Facilitating Co-Design for Extreme-Scale Systems Through Lightweight Simulation

Christian Engelmann and Frank Lauer
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
engelmannc@ornl.gov and FrankLauer@gmx.com

***Abstract*—This work focuses on tools for investigating algorithm performance at extreme scale with millions of concurrent threads and for evaluating the impact of future architecture choices to facilitate the co-design of high-performance computing (HPC) architectures and applications. The approach focuses on lightweight simulation of extreme-scale HPC systems with the needed amount of accuracy. The prototype presented in this paper is able to provide this capability using a parallel discrete event simulation (PDES), such that a Message Passing Interface (MPI) application can be executed at extreme scale, and its performance properties can be evaluated. The results of an initial prototype are encouraging as a simple `hello world` MPI program could be scaled up to 1,048,576 virtual MPI processes on a four-node cluster, and the performance properties of two MPI programs could be evaluated at up to 16,384 virtual MPI processes on the same system.**

***Keywords*-high-performance computing; parallel discrete event simulation; hardware/software co-design; performance evaluation; Message Passing Interface;**

I. Introduction

Recent trends in high-performance computing (HPC) architectures show that, due to the end of processor frequency scaling, performance increases are entirely gained by employing more processor cores. For example, the Jaguar system at Oak Ridge National Laboratory (ORNL) has 224,256 cores (see http://www.nccs.gov/jaguar). Its LINPACK benchmark performance of 1.759 PFlop/s performance was achieved by switching from four- to six-core processors in 2009. Prior to that in 2008, it was upgraded from dual-core to quad-code processors and from 11,508 single- to 18,688 dual-processor nodes. Ongoing planning activities for the road toward multi-petascale and exascale HPC recognize that these increases are going to continue for the next 10 years. An exascale computing system may have up to 1,000,000 compute nodes with 1,000 cores per node by 2018 (see the International Exascale Software Project at http://www.exascale.org). While processors in 2018 may look completely different than today's and may employ a significant amount of hardware threads similar to today's graphics processing units (GPUs), the amount of concurrency a HPC application needs to deal with is immense.

Investigating algorithm performance at this scale becomes rather difficult, especially considering the challenge of non-intrusive scalable profiling using tracing tools. Furthermore, current-generation HPC systems provide not nearly enough concurrency to permit performance evaluation at this scale. The research and development presented in this paper focuses on an alternative approach using lightweight simulation of future extreme-scale HPC architectures to evaluate algorithm performance at extreme scale. While simulation approaches have been used in the past, they did not scale to millions of concurrent threads with the needed amount of accuracy. The presented prototype is able to provide this capability using a lightweight parallel discrete event simulation (PDES), such that a Message Passing Interface (MPI) application can be executed at extreme scale and its performance can be evaluated.

II. Related Work

Discrete event simulation (DES) is a powerful tool for investigating the behaviour of complex environments that have entities interacting in discrete intermittency, such as in computer systems, molecular dynamics, and military battle fields. The operation of such systems is represented as a chronological sequence of events, each marking a change of system state at a specific simulated time. A DES allows to stop, compress, expand, and reverse the virtual time of the simulated state machine as it is separated from wall clock time. Simulations of deterministic systems are repeatable.

A PDES [1], [2] is a parallel implementation of a DES. It needs to maintain causality as transitions are executed simultaneously. Common techniques include to execute the simulation (1) in a conservative fashion that waits until all parts are ahead of the next event's virtual time, (2) in an optimistic way that rolls back the entire simulation in case of a causality violation, or (3) using a time warp algorithm that rolls back only those transitions resulting in a causality violation. A PDES also needs to deal with the trade-off between scalability and fidelity as a reduction of simulation accuracy can reduce the probability of causality violation.

The *Java Cellular Architecture Simulator (JCAS)* [3] was initially developed by our team in 2001 and part of a collaborative effort between IBM and ORNL to investigate scalable and fault-tolerant scientific algorithms for HPC systems with up to 100,000 processors, such as the IBM Blue Gene/L system planned at that time. The first prototype was able to run up to 5,000 virtual processes (VP) on a

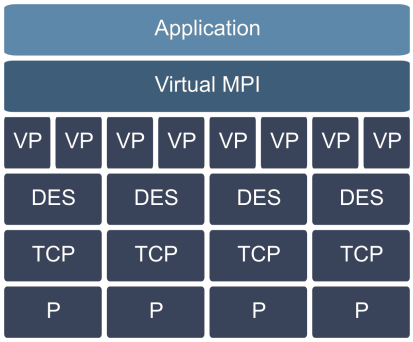

Figure 1. DDES architecture of JCAS

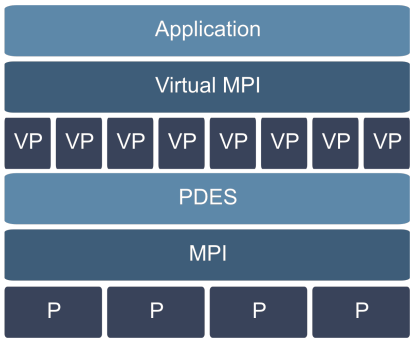

Figure 2. PDES architecture of $\mu\pi$

single native processor (P) solving Laplace's equation. An improved variant ran up to 500,000 virtual processes on a Linux cluster with 5 native processors (1 for visualization and 4 for computation) solving Laplace's equation and the global maximum problem. Its design (see Figure 1) relied on a distributed discrete event simulation (DDES) on top of TCP/IP with a very basic virtual MPI. While this solution was able to run algorithms at scale, it lacked certain important features, such as time-accurate simulation, high performance, support for running the simulator on atop MPI, and a fully functional virtual MPI.

The BigSim [4], [5] project (see http://charm.cs.uiuc.edu/research/bigsim) was initiated in 2001 by the IBM Blue Gene/C project to study programming issues in large-scale HPC systems. The *BigSim Emulator* is build atop Charm++ and Adaptive MPI (AMPI) [6]. It is able to run MPI applications in a highly oversubscribed fashion, such that 100,000 virtual MPI processes are distributed over 2,000 native processors. The BigSim Emulator is meant for application testing and debugging at scale. The *BigSim Simulator* is a trace-driven PDES that models architectural parameters of a HPC system. It is meant for identification of performance bottlenecks, such as load imbalances, communication contention and long critical paths. It supports a variable-resolution processor model, ranging from simple scale factors to interpolation based on performance counters. For the network, it offers a simple latency model or a detailed model of the entire communication fabric. While the BigSim Emulator offers more functionality than JCAS, it scales worse due to the Charm++/AMPI layer. The BigSim Simulator uses a PDES to maintain accuracy, but does not support running native applications.

Other trace-driven PDES solutions for investigating application performance exist. For example, DIMEMAS [7] processes traces from MPIDTrace and generates trace files that are suitable for the two performance analysis tools, PARAVER and Vampir.

$\mu\pi$ [8] (see Figure 2 and http://kalper.net/kp/software/mupi) is a PDES-based system for predicting the performance of parallel programs. It is currently under development and targets various methods for interfacing native applications with the virtual system created by the PDES, such as source code, library and virtual machine grafting. Initial support for MPI applications exists. The strength of $\mu\pi$ is its reliance on the μsik PDES engine (see http://kalper.net/kp/software/musik) that supports conservative and optimistic execution. An early prototype recently ran on 216,000 cores of the Jaguar Cray XT5 at ORNL, successfully simulating over 27 million virtual MPI ranks, each virtual rank containing its own thread context, and all ranks fully synchronized by virtual time. The $\mu\pi$ effort is in parallel to the presented work, both focus on different aspects of investigating the complexities of HPC application performance at extreme scale using a PDES.

The Structural Simulation Toolkit (SST) (see http://www.cs.sandia.gov/sst) is a modular PDES framework on top of MPI for cycle-accurate simulation of novel compute-node architectures, including processor, memory, and network. It is currently under development and its primary goal is to enable the co-design of HPC systems and applications. While the SST scales only to a few number of nodes, its value is in the capability to investigate the performance features of future architectures in detail. It is also able to generate application models for larger-scale simulations, similar to the BigSim Emulator/Simulator combination. The work presented in this paper aims at furthering a similar synergy between small-scale cycle-accurate and large-scale communication-accurate simulations.

The SST project as well as this work are part of the U.S. Department of Energy's Institute for Advanced Architecture and Algorithms (IAA) (see http://www.csm.ornl.gov/iaa). It was established in 2008 to facilitate the co-design of architectures and applications in order to create synergy in their respective evolutions for closing the application-architecture performance gap. The IAA currently targets the development of architecture-aware algorithms and the supporting runtime features needed by these algorithms to solve general sparse linear systems common in many scientific applications. It also focuses on evaluating the algorithmic impact of future HPC system architecture choices.

III. Technical Approach

The research and development presented in this paper is a follow-on effort to the JCAS project that aims at filling the gap between the JCAS, the BigSim Emulator, and the BigSim Simulator. It targets a new PDES-based simulation toolkit that combines the efficiency of the JCAS design for highly oversubscribed execution with the virtual MPI (Charm++/AMPI) concept of the BigSim Emulator and the PDES execution model of the BigSim Simulator. It further incorporates certain design features of $\mu\pi$. In addition, this work is complementing the detailed hardware simulation capabilities at small scale (of a few nodes) of the SST project with more coarse grain simulation capabilities at extreme scale (of millions of nodes).

The presented work aims at executing algorithms in a PDES environment, such that their performance properties and the impact of architectural features can be studied at extreme-scale. The approach focuses on a software architecture similar to $\mu\pi$ (see Figure 2 in Section II). The targeted design features a lightweight PDES layer that offers a VP abstraction and a virtual MPI interface for highly oversubscribed execution scenarios. Similar to our earlier JCAS solution (see Section II) that was already able to run 500,000 VPs on 4+1 native processors, the goal is to enable the execution of millions of virtual MPI ranks on several orders of magnitude smaller HPC resources.

A PDES is employed to keep track of virtual time for every VP and for every message sent between VPs. MPI program execution time is correctly measured based on the virtual time. Performance estimations at different execution scales can be performed accurately despite the slowdown caused by the oversubscription of workload. Furthermore, as the PDES completely virtualizes time measurement, changing the network and processor performance models used by the PDES for virtual time measurement, as exemplified by the BigSim Simulator and the SST, allows to estimate performance on future-generation architectures.

The work presented in this paper focuses on several research and development challenges and demonstrates a first prototype solving these challenges. The first challenge is the design of a lightweight PDES that offers the needed features, such as virtual time measurement and VP messaging, as efficiently as possible. The second is the execution of a large number of MPI ranks as parallel VPs with a minimum performance overhead and without significant changes to the original MPI program. The third challenge is the implementation of a virtual MPI that provides low-overhead VP messaging and takes advantage of the underlying native MPI implementation when possible. The following section details how these challenges were solved and describes the implementation design of a prototype demonstrating the solutions to these challenges.

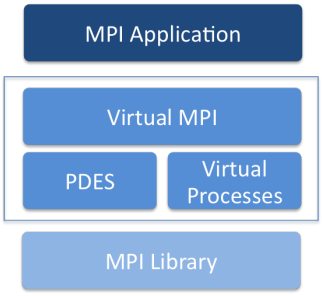

Figure 3. Design of the implemented simulator

IV. Implementation

To enable transparent execution of MPI applications inside a PDES, we have developed a simulator prototype that sits between the MPI layer and the application as an interposition library (see Figure 3). This implementation design provides the most efficient way to deploy the PDES without requiring significant changes to the original MPI application. The simulator utilizes the MPI performance tool interface, PMPI, to intercept MPI calls from the application and to hide all PDES-related mechanisms. The simulator further provides its own C/Fortran MPI application programming interface (API), *i.e.*, the simulator becomes the MPI layer of the application while the native layer MPI is used by the simulator via the PMPI API. The simulator is a native MPI application (distributed as a library) that wraps another MPI application into a virtual execution environment controlled by a PDES. Using preprocessor directives, the simulator API automatically renames the main entry function of a C MPI application. For Fortran MPI applications, the simulator API automatically changes the `program <program name>` directive to `subroutine sim_main`. The simulator library provides its own C and Fortran main entry functions and calls the renamed application's main entry function or subroutine during the PDES execution of a VP. Each native MPI process contains a simulator instance that executes the original MPI application within VPs utilizing a PDES for control and coordination among simulator instances. A MPI application is run in the simulator using the following steps:

1) Replace the C/Fortran MPI header file include directive with the C/Fortran simulator API header file.
2) Compile and link the MPI program with the simulator library (and the original MPI library).
3) Run the MPI program: `mpirun -np <native np> <program> -sim-np <virtual np>`

The simulator is implemented in a mixed C/C++ fashion using two (pthread) threads per instance. One thread constantly receives messages from the native MPI library and processes them accordingly, *e.g.*, queueing VP messages for the PDES and reacting to certain control messages. The other thread is part of the PDES and constantly dequeues

received VP messages. It switches into the context of the message destination VP and processes the message according to the current VP state and the message type/content. It also sends virtual MPI messages in the context of the executing VP to other VPs by locally queueing them or by utilizing the native MPI library to send them to a remote simulator instance. The decoupling of receiving/preprocessing and processing/sending native MPI messages, which contain control and VP messages, allows the PDES to execute a significant number of VPs without being synchronized by the native MPI library. However, this requires multi-threading support in the native MPI library, *i.e.*, the simulator calls `MPI_Init_thread()` and requires `MPI_THREAD_MULTIPLE`.

The developed simulator is comprised of the following three major components. The lightweight PDES performs virtual time management and VP messaging. The VP management component supports the PDES with a mechanism to easily switch between different VP contexts. The virtual MPI provides seamless MPI communication capabilities to applications running inside the simulator without the need for extensive modifications. The design and implementation details of these components are discussed in the following.

A. Parallel Discrete Event Simulator

In thread one of every simulator instance, the PDES receives VP and control messages from the native MPI and processes them accordingly. In thread two, the core mechanism of the PDES separately dequeues every received VP message, switches into the context of the message destination VP, and processes the message according to the current VP state and the message type/content. Utilizing the virtual MPI component, the message processing may involve sending messages to other local or remote VPs and/or to the processing VP itself, *e.g.*, for collective communication operations. The PDES additionally needs to deal with maintaining causality as multiple VPs are executed simultaneously at different simulator instances.

1) Virtual Time Management: For every VP, the PDES maintains a virtual time that is equivalent to the actual execution time of the VP, scaled to the performance of a virtual processor (and memory) as defined by the processor model. Similar to advanced operating system support for process-specific execution clocks, every time slice a VP executes on the native system is measured, adjusted by the processor model, and recorded. Execution of the rest of the simulator (PDES and native communication) is excluded from the time measurement. The current implementation supports a basic scaling processor model that uses the actual VP execution time and applies a scaling factor.

VP messages are transmitted internally by the PDES and its components. The native message latency is not exposed to the VPs. Instead, a network model calculates the virtual latency for every VP message. The message delivery virtual time to a VP depends on the virtual time of the source VP when the message was sent and the calculated virtual network latency. It may also depend on the virtual time of the receiving VP, such as when the VP wants to receive a message that has not been sent yet (wait time). The implemented network model currently supports fixed latency and bandwidth parameters, *i.e.*, a central non-blocking switch topology. A more complex network model supporting different network architectures is under development.

2) Virtual Process Messaging: VPs communicate with each other via virtual MPI messages that are encapsulated in VP messages and routed by the simulator instances from the source VP to the destination VP, which may reside at different instances. As explained earlier, VP messages are sent in the sending VP context through the virtual MPI component by the PDES thread that executes VPs. A message to a local VP is inserted into the local receive queue, which contains all received messages in virtual receive time order. A message to a remote VP is sent to the respective simulator instance using the native MPI, and inserted into the local receive queue by the receiving instance.

To bootstrap the simulation, each simulator instance inserts a special start message into its local receive queue addressed to its first local VP. Upon executing the simulation, the start message is received, re-adressed to the next local VP in line, and re-inserted into the receive queue. The simulated program's main entry function is executed with the appropriate command line argument parameters for every received start message in the context of the addressed VP. The bootstrap guarantees a virtual parallel execution, such that, starting with the main entry function, program chunks between message receives are serially executed separately for every VP until program termination.

Similar to operating system support for multi-threading, the simulator provides a yield capability to manually hand over control to the PDES and to allow other VPs to progress. The yield mechanism inserts a special yield message into the local receive queue addressed to the calling VP with the VP's time stamp. Then the VP waits for the receive of this message, which allows the PDES to process messages with earlier time stamps. While the yield capability is implemented and fully functional, it should be avoided and only used explicitly if really needed by the program. It does slow down the simulators execution, especially at large scale.

3) Maintaining Causality: Maintaining causality is typically the most difficult part of designing a PDES. The simulation can deadlock or degrade in performance if implemented without considering the causality and performance characteristics of the simulated system. The MPI has certain causality and performance features that are reflected in the PDES design of the developed simulator. Considering basic MPI functionality, potential sources of causality errors are limited to `MPI_ANY_SOURCE` and `MPI_Waitsome()` receives. All other MPI communication operations have a

strict deterministic execution semantics, *e.g.*, MPI message delivery order cannot be altered by the MPI layer, every MPI message send requires a matching MPI message receive, and collectives are synchronous operations.

In case of `MPI_ANY_SOURCE`, the message source is unknown to the receiver and the first matching message is received. The PDES needs to determine message order by virtual time in order to deliver the first matching message to the application. This requires to know all potential matching messages at the receiving instance before making the decision. As the number of potential matching messages is unknown, the PDES needs to employ a synchronization mechanism to assure causality. The current simulator implementation does not support `MPI_ANY_SOURCE` at this time as the primary goal was to develop an initial prototype that demonstrates the overall concept. An on-demand conservative synchronization mechanism with deadlock detection is currently under development.

In case of `MPI_Waitsome()`, all receive requests completed at or before the virtual time of the calling VP are returned. The PDES needs to determine the receive completion status for all outstanding receives the VP is waiting on for this particular MPI call that have the potential to complete before making the decision. This can be easily solved by internally waiting for the outstanding receives to complete, re-queueing those that occur in the future of the calling VP, and returning those that occur in the present or past. Note that VP time is not advancing while the PDES waits for the outstanding receives to complete.

B. Virtual Processes

To enable highly-oversubscribed operation, the simulator is employing user-space threads for VPs, similar to the prior JCAS solution (see Section II). In contrast to JCAS, however, the developed solution supports user-space thread stacks as well to allow each VP to execute a native program in its own stack context. This has two consequences. First, all stack variables are maintained separately for each VP. Second, call stacks are maintained separately as well.

The implementation is based on using a single user-space (pthread) stack frame for the PDES thread that executes the VPs. This stack frame is split up to allow each VP to operate in its own stack frame. A context switch between VPs is as simple as saving and restoring stack and base pointer within a dedicated context switch function. Since the function's return address is put on stack upon entering, a VP always saves its call stack before switching over to another VP. As this involves assembly instructions in the C/C++ code of the simulator, the current implementation only supports Intel 32- and 64-bit architectures (x86 and x86_64). This, however, can be easily extended.

To initialize all VP stacks, the simulator switches to the first local VP before receiving its start message. The context switch function creates a copy of the current stack state for each VP, thus assuring that each VP starts with the same state as the first local VP before receiving its respective start message. The initialization routine does take into account that the stack may grow up or down, depending on the actual execution architecture.

While this type of user-space thread support has tremendous performance advantages, *e.g.*, over using a system (pthread) thread for each VP, it comes at the price of loosing the memory protection offered by the hardware and supported by the operating system. Users of the simulator need to be aware of the risk of VP thread stack overflow and resulting VP stack corruption. The simulator provides a command line configuration option to set the VP stack frame size to actual MPI program requirements.

C. Virtual MPI

The virtual MPI intercepts supported calls made by the application to the MPI library and encapsulates each MPI message in a VP message. In the following, supported C and Fortran MPI calls are discussed in more detail.

The `MPI_Init()` and `MPI_Finalize()` functions are empty as the virtual MPI component does not need to be setup or teared down by the MPI program.

`MPI_Comm_rank()` and `MPI_Comm_size()` return the corresponding virtual MPI rank and communicator size. As the other MPI communicator operations are currently not supported, all MPI calls are restricted to `MPI_COMM_WORLD` at the moment. Ongoing work focuses on full support for MPI groups and communicators.

`MPI_Send()` sends a MPI message encapsulated in a VP message, utilizing the native MPI for non-local destination VPs. All basic MPI data types are supported. `MPI_Isend()` does the same, as VP time measurement is not active during virtual MPI calls. It additionally sends a request message to the calling VP, similar to the yield capability, to allow for seamless processing of send requests in conjunction with receive requests by `MPI_Wait()` and `MPI_Waitall()`.

`MPI_Recv()` receives a MPI message encapsulated in a VP message. The first message in the PDES receive queue is processed in the context of the destination VP. If it is not addressed to the calling VP, a context switch is performed to the destination VP. The calling VP receives control back from the PDES once its message is first in the PDES receive queue. Messages that are addressed to the VP, but do not match the receive parameters (source and tag) may exist and are temporarily cached until a matching message is received, and re-queued once it is received. `MPI_Irecv()` just sends a request message containing the receive parameters to the calling VP in order to defer the actual receive to `MPI_Wait()` and `MPI_Waitall()`.

`MPI_Sendrecv()` executes an `MPI_Send()` and then a `MPI_Recv()`. The potential for deadlock, as can be experienced in native MPI implementations with this basic

blocking send/receive approach, does not exist as both operations, sending and receiving messages, are decoupled by threads and a receive queue.

The `MPI_Wait()` and `MPI_Waitall()` functions wait for the request messages previously sent by the respective `MPI_Isend()` and `MPI_Irecv()` calls. `MPI_Waitsome()` implements the mechanism described in Section IV-A3 for ignoring those requests that complete in the future of the calling VP, and returning those that complete in the present or past. The processing of receive requests trigger the actual receive of the VP-encapsulated MPI message.

`MPI_Bcast()` performs a broadcast of a VP-encapsulated MPI message. The VP message is broadcast by the root VP to all simulator instances using a point-to-point broadcast via the native MPI. Local processing of broadcasted VP messages is handled similar to the start message by forwarding the message from one VP to the next. The broadcast capability of the native MPI cannot be used as it is a synchronous operation that interferes with the asynchronous nature of the PDES. The local message forwarding significantly reduces overhead, especially at large scale.

`MPI_Barrier()` waits until all local VPs have reached this call though a basic counter, and then sends a VP message to VP 0 with the time stamp of the last local VP entering the barrier. VP 0 waits for all local and remote VPs and broadcasts a VP message with the time stamp of the last global VP entering the barrier. `MPI_Barrier()` resumes execution once this broadcast message is received.

`MPI_Wtime()` and `MPI_Wtick()` return the current VP time and resolution according to the PDES virtual time management.

V. Experimental Results

We deployed the prototype on a four-node Linux cluster for an initial evaluation. Each node has an Intel dual-core 2.13GHz processor and 2GB of memory. An additional head node with the same configuration was used to avoid interference. All five nodes are connected via Gigabit Ethernet. The system is running the Ubuntu 8.04 64-bit Linux distribution without swap and Open MPI 1.4.2 with multi-threading support. The evaluated simulator implementation is a redesigned and significantly enhanced version of a first prototype that demonstrated the core mechanisms as part of a recent Master's thesis [9] by one of the co-authors.

For the first experiment, we created a micro benchmark that performs one-way MPI message latency measurements between two networked nodes in the cluster using a ping-pong technique. Ten measurements were performed for each metric at increasing payloads and 32kB stack per VP. The virtual network parameters were set to 0μs/∞Gbps, 10μs/∞Gbps, 100μs/∞Gbps, 0μs/1Gbps, and 50μs/1Gbps. The processor scaling factor was set to 1. The results (see Figure 4) clearly show that the fixed virtual network latency and bandwidth remains uninfluenced by the native one-way MPI message latency as it is controlled by the PDES. They also show that the 50μs/1Gbps parameters closely model the native MPI performance.

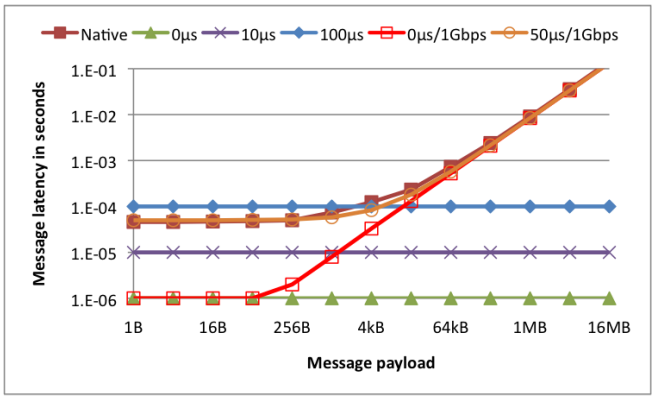

Figure 4. Comparison of native vs. virtual one-way MPI message latency

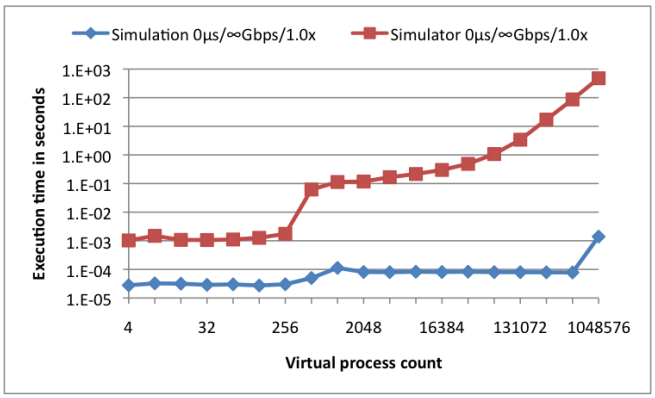

Figure 5. Scaling up the simple `hello world` MPI program to 1,048,576 VPs in the simulator on a four-node Linux cluster

In a second experiment, we created a simple `hello world` MPI program that prints out the MPI rank and size (in `MPI_COMM_WORLD`) for each MPI process. Ten measurements were performed for each metric at increasing scales using one simulator instance on each of the 4 nodes and 4kB stack per VP. The processor scaling factor was set to 1. The results (see Figure 5) overwhelmingly demonstrate the capability of the developed solution as it scales to 1,048,576 VPs, *i.e.*, virtual MPI processes, on a 4-node cluster. The simulation time remains constant at different values for 4-512 VPs and 1,024-524,288. We attribute the lower values at 4-512 VPs to operating system noise influencing the time measurement, since the simulation time is below 100μs and the simulator time is $\approx$1ms. The significant increase of the simulator time to $\approx$100ms at 512 VPs is likely related to output buffer issues as each of the 128 VPs/node is printing a line to the output. The simulator

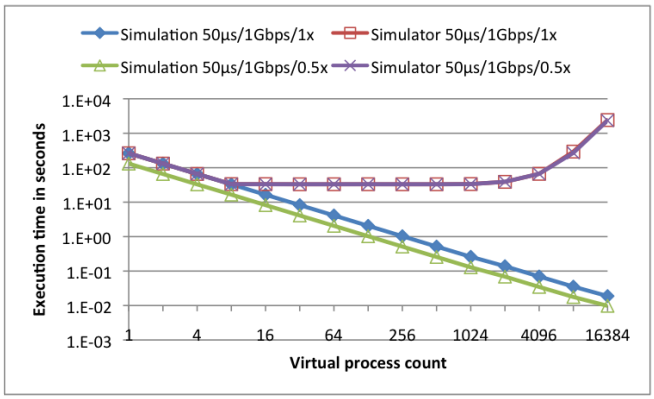

Figure 6. Scaling up the computation-heavy π Monte Carlo MPI application in the simulator on a four-node Linux cluster

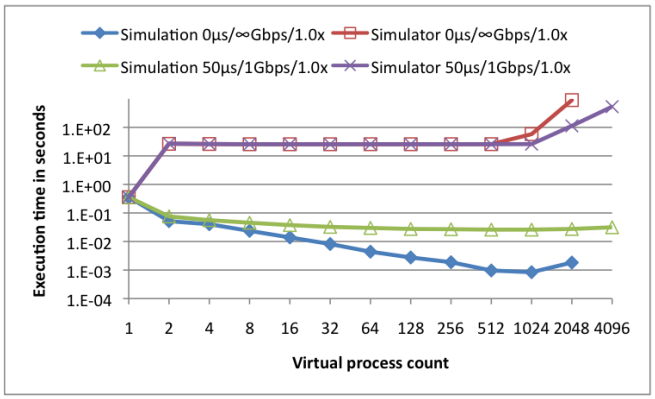

Figure 7. Scaling up the communication-heavy heat equation MPI application in the simulator on a four-node Linux cluster

execution time scales with the number of VPs. At 1,048,576 VPs, the simulation time increases by a magnitude, which is likely caused by the fact that the system is running near its resource limits (1GB memory/node allocated for VP stack alone) with a noise-sensitive MPI program.

The next two experiments focused on scaling up real MPI applications to demonstrate the evaluation capabilities and performance properties of the developed solution. We implemented two orthogonal benchmark MPI applications. The first performs an estimation of π using a Monte Carlo approach. It is a computation-heavy embarrassingly parallel benchmark MPI application that performs communication only at the end to gather the results of all worker tasks at rank 0. The second solves the 1D heat equation in an iterative fashion. It is a communication-heavy closely-coupled benchmark MPI application that performs communication between neighboring MPI ranks at each iteration.

Ten measurements were performed for each metric at increasing scales using 32kB stack per VP, two simulator instances per node (one per core) for the π calculation (as it is computation-heavy), and one per node for the heat equation solver (as it is communication-heavy). The measurements were performed at increasing scales with fixed total workload (strong scaling). The virtual network parameters for the π calculation were set to 50μs/1Gbps. The virtual network parameters for the 1D heat equation solver were set to 0μs/∞Gbps and 50μs/1Gbps to demonstrate the network scaling properties of the simulation and the simulator. The processor scaling factor for the π calculation was set to 1 and 0.5, *i.e.*, twice as fast as the original processor, demonstrate the processor scaling properties of the simulation and the simulator. The processor scaling factor for the 1D heat equation solver was set to 1.

The results for the π calculation (see Figure 6) clearly show that without processor scaling (50μs/1Gbps/1$\times$) the simulation and the simulator have the same execution times with 0% overhead and linear scaling as long as the number of native cores equals the number of VPs. Once the number of VPs exceeds the number of cores, the simulator time remains constant and the simulation time continues to reflect linear scaling. At 4096 VPs, the amount of work performed by the program becomes much smaller than the amount of work required for the last communication step executed by the simulator and hidden to the program by the fixed 50μs virtual network latency. The simulator starts scaling with the increasing communication load. Figure 6 also shows the impact of the processor scaling factor on the simulation execution time (50μs/1Gbps/1$\times$ vs. 50μs/1Gbps/0.5$\times$). The processor scaling factor of 0.5 reduces the simulation execution time to 50% of the original processor.

The 1D heat equation solver results (see Figure 7) show a similar picture. Moving from 1 to 2 nodes simply demonstrates the difference from using no MPI communication at all to MPI communication between neighboring MPI ranks. As this application is communication-heavy, the difference between the simulator time, which includes the native communication workload, and the simulation time, which only includes the virtual communication workload, is clearly visible. Nevertheless, the simulation time shows linear scaling for 2-1024 VPs for the 0μs/∞Gbps/1$\times$ network model parameter set and the expected polynomial scaling for 2-512 VPs for the 50μs/1Gbps/1$\times$ network model parameter set. The simulator execution time remains constant in both simulation scenarios due to the strong application scaling. Starting with 1024 VPs, the communication workload causes contention for the simulator and the virtual network limits the scaling for the 50μs/1Gbps/1$\times$ simulation. The 0μs/∞Gbps/1$\times$ simulation faces a different scaling issue. At 2048 VPs, the initialization phase of the 1D heat equation solver becomes a significant part of the overall simulation execution time. Both simulation runs, 0μs/∞Gbps/1$\times$ and 50μs/1Gbps/1$\times$, create different contention patterns at the simulator as the global message order is defined by the different network model parameter sets.

VI. Conclusions and Future Work

We have implemented a new PDES-based simulation toolkit that fills the gap between the JCAS, the BigSim Emulator, and the BigSim Simulator by combining the efficiency of the JCAS design for highly oversubscribed execution with the virtual MPI (Charm++/AMPI) concept of the BigSim Emulator and the PDES execution model of the BigSim Simulator. This work is complements the detailed hardware simulation capabilities of the SST project with more coarse grain simulation capabilities at extreme scale. The developed solution enables the investigation of the complexities of algorithm performance at extreme scale using lightweight simulation. Furthermore, as part of the IAA effort, this solution is a first step toward evaluating the algorithmic impact of future architecture choices to facilitate the co-design of architectures and applications for closing application-architecture performance gap.

Ongoing and future work targets: (1) a better processor model with interpolation using performance counters, (2) a conservative synchronization mechanism with deadlock detection (for `MPI_ANY_SOURCE` receives), (3) a more complex network model (with different topologies), and (4) a more complete virtual MPI support (including communicator and advanced collective communication operations).

Acknowledgment

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[2] K. S. Perumalla, "Parallel and distributed simulation: Traditional techniques and recent advances," in *Proceedings of the* 38^{th} *Winter Simulation Conference 2006*. Monterey, CA, USA: ACM Press, New York, NY, USA, 3-6, 2006, pp. 84–95.

[3] C. Engelmann and G. A. Geist, "Super-scalable algorithms for computing on 100,000 processors," in *Lecture Notes in Computer Science: Proceedings of the* 5^{th} *International Conference on Computational Science (ICCS) 2005, Part I*, vol. 3514. Atlanta, GA, USA: Springer Verlag, Berlin, Germany, May 22-25, 2005, pp. 313–320.

[4] G. Zheng, G. Kakulapati, and L. V. Kale, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," in *Proceedings of the* 18^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2004*. Santa Fe, New Mexico: IEEE Computer Society, Apr. 26-30, 2004.

[5] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kale, "Simulation-based performance prediction for large parallel machines," *Concurrency and Computation: Practice and Experience*, vol. 3, no. 2-3, pp. 183–207, 2005.

[6] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming petascale applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. CRC Press, Dec. 2007, pp. 421–441.

[7] S. Girona, J. Labarta, and R. M. Badia, "Validation of dimemas communication model for MPI collective operations," in *Lecture Notes in Computer Science: Proceedings of the* 7^{th} *European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2000*, vol. 1908. Balatonfüred, Hungary: Springer Verlag, Berlin, Germany, Sep. 10-13 2000, pp. 39–46.

[8] K. S. Perumalla, "$\mu\pi$: A highly scalable and transparent system for simulating MPI programs," in *Proceedings of the* $3rd^{th}$ *International ICST Conference on Simulation Tools and Techniques (SIMUTools) 2010*. Malaga, Spain: ACM Press, New York, NY, USA, Mar. 15-19, 2010.

[9] F. Lauer, "Simulation of advanced large-scale HPC architectures," Master's thesis, Department of Computer Science, University of Reading, UK, Mar. 12, 2010.