

# A Framework for End-to-end Simulation of High-performance Computing Systems

<b>Wolfgang E. Denzel</b> IBM Zurich Research Laboratory Säumerstrasse 4 8803 Rüschlikon, Switzerland +41 44 724 8516 wde@zurich.ibm.com	<b>Jian Li</b> IBM Austin Research Laboratory 11501 Burnet Road 904/6C-018 Austin, TX 78758, USA +1 512 838 8285 jianli@us.ibm.com	<b>Peter Walker<sup>1</sup></b> Open Grid Computing, Inc. 4030 W. Braker Ln. STE 130 Austin, TX 78759, USA +1 512 343 9196 peter@vircion.com	<b>Yuho Jin<sup>1</sup></b> Computer Science Department Texas A&M University, College Station, TX 77843-3112, USA +1 979 845 5439 yuho@cs.tamu.edu
---	---	---	---

## ABSTRACT

We present an end-to-end simulation framework that is capable of simulating High-Performance Computing (HPC) systems with hundreds of thousands of interconnected processors. The tool applies discrete event simulation and is driven by real-world application traces. We refer to it as MARS (MPI Application Replay network Simulator). It maintains reasonable simulation details of both the processors in general and specifically the interconnection network. Among other things, it features several network topologies, flexible routing schemes, arbitrary application task placement, point-to-point statistics collection, and data visualization. With a few case studies, we demonstrate the usefulness of this tool for assisting high-level system design as well as for performance projection and application tuning of future HPC systems.

## Categories and Subject Descriptors

C.2.1 [Computer Systems Organization]: Computer-communication networks – *Network architecture and design*

C.4 [Performance of systems]: Design studies, modeling techniques

I.6.8 [Simulation and Modeling]: Type of Simulation – *Discrete event, parallel*

## General Terms

Performance, Design, Experimentation.

## Keywords

High-performance computing, end-to-end simulation, interconnection network.

## 1. INTRODUCTION

The next generation of High-Performance Computing (HPC) systems will be distributed systems with hundreds of thousands of processing nodes interconnected via large packet-switched interconnection networks. In the design and development of such

new systems, accompanying performance modeling by simulation is indispensable to evaluate the system design options and to help optimize the performance of the processors, the interconnection network and eventually the entire system, including software and HPC applications. Simulation of such huge systems is challenging and needs new approaches.

On the processor side, there are established methods and tools to simulate complete systems including the applications running on the processors. Execution-driven full-system simulation is applied by modeling processors and applications very accurately at a degree of detail down to the clock and instruction level (e.g. MAMBO [1], SIMICS [2]). In this way, precise application execution times, IPCs (Instruction per Cycle), cache miss rates, etc. can be obtained, which are the preferred performance measure for computing systems. Although one can simulate individual processors or small clusters of processors in this way, the degree of detail does not allow the scaling of this kind of simulation to HPC systems of many thousands of interconnected processors: This would be too time- and resource-consuming, if not even practically impossible.

On the other hand, there is the well-established field of switch and network simulation, in particular in the telecom area. In this field, discrete event-driven simulation is typically applied at a higher degree of abstraction by modeling switches or networks of switches as queuing systems at packet-level granularity. Hence such network simulations can reasonably scale to thousands of network ports on reasonably-sized computers. However, there the main intention is to obtain throughput and delay statistics for synthetic statistical traffic models. While this might be sufficient for telecom applications, it is not suitable for the very different characteristics of HPC interconnect traffic or for obtaining application benchmarks.

In a first phase of system design, the individual simulation methods for processors and for switches or interconnection networks still are very good means for optimizing the respective subsystems. However, separately optimized subsystems may not necessarily yield an overall system optimum from the performance and cost point of view. Furthermore, the impact of different network topologies, switch parameters, link and network parameters, and routing, flow control, congestion control and deadlock prevention algorithms on the run time of real-world HPC applications cannot be studied by separate simulations of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SIMUTools '08*, March 3-7, 2008, Marseille, France.

Copyright 2008 ACM, ISBN 978-963-9799-20-2...\$5.00.

<sup>1</sup> This work was performed while the author was with the IBM Austin Research Laboratory.

processors and network. The same holds for different ways to place application tasks to system nodes. All these network-related aspects are becoming increasingly important so that the network performance can keep up with the processor performance under proliferating system size. Too little attention has been paid to these network-related aspects in prior simulation work that focused on the processor side. Hence, end-to-end simulation with sufficient details on both the processor side (including real HPC applications) and the network is desirable for a second phase of system optimization.

However, large-scale end-to-end simulation of HPC systems running benchmark applications on hundreds of thousands of processors communicating across a large interconnection network is a challenge at the same level of detail as is required for system design and development. This requires the right level of abstraction, an appropriate, efficient “light-weight” simulation tool—a tool that can flexibly cope with the numerous design alternatives that may arise during system design, and a tool that allows distributed parallel simulation to cope with the very large scale.

The MARS framework is the result of our effort towards a full-system end-to-end simulation of versions of the PERCS<sup>2</sup> HPC architecture. We decided to build on an existing, event-driven network simulation environment we had previously developed and used for switch and network simulations in telecom applications. Based on the efficient and flexible OMNEST (also known as OMNeT++ [3]) framework, this simulator allowed the simulation of multistage fat-tree or mesh-type packet-switching networks driven by statistical traffic at the appropriate level of detail. We extended this tool to support end-to-end coverage by replacing the existing statistical packet generators with a new abstract computing node model that is driven by real-world application traces. For the larger system sizes, we newly exploited the OMNEST parallel simulation capability.

As the Message Passing Interface (MPI) standard is pervasively used in HPC applications, our application traces are MPI traces, i.e., traces of the MPI calls in the application software. The trace files are recorded per task of the application on a real system that should be similar to the target system, e.g., a previous-generation system. Alternatively, trace files can be generated synthetically based on deep application knowledge. Our simulator allows arbitrary placement of the tasks onto the system nodes, i.e., the replay of a particular task trace file can be associated to any arbitrary computing node in the system. The nodes replay the associated trace files by generating the appropriate semantic actions, which eventually cause I/O messages to be sent or received via the interconnection network or computing time to be spent in the node itself. The computing time is determined by parameters that account for the processor differences between the system traced and the simulated target system. To precisely determine these parameters requires the expertise of the processor developers and/or comparisons with detailed full-system simulations (MAMBO) of a single target processor. Packets injected into the network experience the full effects of the network protocols under investigation. The network time is determined by the

---

<sup>2</sup> As part of the High Productivity Computing Systems (HPCS) initiative sponsored by the Defense Advanced Research Projects Agency (DARPA), IBM is researching and developing the IBM PERCS—productive, ease-to-use, reliable computing system—for implementation by the year 2010.

resulting queuing times and link delays in the simulated network. By replaying application task traces in a semantically correct way—as opposed to just pushing static traffic traces into the network, our model accounts for the impact of the network loop, i.e., responses from peer tasks are waited for to unlock subsequent transmissions reactively. It is difficult, however, to precisely differentiate potential effects of waiting-time-dependent application behavior, which is a well-known issue for trace-driven simulation.

To be useful for the high-level system design of future HPC systems as well for application tuning, the MARS framework maintains an appropriate level of trace-driven simulation details of both the processors in general and specifically the interconnection network. Among other things, it features several network topologies, flexible processor, switch and network adapter models, a rich set of routing schemes, arbitrary application task placement, point-to-point statistics collection, and data visualization. First applications of our tool allowed us to provide useful feedback to system designers. In the following, we demonstrate this with a few case studies. Thanks to the possibility to simulate full-size systems by parallel simulation, we were able to validate the correct full-system function and determine the full-system performance, which in turn enabled the validation of analytical performance estimates. We have simulated up to 65,536 nodes, each with eight processor cores, on a 32-way SMP cluster. We believe that even larger simulations are possible.

Our paper is organized as follows. In Section 2 we describe the simulation methodology and the underlying simulator framework in more detail. In Section 3 we present exemplary results from four case studies to illustrate the capabilities of our simulation environment. Because of space limitation, we skip detailed quantitative descriptions of the simulated system configurations and application characteristics. Instead we focus on qualitative results. In Section 4, we briefly discuss related work, followed by conclusions and a brief outlook in Section 5.

## 2. SIMULATION METHODOLOGY

### 2.1 Simulation Framework

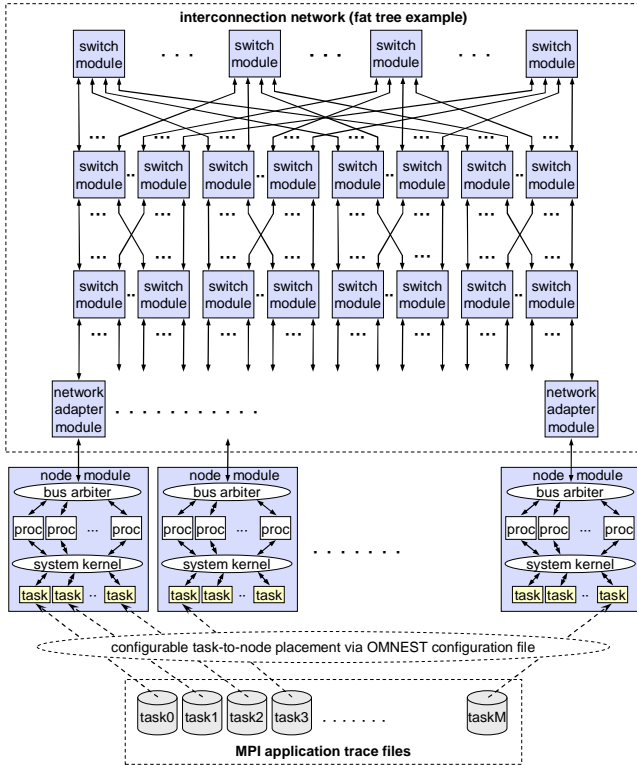
The MARS simulation framework comprises the following set of our own and/or third-party components:

- the OMNEST discrete event simulation core with parallel simulation support,
- network modules describing the overall system model topology, i.e., its subcomponents and the way these are interconnected,
- pluggable modules modeling the subcomponents of the HPC system, i.e., switches, network adapters and computing nodes,
- tools for application-trace collection and/or trace synthesis,
- data/result tracing modules and visualization tools.

In the following we describe the simulation framework, its components and further aspects of the simulation methodology in more detail by using an underlying system model of an exemplary HPC system with a fat tree interconnection network.

### 2.2 Simulation Model

Figure 1 illustrates a high-level overview of the exemplary HPC simulation system model. For the network part, our simulation framework builds around two basic, flexibly configurable and replicable OMNEST modules, i.e., a switch module and a network



**Figure 1. HPC system model example with fat-tree interconnection network**

adapter module. These two basic modules are designed in such a way that any arbitrary interconnection network topology can be flexibly arranged in an OMNEST network description by explicit or algorithmic specification of the connections between multiple instantiations of the switch and network adapter modules. Over time we have created a set of network descriptions for the major network topologies under discussion in the HPC community and in our own projects. For illustration purpose, Figure 1 shows a particular three-level fat-tree topology. Our corresponding OMNEST network description is defined universally for fat trees up to a reasonable maximum number of fat-tree stages. Using conditional module array sizes and conditional connections allows us to parameterize the actually desired number of levels.

The origin of our network simulator was limited to synthetic statistical traffic. For this purpose, each network adapter was fed by a statistical packet generator module. Certainly, this continues to be useful for modeling random-access benchmarks, such as the GUPS (Giga updates per second) benchmark [4]. On the other hand, our new extension for end-to-end simulation replaces the original packet generator module with a new, generic computing-node module that is driven by application traces. To achieve simulation scalability, this node module is confined to a reasonable level of abstraction. As illustrated in Figure 1, each node module is a compound OMNEST module that models multiple processor cores interconnected by a bus model. Furthermore, one or multiple application tasks can run concurrently on the node’s processors. This is controlled by a system kernel. The corresponding task modules perform semantic actions driven by an application trace file. In the OMNEST configuration file, each task of an application, i.e., its associated trace file, can be

assigned to a task module of an arbitrary node module. Thereby one or multiple tasks can be placed onto the same node. Optionally, each task of an application can be placed to multiple nodes. In this way, multiple simultaneously running partitions of the same application can be modeled.

Obeying the causality between messages, the task modules replay the associated trace files by generating the appropriate semantic actions that eventually cause computing time to be simulated in the node itself or MPI messages to be sent or received via a network adapter across the network of switches. MPI messages are sent to the ingress part of the network adapter, where they are segmented into smaller network packets referred to as flits. The flits traverse the switch modules in multiple hops before they eventually reach the egress part of the network adapter, where they are reassembled into the original MPI messages. These are forwarded to the receiving node module, where MPI receive actions are scheduled to wait for them.

## 2.3 Model Subcomponents

### 2.3.1 Switch module

The switch module has a flexible combined input- and output-buffered architecture that can be configured into most popular switch architectures by parameterization. The size of the switch, the number and arrangement of logical queues, buffer sizes, scheduling options, the number of virtual circuits and priority classes, port speeds, or the internal speedup and delays are examples of switch parameters. Further functions supported are credit-based flow control, numerous routing options, and deadlock-prevention mechanisms. In simulations accompanying the development of new switches, it frequently happens that new functions need to be added or others changed. This is flexibly possible in OMNEST because the lowest-level simple module functions are programmed in C++. For other specific cases we have also simplified switch models, for example, a generic InfiniBand switch module.

### 2.3.2 Network adapter module

The network adapter module consists of an ingress and egress part. Similar to the switch module, most realistic adapter architectures can be configured by parameterization. The number and arrangement of logical queues, buffer sizes, scheduling options, and link speed are examples of network adapter parameters.

The ingress part of the network adapter performs the segmentation of MPI messages into network packets, the flits. Correspondingly, the egress part performs the reassembly of flits into MPI messages. Optionally, a re-sequencing function is provided to account for the fact that flows of packets can get out of sequence in multi-path network topologies. Understanding the overhead of re-sequencing is important because it may destroy expected gains of sophisticated multi-path routing schemes.

### 2.3.3 Node module

As described above, the node module provides a model for multiple processor cores and multiple trace-driven tasks running on the processor cores. There are a number of parameters that determine the processor speed, such as the system bus bandwidth, the number of memory controllers, HWMMIO (Hardware Memory Mapped IO) latency, and system call latencies relative to the parameters of the real system on which the application traces

are recorded. One may also set these parameters to project a future system configuration.

An interesting option is to set the processor speed to infinite. In this way, runtime results are obtained that exclusively contain the portion of time spent for I/O and in the network. This allows the impact of the network to be crystallized out and enables good comparisons between different network options. Furthermore, based on the comparison to the runtime with the real, finite processor speed, one can judge the balance between processing and networking, i.e., whether a system is processing-limited, network-limited, or balanced.

### 2.3.4 Task module

The key submodule of the node module is the task module. Its high-level architecture is illustrated in Figure 2. A trace reader function reads the lines from the trace file that is assigned to this particular task and node. As there might exist thousands of trace files that need to be accessed constantly, we apply a caching mechanism to this trace-reading function.

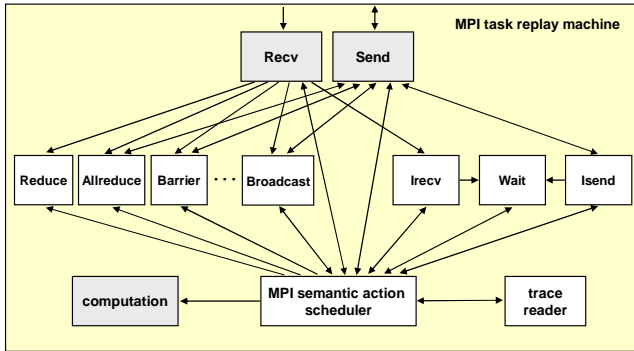


Figure 2. Task module architecture

From a read trace line, a semantic action scheduler determines which kind of MPI-call-related module needs to be called or whether computation time needs to be simulated. For each MPI call type that may occur, a corresponding module must be provided. Point-to-point MPI calls directly involve a *Send* or *Recv* operation to be performed. A *Send* operation eventually causes an OMNEST message to be created that represents the MPI message. The message size is determined by the size indicated in the trace line. Because the target of an MPI message is another task rather than a network address as required for routing in the network, the destination network address for the message created has to be determined by a specific function that finds the network address of the node module the target task is placed to. On the other hand, a *Recv* operation waits for actual reception of a corresponding message. In the case of MPI collectives, the corresponding modules simulate what the MPI library supposedly does: For example, a *Broadcast* operation might be decomposed into multiple sequential point-to-point actions.

For the MPI application traces we have studied so far, we only needed to support about a dozen different MPI calls. In general, however, there are two problems with the approach. One is that MPI defines many dozens of MPI call types, and it simply is a lot of work to write all the corresponding modules, some of which may never be needed. Second, a specific module implementation might not be appropriate if another MPI library is used. The efficiency of different MPI library implementations varies, and

some are optimized for the specific network topology of the system they are provided for. Hence, as far as possible, we try to obtain traces that are not taken at the usual MPI call level but rather at a lower level, where eventual point-to-point messages can be recorded. This has the advantage that MPI-call-related modules are no longer needed in our task module. Second, the simulator implementation remains independent of the specifics of the individual MPI library implementations, although the simulation results still reflect the differences in the MPI software implementations. In this way, the simulator becomes a helpful tool also for the software design of the MPI library for the target system.

## 2.4 Routing Function

For routing, our network simulator framework supports either hop-by-hop or source routing. In the case of hop-by-hop routing, the route determination is performed step-by-step in the switches, i.e., each switch determines through which port it has to route a given packet. In the case of source routing, the route determination is performed in the network adapter once for the entire route from source to destination. The sequence of route hops, also referred to as source route, is then carried in the header of the network packets so that each switch can seize its route decision from the packet header and act accordingly.

In both cases, the actual route decision can be based on algorithms or on routing tables. For the network topologies of interest, we have written specific linear routing algorithms that can be plugged into the switch module when a particular topology is being studied. The advantage of such algorithms is that they can be made to reflect exactly the specifics of the routing in the real system. The drawback is that they have to be written for every new topology or class of topologies. Alternatively, or generally, routing tables as are applied in many real systems can be used. For this case, we use a central control module that can read a routing table file and forward the corresponding table portions to all switch modules (for hop-by-hop routing) or to all adapter modules (for source routing) in the network.

If neither routing algorithms nor routing tables are available or before they become available, the central module of our simulator can also generate a routing table file itself for any arbitrary network topology by using the OMNEST topology exploration concepts. From each switch (for hop-by-hop routing) or each adapter (for source routing) to each destination, the unweighted single shortest-path function is applied to the topology object that incorporates all switches and network adapters. This function is based on the known shortest-path algorithm by Dijkstra. For lack of a weighted shortest-path function in OMNEST, we optionally place one or multiple dummy modules onto network links and incorporate these in the topology object. These dummy nodes have no function other than counting as a node in the OMNEST topology object. In this way we can model a limited weighed shortest-path algorithm by using the existing Dijkstra algorithm. Furthermore, as there might exist multiple shortest paths, we need to find all of them. For lack of a multiple shortest-path function in OMNEST, we do the following. We determine the length of the single shortest path from a specific source switch. Then we apply the single path algorithm iteratively, starting from all neighbor switches the ports of that specific switch lead to. Then, all shortest paths found that have a length that is shorter by one than the initially determined length are alternative shortest paths.

In addition to having our simulator generate routing tables, we can use tables generated externally by other means or tables modified manually.

If multiple paths exist, independently of whether they are determined by a table or by a specific algorithm, we have to apply another step that selects one of the many possible paths. For this we have numerous predetermined options, ranging from static via random or round-robin path selection to state-dependent adaptive schemes based on shortest queues or highest number of available flow-control credits. Note that adaptive routing makes most sense with hop-by-hop routing, which was our motivation to provide hop-by-hop routing. Otherwise, source routing might be desirable because many actual systems apply source routing. However, in many cases, hop-by-hop and source routing are logically identical.

If virtual channels are used, routing may also include a final step of selecting the right virtual channel for the next hop. Specific algorithms for this are needed in certain topologies (although not in fat-tree networks) to prevent cyclic dependency deadlocks.

## 2.5 Task Placement

The task-to-node placement is provided by assigning the number identifying the task to a taskRank parameter of a particular node in the OMNEST configuration file. This can be done explicitly task-by-task such as

```
**.node[0].taskRank = 0
**.node[4].taskRank = 1
**.node[8].taskRank = 2
```

While this allows a truly arbitrary assignment of tasks, it might become tedious for a regular assignment of a large number of tasks. To simplify this, we alternatively use a function assignTask() (defined in the OMNEST distributions file) that automatically assigns all tasks of an application to nodes in a predetermined way depending on the parameter of this function. In this way, only one configuration line is necessary, such as

```
**.node[*].taskRank = assignTask(<parameter>)
```

where <parameter> is a number identifying one of various predetermined schemes, such as sequential assignment (task  $i$  to node  $i$ ) or randomly shuffled assignment. With the help of one additional parameter we can control the assignment of multiple tasks to one node.

## 2.6 Application Traces

From various laboratories and supercomputer centers, we have obtained a number of popular HPC application traces of different sizes (number of tasks) from various application fields, such as ocean modeling (HYCOM, POP), weather research and forecast (WRF), shock-wave physics (CTH), and molecular dynamics, fusion and transport physics (AMBER, CPMD, LAMMPS, GYRO, SPPM, SWEEP3D, UMT2K). Because of their different origins, these traces have more or less comprehensive formats from different tracing tools. An example for such a tracing tool is the Sequoia tool kit [5].

For the purpose of our simulator, we decided to use a simple trace format derived from one of the formats we obtained. Our trace reader can read this format directly. For other trace formats, we decided to write separate programs (Perl scripts) for translating the relevant subset of trace information from the original trace into the format our trace reader understands.

In some cases, traces are very long and unwieldy although the really interesting phase of the trace is relatively short. Then it is necessary to negotiate with the trace provider for some way to identify the interesting phase in the program and only trace this phase. However, it is not trivial to start a trace in the middle of an application. Because of the causal relationship among different tasks, this must happen in a controlled way so that no task waits for messages that have not been traced before.

For another set of applications in the field of weather forecast (DWD, ECMWF T639, ECMWF 4DVar), we use synthesized artificial traces that were generated by application experts with separate programs (Python scripts).

Common to all application traces we have considered so far are very wide message size distributions and similar characteristics in the communication patterns. Message sizes may vary between one Byte and many Megabytes. Figure 3 shows an exemplary message-size histogram measured for the LAMMPS molecular dynamics application. Because of the wide span, the message size range is divided into logarithmically spaced cells in this diagram.

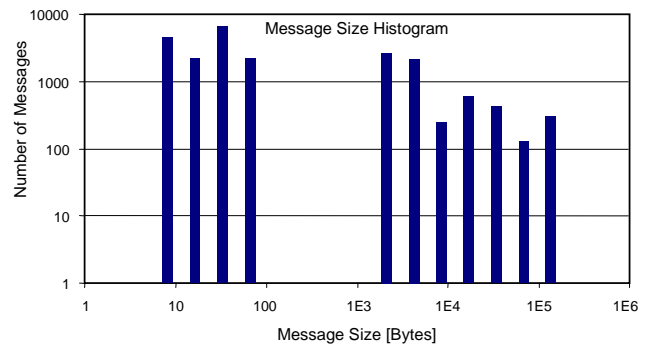


Figure 3. Message size histogram for LAMMPS application

Traffic patterns typically exhibit much more near-neighborhood than far-distance traffic and, when represented in a matrix, generally diagonal patterns. The left diagram in Figure 4 shows a typical example of such a traffic matrix for the LAMMPS application. The high diagonal patterns essentially are made by long messages, whereas the ground floor in this matrix is mainly formed by the short messages. From this characteristic it becomes clear that interconnection network topologies that favor near-neighbor communication are preferable, which leads to the topologies we are considering. However, this only makes sense if the tasks are placed onto the system in sequential order as it is

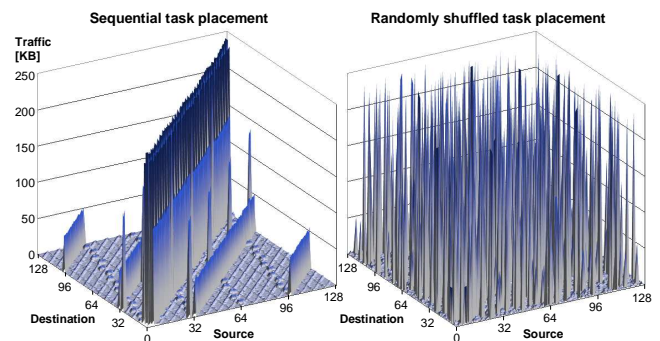


Figure 4. Traffic matrix example for LAMMPS application

done in the left diagram in Figure 4. In contrast, the right diagram shows how the traffic matrix would look if the tasks were placed in a random way onto the system. In this case, the network topology ideally would have to be equally fair to close and far-distance communication patterns. As sequential task placement is not always possible in a real system, we need to be able to study the impact of different task placements.

## 2.7 Parallel Simulation Aspects

OMNEST supports parallel distributed simulation [6] provided that some constraints are obeyed, as outlined in the OMNEST manual or in [7]. The network must be partitioned into segments that can be physically distributed. For this it is useful to define the OMNEST network description as a structure of submodules rather than as one flat network of basic modules. Our basic modules are designed in such a way that they can be arbitrarily nested. For example, relative references such as to the parent module are avoided. The network submodules or sets thereof can then serve as the parallel partitions. In some architectures, the partitioning is naturally given by the physical packaging of the system. An example of this is the hierarchical direct interconnect system shown in Figure 6. In this model, the connectivity in hierarchy level 1 is essentially on-board connectivity, in hierarchy level 2 board-to-board connectivity, and in hierarchy level 3 rack-to-rack connectivity. In parallel simulations of this model, we used parallel partitions of small sets of racks.

The requirement for a look-ahead time by the OMNEST parallel simulation feature is sufficiently fulfilled by the fact that the only connections between physical partitions are network links, which are modeled realistically with a finite delay in any case.

In the process of preparing the simulator for parallel simulation, some caution was required where module or gate objects nested in other sub-modules are accessed. Some OMNEST APIs return the intended object in a sequential simulation, but will return placeholder modules or proxy gates if the target object is in another partition of a parallel simulation. Similarly, when working with OMNEST topology objects, a topology object of switch modules, for example, contains all switch modules of the entire system in a sequential simulation. In a parallel simulation, it contains only the switch modules that are inside the current partition. Understanding and circumventing these facts were essential for successfully simulating in parallel.

## 2.8 Simulation Data Collection

While flits traverse the network, we record in each traversed module various information such as queuing times, time stamps or hop counts inside the OMNEST messages that represent the flits. Once the flits reach the egress side of the network adapter, all kinds of statistical results can be collected from the information carried in the flits. Overall system-wide statistics are collected by default. Point-to-point statistics (i.e., per source/destination pair) can be collected on demand for measures of interest. The statistics collection is performed in a separate statistical measurement module. In case of a sequential simulation there is only one such statistical measurement module, whereas in a parallel simulation we apply decentralized statistics collection by placing one statistical module in each partition. The results collected in this way in a decentralized manner are then sent periodically to the single central control module, where they are consolidated and printed as optionally periodic and eventually as final results.

Periodic results allow the observation of changes of certain measures during the run time of an application (see Section 2.9). The granularity of measurement periods is a parameter. In large systems, some caution is necessary in determining the period and the measures of interest to prevent the generation of impracticably large amounts of data.

## 2.9 Simulation Data Visualization

Simulation data visualization helps understand the communication patterns of MPI applications at run time. Our simulator enables trace collection of point-to-point (P2P) and collective operations among the nodes. For each simulation step of the application, we collect internode I/O operations (the number of send/receive operations), mean message size, standard deviation and total bytes transferred. We also collect this information for collective operations and for a combination of P2P and collective. Furthermore, we track when load balancing is initiated. We also collect data for the case when multiple tasks are on a node. The amount of I/O versus data that leaves the node is a useful indication of how the network load changes when SMP nodes are used.

Such data can then be plotted into frames of pictures, which can be sequenced to animate the run-time communication operations in a movie-like manner. Figure 5 shows a sample frame of the animation of I/O operation distribution in the CTH application, a popular shock-wave physics program developed by Sandia National Laboratories, USA. With data visualization, one can easily spot where the network hotspots are. Then optimal task placement and adaptive routing can be used to fine-tune the system.

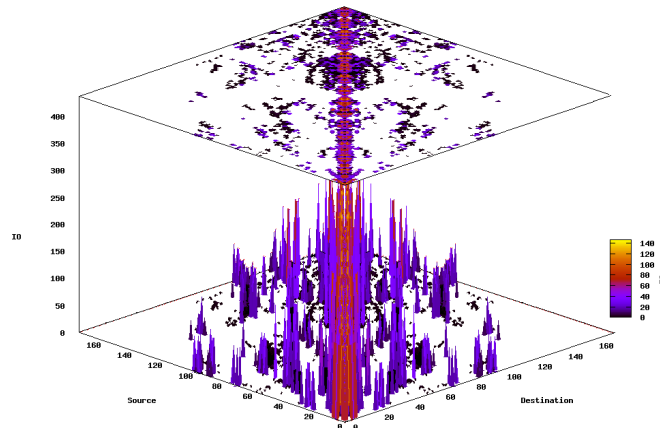


Figure 5. Point-to-point I/O operations of CTH application with 176 tasks

## 2.10 Simulator Portability

The parallel version of our simulator runs on a cluster of SMP machines with the Parallel Operating Environment (POE) of the AIX<sup>3</sup> operating system. The simulator also runs on x86 machines with either Linux<sup>4</sup> or Windows<sup>5</sup> operating systems. It is even possible to run up to about 1000 nodes on a simple

<sup>3</sup> AIX is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

<sup>4</sup> Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

<sup>5</sup> Windows is a registered trademark of Microsoft Corporation in the United States, other countries, or both.

notebook computer. The sequential version of our simulator also works on those platforms. We believe the simulator can also be ported to other hardware and operating systems.

### 3. CASE STUDIES

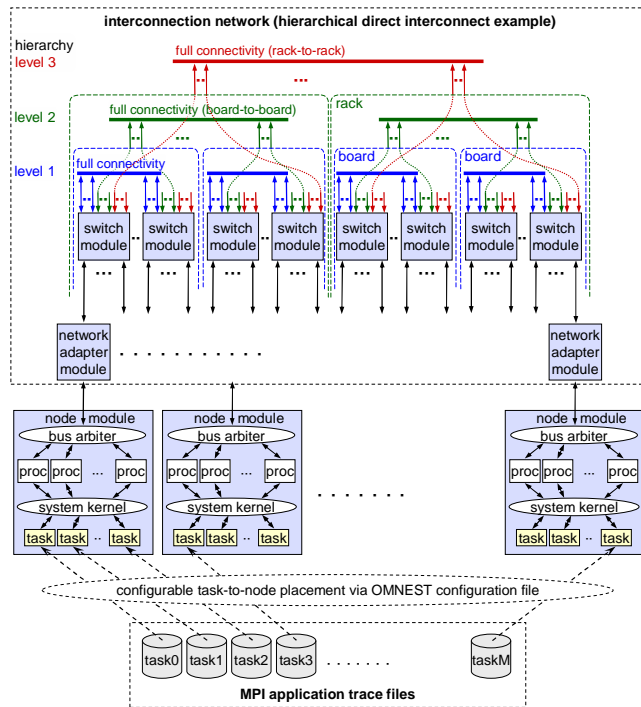
#### 3.1 Interconnect Design

One of the main goals of our simulation environment is to support the design of the interconnection network for an HPC system under development. Simulations of all the options under discussion help decide on the appropriate network topology, switch architecture, routing protocol, and so on. In this context, we applied our tool to study HPC systems with fat-tree interconnection networks, as shown in Figure 1, and variants of this architecture. Examples of questions in this context are:

- What is the optimum size of switch modules and number of fat-tree levels?
- What is the required link bandwidth?
- How should the switch buffers be dimensioned?
- How should the multi-path property of the fat tree be utilized in the routing protocol?
- Can the number of switch modules in higher fat-tree levels be reduced?

A case study related to the last question is exemplarily addressed in Section 3.2 below.

An alternative interconnect architecture we studied is the hierarchical direct interconnect architecture as illustrated in Figure 6. In this architecture, each computing node is assumed to be a multi-core chip, i.e., chip multiprocessors (CMPs). A switch module is associated with one or a few computing nodes. A small set of switch modules are directly and fully interconnected via a first hierarchy level of interconnection links (level-1 links)



**Figure 6. HPC system model example with hierarchical direct interconnection network**

forming first-level groups, e.g., boards. Multiple boards are directly and fully interconnected via a second hierarchy level of interconnection links (level-2 links) forming second-level groups, e.g., racks. Finally, multiple racks are directly and fully interconnected via a third hierarchy level of interconnection links (level-3 links) forming the third-level group, which is the full system. A property of this architecture is that routing between arbitrary nodes may involve multiple intermediate hops via links of the different hierarchy levels. For example, to reach a destination in another rack, a level-1 link hop may be required to reach the appropriate switch module that has a level-2 link to the right first-level group (board) that has a level-3 link to the right second-level group (destination rack) and vice versa on the destination rack. Nonetheless, such a hierarchical complete-graph topology tries to balance the number of hops to reach any other node and the total number of links required to build the network. Examples for questions regarding this architecture are:

- What is the required link bandwidth for each hierarchy level?
- How should the switch buffers be dimensioned per link type?
- How can cyclic dependency deadlocks be prevented?
- What is the routing protocol? Should only direct/shortest routes be used or can indirect/longer routes improve performance?

A case study related to the last question is exemplarily described in Section 3.3 below.

#### 3.2 Fat Tree Case Study

An ideal full fat-tree network (Figure 1) provides the full bisectional bandwidth in all fat-tree levels. In other words, the number of links and the aggregate link bandwidth between levels are constant, and the number of switch modules is the same in all levels except that in the highest level only half as many are needed (here all switch ports are used as downward ports). As the number of levels that must be traversed is a function of the source/destination distance, and HPC applications typically have more near-neighbor than far-distance communication, bandwidth and cost could eventually be saved by reducing the bandwidth or the number of switch modules in the higher levels. The question is how far can this reduction go without significantly impacting performance?

To study this, we performed a series of end-to-end simulations of a fat-tree-based HPC system using some real HPC application traces. Gradually we reduced the bandwidth of each fat-tree level by 25%, 50% and 75% relative to the preceding level. In addition, we studied the impact of different switch module sizes and task-to-node placements.

Figure 7 shows some exemplary results obtained from simulations with one particular HPC application (LAMMPS, a molecular dynamics simulator). The results for other applications are conceptually very similar.

The diagrams show the measured mean system delay of flits as a function of the bandwidth reduction relative to the preceding fat-tree level. The upper diagram is for sequential task-to-node placement (task  $i$  to node  $i$ ), the lower diagram for randomly shuffled task placement. Three different switch system sizes are considered. The tables inserted list the percentages of traffic (number of flits) that traverse up to each of the fat-tree levels.

As we can see, a bandwidth reduction of up to 50% appears feasible if the switch modules are larger than  $8 \times 8$ . The larger the

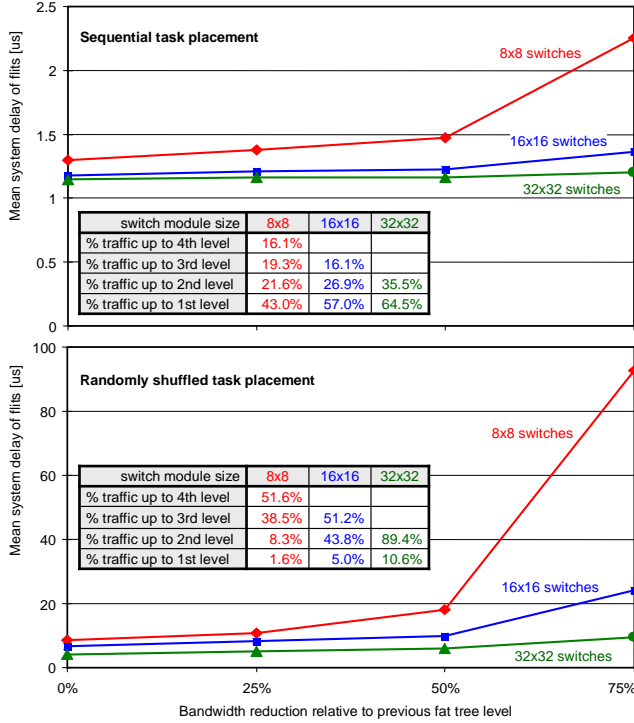


Figure 7. Fat-tree results for LAMMPS application

switches, the more bandwidth reduction is tolerable. This relative behavior appears to be independent of the task placement, although the absolute system delay is significantly worse for random task placement. The tables show that a large portion of the traffic travels up all the levels for random task placement, whereas for sequential placement it is the opposite. In fact, sequential task placement has turned out to be the optimum on fat trees for about a dozen of scientific HPC applications we have simulated so far.

### 3.3 Indirect Routing Study

In the hierarchical direct interconnection model (Figure 6), there is only one shortest path between any pair of nodes on different racks. The single highest-hierarchy link (rack-to-rack link or level-3 link) on this direct route could become a bandwidth bottleneck. More bandwidth could be made available by using additional alternative routes that indirectly lead via third racks. However, these indirect routes have a longer delay because they include more hops, particularly two rack-to-rack hops. Furthermore, alternative indirect routing involves a complexity and cost issue. The question is whether this is worth the effort and under which circumstances.

To study this, we performed a set of simulations based on the hierarchical direct interconnection model. We configured a system of five racks and placed an HPC application across the nodes of two of the racks: Half of the application tasks were placed on the nodes of one rack, and the other half on the nodes of the other rack. Hence, when using only direct routes in a first simulation, half of the tasks have to talk to the other half of the tasks via a single rack-to-rack link. In three more simulation runs, we then allowed additional indirect routes via one, two or three other racks. To have background traffic on these “detour” racks, we placed additional application partitions on each of them.

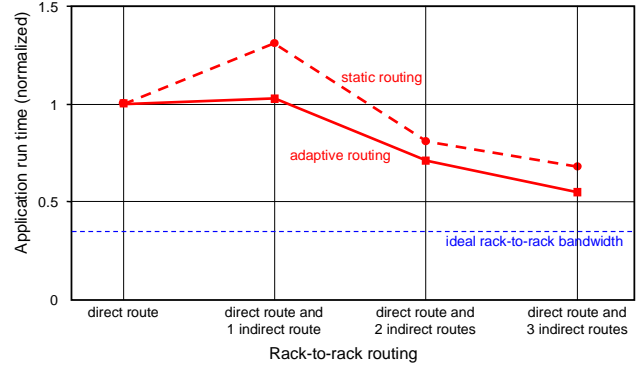


Figure 8. Indirect routing results for LAMMPS application

Another consideration is that if unbalanced direct and indirect routes are used simultaneously, routing with static route selection would distribute the traffic equally among the different paths that might not be the optimum paths. Hence, we also considered the option of adaptive routing that selects the various possible paths based on switch-queue occupancies.

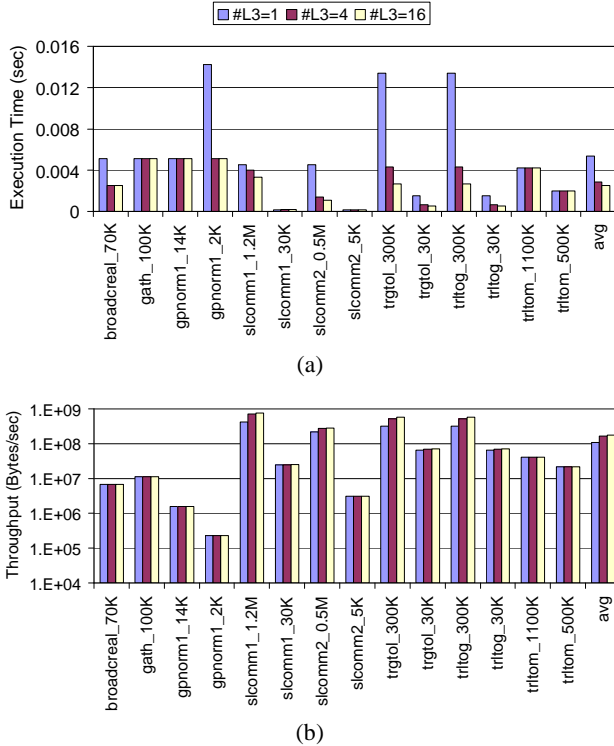
Figure 8 shows application run-time results obtained from simulations with the LAMMPS application, which are again conceptually similar to results for some other applications we tested. The diagram shows the application run time normalized to the run time of the case when only direct routes are used. Furthermore, we applied here the feature of setting the processor speed to infinite to see the pure impact of the network. Hence the application run time is exclusively the time spent for I/O and in the network. We can see that allowing one additional indirect route to the direct route does not yet provide an improvement. On the contrary, for static routing, half of the traffic through the long indirect route is too much and causes a deterioration that can hardly be compensated if adaptive routing were applied. However, for more than one additional indirect route, significant improvements can be achieved in any case, in particular when combined with adaptive routing.

For comparison, we made another simulation in which we artificially increased the bandwidth of the rack-to-rack links by using many of them in parallel so that the system bottleneck is shifted to parts of the system inside the racks. The result of this case yields a kind of ideal bound, indicated by a dashed blue line at the lower portion in the diagram. From this, we can expect that providing the capability for many additional indirect routes will probably not be worth the effort.

Certainly, for a definitive answer on the value of indirect routing, more studies are necessary. Indirect routing may hurt the performance of some other applications, as it uses more links than direct routing for each route, and hence may reduce the effective interconnect bandwidth and increase latency for other applications.

### 3.4 GUPS Benchmark Study

The random access GUPS (Giga updates per second) benchmark from the HPC Challenge suite [4] is a very important performance measure for HPC systems. Although it does not really require trace-driven simulation and the GUPS performance can eventually be obtained analytically, GUPS simulations turn out to be very useful for several reasons. For lack of large enough application



**Figure 9. Run time and throughput of MPI communication patterns in ECMWF in the three-level hierarchically fully connected interconnect (Figure 6). #L3 means the number of level-3 links.**

traces, a GUPS simulation with synthetically generated traffic allows the simulation of the full-scale system. Thereby the parallel simulation scalability can be tested without a need for application traces. Furthermore, effects can be studied and validated that only occur at very large system sizes. Finally, GUPS simulation results can validate the correctness of analytical predictions as well as the correctness of large parts of the simulator.

We have performed parallel GUPS simulations of the hierarchical direct interconnection model (Figure 6). Thereby we were able to validate predicted analytical results up to a system scale of 65,536 nodes. This included the validation of a predicted discontinuity effect at a very large system size above which the second-level connectivity becomes the system bottleneck, whereas below that size the third-level connectivity was the system bottleneck. Furthermore, we observed that the overall GUPS throughput tends to stabilize sooner than individual port buffer and link utilizations across the interconnection network for the uniform GUPS traffic patterns in a large-scale system.

On the one hand, we observed a surprisingly good speedup by parallel simulation. We distributed a small system that originally ran on a single processor onto a 16-node cluster of the same processors. Thereby we observed a superlinear speedup on the order of 17 thanks to the 16-fold aggregate cache capacity and efficient model partitioning. On the other hand, GUPS simulation of a large-scale system of 65,536 nodes certainly requires considerably more computing resources. For example, we used a 32-node cluster with 512 GByte aggregate memory for the full-system simulation. It takes about one hour wall-clock time to

simulate ten microseconds at high GUPS injection rate (i.e., in system saturation). Low injection rate or large packet size simulations run faster.

### 3.5 MPI Application Performance Projection

We have used our simulator to project the performance of some mission-partner MPI applications running on a projected future supercomputer system. One of the applications is ECMWF (European Centre for Medium-range Weather Forecasts). ECMWF is a complicated application with various MPI communication patterns. It has proved to be network-intensive on some existing systems.

We break down ECMWF into its representative MPI communication patterns and simulate them on the three-level hierarchically fully connected interconnect (Figure 6). We compare the execution time and throughput of these MPI patterns of configurations with 1, 4 and 16 level-3 links per pair of level-3 groups (racks). Figure 9 shows the results.

We observe a significant improvement in execution time and throughput when we increase from one to four level-3 links per pair of level-2 groups. However, we see only negligible improvement when we further increase to 16 level-3 links per pair of level-2 groups. The experiment shows that a configuration of four level-3 links per pair of level-2 groups is sufficient to meet ECMWF’s network requirement. Hence, we can downsize the network cost of this application without performance loss. Alternatively, the extra level-3 links can be shut down to reduce power consumption.

## 4. RELATED WORK

Because of space limitations, we only discuss related work that most closely resembles our research.

IBM’s Blue Gene®/L<sup>6</sup> team used an interconnection simulator [8] to model Blue Gene/L’s three-dimensional torus interconnection network during the design phase of the project. They selected a shared-memory parallel simulation approach to interconnect modeling. Message-passing calls of applications are passed to the simulator via traces, which are collected using an IBM unified trace environment trace-capture utility that runs on IBM supercomputer machines. Traces of up to several hundreds of nodes were collected. They used time-driven parallel simulation. In our work, we use the MPI interface to parallelize our discrete-event interconnect simulator. We expect the availability of commodity clusters and the scalability of MPI libraries to make our approach more cost-effective. Our trace synthesizer is able to generate traces for virtually any number of MPI threads. In addition, our simulator features the capability of placing MPI tasks arbitrarily, which can be very useful to tune MPI application on future systems.

BigSim [9] is an interconnect simulator to help optimize applications for larger-scale HPC systems. It uses optimistic synchronization to parallelize the simulation. In our work, we use detailed interconnect models to help also the interconnection network design. We use conservative synchronization to improve development productivity. Furthermore, as conservative synchronization minimizes load imbalance in the simulation, our

<sup>6</sup> IBM and Blue Gene are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

approach can simulate very large-scale systems. In our work, we also collect MPI traces from real machines or synthesize them based on detailed application knowledge.

Dimemas from UPC [10] is an MPI-trace-driven simulator primarily intended to help users develop and tune parallel MPI applications. For that purpose it models the nodes of a parallel machine and the applications running on them at a similar abstraction level as we do. However the network is modeled at a much higher abstraction level in the form of a set of buses. Hence Dimemas is not suited for studying the impact of different network topologies, switch architectures or network protocols, nor would it be useful to help the design and optimization of the interconnection network, the switches and network protocols. A Dimemas simulation produces results in the form of another trace file that serves as input for separate performance analysis tools such as their powerful visualizing tool Paraver.

Sharapov *et al.* [11] presented a performance estimation methodology for an HPC application running on a future HPC hardware architecture. It applies a hierarchical modeling method by combining queuing theory, such as the mean-value analysis model, with cycle-accurate simulation. Their interconnect model is an analytical model, whereas the processor micro-architecture model is cycle-accurate. They analyzed the application behavior with another workload characterization process.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated using our MARS tool that end-to-end simulation is feasible for HPC systems with hundreds of thousands of processors. The tool maintains reasonable trace-driven simulation details of both the processors and the interconnection network. It features several network topologies, flexible routing schemes, arbitrary application task placement, point-to-point statistics collection, and data visualization. With a few case studies, we showed that this tool is very useful for high-level system design, performance projection, and application tuning of future HPC systems.

Currently our models incorporate feedback between the network and the MPI-trace execution by flow control. Saturation trees resulting from heavy congestion would require congestion management that throttles the injection rate. We plan to implement and study congestion control with our tool to help understand this difficult control problem under discussion in the HPC community.

To expand the capabilities for analyzing the large amount of possible simulation results of HPC simulations, we also plan to add a feature to generate result traces in a format that can directly be used by already existing performance analysis and visualization tools, such as the UPC Paraver tool.

Leveraging modeling details and simulation speed of MARS is challenging. Also, minimizing the limitation on interactions between MPI tasks in trace-driven simulation remains a challenge.

## 6. ACKNOWLEDGMENTS

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. Details presented in this paper may be covered by existing patents or pending patent applications.

The authors would also like to thank Lydia Chen, Don DeSota, and Javier Navaridas for valuable discussions and contributions to the work presented in this paper.

## 7. REFERENCES

- [1] Peterson, J. L., Bohrer, P. J., Chen, L., Elnozahy, E. N., Gheith, A., Jewell, R. H., Kistler, M. D., Maeurer, T. R., Malone, S. A., Murrell, D. B., Needel, N., Rajamani, K., Rinaldi, M. A., Simpson, R. O., Sudeep, K., Zhang L. Application of full-system simulation in exploratory system design and development. *IBM Journal of Research and Development*, Vol. 50, No. 2/3, March/May 2006, pp. 321-332.
- [2] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B. Simics: A full system simulation platform. *IEEE Computer*, Vol. 35, No. 2, Feb. 2002, pp. 50-58.
- [3] Varga, A. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'01)*, Prague, Czech Republic, June 2001.
- [4] Luszczek, P., Bailey, D., Dongarra, J., Kepner, J., Lucas, R., Rabenseifner, R., Takahashi, D. The HPC Challenge (HPC) Benchmark Suite. SC06 Conference Tutorial, IEEE, Tampa, Florida, Nov. 2006.
- [5] Vetter, J. S., Bhatia, N., Grobelny, E. M., Roth, P. C. Capturing petascale application characteristics with the Sequoia toolkit. In *Proceedings of the International Parallel Computing Conference (ParCo '05)*, Malaga, Spain, 2005.
- [6] Sekercioglu, Y. A., Varga, A., Egan, G.K. Parallel simulation made easy with OMNeT++. In *Proceedings of the European Simulation Symposium (ESS '03)*, Delft, The Netherlands, Oct. 2003.
- [7] Varga, A., Sekercioglu, Y. A., Egan, G. K. A practical efficiency criterion for the null message algorithm. In *Proceedings of the European Simulation Symposium (ESS '03)*, Delft, The Netherlands, Oct. 2003.
- [8] Adiga, N. R., Blumrich, M. A., Chen, D., Coteus, P., Gara, A., Giampapa, M. E., Heidelberger, P., Singh, S., Steinmacher-Burow, B. D., Takken, T., Tsao, M., Vranas, P. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, Vol. 49, No. 2/3, March/May 2005, pp. 265-276.
- [9] Choudhury, N., Mehta, Y., Wilmarth, T. L., Bohm, E. J., Kal'e, L. V. Scaling an optimistic parallel simulation of largescale interconnection networks. In *Proceedings of the 2005 Winter Simulation Conference (WSC '05)*, ACM, New York, NY, USA, 2005, pp. 591-600.
- [10] Badia, R. M., Labarta, J., Gimenez, J., Escala, F. Dimemas: Predicting MPI applications behavior in grid environments. In *Proceedings of the Workshop on Grid Applications and Programming Tools (GGF '03)*, 2003.
- [11] Sharapov, I., Kroeger, R., Delamarter, G., Cheveresan, R., Ramsay, M. A case study in top-down performance estimation for a large-scale parallel application. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*, New York City, NY, USA, 2006, pp. 81-89.