Improving Multi-Million Virtual Rank MPI Execution in $\mu\pi$

Kalyan S. Perumalla, Alfred J. Park
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
perumallaks@ornl.gov, parkaj@ornl.gov

***Abstract*—$\mu\pi$ (MUPI) is a parallel discrete event simulator designed for enabling software-based experimentation via simulated execution across a range of synthetic to unmodified parallel programs using the Message Passing Interface (MPI) with millions of tasks. Here, we report work in progress in improving the efficiency of $\mu\pi$. Among the issues uncovered are the scaling problems with implementing barriers and inter-task message ordering. Preliminary performance shows the possibility of supporting hundreds of virtual MPI ranks per real processor core. Performance improvements of at least $2\times$ are observed, and enable execution of benchmark MPI runs with over 16 million virtual ranks synchronized in a discrete event fashion on as few as 16,128 real cores of a Cray XT5.**

***Keywords*-Message Passing Interface, Parallel Discrete Event Simulation, Virtual Execution, Exascale**

I. Introduction

Motivation: Here, we attempt to further the state of the art in large-scale simulations of parallel programs by exploring and addressing some of the issues in sustaining very large number of MPI-based control-flows, in the form of millions of virtual MPI ranks. To simulate them, we enhance the efficiency of parallel discrete event simulation methods to support the simulation of a few million ranks on a few thousand of actual cores.

Background: $\mu\pi$ [2] is a process-oriented simulator where each task or thread is represented as a logical process (LP). Each LP is maintained as a unique thread and respective execution contexts (i.e. stack) are preserved. $\mu\pi$ maintains all requisite information in order to suspend and resume simulated virtual MPI ranks as needed by the underlying simulation executive μsik [1]. This allows $\mu\pi$ to multiplex multiple virtual MPI ranks onto each available real processor and support unmodified MPI codes within a simulated environment. $\mu\pi$ runs in a purely parallel discrete event style of execution for the best performance on a wide range of virtual program workloads.

A hierarchical structure is adopted to accommodate the large amount of virtual MPI ranks across the limited amount of processing resources available. Each node may contain multiple processor sockets, and each processor may have multiple cores. Each physical core handles a pre-assigned number of virtual MPI ranks, and these are time-multiplexed on their assigned processing core.

Related Work: A relatively large body of past work covered the problem of simulating computer systems at a high level of fidelity. A survey article [3] captures the details of some of the important performance prediction systems. Disposition relative to several of these past and ongoing works has been documented in our earlier article on $\mu\pi$[2]. Our focus is on software-level experimentation, for either experimenting with existing MPI-based codes to be scaled, or writing skeleton MPI codes from scratch that are designed for scale from the outset, and into which computational meat will be infused later as application-specific computation is included in an evolutionary fashion.

Here, we report some preliminary work on improving the efficiency of $\mu\pi$, in terms of scalable support of MPI's point-to-point message ordering semantics, and increasing the run time efficiency of $\mu\pi$ virtual barrier implementation.

II. Discrete Event Model of FIFO Semantics

MPI implementations must map semantics such as First-In-First-Out (FIFO) message ordering for point-to-point communications, since $\mu\pi$ is, in a general sense, an MPI implementation. The FIFO requirements must be somehow accomplished and supported in $\mu\pi$'s data movement. This raises an interesting challenge in correctly and efficiently mapping the message-ordering to one that operates on timestamp-based ordering of discrete events. The modeled network for data transmission is distinct from any actual network used as a conduit for the simulation itself, and thus must ensure that important MPI FIFO ordering is preserved.

Problem: MPI guarantees FIFO ordering of messages within each communicator regardless of the underlying network. This must be accurately reproduced by the simulator in order to generate correct and repeatable results. In real applications, sending a piece of data typically only requires information about the data itself and to whom to send. If this is translated directly by the simulation environment, incorrect execution will result. This is due to an absence of any semantics and ordering without a full simulated underlying network (e.g., a simple data transmission model that only adds delay incurred by latency and bandwidth).

The core of the problem is that the simulated messaging protocol is stateless. A current message being sent has no prior knowledge of the state of the network. There are various solution approaches to this problem for maintaining a stateful messaging protocol to preserve FIFO ordering, but none of the traditional approaches is scalable.

Scalable Solution Approach: A scalable solution that we developed to the FIFO ordering problem is to record a small amount of pair-wise state of the message sent to the receiver at the sender side only. Effectively this is logically equivalent to a very lightweight network link model that is appended to the sending virtual MPI rank for each destination. There is no need to pre-allocate every potential destination virtual MPI rank; instead, this state is only maintained for the most recently transmitted data. Therefore, memory is only allocated on demand when a transmission is outstanding to an (arbitrary) receiver. Memory is reclaimed on a per-active-receiver basis if any state regarding most recent messages is no longer relevant.

By maintaining the state of the most recent oustanding transmission between itself and the receiver, the proper adjustments to the receive timestamp can be made, and, these simulation messages can be consumed in proper timestamp order on the receiver.

III. Virtual Barrier and Virtual Collectives

Native Barriers Unsuitable for Virtual Barriers: While obvious to a simulation expert, an important aspect that is often misunderstood is that native barriers cannot be employed as-is to simulate a barrier. In other words, although the native hardware may contain a highly efficient barrier implementation, it is not possible for every virtual rank to simply invoke that implementation to realize its virtual barrier functionality. Such blocking on native calls interferes with simulation time advances and, in effect, pollutes the distinction between wall clock time and simulation time. At best, runtime errors such as deadlock conditions arise, and at worst, silent, incorrect results are obtained.

When native implementation cannot be used to improve performance, the only way to improve the scalability and speed of the simulation is to optimize the implementation of the virtual barrier itself by developing simulation-specific enhancements. Here, we undertake precisely such an effort, to design a method that can sustain large number of virtual ranks per real rank (or per real core).

Optimized Virtual Barrier Algorithm: Conventional butterfly barriers must perform pairwise barriers across all processors within the system. Our optimized approach allows efficient local shared memory operations and then only communicating the minimum amount of information between leader processors. The optimized approach effectively reduces two levels of the hierarchy from the butterfly process, which can result in a significant reduction in inter-node communication. Our optimized barrier algorithm is outlined in Algorithm 1. Every virtual rank executes this algorithm as part of the implementation of the virtualized `MPI_Barrier()`.

The key to this algorithm is the use of a variable called `njoined` that is globally visible to all virtual ranks mapped to a core (each core contains its own instance of this variable), initialized to zero. Every rank increments this variable upon entry into the barrier. Here, the number of virtual ranks per core is called LPX. The ranks mapped to a core are referred to as local ranks, and the first virtual rank at any core is the leader rank (i.e. local core ID 0). Since ranks may join the barrier at any arbitrary points in simulation time (and hence in any relative order), exactly two possibilities exist among the local ranks: (1) the leader happens to arrive at the barrier last, or (2) a non-leader arrives at the barrier last. In the first case, the leader detects `njoined` to be equal to LPX, thus the leader knows it has joined last. No additional events are necessary to coordinate the "join" phase of the barrier among local ranks, and can proceed with its leadership role representing all the local ranks. In the second case, with the leader arriving early (by detecting `njoined` is less than LPX), it proceeds to wait on an event reception. When the final local rank arrives last, it detects that is indeed the case and sends an event to the leader, completing the local "join" phase.

The rest is a relatively straightforward use of aggregation at each node (among all cores) to minimize inter-node communication, and then use of a butterfly pattern across nodes for a fast barrier operation. Details are omitted here

Algorithm 1 Optimized Virtual Barrier

```
1:  Integer njoined {global variable initialized to zero}
2:  LPX = {number of virtual ranks simulated per core}
3:  Virtual MPI Barrier operation at every virtual rank
4:  Increment njoined {guaranteed to be atomic due to event loop}
5:  if my rank modulo LPX > 0 then
6:    if njoined == LPX then {I joined last locally}
7:      Send BarrierEvent to local leader {whose rank modulo LPX is 0}
8:    end if
9:    Wait for BarrierEvent from local leader
10: else {I am local leader}
11:   if njoined < LPX then
12:     Wait for BarrierEvent from any local rank
13:   end if
14:   njoined = 0 {reset to zero}
15:   if my local core ID on my node > 0 then
16:     Send BarrierEvent to local core ID 0
17:     Wait for BarrierEvent from local core ID 0
18:   else {I am on local core ID 0}
19:     Wait for BarrierEvent from all local cores
20:     Participate in butterfly communication among cores with ID 0 on all nodes
21:     Send BarrierEvent to all local cores
22:   end if
23:   Send BarrierEvent to all local ranks on my core
24: end if
```

for dealing with non-powers of two, in which care must be taken to avoid long inter-node distance communication for "outlier" nodes that fall in the non-power-of-two region.

This optimization template carries over well to other global collectives as well, such as `MPI_Allreduce()`, which we intend to explore and optimize in future work. The main difference between barrier and other collectives is that other collectives carry data and involve operations on the data in addition to the global coupling. The data, however, can also be accommodated in our barrier algorithm template by storing the per-rank data in a local shared-memory queue during the join phase, and distributed via the same buffers in the release phase.

IV. Performance Study

We turn to an experimental study to measure the performance improvements from the optimizations, and test scalability with respect to the number of virtual ranks. We implemented the optimizations in $\mu\pi$, and tested it on two different platforms: (1) an Infiniband-connected Linux cluster called Frost containing 1,024 cores, and (2) a Cray XT5 machine containing several thousand cores. The former, being the more easily accessible resource of the two machines, was used for an exhaustive number of runs, while the latter was used for selective runs aimed at demonstrating the ability to execute at very large-scale.

Experiment Setup: Frost is an SGI Altix ICE 8200 cluster containing 2 quad-core Intel Xeon X5560 2.8GHz processors per node. SMT was enabled on these nodes, providing 16 hardware threads per node for a total of 2,048 hardware threads across the entire cluster. All experiments filled each node with 16 $\mu\pi$ simulator processes. Each node has 24GB of memory with an Infiniband interconnect. The Intel C/C++ compiler 11.1.059 with `-O3 -ipo -xsse4.2` compilation flags was used to compile all software on this platform.

The Cray XT5 system contains 2 hex-core AMD Opteron 2435 (Istanbul) 2.6GHz processors per node. Each node has 16GB of memory with communications through Cray's SeaStar 2+ router. The Portland Group (pgi) compiler 2.2.73 with `-O3 -fast` compilation flags was used to compile all software all software on this platform.

A `barriertest` benchmark we use for this is aimed at stress-testing multiple items of interest: (a) ability to instantiate and advance millions of virtual ranks on simulation time axis (b) performance under very tight coupling among ranks, especially with regard to stringent characteristics of their virtual interconnection network, and (c) ability for a high level of multiplexing for maximum efficiency (i.e., largest values of LPX reasonably sustained). In the `barriertest`, every rank repeatedly joins a barrier by invoking `MPI_Barrier()`, and querying the time taken by each barrier via the times returned by `MPI_Wtime()`. Also, between each pair of barriers, each rank advances simulation time by one millisecond to model a relatively coarse-grained computation. Since $\mu\pi$ virtualizes the invoked MPI calls, all simulation runs are fully deterministic and repeatable (i.e., observe the same controlled bandwidth and latency effects), despite the challenge of immense non-determinism that is inherent with thousands of threads multiplexed on fewer number of cores. Thus the times of barrier observed by barriertest are repeatable across runs.

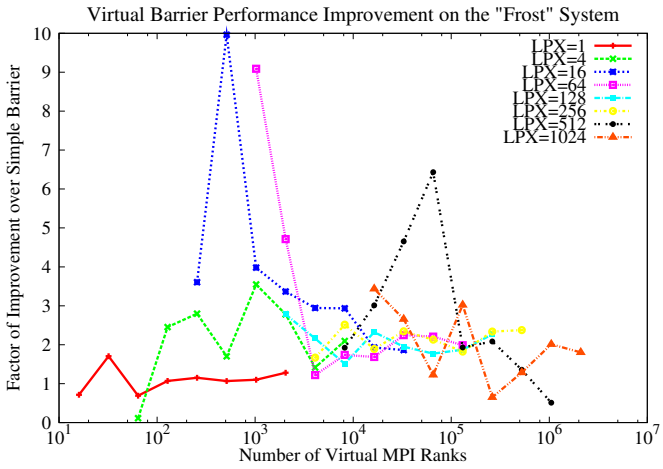

Figure 1. Factor of run time improvement on virtual barrier

Infiniband Linux Cluster: We varied LPX from 1 to 1024, to represent the spectrum from no multiplexing (one-to-one mapping of virtual to real ranks) to the heaviest multiplexing beyond which we see unacceptable degradation in performance due to the limitations on the amount of memory available and the operating system resources. We compare the performance of our optimized barrier with that of a straightforward implementation of barrier in which all ranks participate in butterfly communication. Figure 1 shows the factor of speed improvement of our optimized barrier implementation over the simple barrier. Note that the abscissa is the total number of virtual ranks in the simulation, and hence, the number of actual cores used in the simulation is obtained by dividing the number of virtual ranks by the LPX value.

In the case of small to medium number of virtual ranks, significant performance gain is observed, most near or above a factor of two. In the largest case of over 2 million ranks, obtained with LPX=1024 on all 2,048 hardware threads of Frost, a gain of close to $2\times$ is observed.

Cray XT5: On the Cray XT5, we aimed to show the effects of increased multiplexing efficiency on large core counts. While our prior results utilized 216,000 cores with LPX=128, we aimed to reduce the number of cores nearly by one order of magnitude. Consequently, we experimented with multiple LPX values, and chose the highest value of LPX=1024 beyond which the system became unstable.

Figure 2 shows the elapsed time for `barriertest`

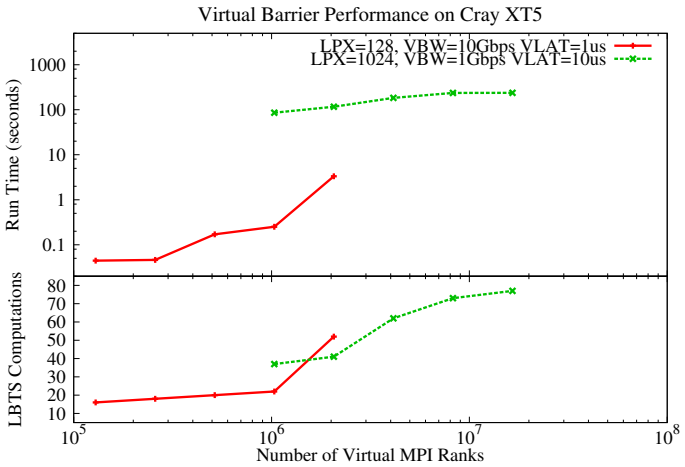

Figure 2. Simulation run time for virtual barrier

on Cray XT5 with increasing number of processors. At LPX=1024 on 16,128 real cores, we simulate over 16 million concurrent virtual MPI ranks, achieving nearly an order of magnitude greater multiplexing efficiency. Two different scenarios of the virtual network are simulated, simply to exercise the simulator with different dynamics (virtual network latency critically determines the amount of lookahead, hence concurrency, available in the parallel simulation). It is observed that the runtime cost of simulating a barrier increases once reaching a million simulated ranks. This is interesting because the various costs contributing to simulation overhead have been minimized (number of events, intensity of inter-node messaging, and amount of thread-switching cost). One of the remaining suspects is the amount of concurrency in the system which may be decreasing with increasing virtual scale. This can sometimes be gleaned from the number of global time computations (also called lower bound on time stamp, or LBTS computations for conservative parallel execution) that the simulation is incurring. The larger the number of LBTS computations, the lower the concurrency.

The number of LBTS computations is plotted against increasing number of virtual ranks, as shown in Figure 2. The trend appears to hold correlation with the runtime, suggesting the LBTS cost as a major part of the runtime. This, in turn, may suggest decreased level of concurrency. It may also suggest the occurence of strange dynamics that may be only activated with the barrier-induced event horizon across processors. One possibility to ameliorate the lack of concurrency is to simulate a more reasonable scenario in which the virtual ranks are not so markedly out of balance with respect to computing load variance, and/or employ larger inter-node latency/lookahead values by exploiting the coarse-grained computation that often intercedes global communications.

V. Summary

As part of work in progress, improvements to $\mu\pi$ event model are being made to enhance its efficiency and scalability. Here, we reported initial experience and experimental results in increasing multiplexing efficiency of $\mu\pi$ to multiple millions of virtual ranks. Traditional approaches or systems have neither attempted nor uncovered scalability problems that we identify here, namely, scalability of FIFO ordering realization for millions of virtual ranks, and the high cost of simplistic approaches for modeling virtual barriers at levels of multiplexing virtual ranks as high as 1024 virtual ranks per real rank/core. Although these issue may appear simple in hindsight, they are critical to realizing large-scale virtual MPI simulations. As part of additional, ongoing work, we are experimenting with larger number of actual ranks/cores used to sustain much larger scenarios, the extension of the improved algorithms to a richer set of collectives, and the development of a theoretical complexity analysis of event cost, threading cost, and inter-processor communication of traditional vs. improved virtual barrier implementations.

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. The authors are grateful to Vinod Tipparaju for helpful comments and discussions.

References

[1] K. Perumalla, "μsik - A Micro-kernel for Parallel/Distributed Simulation Systems," in Proceedings of the IEEE/ACM Workshop on Principles of Advanced and Distributed Simulation (PADS), 2005.

[2] K. Perumalla, "$\mu\pi$: A Scalable and Transparent System for Simulating MPI Programs," in Proceedings of the ICST Conference on Simulation Tools and Techniques (SimuTools), 2010.

[3] S. Plana, I. Brandic, S. Benkner, "Performance Modeling and Prediction of Parallel and Distributed Computing Systems: A Survey of the State of the Art," in Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), 2007.