required by the previous algorithm to find the minimum time factor needed to converge to the behavior of the application.

8. CONCLUSION AND OUTLOOK

In this paper, we introduced a new algorithm to coordinate the sending and receiving of messages via our application interface for MPI-NeTSim and to synchronize the events of real MPI processes with the virtual simulation events. The algorithm does not require a full synchronization since the simulator can be presumed to be the bottleneck, so the new algorithm is therefore quite simple. We explained the steps necessary to adjust the base times to be able to replace the previous time factor technique we used. We showed that with our new technique, we can significantly accelerate the simulation time.

The next step in our work will be the comparison of our results to those of real implementations to judge the predictability of our approach concerning the influence of different link parameters and topologies on the run time. We have also begun to evaluate spreading the MPI processes over physically distributed computers, and thus will be able to increase the number of participating processes without overloading the CPUs. The challenge is to build a universal real-time scheduler to handle the additional, less-predictable messaging delays for interprocess communication.

9. REFERENCES

[1] A. Grau et al. Efficient and scalable network emulation using adaptive virtual time. *IEEE Conf. on Computer Communications and Networks*, pages 1–6, 2009.

[2] D. Gupta et al. Diecast: Testing distributed systems with an accurate scale model. In *In Proc. of NSDI*, pages 407–421, 2008.

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

[4] NAS parallel benchmarks. Available from: http://www.nas.nasa.gov/.

[5] B. Penoff, A. Wagner, I. Rüngeler, and M. Tüxen. MPI-NeTSim: A network simulation module for MPI. In *The Fifteenth International Conference on Parallel and Distributed Systems (ICPADS'09)*, 2009.

[6] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler. MPI-NeTSim: A network simulation module for MPI. *ICPADS'09, Shenzhen, CHINA*, December 2009.

[7] R. Riesen. A hybrid MPI simulator. In *IEEE Cluster Computing*, pages 1–9, Sept. 2006.

[8] I. Rüngeler, M. Tüxen, and E. Rathgeb. Integration of SCTP in the OMNeT++ simulation environment. In *Proc. of OMNeT++'08*. ICST, Brussels, 2008.

[9] M. Tüxen, I. Rüngeler, and E. P. Rathgeb. Interface connecting the INET simulation framework with the real world. In *Proc. of SIMUTools'08*, pages 1–6. ICST, Brussels, 2008.

[10] M. Uysal, A. Acharya, R. Bennett, and J. Saltz. A customizable simulator for workstation networks. In *Proc. of the IPPS*, pages 249–254, 1996.

[11] A. Varga et al. INET Framework. 2010. Available at: http://github.com/inet-framework/inet-doc.

[12] A. Varga et al. OMNeT++ 4.1 Docs. 2010. Available at: http://www.omnetpp.org/documentation.