xSim: The Extreme-Scale Simulator

Swen Böhm and Christian Engelmann
Oak Ridge National Laboratory, Oak Ridge, TN, USA
bohms@ornl.gov and engelmannc@ornl.gov

ABSTRACT

Investigating parallel application performance at scale is an important part of high-performance computing (HPC) application development. The Extreme-scale Simulator (xSim) is a performance toolkit that permits running an application in a controlled environment at extreme scale without the need for a respective extreme-scale HPC system. Using a lightweight parallel discrete event simulation, xSim executes a parallel application with a virtual wall clock time, such that performance data can be extracted based on a processor and a network model. This paper presents significant enhancements to the xSim toolkit that provide a more complete Message Passing Interface (MPI) support and improve its versatility. These enhancements include full virtual MPI group, communicator and collective communication support, and global variables support. The new capabilities are demonstrated by executing the entire NAS Parallel Benchmark suite in a simulated HPC environment.

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

1. INTRODUCTION

Today's leading petascale high-performance computing (HPC) systems, such as the Cray XT5 at Oak Ridge National Laboratory (ORNL), consist of up to several hundreds-of-thousands of processor cores organized in several tens-of-thousands of compute nodes. As processor core frequency scaling is limited by power consumption costs and corresponding thermal dissipation costs, the trend is clearly toward increasing the number of cores in future-generation HPC systems. An exascale system may have up to 1,000,000 compute nodes with 1,000 lightweight processor cores per node by 2018 (see http://www.exascale.org). Parallel applications need to employ more concurrency in order to fully utilize such future-generation HPC systems. Due to Amdahl's law, sequential application parts become more dominant as the level of concurrency grows from today's hundreds-of-thousands to tomorrow's hundreds-of-millions of processor cores, a 1,000× increase. Investigating performance properties at scale is becoming an important part of parallel application development and deployment. The identification of sequential application parts, synchronizing communication and other bottlenecks is essential to enable performance at scale. Application tuning options may include parallelizing sequential parts or replacing one parallel algorithm with another, better scaling, one.

The Extreme-scale Simulator (xSim) [6] is a parallel application performance investigation toolkit that permits running an application in a controlled environment at extreme scale without the need for a respective extreme-scale HPC system. It allows the observation of performance properties in a simulated HPC system with up to millions of concurrent execution threads to identify application scaling issues and to assist in HPC hardware/software co-design. Using a lightweight parallel discrete event simulation (PDES), xSim executes a parallel application with a virtual wall clock time, such that performance data can be extracted based on a processor model and a network model. The processor model scales the virtual time based on actual execution time, while the network model enforces virtual time latencies at the Message Passing Interface (MPI) layer. A MPI application can be executed in a highly oversubscribed fashion on a much smaller HPC system and its performance properties can be investigated properly using virtual wall clock timing, instrumentation and monitoring.

This paper presents recently implemented significant enhancements to the xSim toolkit prototype that provide a more complete MPI support and improve its versatility. The primary goal of this work is to enable the execution of the entire NAS Parallel Benchmark (NPB) [1] suite in the simulated HPC environment to demonstrate its capabilities and to push the implementation to a production-type stage. To achieve this goal, the xSim toolkit prototype is enhanced with full virtual MPI group, communicator and collective communication support, and with global variables support. The NPB suite is a set of C and Fortran MPI benchmarks that utilize a number of more complex MPI calls, such as

MPI_Comm_split() and *MPI_Allreduce()*, as well as global program variables, such as the Fortran *COMMON* block and C variables declared outside of a function. As these features of the NPB suite are also common in today's parallel applications, the xSim toolkit is extended with the respective mechanisms to support them.

2. RELATED WORK

The related work has been discussed in a prior publication [6]. This Section is giving a summary of the original xSim toolkit and of the previously discussed related efforts.

xSim The general design of the xSim simulation toolkit [6] relies on the MPI performance tool interface (PMPI) and its capability to intercept MPI calls from the application. xSim sits between the MPI library and a parallel application as an interposition library utilizing the PMPI mechanism to hide all PDES-related mechanisms, such as virtual time management, virtual process messaging, maintaining causality, and virtual processes management. The interposition library essentially wraps an MPI application into a virtual execution environment controlled by the xSim PDES. xSim intercepts MPI calls coming from the application via its own C/Fortran MPI application programming interface (API), *i.e.*, xSim provides an MPI layer to the application while the native MPI layer is used by xSim via the PMPI API. Each virtual MPI process is encapsulated in a user-space thread that has its own virtual time, while messages between virtual MPI processes are sent and received via the PDES. xSim supports a basic set of MPI functions and is able to run C and Fortran MPI applications. An application is run in the simulator using the following steps:

- Add "#include xsim-c.h" to the C source code, or add "#include xsim-f.h" to the Fortran source code.
- Recompile the application and link it with the xsim library, *e.g.*, "-lxsim", and the respective xsim programming language interface library, *e.g,* "-lxsim-c" for C or "-lxsim-f" for Fortran
- Run the application with "mpirun -np <real process cout> <application> -xsim-np <virtual process count>"

The PDES-driven simulation accounts for the execution time for each virtual MPI process using the actual execution time on the real processor scaled by a processor model. It also accounts for the wait time incurred by communication for each virtual MPI process using a network model. The current implementation [6] is able to scale a basic MPI *hello world* program to up to 1 million virtual MPI processes on a small 8-core compute cluster with 8GB total RAM, using 4GB RAM just for virtual process stack. It is also capable of investigating the performance of basic computational MPI applications to up to 16,384 virtual processors on the same system. Recent enhancements, provide advanced network models for a variety of HPC network architectures, such as star, ring, mesh, torus, twisted torus, tree, and hierarchical combinations, with the appropriate trade-off between simulation scalability and accuracy. The original xSim offers a very primitive MPI support for point-to-point messages in *MPI_COMM_WORLD* only. It also does not support collective operations beyond a basic *MPI_Barrier()* and *MPI_Broadcast()*. Furthermore, the user-space pthread-based virtual process management offers virtual stack and heap without considering global variables. The NPB suite does not run on the original xSim due to these limitations.

JCAS The *Java Cellular Architecture Simulator (JCAS)* [5] is the predecessor of xSim and was developed in 2001 to investigate scalable and fault-tolerant algorithms for large-scale systems. The final prototype was able to run up to 500,000 virtual processes on a cluster with 5 native processors (using 1 for visualization) solving basic mathematical problems. While it was able to run algorithms at scale, it lacked certain important features, such as time-accurate simulation, high performance, support for running the simulator atop MPI, and a fully functional virtual MPI. The JCAS project sparked a new area of research in fault-tolerant algorithms [2, 3, 9].

BigSim The BigSim [11] project was a competitor to JCAS and initiated to study programming issues in large-scale HPC systems. The *BigSim Emulator* is for application testing and debugging at scale and build atop Charm++ and Adaptive MPI [8]. It supports up to 100,000 virtual MPI processes distributed over 2,000 native processors. Similar to JCAS, it does not offer time-accurate simulation. It provides more functionality than JCAS, but scales worse due to the Charm++/AMPI layer. The *BigSim Simulator* is for identification of performance bottlenecks and uses a trace-driven PDES that models architectural parameters of a HPC system. For time-accurate simulation, it supports a variable-resolution processor model and a detailed network model. While it uses a PDES to maintain accuracy, it does not support running native applications.

MPI $\mu\pi$ [10] is a PDES-based system currently under development for predicting the performance of parallel programs. It uses different grafting methods for interfacing applications with the virtual system created by the PDES, such as at the source code, library (currently implemented) and virtual machine level. It supports conservative and optimistic execution based on the μsik PDES engine (see http://kalper.net/kp/software/musik). A prototype was recently tested on 216,000 cores of the Jaguar Cray XT5 at ORNL, providing over 27 million virtual MPI ranks,

each with its own (pthread) thread context, and all ranks fully synchronized by virtual time. xSim is far more advanced as $\mu\pi$ requires an extreme-scale system to simulate an extreme-scale system. $\mu\pi$'s virtual-to-native thread ratio is only 125/core on 216,000 cores, where xSim achieves 125,000/core on 8.

Structural Simulation Toolkit The Structural Simulation Toolkit (SST) (see http://www.cs.sandia.gov/sst) offers cycle-accurate simulation of novel compute-node architectures, including processor, memory, and network. It is a modular PDES framework atop MPI that scales to a few number of nodes. Its value is in the ability to investigate the performance of future architectures and to generate models for larger-scale simulations. Similar to the BigSim Emulator/Simulator, SST and xSim enable this synergy between small-scale cycle-accurate and large-scale communication-accurate simulations.

Further Related Work Other trace-driven PDES solutions for investigating parallel application performance exist and include DIMEMAS [7], which processes traces from MPIDTrace and generates trace files that are suitable for the two performance tools, PARAVER and Vampir. Processing MPI traces at extreme scale is another research challenge and not addressed by the presented work. There are a variety of network architecture simulators, such as ns2/ns3 (see http://nsnam.isi.edu/nsnam and http://www.nsnam.org) and NetSim, (see http://tetcos.com/software.html) that are able to provide network performance metrics at various abstraction levels, such as network, sub-network, and packet traces. These detailed simulators offer high-accuracy/low-scalability results that are not compatible with the low-accuracy/high-scalability approach of xSim.

3. TECHNICAL APPROACH

The presented work focuses on adding three separate improvements to the xSim toolkit: (1) user-defined virtual MPI groups and communicators, (2) virtual MPI collective communication, and (3) global variables support.

MPI Groups and Communicators The approach for the user-defined virtual MPI groups and communicators is as follows. As MPI groups and communicators are inherently linked with each other (there is no MPI communicator without a corresponding group), the improvements target a tracking mechanism for virtual MPI groups and communicators. For each application-defined MPI group, xSim keeps a record of its members and allows respective translation from virtual MPI rank to the native MPI rank of the simulator instance and to the local virtual process rank at this simulator instance. This translation is needed internally by xSim for messaging between virtual MPI processes. The xSim interposition library intercepts all *MPI_Group_...()* calls to provide this functionality. For virtual MPI communicators, a similar tracking mechanism is used to link each application-defined MPI communicator with a corresponding virtual MPI group. xSim intercepts all *MPI_Comm_...()* calls to provide this capability.

MPI Collective Communication MPI collective communication operations, such as broadcast, scatter and gather, are as essential to today's parallel applications as MPI groups and communicators. The approach for adding full MPI collective communication support to xSim adopts a recently developed implementation from another project, MR-MPI [4]. The modular-redundant MPI (MR-MPI) is a solution for transparently executing HPC applications in a redundant fashion. It uses the same PMPI mechanism as xSim for transparently intercepting MPI calls from an application and hiding all redundancy-related mechanisms. In MR-MPI, each native MPI rank may have additional shadow native MPI ranks as redundant copies that are not exposed to the application as independent MPI rank. Similar to xSim, virtual and native MPI ranks do not directly translate to each other and point-to-point as well as collective communication operations need to be aware of this fact. To avoid implementing each collective with the needed support for redundancy in MR-MPI (or oversubscription in xSim), the collective communication operations are simply reimplemented atop the virtual point-to-point communication and virtual communicator operations.

Global Variables The missing support for global variables in xSim is a significant hurdle for its wide-spread use. Many parallel application have global variables to avoid dynamic memory allocation and to facilitate communication between different program modules without the need for direct function calls. The approach for providing this capability to all applications executed in xSim aims at analyzing the application binary itself to identify the size and location of data segments that cause corresponding memory allocations by the system's program loader prior program start. Similar to the existing virtual process stack switch, the global memory segment is kept separate for each virtual process and changed during a virtual process context switch.

4. IMPLEMENTATION

The implementation of the three separate improvements to the xSim toolkit, the user-defined virtual MPI groups and communicators, the virtual MPI collective communication capability and the global variables support, are significant changes to the existing source code base as several core

mechanisms are replaced or changed. In the following, the implementation details of the improvements and the resulting changes to xSim are described in more detail.

MPI Groups and Communicators The implemented improvements add virtual MPI groups to xSim by intercepting all *MPI_Group_...()* calls from the application and by managing virtual MPI groups using an xSim-internal virtual MPI group registry. Each defined group in the registry contains a list of its members identified by their virtual MPI rank in the virtual *MPI_COMM_WORLD* communicator and ordered by their group membership. As all MPI groups are derived from the two predefined groups that are used by *MPI_COMM_WORLD* and *MPI_COMM_SELF*, the respective *MPI_COMM_WORLD* and *MPI_COMM_SELF* groups are created automatically by xSim on startup. All user-defined groups are created by *MPI_Group_...()* operations, using the *MPI_COMM_WORLD* and *MPI_COMM_SELF* groups as starting point, *e.g.*, with *MPI_Group_incl/excl()* or *MPI_Group_union()*. *MPI_Group_rank()* simply returns the index of the group member in the list. To reduce the amount of used resources, the *MPI_COMM_WORLD* and *MPI_COMM_SELF* groups have no list and return the corresponding virtual process MPI rank in the simulated environment for *MPI_Group_rank()* calls with the *MPI_COMM_WORLD* group, or 0 for the *MPI_COMM_SELF* group. *MPI_ Group_size()* simply returns the size of the group's list. For the *MPI_COMM_WORLD* group, this is the total number of virtual MPI ranks. It is 1 for the *MPI_COMM_SELF* group.

The xSim toolkit essentially oversubscribes a single native MPI process with many virtual MPI processes that are managed by the PDES. To reduce the amount of used resources by the group registry, each entry also contains a reference counter. Multiple virtual MPI processes that are located on the same native MPI process may create virtual MPI groups that contain the same members in the same order. If this is the case, they use the same virtual MPI group handle and increase the reference counter accordingly. If a virtual MPI group is freed, its reference counter is decreased and its entry is deleted if the counter reaches 0.

For supporting user-defined virtual MPI communicators, all *MPI_Comm_...()* calls are intercepted and the registry concept for virtual MPI groups is simply reused. Each defined communicator in the virtual MPI communicator registry contains the respective group handle the communicator is associated with and a reference counter to reduce the amount of used resources. The *MPI_Comm_rank()* call simply returns the result of the *MPI_Group_rank()* call of the associated group. *MPI_Comm_size()* relies on *MPI_Group_size()*. An implementation of the rather complex *MPI_Comm_split()* call is available. As all communicator operations, according to the MPI standard, are collective operations, the newly implemented collective communication operations are utilized.

For supporting Fortran applications through C-based MPI stub functions, virtual MPI group and communicator handles are mapped between C and Fortran by intercepting the corresponding *MPI_..._f2c()* and *MPI_..._c2f()* calls. The mapping of handles between both programming languages also considers the existing predefined handles for *MPI_COMM_WORLD*, *MPI_COMM_SELF*, *MPI_COMM_NULL*, *MPI_GROUP_NULL* and *MPI_GROUP_EMPTY*.

As the prior xSim prototype provided no support for virtual MPI groups and only offered a virtual *MPI_COMM_WORLD* communicator, the improvements also caused corresponding changes in the implementation of virtual MPI calls that rely on MPI group or communicator information, such as point-to-point communication calls.

MPI Collective Communication The added support for collective communication is based on the prior implementation of MPI collectives in the redundant MPI solution MR-MPI [4]. All collective communication calls from the application, such as *MPI_Bcast()*, *MPI_Gather()* and *MPI_Scatter()*, are intercepted and re-implemented atop the virtual MPI point-to-point communication calls. A recent addition to the MPI standard, *MPI_Reduce_ local()*, is used for collectives that involve MPI operations on communicated data, such as *MPI_Reduce()*. In general, *MPI_Reduce_local()* executes an MPI operation that is registered with the MPI library, such as the predefined *MPI_SUM* operation, without the need for a collective operation. xSim uses this feature to transparently implement these collective communication calls atop its nonblocking point-to-point communication primitives.

The newly implemented collective communication operations are based on linear algorithms at this point. More advanced algorithms, such as trees, will be considered in the future. As the native communication cost differs dramatically if the destination virtual MPI process resides on the same native MPI process or not, linear algorithms may be more efficient locally. These optimizations and associated trade-offs will be part of future investigations. As the original xSim prototype offered minimal support for collective communication operations, *i.e.*, a hard-coded *MPI_Bcast()* and *MPI_Barrier()* only, the collective communication module was re-implemented entirely.

Global Variables The implemented capability to allow each virtual MPI process to have its own global variables is

based on two mechanisms. The first identifies the data segments of initialized and uninitialized global program memory in the application executable file. As this depends on the executable file format, the initial implementation presented in this paper focuses on the widely-used ELF format, *i.e.*, the format used in Linux, and offers a capability to extend this solution to other formats, *e.g.*, the Mach-O format used in Apple's MacOS. Based on the identified global memory in the application executable, the corresponding virtual start and end addresses of the global memory of the executed program are calculated and a global memory backup space is allocated for each virtual MPI process.

The second mechanism deals with managing the global memory for each virtual MPI process. In general, xSim uses a single user-space memory region within each native MPI process as stack for the thread (pthread) that executes the associated virtual MPI processes [6]. This stack is evenly split across all virtual MPI process and every virtual MPI processes context switch includes changing the thread stack and base pointer, *i.e.*, switching the stack frame. The first context switch initializes the stack memory of all virtual MPI processes with the stack content of the active virtual MPI process. As the first context switch is initiated by the PDES when calling the program's *main()*, all virtual MPI processes start at the same initial condition. For the global memory, the first context switch similarly initializes the global memory backup space of all virtual MPI processes with the global memory content of the active virtual MPI process. All subsequent context switches save the global memory content of the active virtual MPI process in its global memory backup space and restores the global memory content of the next active virtual MPI process from its global memory backup space.

The global memory management is based on copying. This technique was preferred as it is simple and less error prone than, for example, binary rewriting the instructions that access global memory at runtime. It comes with the drawback of additional time spent for copying during context switches. As the xSim PDES itself uses global memory, it was modified to be virtual MPI process context switch safe by accessing all PDES state through pointers only, which are the same for all virtual MPI processes. The examination of the application executable file is executed prior to PDES startup and part of the overall xSim configuration setup procedure. The global memory management required only a slight modification of the virtual MPI process management.

5. EXPERIMENTAL RESULTS

The evaluation strategy for the modified xSim toolkit included a series of performance tests of the NPB suite on the native hardware and within the simulation.

Hardware Environment xSim and the NPB suite were deployed on a 16-node cluster. Each node has two 2.4GHz AMD Opteron processors with 4-cores each, *i.e.*, 8 cores per node, and 8GB RAM, *i.e.*, 1GB per core. The system has a total of 128 cores and 128 GB RAM. The network interconnect is dual non-blocking Gigabit Ethernet with a central switch, which was not bonded for the native NPB suite evaluation to get a performance baseline at 1Gbps and bonded for the xSim toolkit evaluation to get maximum performance for running the NPB suite within xSim.

Software Environment The cluster is running Ubuntu 10.04.1 LTS Linux and Open MPI 1.5 with multi-thread support. For evaluating the modified xSim toolkit, the entire NPB suite version 3.3.1 is executed at various scales (virtual MPI process counts). The NPB suite is a set of 9 individual MPI benchmarks with different communication/computation ratios and communication patters. Two (DT and IS) are implemented in C, while the others are in Fortran. All 9 utilize collective communication, such as *MPI_Barrier()*, *MPI_Bcast()*, *MPI_Reduce()*, *MPI_Allreduce()*, *MPI_Alltoall()* and *MPI_ Alltoallv()*. BT and SP also rely on *MPI_Comm_ dup()*, while FT, BT and SP further invoke the most complex communicator operation *MPI_Comm_split()*. The virtual execution times measured by each benchmark and the native execution times measured by xSim for selected benchmarks are collected and averaged over 10 runs for each parameter configuration.

Experiments The first series of experiments evaluate the scaling performance of the NPB suite on the experimentation platform without xSim. The purpose of this evaluation is to get a performance baseline that can be compared to the execution with xSim. Figure 1 illustrates the measured execution time in seconds for each benchmark executed with the problem size A. Since the BT and SP benchmarks require a squared MPI process count, while the other benchmarks require the MPI process count to be a power of two, the diagrams are split between both groups. With the execution time plotted on the y-axis in log scale, it is easy to see that all benchmarks initially show strong scaling. This is expected as the NPB benchmarks are strong scaling, *i.e.*, the problem to solve is constant in size and split up between more and more MPI processes when scaling up. The EP and CG benchmarks plateau at 64 MPI processes and their performance worsens with 128 as the problem size is too small and communication starts to dominate. IS shows this behavior already at 8, as it performs a communication-intensive integer sort. The others start to degrade at 16. For FT and LU, the performance improves from 64 to 128 MPI processes, which may be caused by a better match of the prob-

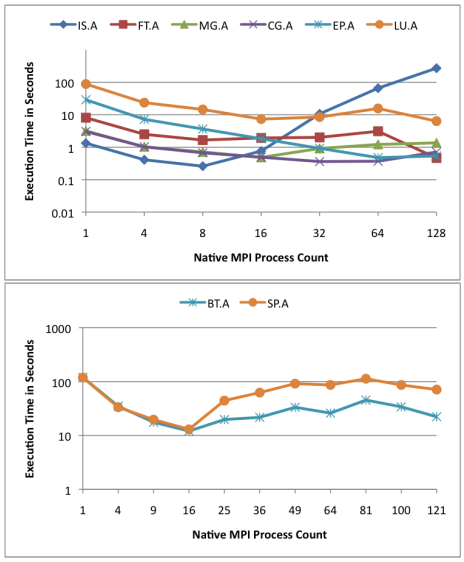

Figure 1. NPB Class A With Open MPI

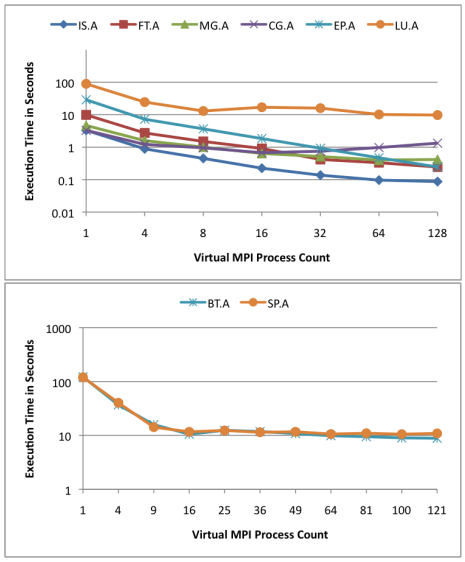

Figure 2. NPB Class A With XSim

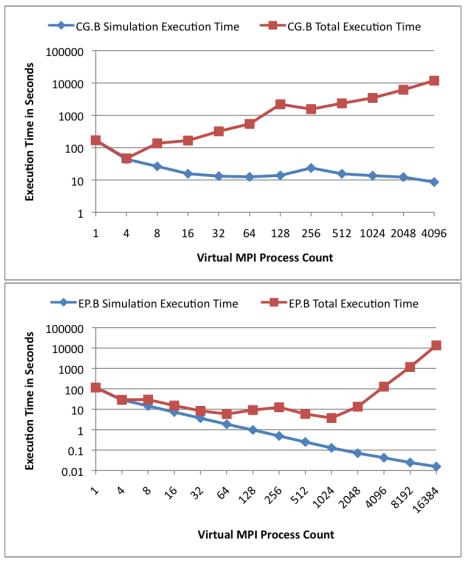

Figure 3. NPB Class CG/EP Class B With XSim

lem size to the number of MPI processes resulting in less communication. BT and SP show an interesting behavior from 25 MPI processes on. This may be caused by communication contention within the compute nodes in combination with the problem size being too small. Overall, the experiment shows some interesting features.

The second series of experiments evaluate the scaling performance of the NPB suite with xSim. The xSim processor model is set to a scaling factor of 1 and the xSim network model is configured for a star topology with 50μs latency and 1Gbps bandwidth. While the network model closely resembles the actual Gigabit Ethernet switch performance for MPI [6], it does not consider that multiple cores are located on the same processor with much smaller latency and much higher bandwidth to each other.

Figure 2 illustrates the measured virtual execution time in seconds for each benchmark executed with the problem size A. Comparing Figures 1 and 2, it is easy to see that EP, LU and CG scale roughly similar. IS does not show the performance drop beginning with 16 virtual MPI processes, while FT and MG show similar, but slightly different, performance. BT and SP show the same performance until 16 virtual MPI processes, however, they plateau instead of displaying a degraded performance. The differences may be caused by the network model not matching the actual hardware topology or not considering communication contention. It is also possible that the native MPI does not use linear collectives, while xSim's virtual MPI does.

The third set of experiments evaluate CG and EP at problem size B with oversubscription in xSim. Figure 3 plots the simulation execution time and the total execution time of the simulator for CG and EP, respectively. The communication-intensive CG is scaled to 4,196 virtual MPI processes and the embarrassingly parallel EP is scaled to 16,384. EP shows strong scaling in the simulation despite the 128$\times$ oversubscription, while GC levels off quickly due to the increasing communication/computation ratio. The simulator takes longer to finish the more virtual MPI processes exist. This behavior was already discussed in a prior publication [6] using MPI micro-benchmark applications.

The last set of experiments (Figure 4) evaluate CG and EP at problem size B on the native system and in xSim to asses the introduced overhead. While there is virtually no overhead for EP, CG experiences an overhead of up to a magnitude with 128 native/virtual MPI processes.

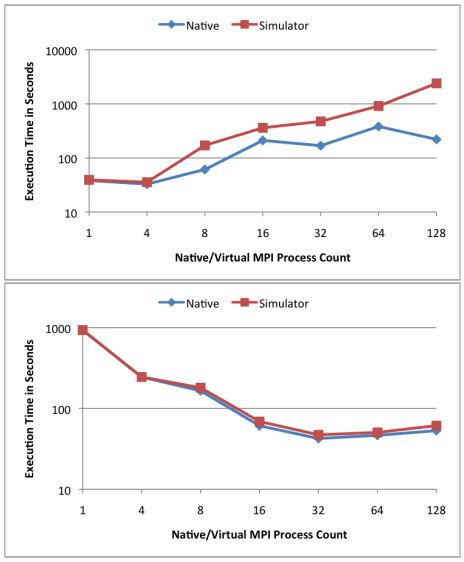

Figure 4. NPB Class CG/EP Class B With XSim

6. CONCLUSIONS AND FUTURE WORK

This paper presented significant enhancements to the xSim toolkit prototype, such as full virtual MPI group, communicator and collective communication support, and global variables support, which improve its capabilities and versatility. These new capabilities were demonstrated by executing the entire NAS Parallel Benchmark suite in a simulated HPC environment and by comparing its performance in the real world with the performance displayed in the simulated HPC environment. The experimental evaluation show that the implemented enhancements work and provide an additional benefit to the xSim toolkit. As already indicated in the paper, future work will focus on more efficient collective operations. Further planned efforts also target fault injection and emulation of the MPI 3 fault tolerance semantics to enable hardware/software codesign for fault tolerant algorithms [2, 3, 9]. Furthermore, next steps will also include more detailed analyses of HPC applications using xSim.

ACKNOWLEDGEMENTS

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at ORNL, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725. The U.S. 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] Advanced Supercomputing Division, National Aeronautics and Space Administration (NASA), Ames, CA, USA. "NAS Parallel Benchmarks (NPB) documentation", 2009.

[2] G. Bosilca, Z. Chen, J. Dongarra, and J. Langou. "Recovery patterns for iterative methods in a parallel unstable environment". *SIAM Journal on Scientific Computing (SISC)*, 30(1):102–116, 2007.

[3] Z. Chen and J. J. Dongarra. "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources". In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, p. 10. IEEE Computer Society, 2006.

[4] C. Engelmann and S. Böhm. "Redundant execution of HPC applications with MR-MPI". In *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*. ACTA Press, Calgary, AB, Canada, 2011.

[5] C. Engelmann and A. Geist. "Super-scalable algorithms for computing on 100,000 processors". In *Lecture Notes in Computer Science: International Conference on Computational Science (ICCS), Part I*, Vol. 3514, pp. 313–320. Springer Verlag, Berlin, Germany, 2005.

[6] C. Engelmann and F. Lauer. "Facilitating co-design for extreme-scale systems through lightweight simulation". In *IEEE International Conference on Cluster Computing (Cluster): Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC)*, pp. 1–8. IEEE Computer Society, 2010.

[7] S. Girona, J. Labarta, and R. M. Badia. "Validation of dimemas communication model for MPI collective operations". In *Lecture Notes in Computer Science: European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, Vol. 1908, pp. 39–46. Springer Verlag, Berlin, Germany, 2000.

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

[9] H. Ltaief, E. Gabriel, and M. Garbey. "Fault tolerant algorithms for heat transfer problems". *Journal of Parallel and Distributed Computing (JPDC)*, 68(5):663–677, 2008.

[10] K. S. Perumalla. "$\mu\pi$: A highly scalable and transparent system for simulating MPI programs". In *International ICST Conference on Simulation Tools and Techniques (SIMUTools)*. ACM Press, New York, NY, USA, 2010.

[11] G. Zheng, G. Kakulapati, and L. V. Kale. "BigSim: A parallel simulator for performance prediction of extremely large parallel machines". In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2004.