μπ: A Scalable and Transparent System for Simulating MPI Programs

Kalyan S. Perumalla
Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
perumallaks@ornl.gov

ABSTRACT

μπ is a scalable, transparent system for experimenting with the execution of parallel programs on simulated computing platforms. The level of simulated detail can be varied for application behavior as well as for machine characteristics. Unique features of μπ are repeatability of execution, scalability to millions of simulated (virtual) MPI ranks, scalability to hundreds of thousands of host (real) MPI ranks, portability of the system to a variety of host supercomputing platforms, and the ability to experiment with scientific applications whose source-code is available. The set of source-code interfaces supported by μπ is being expanded to support a wider set of applications, and MPI-based scientific computing benchmarks are being ported. In proof-of-concept experiments, μπ has been successfully exercised to spawn and sustain very large-scale executions of an MPI test program given in source code form. Low slowdowns are observed, due to its use of purely discrete event style of execution, and due to the scalability and efficiency of the underlying parallel discrete event simulation engine, μsik. In the lAZargest runs, μπ has been executed on up to 216,000 cores of a Cray XT5 supercomputer, successfully simulating over 27 million virtual MPI ranks, each virtual rank containing its own thread context, and all ranks fully synchronized by virtual time.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance – *Simulation*, *Operational Analysis*; D.4.4 [**Operating Systems**]: Communications Management – *Buffering*, *Message Sending*, *Network Communication*; D.4.8 [**Operating Systems**] Performance – *Operational Analysis*; I.6.1 [**Simulation and Modeling**] General; I.6.3 [**Simulation and Modeling**] Applications

General Terms

Parallel algorithms, Performance, Experimentation

Keywords

MPI, Virtual Execution, Supercomputing, Synchronization

1. INTRODUCTION

μπ is designed to solve the following problem: Given a parallel application A_α, estimate the characteristics it would exhibit if it were executed on a (imaginary, simulated) machine M_α. The tool for such experimentation, which in general is a parallel application A_β in its own right, is to be executed on another (real) machine M_β in an as-fast-as-possible manner. μπ is such an A_β system, which we design as a parallel discrete event simulation (PDES) system that simulates the execution of any complex Message Passing Interface (MPI)-based application A_α on any user-configured imaginary machine M_α. μπ itself executes in a scalable way on any suitable parallel cluster or supercomputing platform M_β such as a Cray XT5 or Blue Gene/P system.

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.
Copyright 2010 ICST, ISBN 78-963-9799-87-5.

Typically, experimentation with A_α is motivated by the desire to observe metrics such as computation time, blocked time, memory consumption, and network load. Additional aspects include software engineering concerns such as uncovering software shortcomings or other unknown behavior when the application is executed on a larger number of processors than feasible before.

Ideally, the experiments should be (a) repeatable, (b) accurate, and (c) fast, in that order of preference. However, when the complexity of the application A_α and/or the machine M_α increases, it becomes a harder endeavor to satisfy all the preceding three criteria at once. In particular, when the targeted number of processing elements in M_α is on the order of 10^5-10^7 (comparable to existing and emerging peta-scale and exa-scale platforms), a specially designed scalable and efficient experimentation system is needed to meet the goals. μπ is designed to fill this need.

The name μπ is a Greek abbreviation for the acronym MUPI standing for **micro** parallel **p**erformance **i**nvestigation system.

Here, we document a snapshot of work in progress in our design, development and usage of μπ for large-scale experimentation of MPI-based parallel scientific codes. The focus of this report is on documenting proof-of-concept scaling results from a prototype μπ implementation.

1.1 Motivation

In reference [1], the need for simulating the execution of parallel programs is well articulated, based on the fact that the average useful lifetime of a parallel machine is 5 years, whereas that of a parallel application (source code) is up to 20 years. There are many additional motivating factors that are well documented in the literature (please see the related work in Section 1.3). An environment for debugging, testing and customizing existing applications to new parallel hardware and software platforms would be useful to improve the utilization of many resources. Purely analytical approaches alone are inadequate to effectively experiment with parallel applications, due to the increasing scale and widening diversity of hardware platforms, middleware systems, software interfaces, and parallel algorithms. Ideally, an existing application would be minimally modified and simply executed on a newly envisioned parallel system before that new system is built. Not only can the application benefit from the performance insights, but also the system designers can use the observed behavior to improve the system design.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8692
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8692

Important considerations in the design of μπ are support for memory-constrained programs, large number of real processors, and controlled execution of actual, unmodified codes on virtual platforms with one to two orders of magnitude more numerous MPI ranks than the number of cores available in real hardware.

1.2 Scaling Challenges

Scaling the simulator to very large number of processors requires new scalable methods. Such scaling issues of the simulator have not been as important in earlier systems either due to use of relatively small number of processors or due to abstraction approaches that rely on traces or execution models without having to execute actual applications on simulated machines.

An example of the simulator scaling challenges is in the virtualization of the **`MPI_Barrier()`** method. When the application invokes the barrier call, the call is trapped by the simulator. The elapsed time for satisfaction of the barrier is simulated, with either a user-specified or algorithm-induced total delay. A user-specified runtime cost for barrier calls could be used to experiment with a desired or fixed (e.g., from a design goal) global reduction latency, or an algorithm-induced latency could be simulated by executing an actual reduction algorithm over a simulated (point-to-point or hardware-optimized) network.

Another challenge arises in the actual transfer of data exchanged by the MPI ranks at runtime. In order to support unmodified code execution, μπ must implement actual exchange of bytes across MPI ranks initiated via MPI communication primitives such as **`MPI_Send()`** and **`MPI_Recv()`**. In effect, the simulator acts as the underlying network layer that performs packetization for large message payloads, and undertakes buffering and reordering as needed. Such concerns are absent in lower fidelity systems that do not need to support actual data exchanges due to use of abstractions.

Yet another challenge in multiplexing multiple virtual MPI ranks on each core is the cost of scheduling and de-scheduling threads within the simulator. For minimizing overall slowdown, the simulator must be optimized to multiplex the virtual MPI ranks in a (virtual-) time-synchronized fashion.

1.3 Related Work

μπ is focused on experimentation with very large (virtual) parallel computing configurations, especially with hundreds of thousands of (virtual) processor cores. We are not aware of any systems that are able to sustain experiments at such a high level of scale.

Several related systems have been reported in the literature for experimenting with applications on simulated machine configurations. A selected list includes FASE [2], MPI-Sim [3], POSE [4], PDES [5, 6], Direct Execution[7-9], optimistic simulation [10], task graphs [11], synchronization-optimizations [12], BigSim [13, 14], EMPOWER [15], and hybrid modeling [16]. Full system simulators such as Wisconsin Wind Tunnel [17], and SIMICS [18] may impose unacceptable levels of slowdowns for the purpose of large parallel program execution, making them difficult to use in practice for virtual machine configurations containing 10^5-10^6 processor cores. A survey article [1] captures the details of some of the important performance prediction systems, including PAL, Performance Prophet, POEMS, POSE, and RSIM. Recently, the hurdle of simulation speed is addressed via parallel simulation for instruction-level models [19]. Trace-based performance modeling is another commonly-used method for experimenting with simulated machine configurations. Most aforementioned systems have been limited by the sizes of the parallel machines available at their times. Almost all the systems reported so far were exercised with fewer than 1,000 processor cores used to execute the simulation, although much larger *virtual* configurations were evaluated in experiments. Among the most prominent in scaling the virtual configurations is the BigSim/POSE [13, 14] efforts that reported low slowdowns in experiments of 64,000 virtual processor simulated on 512 processors. Our work differs from all the related work in our focus on combining the goals of executing un-abstracted, unmodified applications with very large simulated configurations. The work that comes closest to our goals is the LAPSE system [7] that was designed to support Intel Paragon applications using Intel's communication interfaces. Our work differs in that we focus: (a) on MPI programs, (b) on very large numbers of MPI ranks (10^4-10^5 real, 10^5-10^7 virtual), and (c) on using multi-threading instead of UNIX processes for virtualizing the multiplexed identities of the application processes.

1.4 Contributions

To the best of our knowledge, ours is the first to apply process-oriented simulation [20-22] to simulate MPI programs. We employ multi-threading to sustain and multiplex many simulated MPI ranks on each processor used by the parallel simulator. Efficient, time-synchronized scheduling and de-scheduling are realized to implement the overall virtual execution, resulting in very efficient emulation. Our work is also among the first to execute unmodified MPI applications on thousands of (actual) processor cores, with one or more simulated MPI ranks per core. The support for *scalable* implementation of *virtual* MPI primitives (MPI calls made by the application, trapped and supported by the simulator) is also novel. Quantitatively, μπ is the only system to report feasibility and actual time-synchronized execution of millions of virtual, unmodified MPI ranks. No system reported in literature approaches this scale, even for the simplest *unmodified* MPI benchmarks. The closest work is the seminal LAPSE system of the 1990's tested on 512 virtual tasks on an Intel Paragon.

The factors that make μπ scale better than others are (a) better implementation of process-oriented execution needed for virtualizing unmodified MPI code, (b) superior efficiency of the PDES engine helping scale to orders of magnitude larger platforms than other MPI simulator tools in the literature, and (c) scalable design and *implementation* of the Message Passing *Interface*.

It is important to note that the focus of this paper is the design and preliminary evidence of feasibility on large platforms of the simulation tool. This article does not address validation issues. Moreover, this paper documents work in progress; hence it does not yet include a comprehensive performance evaluation.

2. DESIGN

Recall that the subject application A_α is to be virtually executed on some P_α processors of the imaginary (guest) machine M_α, but needs to execute on (typically much smaller) number of processors P_β of a real (host) machine M_β. For example, if the application is to be experimented on 100,000 processor cores, but only 10,000 cores are available for experimentation, then the application must be somehow "fooled" into the view that it is executing on 100,000 cores. Since more than one virtual core is "simulated" on each real core, timing results perceived by the application have to be adjusted to erase the effects of multiplexing. Message buffering and re-ordering have to be similarly accurately recreated as they would have occurred in the envisioned machine network, even though messages might arrive faster or slower on the host machine compared to the envisioned machine. The full design involves three components: (1) grafting (2) time pacing (3) communication buffering and reordering. Each of these components is described next.

2.1 Grafting

The grafting portion of the prediction system deals with the software interface-related mismatch between the guest parallel machine M_α and the host parallel machine M_β. The original software interface (e.g., MPI) that the application uses should be retained, but the implementation must be changed (preferably, transparently) to reflect the behavior of the new machine. We call the redirection as a grafting process. There are multiple grafting methods, each with its own merits and demerits. A source code-based grafting method is best when the source code of the application is available. A library redirection-based grafting method is best to deal with application functionality that is available only in the form of pre-built object libraries; however, depending on the complexity of the libraries involved, the grafting systems can involve significant amount of systems work. A virtual machine-based grafting is possible for the highest levels of transparency in the grafting process, but also is the most expensive with respect to runtime cost.

In the initial version of μπ, the source code-based grafting method is supported. Its usage and implementation are described here. When the source code is available for the MPI application, μπ supports grafting with the use of the header file **`mupi.h`** and library **`libmupi.a`** that μπ supplies. In fact, this can be easily made transparent by using installation-specific customization of the native MPI header file and linked libraries. Similar customization is used for MPI-based FORTRAN.

The application is compiled and linked as usual. The resulting executable will be a μπ simulation, which executes in virtual mode. On platforms on which μπ itself executes over MPI, "**`mpirun -np 4 myprog -nvp 32`**" runs **`myprog`** on 32 virtual ranks, simulated by μπ on 4 real cores.

In **`mupi.h`**, the original MPI routines are redefined, via macro substitution, to μπ calls of correspondingly identical signatures, so that μπ can execute them on a simulated platform. The original main routine is renamed and automatically invoked by μπ for each virtual MPI task of the application (command line arguments are duplicated for each virtual task invocation). μπ allocates and creates a separate stack context for each virtual MPI task, and maintains the full stack context between MPI calls by the task.

The MPI routines implemented, in C/C++ and FORTRAN, as of this writing are **`MPI_Init()`**, **`MPI_Finalize()`**, **`MPI_Comm_rank()`**, **`MPI_Comm_size()`**, **`MPI_Barrier()`**, **`MPI_Send()`**, **`MPI_Recv()`**, **`MPI_Isend()`**, **`MPI_Irecv()`**, **`MPI_Waitall()`**, **`MPI_Wtime()`**. Only the **`MPI_COMM_WORLD`** communication group is currently recognized. The set is being expanded, which is mostly developmental in nature.

2.2 Time Pacing

Each virtual MPI rank is realized as the threaded simulation process of μsik. As mentioned earlier, for every MPI rank, μπ traps all MPI calls, and acts as the scheduler that ensures execution is ordered by simulation time. An important MPI call is the **`MPI_Wtime()`** that gives the invoking MPI rank a snapshot of current "wall clock" time: μπ returns the simulation time, instead of the wall clock time, making the rank synchronized with the timelines of the rest of the simulation. Any blocked call, such as **`MPI_Barrier()`**, **`MPI_Recv()`** or **`MPI_Wait()`**, is realized using the "hold" primitive of process-oriented simulation. The equivalent "hold" primitive in μsik is the **`wait(WaitContext &)`** which is used by μπ's trapped MPI calls to either advance simulation time unconditionally, or advance simulation time until a given condition is satisfied (e.g., message arrival).

2.3 Buffering and Reordering

Just as in a real distributed system operation, MPI message data may arrive at various moments in time. In the simulation, the data arrives as data events, which need to be buffered inside the simulator until a corresponding (local) virtual MPI rank queries and or accepts the data using an appropriate MPI call. μπ handles all the buffering (in an assembly queue and a ready queue, for each MPI rank), and performs reordering internally by time-stamp order, and supplies the data as part of the appropriate MPI invocations.

2.4 Machine Specification

The characteristics of the envisioned machine can be specified programmatically, as an object library linked to the μπ simulation. The library can be invoked by the user as part of an experimentation system to suitably customize the scenario in a series of experiments. For unsophisticated usage of μπ, a simple machine performance model is provided (the μπ design allows for far more sophisticated machine models). In this simplified usage, the user can specify (via environment variables) a point-to-point bandwidth and a point-to-point latency.

3. IMPLEMENTATION

The μπ software is written C/C++, as an application of the μsik PDES engine [23]. Both μπ and μsik are portable to a large number of platforms. μπ has been tested on MPI-based platforms, including Linux, Mac OS X, Blue Gene/P, and Cray XT4/XT5.

The timing model aspect of MPI program simulation deals with how the computing time is accounted and incorporated into the simulation. The time consumed by computation in the application between MPI calls must be used by the simulator to advance simulation time once μπ is entered via an MPI call trap. The communication time is the time spent inside MPI before returning control back to the application. Such a framework is depicted in Figure 1.

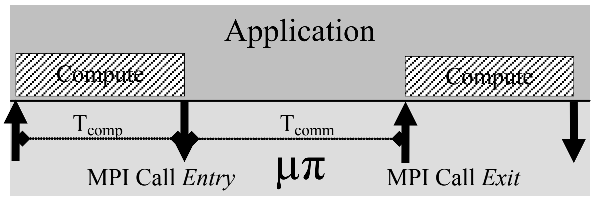

Figure 1: Time accounting framework used by μπ

It is important to note that this is only a framework (interface), and does not constrain the complexity of the timing models (implementation) that could be employed. Any arbitrary level of timing detail can be incorporated. Compute-time can be charged to application with arbitrarily complex models (example: full-system simulation of instructions on the side, cache effects, processor speed, etc.), to the extent the user is willing to suffer the consequent run time overhead. Similar argument holds for network effects. Our simulation tool core is unaffected; detail can be added or abstracted by user, depending on level of detail needed. Thus, the framework in Figure 1 only depicts generalized, streamlined hooks, but do not constraint the model.

A simple way to use the framework is one in which the guest and host processor architectures are identical. In this case, the computing time T_{comp} can be automatically gleaned by µπ simply based on the amount of elapsed system time between the most recent MPI call exit and the current MPI call entry. µπ then simulates the elapse of that amount of simulated time by using the **`wait(dt)`** primitive of the corresponding µsik process to "hold" **`dt`** amount of simulated time.

To simulate other processor architectures, the gleaned time can be adjusted appropriately with a scaling correction factor (e.g., based on ratio of clock speeds). Even more sophisticated schemes are possible, via heterogeneous processor emulation, albeit at a significant performance cost, but such more elaborate schemes are also conceivable to incorporate into µπ if deemed important.

This framework can also be used to increase simulation efficiency (without affecting the program dynamics) by simply estimating the computation time and invoking a µπ primitive called **`MPI_Elapse_time(dt)`** that is provided to avoid consuming host processor's cycles simply to simulate guest processor's cycles for virtual time period of **`dt`** units.

4. PERFORMANCE STUDY

A functional prototype of µπ is operational, and has been tested on a variety of platforms. Here, we focus on simulator performance, and relegate validation studies to another document. The criteria of interest are how the runtime efficiency fares in some of the most stringent (worst case) hardware scenarios. We choose a parameterized benchmark to test the simulator performance effects, especially uncovering the synchronization and rank multiplexing effects.

The experiments were executed on two platforms: (1) a Cray XT4 consisting of quad-core Opteron Budapest processors and 8GB of memory per node, with nodes connected by a SeaStart2 interconnect, and (2) a Cray XT5 with 224,256 compute cores spanning nodes containing two hex-core AMD Opteron processors, 16GB memory, all nodes being interconnected by a SeaStar 2+ router.

The benchmark is a ping benchmark designed to test the scalability of the most common and basic MPI primitives, namely, **`MPI_Barrier()`**, **`MPI_Send()`** and **`MPI_Recv()`**. Each rank receives a message from its left neighbor and sends a message to its right neighbor. After every receive-send pair of operations, a barrier is invoked all ranks. The number of bytes sent in each message is doubled after every round. The parts challenging to the simulator are the fast simulation of the barrier for every round, and the communication of the messages in the correct order at every rank.

It is well-known [5, 7, 9] that lookahead [20] is one of the most limiting factors in parallel system simulation. Here, we present one set of results on the worst-case scenario of zero-lookahead, and another set of results corresponding to a low, 10µs lookahead (from inter-node latency). Inter-processor networks in both sets also represent a high 1Gbps bandwidth for every inter-rank message. Thus, the results here can be viewed as worst case; hardware configurations. Larger latencies in practice can hence be expected to be simulated even faster.

4.1 Zero Lookahead Scenarios

Two scenarios are chosen: (1) communication-intensive, in which every MPI rank simulates 1 millisecond worth of computation and 8KB worth of data exchange with a neighbor for every epoch (2) computation-intensive, in which every MPI rank simulates 100 milliseconds worth of computation and 1KB worth of data exchange. Each epoch is guarded by every MPI rank invoking **`MPI_Barrier()`**.

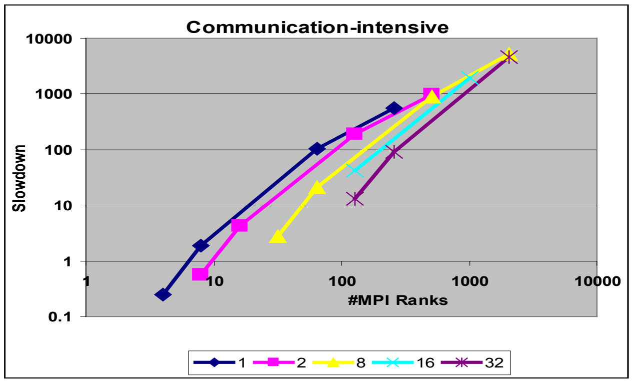

Figure 2: Runtime slowdown in the communication-intensive scenario

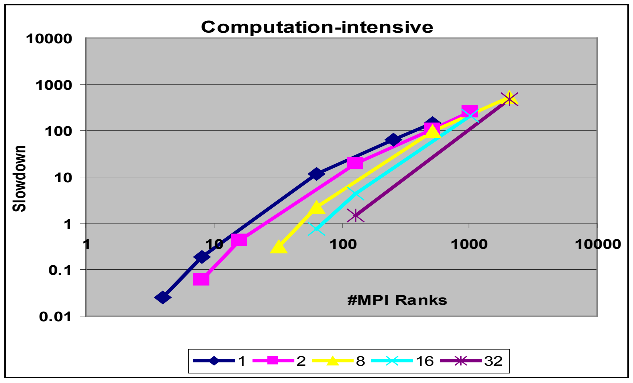

Figure 3: Runtime slowdown in the computation-intensive scenario

The performance metrics of interest are the slowdown factor, the total number of events processed, and a measure of the amount of synchronization performed per event. The slowdown factor is the ratio of the elapsed time taken by the simulation to the virtual time at the end of simulation in the simulated machine. It is clear that the smaller the slowdown factor the faster the simulation. Also, a slowdown less than unity in fact implies speed up, in which the simulation is performed faster than the execution on the simulated machine. The number of events is important to observe how much overhead the simulator induces via events for simulating data transmission and other internals such as a tree (butterfly) pattern-based barrier. The number of events per synchronization (i.e., computation of a lower bound on time stamp (LBTS) needed for parallel time-ordered execution) is an important measure of the amount of concurrency available in the simulation (which is directly related to lookahead). A higher lookahead value can be expected to give larger number of events per LBTS computation.

The runtime performance in terms of slowdowns (or speedups) is shown in Figure 2 and Figure 3 for the two scenarios respectively. The slowdown factor is plotted against the number of simulated MPI ranks. Each line represents the ratio of the number of simulated ranks to the number real cores used for simulation (e.g., the purple line represents a ratio of 2, with two virtual MPI ranks simulated by μπ for each real core on which μπ executes). As expected, the computation-intensive scenario is simulated much faster than the communication-intensive scenario because the computation is simply an elapse of simulation time (which is achieved by scheduling an event into the future for each leap in computation time), and because of the smaller number of data events infrequently exchanged. In fact, the simulation is observed to experience speed up on smaller numbers of MPI ranks.

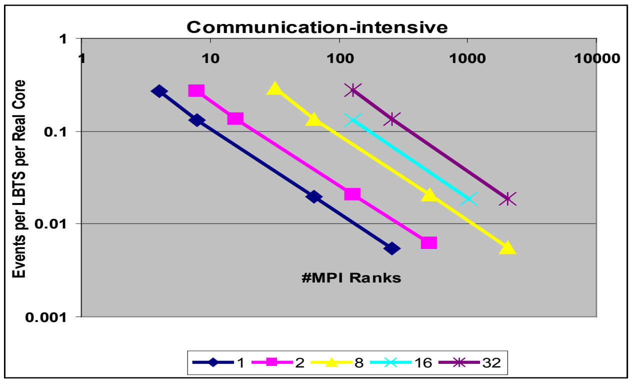

Figure 4: Events processed per synchronization in the communication-intensive scenario

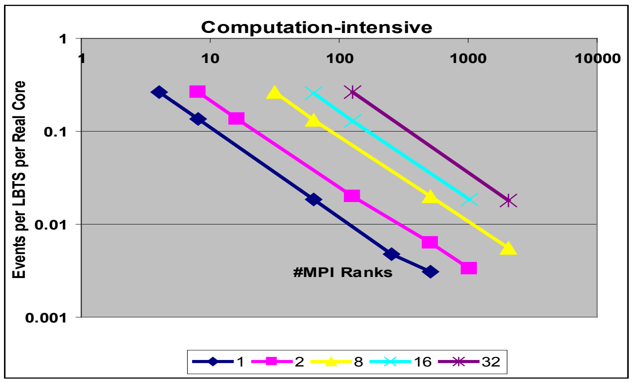

Figure 5: Events processed per synchronization in the computation-intensive scenario

Note that these are under the most stringent lookahead constraints, namely, assuming extremely fast inter-rank network connections in the simulated machine hardware. Slower machine configurations experience much less simulation overhead, due to greater concurrency permitted by inter-rank latency distances. The fact that it is the low lookahead (and hence higher synchronization cost) that contributes to slowdowns on larger configurations (e.g., 2048 MPI ranks) is discerned by observing the number events available to process by μπ per parallel synchronization operation. Figure 4 and Figure 5 show the decrease in the amount of concurrency permitted by the simulated machine, on communication-intensive and computation-intensive scenarios respectively. Due to increasingly larger number of staggered events across the virtual MPI ranks (hence, across μπ real cores), zero lookahead necessitates synchronization often. Thus, the number of synchronization operations increases and the number of events processed per LBTS computation per core decreases. This represents the worst case, and is easily improved on less stringent simulated-machine specifications (e.g., clusters).

4.2 Scaling Scenarios

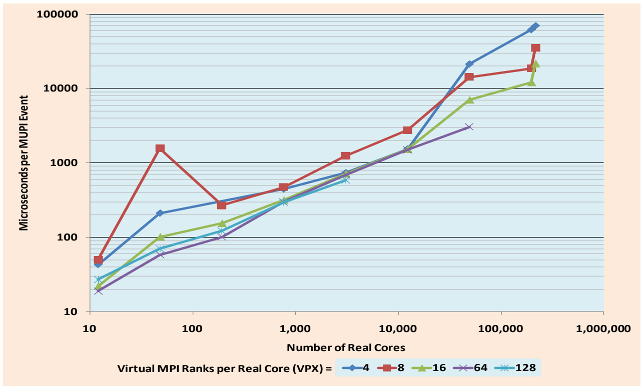

Figure 6: Average run time cost incurred per event

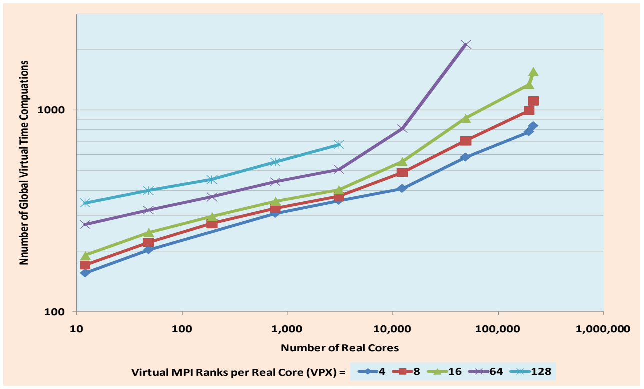

Figure 7: Sub-linear trend of increase in the number of synchronization operations with number of real cores

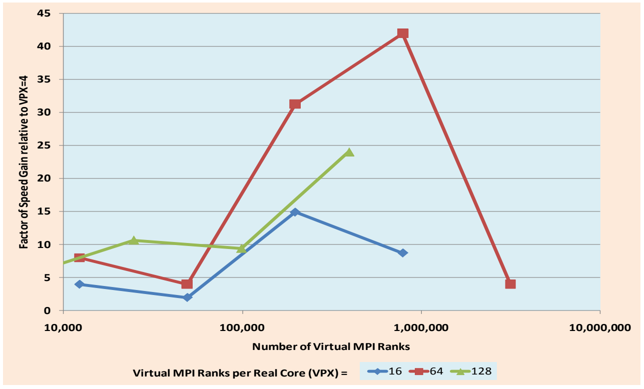

Figure 8: Non-linear nature of performance with variation in number of virtual ranks per real core

The next set of results is designed to highlight μπ's scalability. The computation-intensive scenario of the previous set of results is used and scaled on Cray XT5 with two variables: (1) number of real cores used in the parallel simulation, and (2) the number of virtual MPI ranks mapped to each real core. Figure 6 shows the run time cost in microseconds incurred per event. It is seen that the average increases by one less order of magnitude than the number of cores. Performance improvement is clearly possible;

we are currently investigating this. Figure 7 shows the number of synchronization operations against the number of real cores. An insignificant growth is observed in synchronization operations for the simulation with the number of cores, suggesting very efficient operation with respect to global simulation synchronization of μsik even on the largest number of real cores. Figure 8 shows a very interesting, non-linear relation between the number of virtual MPI ranks mapped to each real core and the number of virtual MPI ranks being simulated. While performance is better on smaller number of cores with larger number of ranks per core, the same is untrue on very large number of cores. Although we are still investigating this phenomenon in greater detail, our explanation for this is that there is a competitive relation between two effects: (a) the shared-memory effect of communication by virtual MPI ranks mapped to the same cores, and (b) the complex variation in the cost of virtual time synchronization.

5. SUMMARY AND FUTURE WORK

Investigation of execution effects of existing applications on new parallel platforms is gaining more importance recently with the rapidly increasing sizes of parallel computing installations. The need to experiment with MPI codes is immediate, yet few simulation tools exist to help execute complex parallel codes at the scales of 10^4-10^7 MPI ranks on simulated machine configurations. μπ is being developed towards meeting this need, towards the ultimate goal of taking unmodified MPI applications' source code and executing it on a simulated machine with up to 10^5-10^7 imaginary MPI ranks. Here we report work in progress in its design and preliminary performance characteristics. The results are very encouraging, providing acceptable levels of slowdowns even under the most stringent simulated machine configurations (zero lookahead). On scenarios with a virtual network latency of 10μs, μπ sustained over 27 million virtual ranks, and execute on up to 216,000 cores of a Cray XT5. We are currently porting NAS benchmarks and other complex applications to execute on large simulated computers in μπ.

Acknowledgements This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Dept. of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. 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 U.S. Government purposes. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, supported by the Office of Science of the U.S. Dept. of Energy under Contract No. DE-AC05-00OR22725. The author is grateful to Sudip Seal and Vinod Tipparaju for many helpful comments and discussions.

REFERENCES

[1] S. Pllana, *et al.*, "Performance Modeling and Prediction of Parallel and Distributed Computing Systems: A Survey of the State of the Art," International Conference on Complex, Intelligent and Software Intensive Systems, 2007.

[2] E. Grobelny, *et al.*, "FASE: A Framework for Scalable Performance Prediction of HPC Systems and Applications," *Simulation,* vol. 83, 2007.

[3] S. Prakash and R. Bagrodia, "MPI-Sim: Using Parallel Simulation to Evaluate MPI Programs," in *Winter Simulation Conference*, 1998.

[4] T. Wilmarth, *et al.*, "Performance Prediction using Simulation of Large-Scale Interconnection Networks in POSE," in *Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[5] J. Liu, *et al.*, "Performance Prediction of a Parallel Simulator," in *Workshop on Parallel and Distributed Simulation*, 1999.

[6] K. S. Perumalla, *et al.*, "Performance Prediction of Large-scale Parallel Discrete Event Models of Physical Systems," in *Winter Simulation Conference*, 2005.

[7] P. Dickens, *et al.*, "Parallelized Direct Execution Simulation of Message-Passing Programs," *IEEE Transactions on Parallel and Distributed Systems,* vol. 7, 1996.

[8] M. Hibler, *et al.*, "Feedback-Directed Virtualization Techniques for Scalable Network Experimentation," University of Utah, Technical Report, 2004.

[9] J. Liu, *et al.*, "Simulation validation using direct execution of wireless Ad-Hoc routing protocols," 18th Workshop on Parallel and Distributed Simulation, 2004.

[10] T. Phan and R. Bagrodia, "Optimistic Simulation of Parallel Message-Passing Applications," 15th Workshop on Parallel and Distributed Simulation, 2001.

[11] C. Roig, *et al.*, "Modeling Message-Passing Programs for Static Mapping," 8th Euromicro Workshop on Parallel and Distributed Processing, 2000.

[12] S. Prakash, *et al.*, "Asynchronous Parallel Simulation of Parallel Programs," *IEEE Transactions on Software Engineering,* vol. 26, 2000.

[13] G. Zheng, *et al.*, "Simulation-based Performance Prediction for Large Parallel Machines," *International journal of Parallel Programming,* vol. 33, 2005.

[14] G. Zheng, *et al.*, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," International Parallel and Distributed Processing Symposium, 2004.

[15] P. Zheng and L. M. Ni, "EMPOWER: a scalable framework for network emulation," in *International Conference on Parallel Processing*, 2002.

[16] S. Pllana, *et al.*, "Hybrid Performance Modeling and Prediction of Large-Scale Computing Systems," International Conference on Complex, Intelligent and Software Intensive Systems, 2008.

[17] S. K. Reinhardt, *et al.*, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. vol. 21, 1993.

[18] P. S. Magnusson, *et al.* (2002, 2002/02/01) Simics: A Full System Simulation Platform. *IEEE Computer*. pp. 50-58.

[19] E. A. Leon, *et al.*, "Instruction-level Simulation of a Cluster at Scale," Supercomputing, 2009.

[20] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*: Wiley Interscience, 2000.

[21] K. S. Perumalla and R. M. Fujimoto, "Efficient Large-Scale Process-Oriented Parallel Simulations," in *Proceedings of the Winter Simulation Conference*, ed, 1998.

[22] T. J. Schriber, *Simulation Using GPSS*. John Wiley & Sons, 1974.

[23] K. S. Perumalla, "μsik - A Micro-Kernel for Parallel/Distributed Simulation Systems," in *Workshop on Principles of Advanced and Distributed Simulation*, 2005.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8692
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8692