A Simulator for Large-scale Parallel Computer Architectures

Curtis L. Janssen,[i] Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo

Sandia National Laboratories
Livermore, CA 94551-0969

Keywords: computer architecture simulation, network models, network congestion, Message Passing Interface

Accepted for publication in "International Journal of Distributed Systems and Technology". Expected publication date is January 2010.

SAND2009-2279C

Abstract

Efficient design of hardware and software for large-scale parallel execution requires detailed understanding of the interactions between the application, computer, and network. We have developed a macroscale simulator (SST/macro) that permits the coarse-grained study of distributed-memory applications. Currently, applications using the Message Passing Interface (MPI) are simulated; however, the simulator is designed to allow inclusion of other programming models. The simulator is driven from either a trace file or a skeleton application. Trace files can be either a standard format (Open Trace Format) or a more detailed custom format (DUMPI). The simulator architecture is modular, allowing it to easily be extended with additional network models, trace file formats, and more detailed processor models. We describe the design of the simulator, provide performance results, and present studies showing how application performance is affected by machine characteristics.

1 Introduction

The degree of parallelism that must be exposed to efficiently utilize modern large-scale parallel computing systems is intimidating. Because individual processor performance gains are currently achieved primarily through multiple cores on a chip and multiple threads of execution in a core, the rate at which parallelism must be exposed by an application will increase as a function of overall machine performance relative to historical trends. This results in greater design complexity for both machine architects and application software developers. The use of simulation, however, can aid both in their efforts to obtain high utilization from future computing platforms.

Simulation is already used extensively in the design of computing systems for both functional verification and timing estimation. As an example of the range of capabilities available, including just a few examples of open-source timing simulators, there are processor simulators (Binkert, et al., 2006; M5Sim), memory simulators (Jacob; Wang, et al., 2005), and network ns-3 (ns-3).

Several simulators have been developed to generate performance estimates for high-performance computing architectures. These range from high-fidelity and computationally expensive simulators for measuring performance between two nodes (Rodrigues, et al., 2003; Underwood, Levenhagen, & Rodrigues, 2007) to lower-fidelity and lower-cost simulators that can estimate performance on large-scale machines. These lower-fidelity simulators use a variety of approaches to generate the application's processor and network workload including tracing, direct execution, and the use of skeleton applications. Additionally, the flow of data through the network is modeled with varying fidelity. In the present paper we are concerned with lower-fidelity and lower-cost simulation techniques to enable simulation at very large scales, and we will briefly discuss these simulator variants in more detail, giving examples of simulators supporting each capability before

turning to a detailed description of our simulator in Section 2.

When an application is traced, the full program is run in order to collect information about how it executes. The resulting data is output into a trace file, which contains data such as the time spent in computation and the amount of data sent and received by each node. This trace file is read by the simulator, allowing it to replay the run, adjusting the simulated times to account for differences between the simulated machine and that which was used to collect the traces (Zheng, Wilmarth, Jagadishprasad, & Kale, 2005). In the case of Message Passing Interface (MPI) (Message Passing Interface Forum, 2008) traces, events that are higher level than simple sends and receives are recorded, such as all-to-all broadcast or all-to-one reduce. These network events along with associated parameters are logged without the details of the underlying messages that are used to implement the operation. It is the responsibility of the simulator to either convert these higher-level operations into the low-level messages that implement the operation or to provide an appropriate timing model that does not require simulation of the low-level messages.

In the direct execution approach the full application is run on each node (Prakash, et al., 2000; Riesen, 2006; Zheng, et al., 2005). This is different from normal benchmarking because, instead of real time, a virtual time is used to determine the execution time. The virtual time is computed by using a network model to estimate communication times. The contribution to the virtual time due to processor execution can be determined simply by using the measured real time for non-communication work or by using a processor model. This model can be informed by measurements of actual application processor utilization or more detailed processor simulations.

The third approach to generating the machine's workload does not use a full application. Instead a so-called skeleton application is used that provides enough information to the simulator for it to model both computation and communication. This takes advantage of the fact that the computations needed to determine program flow are a small subset of the total number of computations needed by typical high-performance computing applications. The skeleton application can be constructed in a variety of ways. An application programmer could directly program a skeleton application, giving the programmer the opportunity to experiment with different algorithms before having to write the full application. Existing applications can be skeletonized by replacing portions of the code doing computation with calls that instruct the simulator to account for the time implicitly (Susukita, et al., 2008). Skeleton applications can also be constructed using automated analysis tools, for example, using compiler analysis techniques to abstract away portions of the application (Adve, Bagrodia, Deelman, & Sakellariou, 2002). Skeleton applications have the advantage of capturing the essence of the application in sufficient detail to enable reasonably accurate simulation while being much less expensive than running the application.

Various approaches are also taken to model the network layer. These range from relatively simple models that only consider endpoint congestion (Prakash, et al., 2000) to accurate models that treat the flow of data through the network in detail (Benveniste & Heidelberger, 1995; Petrini & Vannesch, 1997; Zheng, et al.,

2005). In endpoint congestion models the only network bottlenecks are the nodes. If two messages arrive simultaneously at a node, only one at a time can be received, and the delay in reception of the second message is determined from simple network performance characteristics such as the latency and bandwidth. This model does not reflect the fact that internal to the network fabric there can be contention for resources. Detailed network models are aware of the machine's network topology and use this information, along with other details such as routing algorithms, to estimate message arrival times. Both approaches are useful in that the endpoint congestion model provides an inexpensive way to obtain an optimistic performance estimate while the more detailed models take into account the impact on performance of machine topology and process layout effects.

In the present work we describe a macroscale simulator for estimating the performance of large-scale parallel machines. The goals of the simulator are to assist in system design and application development. The simulator is modular, permitting multiple computation and communication models to be employed. This will allow the study of architectures at a variety of fidelities so we can trade off the computational cost of doing a simulation against the accuracy of the result. The simulator will be distributed under an open-source license to maximize its usefulness to the high-performance computing community. We focus on an extremely lightweight implementation, rather than enabling parallelism in the simulator itself. Parallelism can be easily introduced when performing independent simulations of architecture variants. We also provide a detailed MPI model that converts the high-level MPI events into the necessary communication operations. Because the MPI capability is implemented to be modular, it is simple to investigate the relative performance of various MPI algorithms. The simulator is designed to allow the use of alternative programming models, as well.

Our work is done in the context of a larger project to develop a parallel multiscale simulator that permits users of the simulator to select the desired level of fidelity for each component of the machine. This larger project is an outgrowth of the Structural Simulation Toolkit (SST) (Rodrigues, et al., 2003; Underwood, et al., 2007) and the macroscale components described herein will be referred to as SST/macro to distinguish them from the existing microscale SST components.

2 The Macroscale Simulator

The execution of an application on a parallel machine can be represented as a collection of computation and communication events. These events have complex but known dependencies; for example, a synchronization event must occur in all parallel tasks before any task can move forward. We model the execution of these events using a discrete event simulator. Using models to determine the duration of these computation and communication events, the simulator determines event completion times. Thus, the message timing of applications is determined, allowing the efficiency and scalability of applications to be examined.

We avoid the synchronization overheads incurred by parallel discrete event simulation and implement an extremely lightweight simulator within a single kernel thread. Application tasks are modeled using lightweight threads, allowing the

simulator to maintain the complex states of numbers of tasks ranging into the millions. Application task threads use a well-defined interface layer to generate simulation events, reproducing the coarse-grained communication and computation loads of real applications. This lightweight implementation allows us to simulate up to 200,000 MPI send/receive pairs per second on a single workstation, with a memory footprint that scales linearly with the number of peers.

Figure 1 illustrates the high-level design of the simulator. The process layer supports two execution modes, skeleton application and trace-driven, using lightweight threads. Task threads create communication and compute kernels which are parameterized with data for a particular communication operation or compute block. Kernels for MPI operations, for instance, require the arguments to the MPI call, while compute kernels require a description of the CPU instructions to be simulated. Tasks interact with the simulator back-end by pushing kernels down to the interface layer. The interface layer coordinates interaction with network and CPU models and handles the scheduling of resulting events on the simulator back end. The interface layer includes servers, such as **`mpiserver`** which manages interaction with the network model in MPI contexts. When kernels are completed, the process layer receives callbacks via request objects.

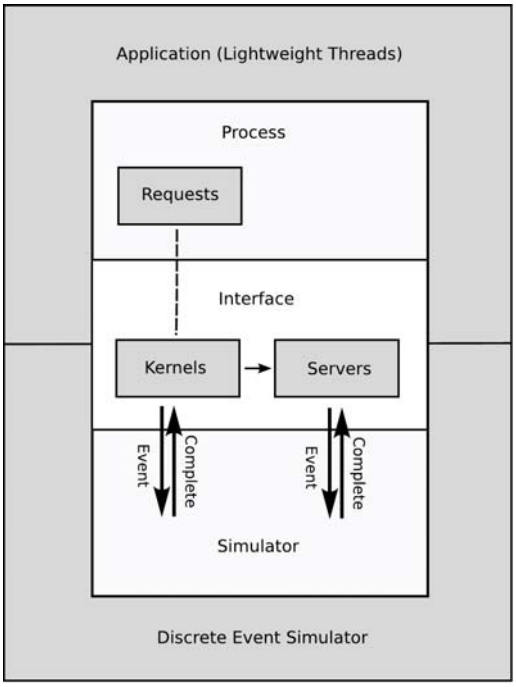

Figure 1: Application threads create communication and compute kernels and push them down to the interface layer. The interface layer schedules events on the discrete event simulator, possibly using calls to servers. Callbacks are made to the process layer via request objects when events complete.

SST/macro is implemented in C++, allowing a flexible, modular design that provides opportunities for modification and extension. The inheritance diagram

provided in Figure 2 highlights the flexibility of our design in the context of kernel objects. As specified by the **`kernel`** base class, all kernels have **`start()`** and **`complete()`** methods and maintain a list of event handlers which require notification of the kernel's completion. These are the only methods required by the simulator to incorporate kernels as discrete events. The various specializations of the kernel class handle the specific requirements of particular operations by defining the **`start()`** and **`complete()`** methods. While calling **`start()`** on an **`mpisendengine`** kernel results in a call to **`mpiserver::send()`**, invoking a network model to determine delays, a call to **`start()`** on a **`computekernel`** results in a call to the node model associated with the task, invoking a processor model. By encapsulating implementation details behind well-defined interfaces, modules within the simulator can easily be replaced; for example, alternate programming models could be simulated by replacing the MPI interface layer with an interface layer supporting a different parallelization model.

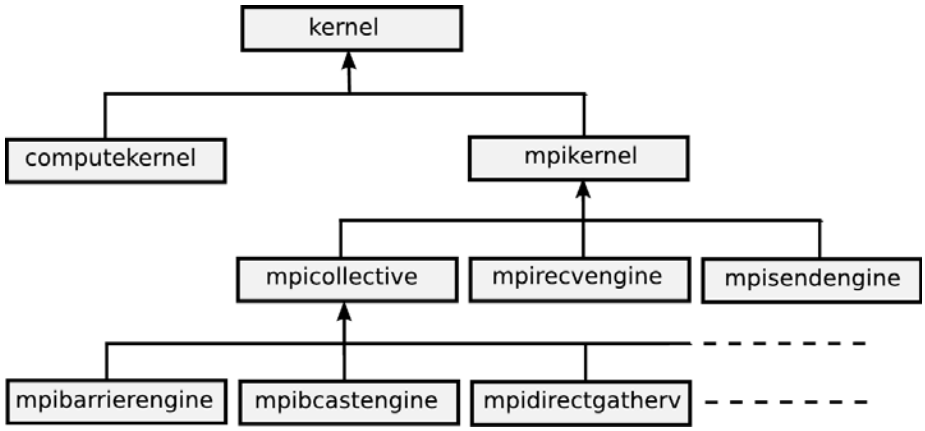

Figure 2: Inheritance diagram for kernel objects. Kernel specializations handle the specific requirements of particular operations while providing a uniform interface to the discrete event simulator.

The MPI Model

The Message Passing Interface (MPI) (Message Passing Interface Forum, 2008) provides a standard interface for programming distributed memory parallel machines in a portable and efficient manner. The MPI interface currently consists of over 200 function calls providing a rich set of communication primitives. Two of the most common sets of primitives include those for point-to-point communication and those for collective communication. Point-to-point operations are available for a variety of modes, including buffered, synchronous (will only complete after the matching receive is posted), and ready (the matching receive must be posted before the send) modes. For each of these send modes, as well as the receive calls, there are blocking (which have completed when they return) and nonblocking (a separate call is used to check for completion) versions. In addition to these point-to-point calls, collective operations that involve groups of processors are commonly employed. These include such operations as all-to-all broadcast, all-to-one reduce, all-to-one gather, and so on. Both the point-to-point and collective

operations are typically implemented using a relatively small set of point-to-point communication primitives which are specific to a particular networking technology.

Skeleton applications and MPI trace files typically provide information about only MPI calls and their associated argument lists. This means that no information is available to the simulator about the low-level point-to-point messages that a particular MPI library uses to implement an operation. The timings due to the low-level operations must be modeled by the simulator, and this presents us with the opportunity to implement a variety of models, at varying levels of fidelity, to represent the MPI operations. At the low-fidelity, low-cost end, MPI collectives can be treated without consideration of the low-level MPI implementation. An analytic or empirical performance model could be used to determine when each process will complete the operation, and a single simulator event to continue execution of all processes at the appropriate virtual time would be inserted into the event queue. A higher-fidelity approach, which is implemented in SST/macro, is to have the simulator schedule events needed for all of the low-level data transfers in the same way an MPI implementation would. If this is done while also using a network model that includes congestion effects, then the effect of congestion on the collective operation time is estimated by the simulator. In this way, the effects of changes in the MPI implementation can be studied using the simulator.

Lightweight application threads perform MPI operations through calls to the interface layer, resulting in determination of completion times for the required events by the network model and the scheduling of these events with the discrete event simulator. A timeline detailing the chronology of events scheduled by two lightweight application threads performing typical operations is shown in Figure 3. When an MPI send or receive operation is performed, the thread yields until the appropriate event is executed by the simulator indicating that enough simulation time has elapsed for the data to have been sent or received. Likewise with computation, the application trace or CPU model determines when a computation operation completes and schedules a completion event with the simulator. The application thread performing the computation yields until this completion event is triggered.

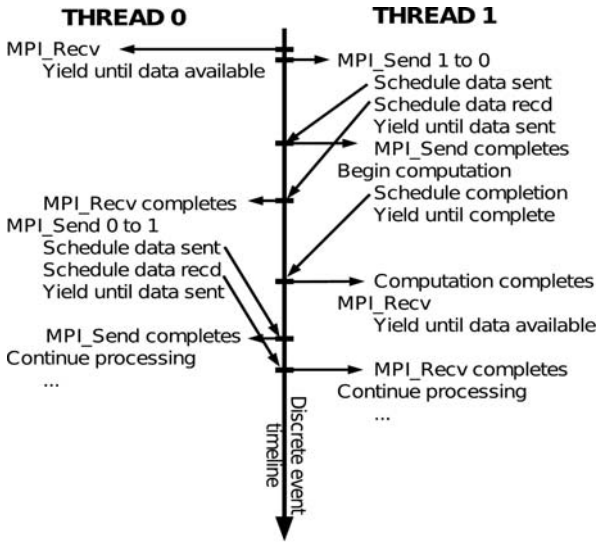

Figure 3: A timeline for the interaction of two lightweight application threads with the discrete event simulator for MPI send/receive operations and computation.

A simplified class collaboration diagram for the SST/macro MPI components is shown in Figure 4. Application threads that utilize MPI are instances of classes inheriting from the **mpiapp** class, which in turn inherits from the simulator's generic class for threads, **thread**. Three such MPI applications are shown in the figure: **mpipingpong** (a simple ping-pong skeleton application), **minimd** (a skeleton molecular dynamics application which is discussed in Section 2), and **mpitrace** (a trace file reader which is discussed in Section 2). Each MPI application object references an **mpiapi** object, which provides the MPI application programming interface. The **mpiapi** object uses an **mpistrategy** object to simulate MPI communication by building the appropriate **kernel** objects (shown in Figure 2). The **mpistrategy** object has a collection of strategies specialized for particular operations. For example, implementations of the **MPI_Barrier** function are specializations of the **mpibarrierstrategy** abstract base type. Specialized implementations of barrier strategies can be provided, or the provided **mpicorebarrier** specialization can be used. This specialization can provide a high-fidelity barrier implementation typical of many actual MPI implementations or it can perform a low-fidelity barrier that requires minimal processor time to simulate. These low-fidelity collective operations are currently only used to synchronize the nodes after **MPI_Init** is called. Otherwise, the high-fidelity MPI core operations are used in the results presented herein.

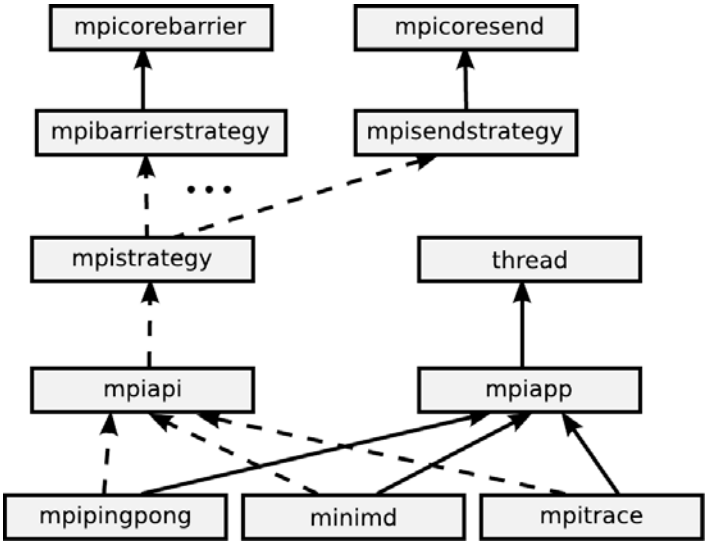

Figure 4: Collaboration diagram for MPI components. Solid lines indicate inheritance (is-a relationships) and dashed lines indicate containment (has-a relationships).

The Network Model

As the next generation ultra scale systems increasingly rely on higher concurrencies, the effect of the interconnection network on the overall system performance becomes even more important, especially for applications with intense communication loads. Consequently, there has been renewed interest in interconnect design in the computer architecture community with many new ideas

and promising results being reported. It is important to predict the performance of these proposed interconnect ideas on modern high-performance computing applications. We believe our simulator will be instrumental in this respect, as the software framework allows easy integration of new methods to enable experiments with proposed techniques on large scale, real world applications.

In keeping with the focus on modularity in the simulator design, the network system is designed as a separate module in order to provide the flexibility required to support a rich set of techniques. Moreover, the network can be simulated in arbitrary detail, which allows trading off between fidelity and runtime of simulations. For example, the network could simply be a latency-based model, which assigns a pre-specified delay to each message, or a cycle-level model that captures the finer details of a router. The general framework can support any topology, routing algorithm, etc., and can be easily tuned for network parameters such as bandwidth and latency. Below, we discuss the basic components of the currently available network system, which we used in the experiments described in Section 3.

Topology and Routing

Within SST/macro, an instance of the network object is defined by its topological description (i.e., the connections between routers/processors) and a routing method to compute a path for a message between two processors. As illustrated in Figure 5, we can currently support torus, fat-tree, hypercube, Clos, and gamma topologies, detailed descriptions of which can be found in (Dally & Towles, 2004). The **`product`** object in this figure enables producing tori of different dimensions. To define a new interconnect, the user needs to provide a method to build the topology of the interconnect and a routing method to compute a path for a message between two processors. The system will take care of the details of congestion as we will explain in the next section.

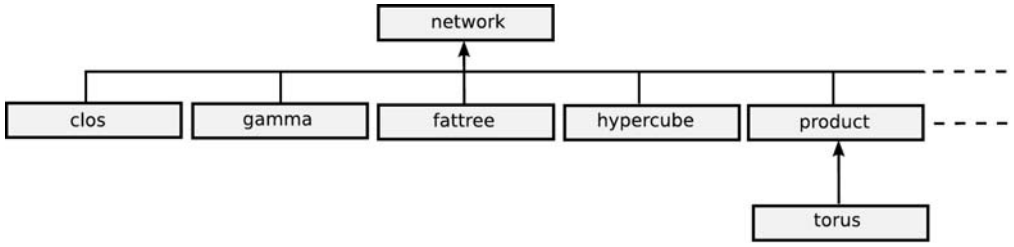

Figure 5: Inheritance diagram for network objects.

The current network module is designed for maximum runtime efficiency. The routing algorithms are static (i.e., messages between two processors always follow the same path, regardless of network status) and follow the shortest path on the network. Bandwidth is allocated on all links in the message path for the entire time required to transmit all of the data.

Congestion Modeling

Recent studies (Kamil, Oliker, Pinar, & Shalf, 2009; Shalf, Kamil, Oliker, & Skinner, 2005) show that many target applications that can reach extreme scales

display local but irregular communication patterns. Such applications commonly suffer from network congestion due to hot spots in the network; thus, modeling congestion is crucial for the accuracy of a network simulator.

From a modeling perspective, irregular communication patterns pose the biggest challenge, as the network congestion cannot be predicted a priori. Different parts of the network can be congested at different times, and the congestion depends not only on network features, such as topology, bandwidth, and routing algorithm, but also on processor features, as they determine when the messages are injected into the system. Thus, for such irregular applications, detailed simulation that accounts for the routes of messages and congestion on individual network components is essential.

Congestion modeling for regular communication patterns is relatively easier, as message delays are predictable. For instance, for computations on a regular finite element grid with only boundary communication, the communication pattern of the application will match perfectly with a mesh interconnect. Thus, there will not be any congestion, and it is sufficient to know the latency of a message between two neighboring points. At the other extreme, when all-to-all communication is performed, the message delays can be predicted as a function of network parameters a priori. The ability to predict the message delay can significantly simplify the network simulation, providing huge speedups in simulation runtime.

As a general purpose tool, SST/macro can support both approaches. The **`full`** interconnect object corresponds to the fully connected interconnect, which provides a congestion free network. For the purposes of simulation, it is the same as a latency-based network model. The **`sharedcircuit`** interconnect object, on the other hand, is designed to model congestion on a link-by-link basis by modeling he communication between each pair of nodes as a continuous flow of data. Each flow has bandwidth allocated to it in such a way that the sum of the bandwidth of all flows passing through a given network link does not surpass the bandwidth of the link. In our current implementation, the oldest active flow receives all the bandwidth it can use. The next oldest flow is allocated any remaining bandwidth it can use, and so on. In this way model network congestion can be modeled efficiently while still obtaining reasonable accuracy as will be shown in Section 3.

Trace File Driven Simulation

SST/macro is able to generate network traffic and processor workloads using trace files that record MPI calls and the time spent performing computation between MPI calls. Currently, two trace formats can be processed: Open Trace Format (OTF) (Knupfer, Brendel, Brunst, Mix, & Nagel, 2006) and DUMPI. The **`mpitrace`** class (see Figure 4) reads the trace file using the **`parser`** abstract base class which has specializations for both file formats. Additional formats can be added easily by providing additional **`parser`** specializations. Different trace formats provide differing levels of detail about the MPI call signature, and this impacts the accuracy of the simulator.

Open Trace Format traces are collected by linking the target application with the VampirTrace library (VampirTrace). VampirTrace uses the PMPI interface to intercept MPI calls, and it records trace information in OTF formatted trace files.

VampirTrace writes a binary file for each process. The OTF standard provides an interface specification for reading and writing OTF trace files. The binary files can be read by the **`otfdump`** utility provided with the VampirTrace library. This utility produces ASCII encoded files which the simulator uses as input.

The DUMPI format is a custom MPI trace file format, recorded in binary, which has been developed as part of the SST/macro simulator. Like OTF, DUMPI files are obtained by linking the application with a library that uses the PMPI interface to intercept MPI calls. The DUMPI format records more information than OTF, including the full signature of all MPI-1 and MPI-2. With this additional detail we are able to more accurately simulate an application. The DUMPI format also records return values and MPI request information. This allows error checking and permits us to match immediate mode MPI operations with the MPI operations that complete them. In addition, DUMPI allows individual functions to be profiled instead of the entire program. Processor hardware performance counter information can also be stored in DUMPI files using the Performance Application Programming Interface (PAPI). This allows information such as cache misses and floating point operations to be logged. Such data is recorded both within and between MPI calls. This information will be used by the simulator in more detailed processor models, as they are made available.

Skeleton Applications

Trace files are generated with specific application input and parallel task configuration, yielding a detailed profile of one particular run. Through the manipulation of parameters used to model the hardware and swapping in different messaging models and strategies, trace-driven simulation can contribute significantly to performance optimization and hardware design at parallelism scales on the same order as that used to generate the trace. However, the challenge of optimizing codes or designing hardware for extreme scales requires simulation capabilities long before hardware is actually available for trace file generation. Additionally, many distributed-memory codes have branch statements that are dependent on which of a set of requests was matched at a given stage. These execution details cannot be adequately captured in trace-driven execution, since the trace file reader cannot retroactively redirect control flow in the application.

Direct execution is an elegant strategy for generating traces at ultra scale on more readily available hardware, but the requirement of running the full application hampers parameter studies and limits the scale that is ultimately achievable. Though creating skeleton applications requires a greater programmer effort than trace-driven simulation or direct execution, driving the simulator from a skeleton application provides an immensely powerful approach to evaluate efficiency and scalability at extreme scales and to experiment with code reorganization or high-level refactoring without having to rewrite the numerical part of an application. This is further facilitated by the reduction in code size that happens when the bulk of computation is removed.

As a basic parallel application, consider a simple ping-pong between pairwise ranks in a parallel system (rank 0 exchanges data with rank 1, rank 2 with rank 3, etc.).

The implementation of this program on the simulator, given in Figure 6, looks almost identical to the native MPI implementation, except for differences in the syntax of MPI calls. Building on this basic skeleton application, it is easy to test the effects of varying network topology, hardware layout (e.g. processors per node), indexing strategies for node allocation, etc. Using a contention-free network model, **`mpipingpong`** has been run with up to 16M nodes on a single workstation processor with a memory footprint of roughly 4KiB for each MPI peer.

```
void mpipingpong::run() {
  this->mpi_->init();
  mpicomm world = this->mpi_->comm_world();
  mpitype type = mpitype::mpi_double;
  int rank = world.rank().id;
  int size = world.size().id;
  if(! ((size % 2) && (rank+1 >= size))) {
    // With an odd number of nodes, rank (size-1) sits out
    mpiid peer(rank ^ 1); // partner nodes 0<=>1, 2<=>3, etc.
    mpiapi::const_mpistatus_t stat;
    for(int half_cycle = 0; half_cycle < 2*niter; ++half_cycle) {
      if((half_cycle + rank) & 1)
        mpi_->send(count_, type, peer, mpitag(0), world);
      else
        mpi_->recv(count_, type, peer, mpitag(0), world, stat);
    }
  }
  mpi_->finalize();
}
```

Figure 6: Main run loop for a pairwise-exchange MPI ping-pong skeleton application.

A more significant application is the skeletonization of miniMD, a molecular dynamics micro-application from the Mantevo project (Mantevo). The full miniMD application is reasonably small, at 1830 source lines of code, and the skeleton application is one-quarter the size at 456 lines. Most of the key computations in miniMD get collapsed down to simple **`compute(...)`** calls, while all MPI calls and control logic relevant to execution patterns are retained. The skeleton version of the time integrator in miniMD (Figure 7) provides an example of how this mixture of pre-evaluated timing information and original program logic can be used to drive the simulator.

```
void minimd::integrate::run(shared_ptr<atom> atm,
  shared_ptr<force> frc,
  shared_ptr<neighbor> nbr, shared_ptr<comm> cmm,
  shared_ptr<thermo> thm, shared_ptr<timer> tmr)
{
  mpiid rank = mpi_->comm_world().rank();
  for(int n = 0; n < this->ntimes; ++n) {
    env_->compute(interpolator->get("integrate::run", 0));
    if((n+1) % nbr->every) {
      cmm->communicate(atm);
    }
    else {
      cmm->exchange(atm);
      cmm->borders(atm);
      nbr->build(atm);
    }
    frc->compute(atm, nbr);
    env_->compute(interpolator->get("integrate::run", 1));
    if(thm->nstat)
      thm->compute(n+1, atm, nbr, frc);
    }
}
```

Figure 7: Time integrator from the skeletonized miniMD application.

The two calls to **`env->compute(...)`** simulate the actual time integration in miniMD. The interpolated time values for these calls come from a parametric evaluation of miniMD performance, but they could just as well be obtained from detailed microprocessor simulations, runs on emulator systems such as QEMU, or constitutive performance models. These time values can also be scaled or have noise added to them to study the effects of load imbalance or rogue OS noise. This skeletonization effort is being used as a development platform for analyzing and instrumenting more significant application codes.

3 Results

In this section we present performance results for our simulator, both in terms of the ability of the simulator to reproduce measured machine performance and in terms of the computational expense of running the simulator. We also use the simulator to determine the sensitivity of application runtimes to changes in machine characteristics in order to demonstrate the power of simulation in understanding application performance.

Experimental Setup

Performance studies were carried out on two separate platforms. Parallel studies of AMG were performed on Sandia's RedStorm Qualification (RSQ) machine, which

consists of 32 dual processor compute nodes and 48 quad processor compute nodes. These are respectively based on 2.4 GHz dual-core AMD Opteron and 2.2 GHz quad-core AMD Opteron processors. RSQ consists of a single cabinet, in which case the interconnect is reduced to a 2D mesh of dimension 4x24. The mesh is wrapped in the larger dimension. The link bidirectional bandwidth of the mesh is 9.6 GB/s and the bandwidth of the HyperTransport (HT) that connects the router chip to the processor is 3.2 GB/s in each direction. Parameters for use in the simulation of AMG on RSQ were determined by running MPI benchmarks on the system. The simulator models the bidirectional links in RSQ as a pair of unidirectional links, thus only data for unidirectional bandwidths were collected. The communication bandwidth between nodes on an otherwise idle network is limited by the HT link between the processor and the router chip, and the measured unidirectional bandwidth was in this case 1823 MB/s. When two pairs of nodes communicate and the network traffic for each of these pairs is routed over a single router-router network link, the measured aggregate unidirectional bandwidth over the shared router-router link was 3245 MB/s. The nearest neighbor latency was measured at 4.44 μs. We also measured the bandwidth and latency for MPI communication between a pair of processes on the same node to be 6115 MB/s and 2.8 μs. These results were obtained using only the quad-core nodes when the machine was otherwise idle. For applications, we used the **`mpipingpong`** skeleton application described in Section 2 and the AMG2006 benchmark (Henson & Yang, 2002). AMG2006 is a parallel implementation of the Algebraic MultiGrid method. It was developed at Lawrence Livermore National Laboratory, and is part of the Sequoia benchmark suite (ASC Sequoia Benchmark Codes). The code is written in ISO standard C using MPI for parallelization. The algebraic multigrid method is commonly used to solve sparse systems of linear equations, especially those that arise in applications of finite element methods. The dominant computational kernel is sparse matrix vector multiplication; thus, the memory bandwidth is the main factor that determines performance. For parallelization, each processor is assigned a portion of the finite elements (subdomain) and the associated variables/equations in the sparse matrix. Communication is required to exchange boundary information between subdomains. The average MPI message size for these noncollective calls is around 2-10 KB. Collective calls dominate the total communication time, as they take around 90% of the total MPI time. More detailed information about AMG2006 can be found at (AMG benchmark summary).

Validation of the Simulator

The simulator was validated on results from a range of AMG configurations using the latencies and bandwidths measured for RSQ above. The ranks were laid out along the nodes of the mesh, traversing the shorter dimension first. We used processor counts of powers of 2 from 8 to 128. These were run using 1 processor per node (ppn), 2ppn, and 4ppn. Two logical grid decompositions were used, a 1D decomposition and a 3D decomposition. Traces were collected using the lightweight DUMPI library. Figure 8 shows the measured simulated walltime versus the measured elapsed walltime with the simulation driven from these DUMPI traces.

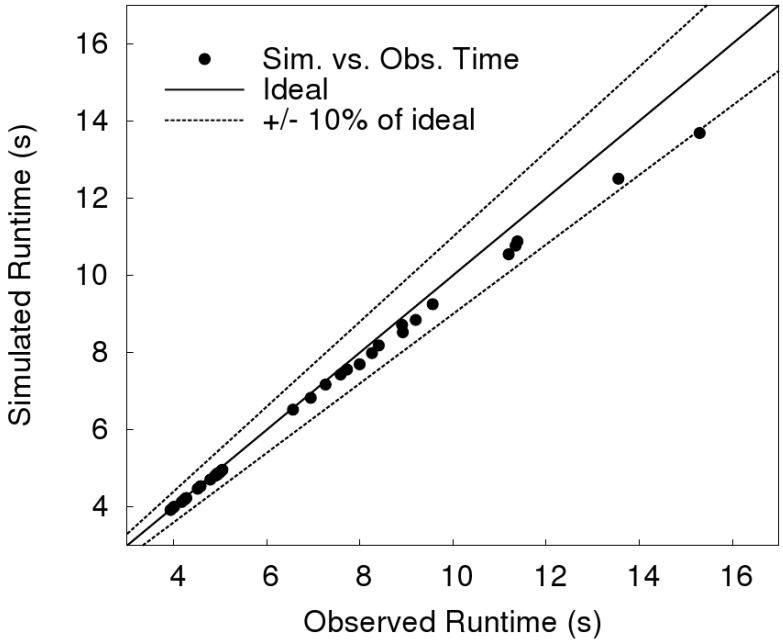

Figure 8: Comparison of simulated and observed runtimes for the AMG2006 program for a variety of node counts and decompositions.

Capabilities of the Simulator

We begin highlighting the capabilities of SST/macro by describing two sets of benchmark simulations of the **`mpipingpong`** skeleton application. The first is used to measure how much processor time the simulator itself requires, and the second illustrates how the simulator can be used to study machine performance. Using a contention-free network model to focus on the process layer performance, simulations were performed with up to 2^{20} peers. Figure 9 demonstrates the high performance context switching that our lightweight thread-based process implementation can achieve. An MPI ping/pong send/receive pair can be simulated in about 5 μs of time. Nearly 1,000 processors can be simulated before the third level cache no longer holds the simulator's data, at which point walltimes begin to increase significantly. After the third level cache size is exceeded, the cost of simulating a send/receive pair levels off to around 10.5 μs. Figure 10 illustrates the results of simulations using a fat-tree network with 4 levels and a radix of 24. The effects of traffic congestion on the performance of the **`mpipingpong`** application are very clearly observed as the number of nodes surpasses 12, which is the number of nodes connected to a single radix 24 crossbar in the fat-tree.

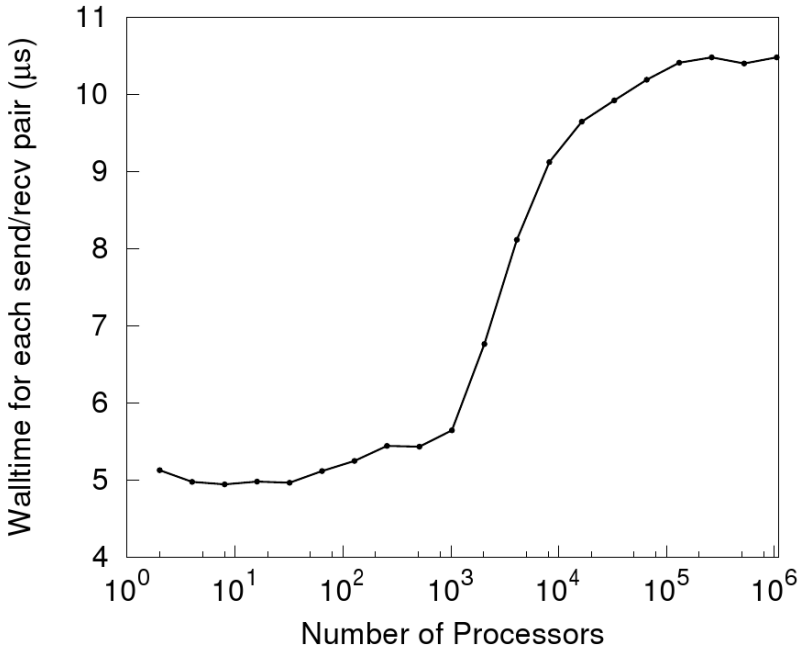

Figure 9: Performance of **`mpipingpong`** simulations using a contention-free network model with up to 2^{20} peers sending a total of 4M messages. The step in the performance curve corresponds to the point at which the program and its data no longer fit into third level cache.

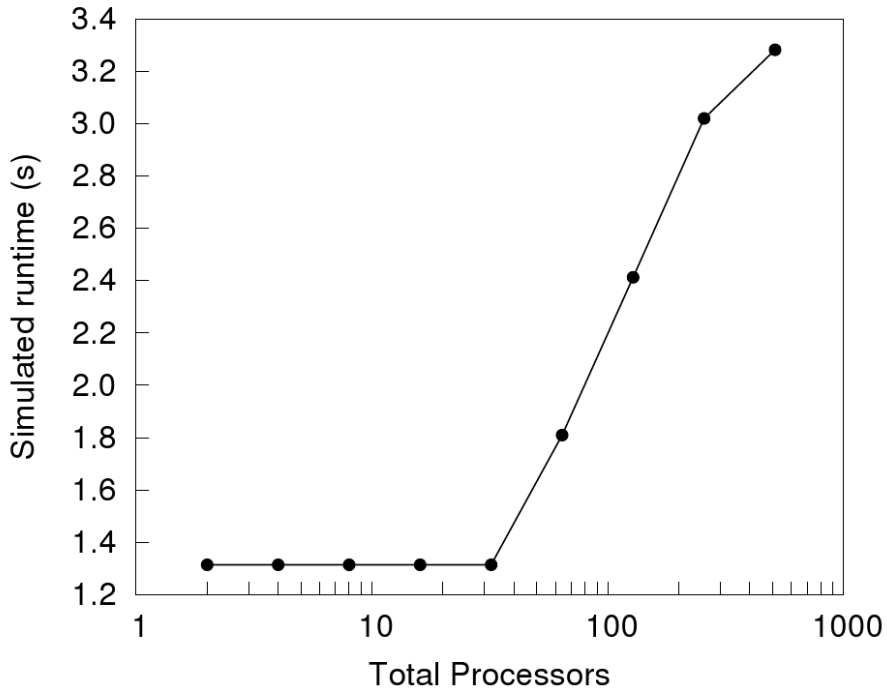

Figure 10: Traffic congestion in **`mpipingpong`** simulations with random MPI rank assignment on a fat-tree network (4 levels, radix 24). Each node sends $65536/n_{proc}$ messages.

The modular and high performance infrastructure, and growing collection of hardware and software modeling capabilities provided by SST/macro create a powerful platform for exploring hardware and software design. We present here the results of several parametric studies which demonstrate this ability to develop insights by rapidly performing simulations spanning design spaces. The following parameter studies were performed using AMG traces and simulator parameters as described above. Figure 11 and 12 respectively illustrate the sensitivity of simulated AMG execution times to latency and bandwidth variations. The AMG simulation was done using DUMPI traces collected using 128 processors of RSQ. Varying only latency and holding all other parameters constant, we see that for latencies on the order of 10 μs the predicted runtimes are fairly insensitive to changes in the latency. Reducing latencies even further produces very little benefit in runtime, while at latencies of 100 μs performance the runtimes begin to increase sharply. Varying network bandwidth while holding the latency constant, we find that at 1 GB/s or greater bandwidth, little performance variation is seen in the AMG runtimes, and all of the network topologies perform similarly. When the bandwidth falls significantly lower than 1 GB/s, runtimes significantly increase, and variation is observed among the network topologies, with the torus topology giving slight longer execution times than fat-tree or crossbar topologies.

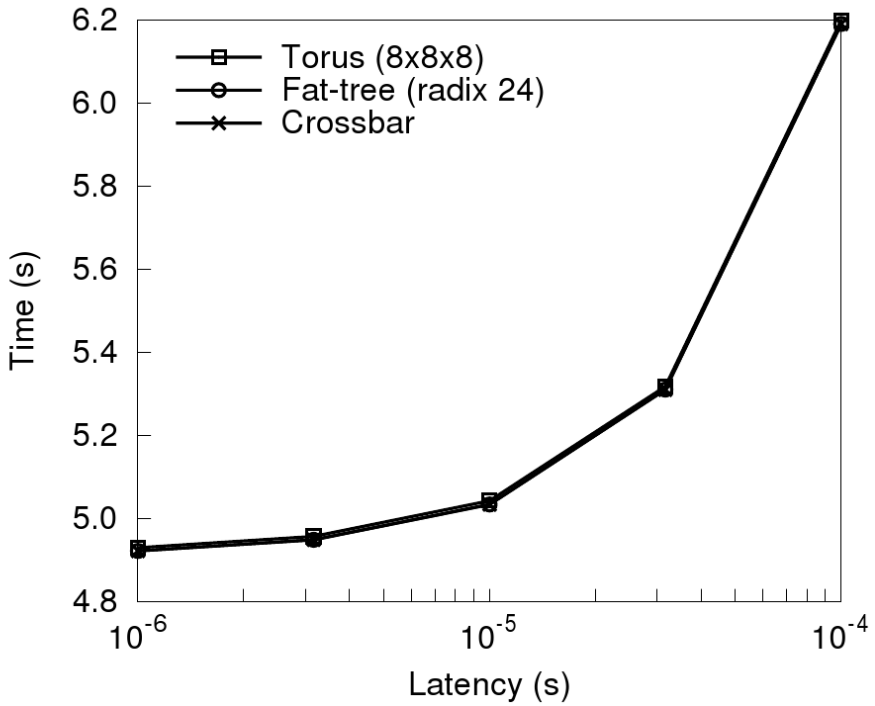

Figure 11: Studies of the sensitivity to network latency of trace-driven AMG simulations using 128 nodes with a single processor per node. Time is measured as latency is varied holding the bandwidth constant at 1 GB/s.

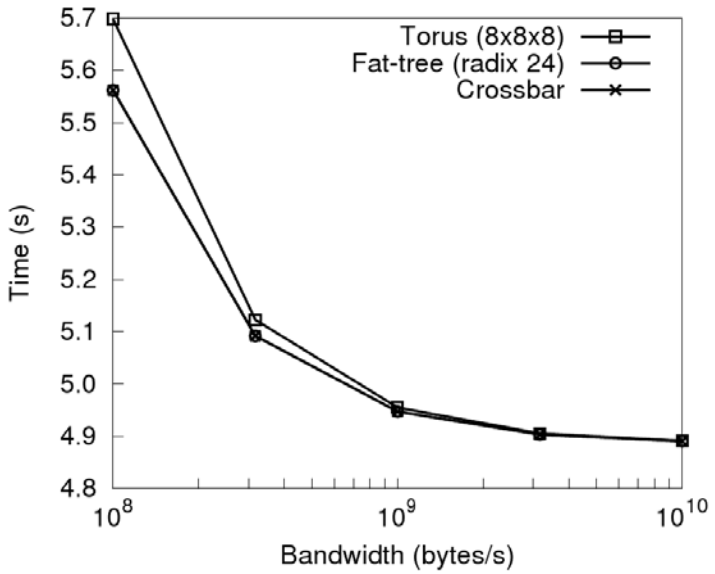

Figure 12: Studies of the sensitivity to network bandwidth of trace-driven AMG simulations using 128 nodes with a single processor per node. Time is measured as bandwidth is varied holding the latency constant at 3 μs.

While these initial parameter studies focus on network characteristics, as more processor and messaging models are made available within SST/macro, this type of sensitivity study will be possible for a wide range of hardware and software parameters.

4 Conclusions

We have described SST/macro, a macroscale simulator for the coarse-grained simulation of applications running large-scale parallel computers. The simulator is designed to assist in the development of computing architectures and applications. The simulator has a flexible architecture allowing treatment of different hardware and software components at various fidelities. Our implementation is extremely lightweight, enabling large-scale systems to be simulated on a single processor. We also provide a flexible approach to modeling MPI that can be used to easily investigate the effect on performance of changes to the MPI library and do not preclude the investigation of alternative programming models. The simulator can be driven through trace files collected by running applications on an existing machine or by skeleton applications which provide enough information for the simulator to predict the corresponding applications' execution times. We have found that the simulator reproduces actual runtimes with an error that is typically less than 10%.

5 Acknowledgements

The authors would like to thank Ida M. B. Nielsen for helpful comments.

This work was supported by the US Department of Energy's National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94-AL85000.

Bibliography

Adve, V. S., Bagrodia, R., Deelman, E., & Sakellariou, R. (2002). Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures. *Journal of Parallel and Distributed Computing, 62*(3), 393 - 426.

AMG benchmark summary. from https://asc.llnl.gov/sequoia/benchmarks/AMG_summary_v1.0.pdf

ASC Sequoia Benchmark Codes. from https://asc.llnl.gov/sequoia/benchmarks/

Benveniste, C., & Heidelberger, P. (1995). *Parallel simulation of the IBM SP2 interconnection network.* Paper presented at the 1995 Winter Simulation Conference, 345 E 47th St., New York, NY 10017.

Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., & Reinhardt, S. K. (2006). The M5 Simulator: Modeling Networked Systems. *IEEE Micro, 26*(4), 52-60.

Dally, W., & Towles, B. (2004). *Principles and Practices of Interconnection Networks*: Morgan Kaufmann.

Henson, V. E., & Yang, U. M. (2002). BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics, 41*.

Jacob, B. DRAMsim: A Detailed Memory-System Simulation Framework. from http://www.ece.umd.edu/dramsim/

Kamil, S., Oliker, L., Pinar, A., & Shalf, J. (2009). Communication Requirements and Interconnect Optimization for High-End Scientific Applications. *IEEE Transactions on Parallel and Distributed Systems, 99*(1).

Knupfer, A., Brendel, R., Brunst, H., Mix, H., & Nagel, W. E. (2006). Introducing the open trace format (OTF). In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot & J. Dongarra (Eds.), Computational Science-ICCS 2006. 6th International Conference. Proceedings, Part II (Lecture Notes in Computer Science Vol.3992)

M5Sim. The M5 Simulator System: A modular platform for computer system architecture research. from http://www.m5sim.org

Mantevo. from https://software.sandia.gov/mantevo/

Message Passing Interface Forum. (2008). MPI: A Message-Passing Interface Standard: Version 2.1. from http://www.mpi-forum.org

ns-3. The ns-3 network simulator. from http://www.nsnam.org/

PAPI. from http://icl.cs.utk.edu/papi/

Petrini, F., & Vannesch, M. (1997). SMART: A simulator of massive architectures and topologies. *Euro-PDS*, 185-191.

Prakash, S., Deelman, E., & Bagrodia, R. (2000). Asynchronous parallel simulation of parallel programs. *IEEE Transactions on Software Engineering, 26*(5), 385-400.

Riesen, R. (2006). A Hybrid MPI Simulator, *IEEE International Conference on Cluster Computing (CLUSTER'06)*.

Rodrigues, A., Murphy, R., Kogge, P., Brockman, J., Brightwell, R., & Underwood, K. (2003). *Implications of a PIM architectural model for MPI.* Paper presented at

the IEEE International Conference on Cluster Computing, Los ALamitos, CA, USA.

Shalf, J., Kamil, S., Oliker, L., & Skinner, D. (Artist). (2005). *Analyzing Ultra-Scale Application Communication Requirements for a Reconfigurable Hybrid Interconnect*.

Susukita, R., Ando, H., Aoyagi, M., Honda, H., Inadomi, Y., Inoue, K., et al. (2008). *Performance prediction of large-scale parallel system and application using macro-level simulation.* Paper presented at the SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA.

Underwood, K., Levenhagen, M., & Rodrigues, A. (2007). *Simulating Red Storm: challenges and successes in building a system simulation.* Paper presented at the 2007 IEEE International Parallel and Distributed Processing Symposium (IEEE Cat. No.07TH8938), Piscataway, NJ, USA.

VampirTrace. from http://www.tu-dresden.de/zih/vampirtrace

Wang, D., Ganesh, B., Tuaycharoen, N., Baynes, K., Jaleel, A., & Jacob, B. (2005). DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News, 33*(4), 100-107.

Zheng, G., Wilmarth, T., Jagadishprasad, P., & Kale, L. (2005). Simulation-based performance prediction for large parallel machines. *International Journal of Parallel Programming, 33*(2-3), 183-207.

---

[i] Corresponding Author. Email:cljanss@sandia.gov