SAND2008-7867C

Instruction-Level Simulation of a Cluster at Scale

Abstract

Accurate simulation is necessary to evaluate new architectural features. However, single-node, instruction-level simulation cannot predict the performance or behavior of a parallel application on a cluster or a supercomputer.

We present a scalable cluster simulator that couples a cycle-accurate node simulator with a supercomputer network model. Our simulator leverages the computational resources of a cluster by executing individual instances of IBM's Mambo PowerPC simulator on hundreds of nodes.

We integrated a network and NIC model into Mambo and transfer application data between simulated machines using the transport layer of the cluster we run on. We model the network instead of fully simulating it, which decouples the individual node simulators enough to make our design scalable.

Our simulator is able to run unmodified parallel message passing applications on hundreds of simulated nodes. We can change network parameters, inject network traffic directly into caches and use different policies to decide when that is an advantage over depositing the data in main memory. Mambo itself can simulate various numbers of CPUs, different cache and memory sizes, and has a multitude of parameters that allow us to simulate a wide variety of different parallel systems.

The main purpose of this paper is to describe our simulator in detail, evaluate it, and demonstrate its scalability. We also demonstrate the suitability of our cluster simulator for architecture research by showing the impact of cache injection on parallel application performance.

1. Introduction

The study and implementation of novel architectures play an important role in the development and advancement of applications on future-generation, high-end computing systems. Researchers are exploring potential architectural changes for clusters, including a wide range of techniques such as hardware matching support for scalable communication libraries (Brightwell et al. 2006), various techniques for injecting incoming communication messages into processor caches (Bohrer et al. 2004a; Huggahalli et al. 2005; León et al. 2007), and radical architectural changes like processor-in-memory (Rodrigues et al. 2005). Unfortunately, the impact of such changes is difficult to predict analytically due to the complex interactions between the architecture, operating system, system libraries, and applications.

Architectural simulators that examine the impact of system changes on application performance have not historically scaled well. For example, coarse-grained simulators skew dramatically when these changes are scaled up over tens or hundreds of systems. Similarly, cycle-accurate simulators which can accurately model every event to nanosecond accuracy in a single system, scale up poorly. Their running time increases dramatically even for a small number of processors. This limits designers in their ability to study how architectural changes affect scientific application performance as a cluster grows in scale.

To address this problem, we present an MPI-based cluster simulator designed to enable studies of architecture/operating system/application interactions on current and future architectures. Unlike previous work, our cluster simulator architecture uses existing clusters to simulate future clusters by coupling a cycle-accurate full-system node simulator[1] with an MPI-based high-performance network model.

The contribution of this paper is the design and implementation of a scalable cluster simulator that can be used to analyze the impact of novel architectures on parallel application performance. We describe the design of our network and NIC model that couples the individual node simulators in Section 2 and evaluate the resulting cluster simulator in Sections 3 and 4. We analyze three cache injection policies to demonstrate the capabilities of our simulator (Section 5).

2. Cluster simulator

To study the impact of future architectures on scientific applications at scale, we built a flexible apparatus to simulate a cluster of machines based on a cycle-accurate simulator. This cluster simulation infrastructure allows us to leverage the parallel computation capabilities of current clusters to simulate future parallel architectures or current ones with different performance characteristics.

Our cluster simulation infrastructure consists of the following components: a multiprocessor full-system simulator on each each node of the physical cluster, an OS and communication libraries for the simulator, a shim layer, a modeled NIC, a modeled network, and a runtime environment to launch the simulated cluster. Since we will be referring to two clusters – the simulated cluster and the physical cluster where the simulated cluster runs – we will use the following terms for the rest of this paper. A *node* refers to a physical host within the real cluster; a *machine* refers to a full-system simulator running on a node; and a *NIC* refers to a modeled NIC that connects machines within the simulated cluster.

[1] A *full system* in this context means all the components that make up a processing element (PE) of a cluster node: CPU(s), caches, main memory, and the components that connect these parts.

Our simulated cluster is launched on a physical cluster by executing one instance of a machine per node. The machines communicate with each other over a modeled network through a modeled NIC. The NIC serves as a bridge between a (simulated) machine and a (physical) node. The NIC uses the existing messaging system on the physical cluster to communicate with NICs running on other nodes.

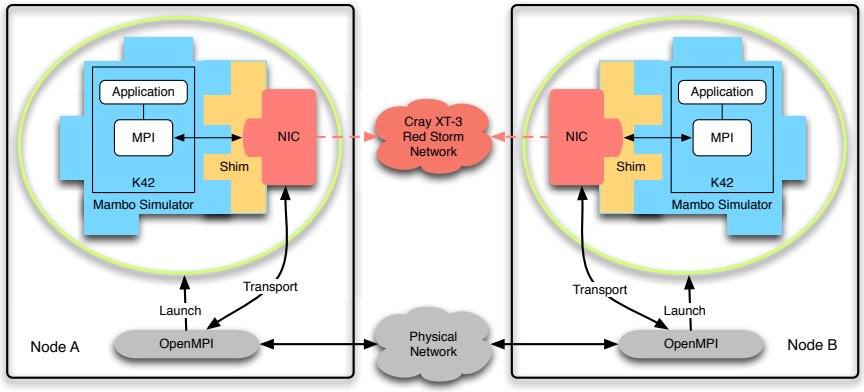


Figure 1. Two simulated machines communicating with each other through the physical cluster's network.

Figure 1 illustrates the architecture of our apparatus. On each physical node of a cluster we launch an instance of IBM's Mambo full-system simulator (see Section 2.1 below). Each Mambo instance runs the K42 operating system. The application under test uses the MPICH (Gropp et al. 1996) implementation of MPI to communicate with the system. This version of MPICH uses a device layer we call MIAMI to interact with its peers on other nodes. It does that by interacting with the local NIC model which, in turn, uses the physical system's MPI layer to exchange messages with other nodes on behalf of the simulated application. The physical transport in our case is the OpenMPI (Graham et al. 2005) version of MPI running over Myrinet, Infiniband, and Ethernet, depending on which cluster we run our tests on.

The full-system simulator provides a cycle-accurate simulation of a single-node, multiprocessor system. This machine provides a platform to study architectural features and configurations not yet available in current hardware. Our goal in developing a cluster simulation infrastructure is to leverage existing single-node simulators in a cluster setting.

2.1 Mambo

The full-system simulator we use in our infrastructure is an augmented version of IBM's Mambo full-system simulator (Bohrer et al. 2004b) that provides cache injection of incoming network messages (Bohrer et al. 2004a). A simulated machine is a multi-core, cache-coherent, distributed shared memory system (Sinharoy et al. 2005). We run the K42 research operating system (Appavoo et al. 2005) on Mambo.

Table 1 shows the configuration of Mambo we have chosen for the experiments presented in this paper. One of the key features of a cycle-accurate simulator is that many of its configuration parameters can be changed. Mambo is no exception and we plan to use additional cores and different cache organizations to simulate different machines in the future.

Table 1. Simulated system configuration.

Feature	Configuration
Simulator	Mambo PowerPC full-system simulator
Architecture	Power5 with cache injection to L2/L3
Processor	1.65 *GHz* frequency
L1 I/D cache	64 *KB*/32 *KB*, 2-way/4-way
L2 cache	1.875 *MB*, 3-slice, 10-way, 10 cycle latency
L3 cache	36 *MB*, 3-slice, 12-way, 80 cycle latency
Cache line	128 *B*
Main memory	1024 *MB*, 230 cycle latency
OS	K42
Comm. Lib.	MPICH-MIAMI w/OS-bypass & 0-copy
Network	Cray XT-3 Red Storm

2.1.1 Fast-forward mode

Cycle-accurate simulation is expensive in terms of time. Slowdown factors of several orders of magnitudes are common. Mambo spends a large portion of its time simulating the operation of caches. Cache simulation can be turned off in Mambo to make it run faster. The simulated machine then behaves as if no caches were present and that it can access main memory at L1 cache speeds.

For many experiments that is a useful feature that does not impact the end result. For example, in studies that focus on network characteristics, it may not be all that important how fast the application itself executes.

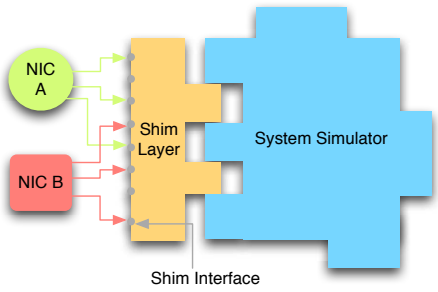
For accurate machine simulations however, cache simulation needs to be on. We have added the ability to fast-forward in our cluster simulation infrastructure. During application startup, for example, we would like the simulation to proceed as quickly as possible. When execution reaches the inner kernel, we activate the cache simulation. In Section 4.3 we will demonstrate this feature. We run through the application setup phase with cache simulation disabled, and then turn it on before we reach the computational kernel we are interested in evaluating.

Fast-forward can also be used to let an application restart from an earlier checkpoint. While reading the restart data and initializing, cache simulation can be turned off. Once the calculation resumes, we enable cache simulation. This feature enables us to simulate longer application runs in less time.

For many of the experiments in Section 4 we ran with cache simulation turned off, since for most of those experiments we were not interested in the actual simulation results from Mambo. The experiments in Section 5 were done with cache simulation turned on. We have implemented fast-forward mode for the AMG application and use it for all experiments with AMG, unless we specifically say otherwise.

2.1.2 Interfacing Mambo with the NIC

A machine and its associated NIC interact through a *shim layer*. This layer provides a bidirectional path between the simulated machine and the NIC (see Figure 2). On one side, a machine communicates with its NIC through memory mapped registers. Using this mechanism, a user process can interact with the NIC directly, bypassing the OS and/or hypervisor if needed. Access to these registers is controlled by the OS and/or hypervisor. On the other side, a NIC communicates with its machine through a well-defined interface called the *shim interface*. The table in Figure 2 shows the operations provided by the shim interface. The implementation of this interface is simulator dependent. In addition to the operations listed in the table, the shim layer provides functions to load and unload network controllers and other devices at run time.



Function	Description
memory_read/write	read/write to main memory
cache_write	write to L2/L3 cache
schedule_job	launch job on host
delay_cycles	time delay on host
raise_interrupt	I/O interrupt
memory_mapped_I/O	functions to trigger on regs

Figure 2. The shim layer provides the glue between the system simulator and the NIC.

2.2 The network and NIC model

A NIC connects its local machine with the rest of the machines running on the cluster. It acts as a bridge between its machine and the physical node it is running on. The NIC uses the host node's transport layer (MPI) to communicate with NICs running on other nodes of the system.

Each message sent between simulated machines is augmented by the NIC with a timestamp and delay information about the modeled network. The NIC then sends that information together with the actual data using the transport layer provided by the host cluster. The receiving NIC waits to deliver the message to the simulated machine until its clock reaches the message's timestamp plus the modeled network delay.

This is possible because the network of the physical cluster looks lighting fast in comparison to the very slow running simulated machines. The NICs can deliver messages at practically any latency and bandwidth in simulated time. This allows us to present the simulated machines with any type of network (physically possible or not) of our choice.

2.2.1 Synchronization

The clocks on the nodes of a typical cluster are not synchronized and drift over time. Furthermore, the performance of nodes in a cluster is not the same and impacts the performance of the simulated machines. Therefore we need to synchronize the simulated machines once in a while so that none of them get too far ahead of the others in the system.

We accomplish this synchronization by executing a barrier operation at specific intervals of the simulation clock. For example, if the synchronization interval is set at 50,000, then the simulated machines will execute the barrier when they reach 50,000, 100,000, 150,000, and so on, cycles. While the machines wait for each other in the barrier, the local simulation clocks are stopped.

The delay caused by the barrier is not visible to the simulated application. However, the barrier has externally visible effects. If one of the machine simulators is slow in reaching the barrier, because it writes lengthy debug information to an external disk for example, all the simulated machines are delayed by that. In other words, the overall execution time of our cluster simulator represents the slowest path through all instances of the individual machine simulators. We will evaluate the impact of synchronization in Section 4.

2.2.2 Network model

To decide when to deliver messages, we use the network model that is part of Seshat (Riesen 2006) and has the characteristics of a Cray XT-3 Red Storm network. Seshat is an execution driven discrete event simulator to study application behavior under varying network characteristics.

The current network model does not take network topology into considerations. Therefore it is not capable to emulate congestion. Two configuration parameters allow us to vary the bandwidth and latency of the model. This enables us to simulate faster or slower networks.

2.2.3 NIC model

To run parallel applications on our simulated cluster, we developed a minimal API needed to support MPI: *MIAMI* – Minimal Interface for An MPI Implementation. The asynchronous MIAMI API is shown in Table 2. It supports OS-bypass and allows for zero-copy MPI transfers.

Table 2. MIAMI API

Function	Description
int **init**(void);	initialize
int **finalize**(void);	clean up
int **size**(void);	number of processes in job
int **rank**(void);	my rank in job
double **clock**(void);	time in seconds
int **tx_start**(void **buf*, int *len*, int *cntxt*, int *tag*, int *dst*, int *lsrc*);	start a send
int **stx_start**(void **buf*, int *len*, int *cntxt*, int *tag*, int *dst*, int *lsrc*);	synchronous send
int **tx_done**(int *handle*);	check send completion
int **rx_start**(void **buf*, int *len*, int *cntxt*, int *tag*, int *src*);	post a receive
int **rx_done**(int *handle*, int **len*, int **tag*, int **src*);	check receive completion
int **rx_probe**(int **flag*, int **len*, int *cntxt*, int **tag*, int **src*);	probe for message arrival

We implemented a driver on our simulated NIC for the eleven-function MIAMI API, and also created a corresponding device layer for MIAMI in the MPICH implementation of MPI that runs as part of the simulation. Figure 3 shows how a simulated application interacts with the local NIC. The application makes calls into the MPICH library which in turn uses the MIAMI API to interact with the NIC model.

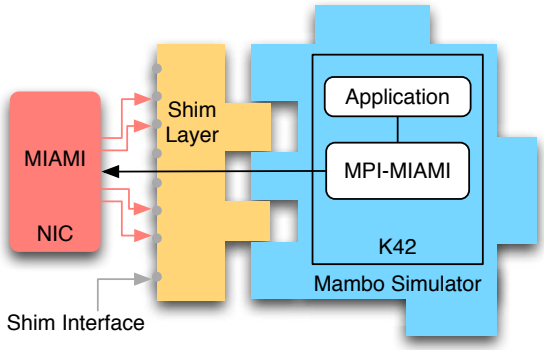


Figure 3. An application on a simulated machine interacting with the local NIC.

Most MIAMI functions in the NIC are implemented straightforward as calls into the physical system's transport layer. In our case that layer is OpenMPI running over Myrinet, OpenIB, or Ethernet depending on the cluster we are using for our experiments.

Several factors complicate the implementation of MIAMI in our NIC. Our simulation infrastructure is in itself a complete parallel application, and it must not use the underlying MPI in an illegal manner. For example, the infrastructure might be in the middle of a point-to-point transfer on behalf of the simulated application when Mambo's clock reaches a synchronization interval and forces a barrier. Other nodes may still be sending or waiting for messages before they reach their barrier points. It is important not to cause deadlock or violate the rules of the MPI standard in such situations.

Another complicating factor is that we have to send envelope information along with each message the simulated application sends. We use information inside that envelope to decide when to deliver a particular message. In order to see the envelope information we have to receive the message into a buffer so we can examine it. This is made difficult when messages arrive unexpectedly and the simulated application does not post a receive until much later. The NIC has to perform careful buffer management and message matching while maintaining good performance and scalability.

3. Experimental setup

We ran our experiments on two large clusters at *(removed for blind review)*. In this section, we briefly describe the characteristics of these machines, and list the benchmarks and applications we used to evaluate our infrastructure.

3.1 Test platforms

Cluster Θ is comprised of *(removed for blind review)* compute nodes. They are dual 3.6 *GHz* Intel EM64T processors with 6 *GB* of RAM. Cluster Θ's network is an Infiniband fabric with a two level Clos topology. The nodes run Red Hat Enterprise Linux with a 2.6.9 kernel and use Lustre as the parallel file system. We use the version 1.2.7 OpenMPI library and the version 1.3.1 OFED library to connect to the Infiniband fabric.

Cluster Σ has *(removed for blind review)* nodes and each has dual 3.4 *GHz* Intel EM64T processors with 2 *GB* of RAM. It uses a Myrinet for its interconnect. It also runs Red Hat Enterprise Linux with a 2.6.9 kernel and runs OpenMPI version 1.2.2 which uses version 1.2.7 of the MX library to interact with the Myrinet.

Both of these systems have dual CPU nodes. Due to the way K42 interacts with Mambo, we are limited to run a single simulator on each node. That means we currently need 256 cluster nodes to simulate 256 nodes.

3.2 Benchmarks and applications

We chose five different benchmarks and applications to test and evaluate our simulator.

3.2.1 IS from the NAS parallel benchmark suite

IS is the well-known integer sort benchmark from the NAS parallel benchmark suite (Bailey et al. 1995; der Wijngaart 2002). We used version 2.4 for our experiments and chose IS because it has a very short runtime. This makes IS very suitable for cycle-accurate simulations which take many hundred to thousand times longer to finish than the native execution time of the tested benchmark.

The second reason we chose IS is that it is a C code. We currently do not have a Fortran cross compiler available for our test environment. This prevents us from running the remaining NAS parallel benchmarks which are written in Fortran.

3.2.2 AMG from the Sequoia acceptance suite

AMG is an "algebraic multigrid solver for linear systems arising from problems on unstructured grids" (Lawrence Livermore National Laboratory 2008). It is one of several benchmarks used by Lawrence Livermore National Laboratory (LLNL) in its request for proposals and acceptance of the Sequoia supercomputer. Sequoia will be LLNL's next Advanced Simulation and Computing (ASC) machine.

We chose AMG because it is a communication intensive application which can, for large problem sizes, spend 90% of its execution time using the MPI library to transfer messages. Doing this, AMG uses mostly MPI collective operations. A small percentage of communications are point-to-point messages of relatively small size (2 – 10 KB) (Lawrence Livermore National Laboratory 2008).

AMG contains several different solvers which can be selected from the command line. The data presented in this paper is from solver 0 (the default) and solver 1 runs. We chose those two solvers because they seem to place more demand on the memory subsystem than the other solvers. Cache injection (discussed in Section 5) should benefit these solvers more than the others available in AMG.

AMG has three distinct phases of operation. The solver runs in the third phase, while the first two phases are used for problem setup. We augmented AMG so it can run in fast forward mode (discussed in Section 4.3). We let AMG run without cache simulation during most of its setup phases and turn on cache simulation a little before we enter the solve phase. Turing cache simulation on a little early is necessary to let the caches warm up before we enter the solve phase.

3.2.3 LAMMPS from the Sequoia acceptance suite

LAMMPS (Plimpton 1995) is a classical molecular dynamics code developed at Sandia National Laboratories. For our experiments we use the embedded atom method (EAM) metallic solid input script used by the Sequoia benchmark suite and provided on the LAMMPS web site (Sandia National Laboratory 2008a).

3.2.4 FFTW

FFTW (Fastest Fourier Transform in the West) is a C library for computing the discrete Fourier transform (DFT) in one or more dimensions (Frigo and Johnson 2005; Frigo 1999). We use the MPI parallel FFTW version 2.1.5 (MPI transforms are available only in this version). FFTW allows the computation of different types of transforms, including normal and transpose order, and with and without work space. The output data computed by these transforms maintain the same ordering as the input data for normal, and transpose order for the transpose transform. The work parameter uses `MPI_Alltoall` communication at the expense of extra storage space, while no work uses point-to-point communication. We use the speed test provided by the parallel FFTW to benchmark complex multi-dimensional transforms on several processors. The results reported in this paper use normal order without work space.

3.2.5 HPCCG

HPCCG is one of the micro-applications of the Mantevo project (Sandia National Laboratory 2008b). Micro-applications are "small, self-contained programs that embody essential performance characteristics of key applications." HPCCG is intended to be the "best approximation to an unstructured implicit finite element or finite volume application in 800 lines or fewer." These characteristics make HPCCG ideal for our purposes.

4. Cluster simulator evaluation

In this section, we evaluate several characteristics of our cluster simulation infrastructure. Our goal is to build a flexible apparatus that is scalable, performs well and reliably, and produces valid results. We present some initial experiments we have conducted using our cluster simulation infrastructure in the next section. Here we evaluate the simulator itself.

4.1 Repeatability

Since cycle accurate simulation is time consuming it is not always possible to repeat a single experiment very many times to avoid measurement errors. It is therefore important to know how repeatable individual results are and how often outliers occur.

Table 3 shows the application reported run times for several of our applications. Since we are running on production clusters, an important consideration is whether running on the same set of nodes produces different results, than running on whatever nodes the batch system allocates to us.

Table 3. Variations in reported simulation time.

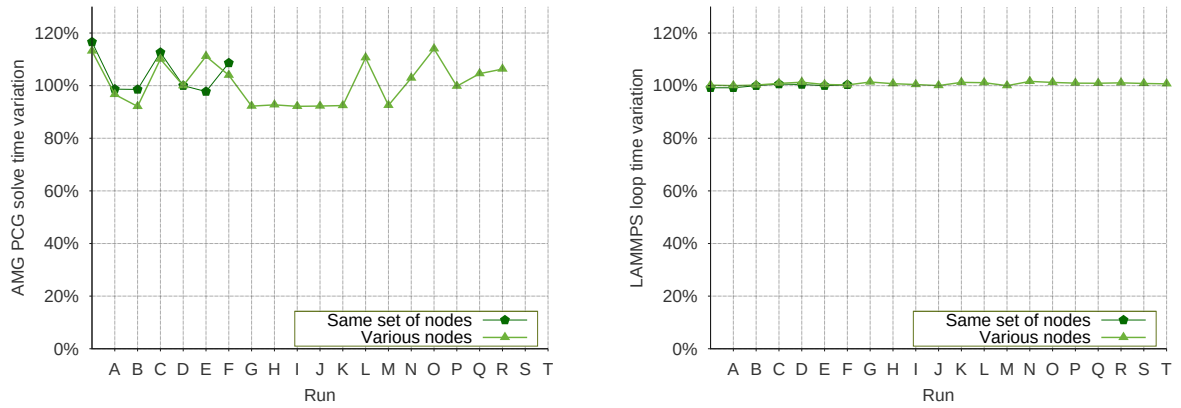
Program	Minimum	Mean	Average	Maximum	SD	Runs
AMG, 8 nodes, same nodes	-0.92%	33.2 *ms*	35.2 *ms*	+18.17%	3.4 *ms*	7
AMG, 8 nodes, batch	-7.70%	36.0 *ms*	36.5 *ms*	+14.24%	2.9 *ms*	19
HPCCG, 64 nodes, same nodes	-0.66%	80.6 *ms*	81.2 *ms*	+4.06%	1.3 *ms*	2×7
HPCCG, 64 nodes, batch	-1.34%	80.9 *ms*	81.2 *ms*	+3.72%	1.1 *ms*	58
LAMMPS, 16 nodes, same nodes	-0.78%	480.2 *ms*	480.1 *ms*	+0.63%	2.7 *ms*	7
LAMMPS, 16 nodes, batch	-0.85%	480.4 *ms*	480.2 *ms*	+0.75%	2.4 *ms*	24
IS, 64 nodes, 7 same nodes	-0.00%	750.0 *ms*	750.0 *ms*	+0.00%	0.0 *ms*	6×7
IS, 64 nodes, batch	-0.00%	750.0 *ms*	750.0 *ms*	+0.00%	0.0 *ms*	9

We therefore ran several times in two different modes. In “batch” mode we repeatedly submit the job and let the system determine which nodes are available for each subsequent run. In “same nodes” mode we submit a single batch script and run the same benchmark seven times. Each of the seven runs is then using the same set of nodes allocated for the duration of the run.

For each application and mode we calculate the minimum, mean, average, maximum, and standard deviation of the running time each benchmark reports. The minimum and maximum are expressed as a percentage of the mean.

There is no significant difference whether we run on the same set of nodes, or let the batch system allocate nodes for us. However, there is a difference in the behavior of the benchmarks. While IS and LAMMPS show almost no variation from one run to the next, HPCCG and AMG fluctuate more. Figure 4 shows the difference between running AMG several times and running LAMMPS several times in a row.

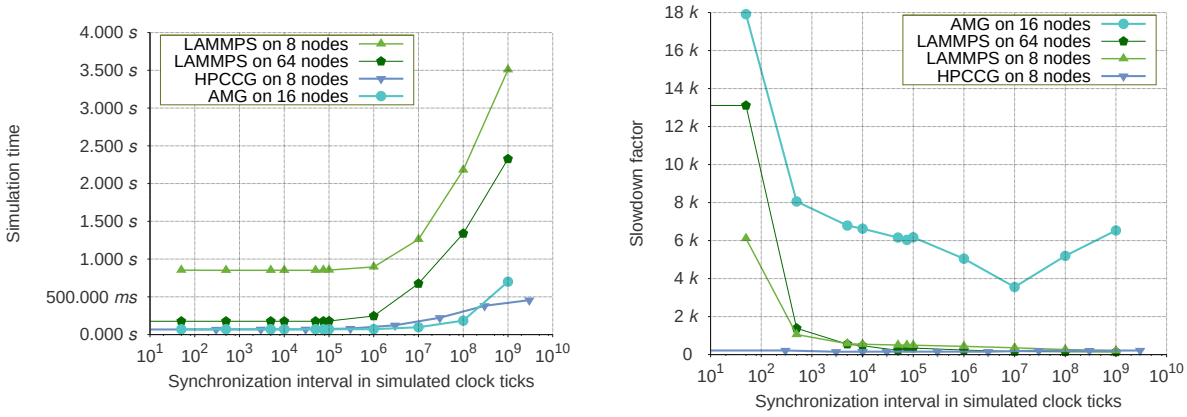
AMG is much less deterministic. Indeed, in many of our experiments we noticed AMG producing outliers, requiring more runs to observe trends in a given experiment.



(a) AMG on 8 nodes

(b) LAMMPS on 16 nodes

Figure 4. Examples of reported simulation times over several runs.



(a) Impact on simulation time

(b) Impact on wall-clock time

Figure 5. Frequency of synchronization impacts reported simulation time and measures wall-clock time.

4.2 Impact of synchronization interval

In Section 2.2.1 we discussed how our simulation infrastructure keeps the individual machines synchronized. The choice of synchronization interval has an impact on how long a simulation takes and the accuracy of the simulation.

For the results shown in Figures 5 and 6 we ran LAMMPS, simulating 4,000 atoms, on 8 and 64 nodes, HPCCG on 8 nodes with a problem size of $10\times10\times10$, and AMG with solver 0 and problem size 1. Since we are only interested in the synchronization impact, we turned cache simulation off for LAMMPS and HPCCG. We ran AMG in fast-forward mode with cache simulation off until just before the solver phase which AMG measures.

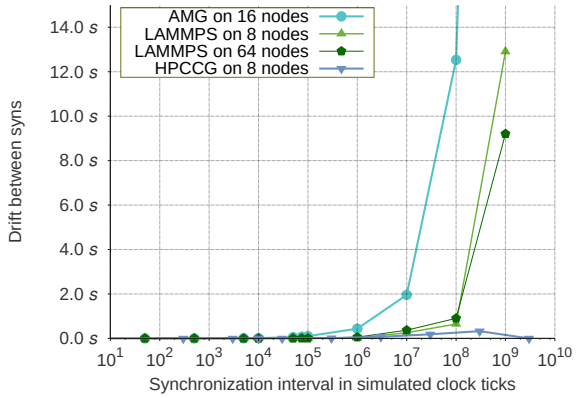


Figure 6. Impact of synchronization interval on drift.

In Figure 5(a) we increase the synchronization interval along the x-axis and plot the reported simulation time. The graph shows that if we increase the interval beyond about 10^6 clock cycles, the simulation results start to deviate from the results obtained when the nodes are more tightly synchronized.

We calculate slowdown by dividing the simulation time between `MPI_Init` and `MPI_Finalize` into the wall-clock time between the calls to those two functions. Figure 5(b) shows that increasing the interval between node synchronizations lowers the slowdown factor because the wall-clock time required to run the simulation goes down. Our simulation infrastructure is in itself a parallel application. By increasing the synchronization interval we reduce the amount of message passing and synchronization and, therefore, increase performance.

Since we should not increase the synchronization interval beyond 10^6 clock cycles, but want to keep it as high as possible for performance reasons, a range of 10^5 to 10^6 clock cycles seems appropriate. We chose 50,000 for all the results reported in this paper.

Before each synchronization we read the local wall-clocks. During the synchronization we find the lowest time value and subtract it from the highest time value. This measures how much these two extreme nodes have drifted from each other during the previous interval. We keep a running total and report the average drift at the end of the simulation.

Figure 6 shows the impact the synchronization interval has on drift. Some applications are more prone to drift than others, with drift becoming a problem at an interval of about 10^6 clock cycles.

Synchronizations occur at specific simulation time intervals. If the nodes drift too much from each other, fast nodes will have to wait longer for others to reach their synchronization points. The simulation clock is stopped during these waits.

Drift is caused by the environment outside our simulation framework. Different nodes run at slightly different speeds, but a bigger factor is OS noise and I/O, to write logging information for example. While a node is busy writing to an external file system, or doing OS housekeeping tasks, the simulation clock is not advancing. When that node joins the others in the next synchronization it will show up late and we can measure that delay.

If the synchronization intervals are high (greater than 10^6 clock cycles), faster nodes can get too far ahead. That means our network model may deliver some messages late in simulation time. This in turn leads to delays in simulation time at the receiving node. That is why drift in wall-clock time has an impact on simulation time. By choosing a good synchronization interval we can keep drift to a minimum and prevent artificial delays in simulation time.

Another way to look at this is that our simulation framework gang schedules the simulated nodes. It can only do that, if we keep the synchronization interval sufficiently small; i.e., below 10^6 clock cycles.

4.3 Fast-forward evaluation

In Section 4.3 we described the ability of our infrastructure to disable cache simulation and let the simulation progress faster. At a strategic point in the application, we turn cache simulation back on to properly evaluate an inner kernel or restart a calculation after we have reinitialized from a checkpoint for example.

When turning cache simulation on, the caches will be empty and need some time to warm up. To avoid inaccurate results, it is necessary to turn cache simulation on a little before the program section of interest. A statement inserted into the source code of the program under test accomplishes that.

We ran AMG on 8 nodes with solver 0 and different workloads with cache simulation turned off, always on, and only on during the calculation phase. We obtained the results shown in Table 4. For fast-forward mode we turned cache simulation on during the setup phase to allow the caches to warm up before entering the calculation phase.

Both wall-clock and simulation time are measured between `MPI_Init()` and `MPI_Finalize()`. The slowdown factor is the wall-clock time divided by the simulation time. The results in Table 4 are the average of three runs each (problem size 3 is six runs for each simulation).

The “Solver only” column is the time AMG reports being in the solve phase. During that phase cache simulation is turned on in the “always on” and the fast-forward mode. Therefore, the solver times reported

Table 4. Cost of cache simulation for 8-node AMG

Problem size	Cache simulation	Wall-clock	Simulation	Solver only	Slowdown factor
1 1 1	Always off	0:00:19	123.465 *ms*	29.209 *ms*	158
	On during solve	0:10:13	137.880 *ms*	33.626 *ms*	4,447
	Always on	0:14:32	137.560 *ms*	33.258 *ms*	6,340
3 3 3	Always off	0:02:48	535.781 *ms*	233.962 *ms*	313
	On during solve	1:25:43	753.849 *ms*	264.116 *ms*	6,822
	Always on	1:40:32	786.404 *ms*	264.944 *ms*	7,671
4 4 4	Always off	0:06:15	1.118 *s*	522.955 *ms*	336
	On during solve	3:31:09	1.780 *s*	610.965 *ms*	7,118
	Always on	3:58:37	1.854 *s*	614.203 *ms*	7,721

in those two modes should be the same. The difference for each problem size in Table 4 is less than 1%, which is well within the running time variations of AMG.

Using fast-forward mode to advance a simulation to the point of interest should help us get the same simulation results as if we had run the entire simulation with cache simulation turned on. Fast-forward mode for AMG for the cases we measured saves up to 27 minutes or up to 30% for the small problem size.

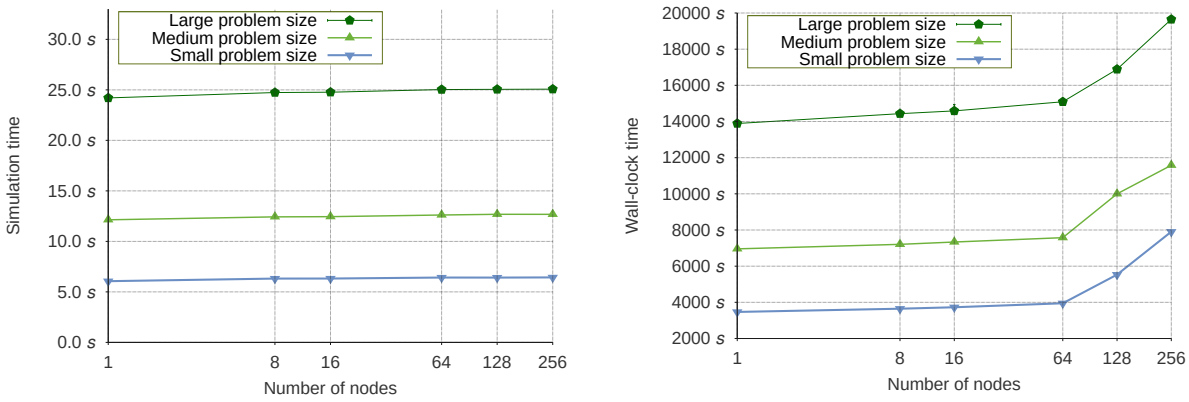
4.4 Scaling

We have mentioned before that our infrastructure is itself a parallel application. Due to the tight synchronization of the simulated nodes the parallel performance of the infrastructure is directly tied to the speedup and parallel efficiency of the simulated application.

LAMMPS scales very well as can be seen in Figure 7(a). The graph plots the reported LAMMPS time against the number of nodes we ran on. In weak scaling mode we increase the overall problem size with the number of nodes available for the computation. This keeps the problem size on each node constant. The flatness of the curve attests to the fact that LAMMPS has very little (communication) overhead as more nodes are added.

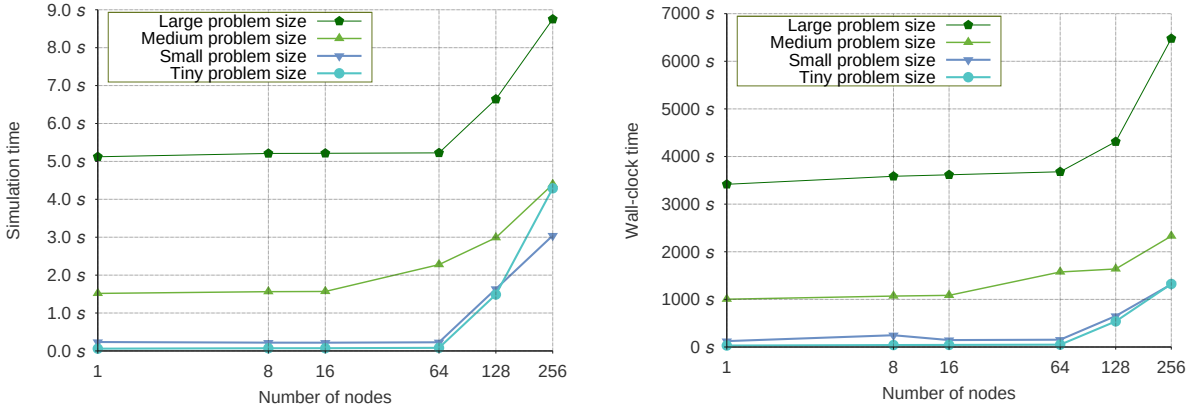
We ran with cache simulation turned off and show one run per problem and node size. The exception is the large problem size for which we ran three times for each node size. The minimum, maximum, and mean are plotted using error bars. However, the differences are so small that the error bars are not visible behind the plot points.

In Figure 7(b) we show the wall-clock time of our simulation infrastructure for the LAMMPS runs in Figure 7(a) (from `MPI_Init` to `MPI_Finalize`). Our simulator has some overhead due to the frequent



(a) Simulation time **(b)** Wall-clock time

Figure 7. Simulation and wall-clock time of simulator running LAMMPS.



(a) Simulation time **(b)** Wall-clock time

Figure 8. Simulation and wall-clock time of simulator running HPCCG.

synchronizations which require all-reduce communications every 50,000 simulation clock cycles. Nevertheless, it scales very well and is suitable for simulating well-scaling applications, such as LAMMPS, to several hundreds and thousands of nodes.

Figure 8 shows the running time of the simulated HPCCG micro-application and the behavior of our simulation infrastructure in wall-clock time. Each plot point represents a single run of HPCCG with cache simulation turned off. The simulated HPCCG does not scale quite as well as LAMMPS. Comparing Figure 8(a) and (b), we can see that the inefficiencies of our simulation infrastructure are masked by HPCCG's parallel performance characteristics. Despite this, our infrastructure scales well enough to run HPCCG on several hundred nodes.

5. Cache injection

The purpose of creating the simulation infrastructure described in this paper is to conduct experiments where cycle-accurate node simulation is important when evaluating message-passing parallel applications. In this section, we look at cache injection and its impact on parallel applications.

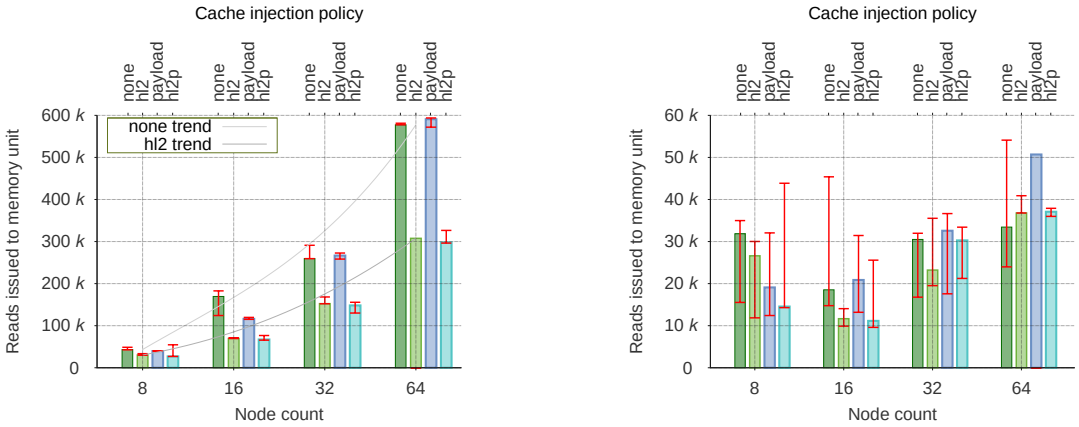
Our version of Mambo has the capability of letting the NIC inject data directly into the L2 or L3 caches. Writing to memory is performed by issuing write-invalidate bus transactions. Writing to a cache is performed in chunks of one cache block and the state of the resulting block is set to clean exclusive (Tendler et al. 2002). Writes of less than one block are handled by a write with flush operation (flush the cache line first and then write the data into memory). Writes to a cache require the physical address of the destination to be block-aligned. Thus, writing incoming network data to a cache may involve writing the first few words using write with flush until the destination address is cache aligned, then writing full blocks to the cache. Currently, all writes to a cache also update main memory.

When to inject data into the cache is a current topic of research. Injecting network data before it is needed will displace current data forcing a reload of that data plus a reload of the network data later on. Which network data to inject is also a question of interest.

We can inject entire messages with the risk of displacing too much data of the current working set. We can inject only MPI envelope information from the message header, such as source, tag, and length information about the message, or we can inject both the payload and the header.

We will look at four different injection policies. “None” is no cache injection at all, which is the base case to which we will make comparisons. “Hl2” injects the message headers into the L2 cache. “Payload” injects the body of the message into the L3 cache as long as the payload is at least 128 bytes (a cache line), but not more than half the L3 cache size. The fourth policy, “Hl2p”, combines header injection into the L2 cache with payload injection into the L3 cache.

The version of the NIC we used for these experiments injects data at the time of a successful return from a user-level call to `rx_done` (see Section 2.2.3). That is, the data has arrived at the destination and the network model has determined that the (simulated) time for delivery has arrived. The next time after that, when the application asks whether a particular message has arrived (using `rx_done`), the NIC injects the data and returns success to the application query. This approach should increase the likelihood that injected data will be used right away, since the application has just asked whether it was available.



(a) AMG, solver 0, -r 1 1 1

(b) FFTW

Figure 9. Memory pressure is influenced by cache injection policy.

5.1 Memory pressure

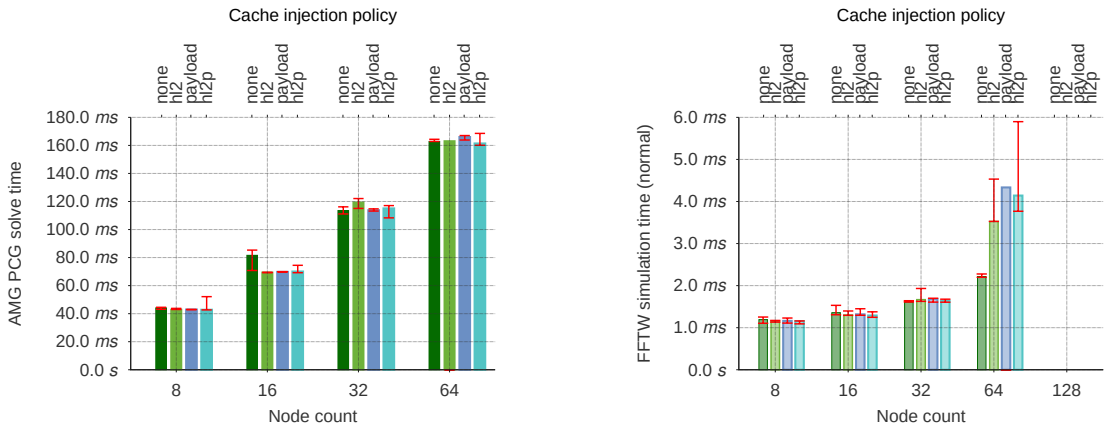
One of the goals for cache injection is to reduce the number of times data has to travel across the memory bus between arrival at the NIC and consumption by the CPU. In other words, we hope that cache injection relieves pressure on memory by serving more of the data directly from cache.

We can evaluate this by counting the number of read requests to the memory unit. Only reads that cannot be satisfied from one of the cache levels result in a countable event. In Figure 9 we show the number of reads issued for the four cache injection policies on increasing number of nodes.

Both AMG and FFTW were run in weak scaling mode where the problem size per node is kept constant. The number of reads we report are for the entire run of FFTW and the solve phase for AMG. Each bar in Figure 9 is the result of three runs. We show the median as a colored bar, and the minimum and maximum as error bars.

Studying Figure 9(a) we see that without cache injection, the number of reads to main memory increases exponentially as the node count goes up. The upper trend line plots a smooth curve among the no-cache injection data points. The lower trend line follows the hl2 data points.

It is clear that injecting header information into the L2 cache greatly reduces memory pressure as the node count goes up, when compared to no cache injection. The reason the payload only injection policy shows no benefit is because for the small problem size we use here, most of the messages are smaller than 128 bytes and will not be injected. The hl2p policy shows the same benefit as hl2 because the headers are injected as in hl2, but the payload is not, since it is too small. The FFTW data in Figure 9(b) shows a similar benefit.



(a) AMG, solver 0, -r 1 1 1

(b) FFTW

Figure 10. Performance of AMG and FFTW using different cache injection policies.

5.2 Performance

Reduced memory pressure should lead to better application performance. As Figure 10 shows, this is not the case for our experimental runs. The reason is again the small problem size. The memory subsystem has plenty of capacity to support the additional reads that become necessary as we increase the node count because of the small problem size.

6. Related work

The cluster simulator we describe in this paper combines a discrete event multiprocessor full-system simulator with a NIC and a network model implementation. Discrete event simulation has been a topic of study for many years. We refer the reader to a textbook (Banks et al. 2000) which devotes a chapter to systems simulation.

Parallel discrete event simulation has also been explored extensively and many techniques are in use to limit interactions between distant parts of a system. These interactions are necessary to synchronize clocks. The event queues need to be distributed for a simulator to scale. This, however, increases the need for additional synchronization. Fujimoto (Fujimoto 1990) provides a very nice description of the problems involved in parallel discrete event simulation.

Asynchronous distributed simulation (Chandy and Misra 1981) is one way to address the problem of synchronizing distributed parts of the same simulator. Improvements upon this work include (Greenberg et al. 1996), and simulating large-scale systems is summarized in (Fujimoto et al. 2003).

However, the problem of scalability remains. The larger and the more accurate the simulation, the longer it takes. Some researchers turn to modeling instead (Hoisie et al. 2000, 2006; Kerbyson et al. 2001). While this is more efficient, it is also less accurate and less likely to predict future systems' performance precisely when compared to detailed discrete event simulation.

We believe that a hybrid approach of simulation and modeling can yield accurate results within reasonable time frames using compute resources such as a cluster or a small supercomputer. As we have explored in Section 4.2 we still need to synchronize. However, it is much less frequent than what would be necessary, if we had distributed, but coupled, event queues.

7. Summary and future work

In this paper we describe an infrastructure that allows systems researchers to study the impact of architectural changes on scientific, parallel, application performance. This infrastructure is designed to:

- leverage current single-node simulators into a cluster infrastructure;
- enable simulation of recent and future cluster architectures, including techniques to improve application performance and scalability;
- allow system designers to better understand the interactions between the OS, parallel applications and the NIC, as well as between nodes; and, finally,
- accurately simulate a cluster at scale.

The results presented in this paper indicate that our infrastructure can indeed meet these goals. We will continue making improvements to the simulation infrastructure itself. For example, we would like to replace K42 with Linux and make launching a simulation more straight forward. We also need to enable more than a single node simulator instance per multicore node.

Many opportunities exist for future work with the simulator described in this paper. We have only scratched the surface of exploring the potential benefits of cache injection, and plan to explore cache injection policies for multicore architectures. We also intend to assess the impact of different network characteristics, such as different latencies and bandwidths, on applications and their interactions with the memory system.

Mambo is able to simulate a multi-processor multicore machine. This kind of architecture will become prevalent in the next-generation supercomputers and will significantly change the flops to network bandwidth ratio. We want to evaluate the impact of such an architecture on parallel application performance.

References

J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source Linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

D. Bailey, T. Harris, W. Saphir, R. v. d Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

J. Banks, J. S. C. II, B. L. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, Inc., 3rd edition, 2000. ISBN 0-13-088702-1.

P. Bohrer, R. Rajamony, and H. Shafi. Method and apparatus for accelerating Input/Output processing using cache injections, March 2004a. US Patent No. US 6,711,650 B1.

P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo – a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004b. ISSN 0163-5999.

R. Brightwell, T. Hudson, K. Pedretti, and K. Underwood. An accelerated implementation of portals on the cray seastar. In *Proceedings of the Cray Users' Group Annual Technical Conference*, Lugano, Switzerland, May 2006.

K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981. ISSN 0001-0782.

R. F. V. d Wijngaart. Nas parallel benchmarks version 2.4. NAS Technical Report NAS-02-007, Computer Science Corporation, NASA Advanced Supercomputing(NAS) Division, NASA Ames Research Center, 2002.

M. Frigo. A fast Fourier transform compiler. In *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 169–180. ACM, May 1999.

M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990. ISSN 0001-0782.

R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. Large-scale network simulation: How big? how fast? In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pages 116–123, October 2003.

R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, September 2005.

A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Superfast parallel discrete event simulations. *ACM Trans. Model. Comput. Simul.*, 6(2):107–136, 1996. ISSN 1049-3301.

W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A general predictive performance model for wavefront algorithms on clusters of SMPs. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, pages 219–228, Washington, DC, USA, 2000. IEEE Computer Society.

A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin. A Performance Compariosn through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple. In *in Proc. IEEE/ACM SuperComputing*, Tampa, FL, November 2006.

R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, Madison, WI, June 2005.

D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–48, Denver, CO, 2001. ACM Press.

Lawrence Livermore National Laboratory. ASC Sequoia benchmark codes. `https://asc.llnl.gov/sequoia/benchmarks/`, Apr. 22 2008.

E. A. León, K. B. Ferreira, and A. B. Maccabe. Reducing the impact of the memory wall for I/O using cache injection. In *15th IEEE Symposium on High-Performance Interconnects (HOTI'07)*, Palo Alto, CA, August 2007.

S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117(1):1–19, 1995.

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

A. Rodrigues, R. Murphy, R. Brightwell, and K. D. Underwood. Enhancing NIC performance for MPI using processing-in-memory. In *Workshop on Communication Architectures for Clusters*, Denver, CO, April 2005.

Sandia National Laboratory. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Nov. 6 2008a.

Sandia National Laboratory. Mantevo project home page. `https://software.sandia.gov/mantevo/`, Nov. 6 2008b.

B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, January 2002.