Best Paper Award, Summer Simulation Conference SCS (SPECTS)'14
International Symposium on Performance Evaluation of Computer
and Telecommunication Systems, July 6-10, 2014, Monterey, CA

Simulating Billion-Task Parallel Programs

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

Alfred J. Park
Microsoft Corporation
Redmond, Washington, USA
Email: alfpark@outlook.com

***Abstract*—In simulating large parallel systems, *bottom-up* approaches exercise detailed hardware models with effects from simplified software models or traces, whereas *top-down* approaches evaluate the timing and functionality of detailed software models over coarse hardware models. Here, we focus on the top-down approach and significantly advance the scale of the simulated parallel programs. Via the direct execution technique combined with parallel discrete event simulation, we stretch the limits of the top-down approach by simulating parallel programs with hundreds of millions of tasks. Although the scaling issues and solutions presented here are generally applicable, we focus on message passing interface (MPI) programs. Using a timing-validated benchmark application, a proof-of-concept scaling level is achieved to over 0.22 billion virtual MPI processes on 216,000 cores of a Cray XT5 supercomputer, representing one of the largest direct execution simulations to date, combined with a multiplexing ratio of 1024 simulated tasks per real task.**

I. Introduction

Parallel programs with millions of tasks are already a reality (e.g., over 1.5 million MPI ranks can be instantiated on the Sequoia Blue Gene supercomputer[1] that has as many processor cores). Following the scaling trends, support for much larger number of tasks are targeted by supercomputing installations within the next few years. With the increasing parallelism scales of interest, simulation advancements are needed to meet the goals of modeling fidelity, system scale and simulation speed in experimentation with future large-scale parallel programs.

Parallel systems being very complex systems, experimentation with their designs and performance evaluation requires the use of a variety of ways and methods [2], depending on the purpose or use-cases. We broadly classify the simulation-based methods into two approaches. *Bottom-up* approaches such as full-system or cycle-accurate modeling use detailed hardware models (e.g., of caches, processors, memory, and network interface cards) driven by simplified program loads or by traces (e.g., Wisconsin Wind Tunnel [3], SIMICS [4], EMPOWER [5], FASE [6], hybrid [7], WARPP [8], OMNeT++ [9], and micro SST [10], [11]). While being effective for designing the individual hardware elements, they are infeasible for simulating the runtime operation of very large parallel programs due to limitations of scale and speed. *Top-down* approaches use actual software or detailed models of programs and execute them on simplified hardware models (e.g., MPI-Sim [12], POSE [13], BigSim [14], [15], and LogGOPSim [16]). They are useful to exercise actual parallel programs at the largest scale – either full applications, benchmarks, program prototypes, or program skeletons, all of which relate more directly to the parallel programmer – executed with user-chosen levels of functionality and timing details.

With top-down approaches, prototypes of actual parallel programs can be written, compiled and tested at futuristic scales, with user-specified hardware configurations. Verification and exchange of findings become trivial, via exchange of actual application or prototype source code. Early debugging and testing of algorithms and optimizations are facilitated by deterministic execution (which can be enabled by the simulator) and zero-perturbation instrumentation.

The key contribution of the present article lies in scaling: the achievement of the largest scale to date for timing-accurate, software-level direct execution of actual parallel programs. We describe our effort to push the simulation capacity of top-down experimentation for very large-scale parallel programs. Our approach involves exploring, uncovering and addressing some of the issues in sustaining very large numbers of control-flows, in the form of millions of virtual MPI processes. Since the primary focus of this paper is on proof-of-concept scalability to very large number of tasks, additional important factors are not considered here, such as modeling the effects of GPUs/accelerators, different processor types, file system or disk input/output effects. However, it is possible to incorporate them later in a more complete system.

To realize a scalable top-down simulation of parallel programs, we employ a combination of *direct execution* [17], [18], [19] and parallel discrete event simulation [20], [21]. Our approach is realized in $\mu\pi$ [22], a *process-oriented* simulator in which each virtual MPI process is represented as a logical process (LP). Each LP is maintained as a distinct thread of control flow, with its own execution context (e.g., stack). $\mu\pi$ maintains all requisite state to suspend and resume simulated virtual MPI process as needed by the underlying simulation executive μsik [23]. This allows $\mu\pi$ to multiplex multiple virtual MPI processes onto each available processor and support unmodified MPI codes within a simulated environment. The simulator process on each physical core hosts a user-specified number of virtual MPI processes multiplexed by simulation time on that core. A hierarchical structure is adopted to accommodate the large number of virtual MPI processes across the limited amount of processing resources

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.

available. This structure follows naturally from typical high performance computing environments. Each node may contain multiple processor sockets, and each processor may have multiple cores. For simplicity, the number of cores within a node is flattened across all sockets to a single level. Typically in a real MPI application, each MPI task or thread within the MPI program is assigned to a single core. However, since we are simulating a much larger MPI program on a virtual platform, these tasks are virtualized. Each physical core handles a pre-assigned number of virtual MPI processes, and these are time-multiplexed on their assigned processing core. To make simulation-based experimentation usable, we focus on achieving very high efficiency in both multiplexing as well as total runtime, using efficient discrete event methods for simulating millions of MPI processes on thousands of actual cores.

A. Organization

The rest of paper is organized as follows. In Section II, some of the scaling issues and solutions are presented to significanly increase the overall number of control flows as well as the multiplexing ratio (number of simulated tasks per real task). The challenge of efficiently modeling at scale the first-in-first-out semantics of the simulated communication interface is addressed. The issue of modeling global operation primitives such as barriers and reductions is identified and addressed. Following that, a detailed performance study is presented in Section III, with results from a timing validation effort and a scaling exercise to simulate up to 0.22 billion MPI tasks. Finally, a summary and some future work are described in Section IV.

II. Scaling Issues and Solutions

Here, we identify two important issues that arise in increasing the scale of the simulations both in multiplexing ratio and in the aggregate number of control flows (MPI tasks). One is the issue of correctness in message ordering semantics and the second deals with efficiency of multiplexing.

A. Discrete Event Model of Ordered Matching Semantics

MPI is a messaging protocol that can sit atop various data transmission layers such as TCP and shared memory. The variety of layers on which the MPI protocol can transmit data implies that the MPI implementation itself cannot assume that underlying layers will preserve certain data transmission characteristics. Thus, the semantics of certain MPI communications must be clearly defined by the standard, which implementations must meet. One such characteristic is the ordered matching of successive messages between two ranks of the same communicator, with the same tag and no wildcards; this is a common communication case. 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 guarantees, such as ordered matching, are preserved. Ordered matching of messages within a communicator is non-trivial to implement in a virtual execution when scaled to millions of tasks. Since $\mu\pi$ multiplexes virtual MPI processes, application data movement is required to be modeled by $\mu\pi$ via simulated events, thereby requiring a scalable solution to enforcement of ordered matching guarantees in the virtual execution.

1) Problem: MPI guarantees ordered matching of messages between two ranks within each communicator with a single tag and no wildcards 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).

Figure 1 shows incorrect event processing without proper measures in place to maintain ordered matching. An `MPI_Isend()` with a data payload of 1MB is sent first from $Rank_i$ to $Rank_j$ followed immediately by another `MPI_Isend()` with a data payload of 1KB between the same pair of MPI processes. Clearly, the message with 1MB takes a longer transmission time through the network than the 1KB message, due to the size of the payload and available bandwidth. In the incorrect scheme, when $Rank_j$ posts a data receive, it will receive the 1KB message first, instead of the 1MB message which the application is expecting. This is a clear case of a message overtaking another, when the 1MB should be returned by the first `MPI_Irecv()` call followed by the 1KB message, due to the MPI semantics of ordered matching. Thus, without proper knowledge of outstanding sends in the network, strict ordered matching of messages for this scenario at the receivers is not possible.

The core of the problem is that this simple simulated messaging protocol is stateless while the semantics require state. A current message being sent has no prior knowledge of the state of the network. There are different solution approaches to this problem by essentially maintaining a stateful messaging protocol to preserve ordered matching. We present a few non-scalable solutions for illustration, followed by our scalable, efficient approach.

2) Non-scalable Solutions:

1) Network Link Model: One approach to ensuring ordered matching is to create a network link model between each point-to-point communication endpoint as widely used in network simulation. This allows all messages sent between two MPI processes to be funneled through the link model where appropriate characteristics can be applied to any event. Although this approach can allow for arbitrary complexity and fidelity of certain network properties, when dealing with tens of millions of virtual MPI processes, memory usage can become excessive, preventing scalability.
2) Sequence Numbers: Another non-scalable approach is to allocate message sequence counters for all possible senders at each virtual receiver process. When a new message arrives, the message is only delivered to the virtual process if the message is in-order. Otherwise, the message is buffered in a priority queue. This approach causes a few problems. First, messages must be buffered on the receiver side until all preceding messages are processed, which can result in increased memory usage. Second, there is an increase in the

actual size of the event that must be sent by the size of the message counter. And finally, perhaps the most damaging consequence, is the requirement of two data structures of size $O(n)$, where n is the number of virtual MPI processes on both the sender and receiver. The sender needs to keep track of the current sequence number between itself and every other receiver in the simulation. Similarly, this is also required on the receiver to know the in-order message count on a per-sender basis. Clearly, this solution will not scale well for very large scenarios.

The network link model approach encompasses undue modeling effort and computational effort in scenarios where point-to-point delays are sufficient (e.g., with a user-specified barrier time or probabilistic network latency). The sequence number approach, while being computationally light, requires excessive memory to maintain the sequence number state on a MPI process-pair (sender,receiver) basis. For example, in a 1 million virtual MPI process simulation performed with 1,000 virtual MPI processes per core of a 12-core node used for simulation, each MPI process will need to maintain up to 1 million sequence numbers, requiring total state of over 1 billion sequence numbers per node, which is an excessively large fraction of memory consumed on a node.

3) Scalable Solution Approach: A scalable solution to the ordered matching problem is to essentially record a minimal amount of pair-wise state of the message sent to the receiver at only the sender side. Effectively, this is a very lightweight network link model that is appended to the sending virtual MPI process for each destination. There is no need to pre-allocate every potential destination virtual MPI process; instead this state is only maintained for the most-recently sent data. Therefore, memory is only allocated on demand when a send operation is outstanding to any receiver. Memory is reclaimed if any state regarding previous messages is no longer relevant. By keeping the state of the last message sent between itself and the receiver, proper adjustments to the receive timestamp can be made, thus these simulation messages can be consumed in proper timestamp order on the receiver. This scalable approach to preserve ordered matching with MPI messaging is outlined in Algorithm 1.

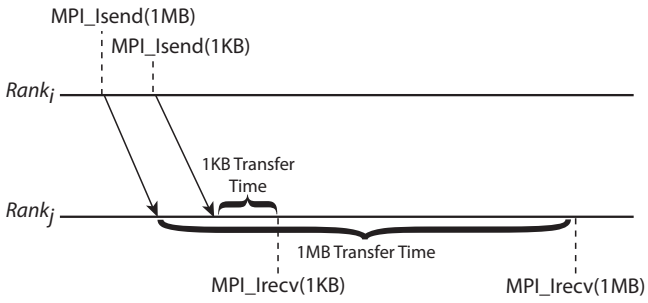

Fig. 1. Incorrect Simulation that Violates Order Matching

Figure 2 shows our correct and efficient approach to order matching between two MPI processes. In the illustrated scenario, $Rank_i$ sends two consecutive messages to $Rank_j$. Assuming there are no prior outstanding messages between the two MPI processes, the first message is sent and received without any additional delay with a send timestamp of $SendTS_1$ and a receive timestamp of $RecvTS_1$. At $Rank_i$, three pieces of information are recorded when a message is sent: the destination ($Rank_j$), send and receive timestamp. When the second message is sent from $Rank_i$ to $Rank_j$, the prior send information is queried. $LatencyDelay_2$ is considered concurrent and can be overlapped with any previous outstanding sends. The remaining $BandwidthDelay_2$ that overlaps the prior outstanding send (i.e. $LatencyDelay_1 + BandwidthDelay_1$) must be proportionally delayed by that additional amount represented as $BusyDelay_{1,2}$. This amount is added on to the existing total computed delay time for the sent message, and the receive time for the second message is set as $RecvTS_2$. The old state that contains the information about the prior message is overwritten with new message send/receive information.

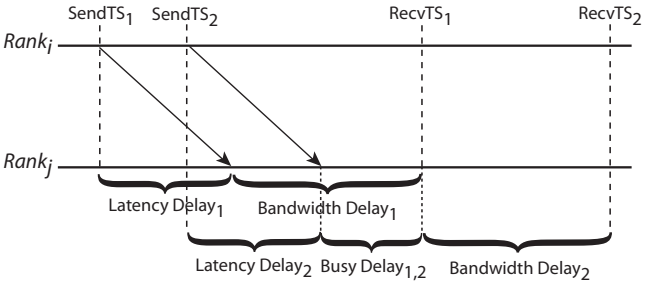

Fig. 2. Simulation with Correct Order Matching

Algorithm 1 Scalable Order Matching of Virtual MPI Messages

1: State variables: A set S (initially null) of triples at every MPI process, where a triple is defined as: (`destination rank`, `source timestamp`, `destination timestamp`)
2: `MPI_Send(torank, msgsize)`:
3: Flush every entry in S whose `destination timestamp` $<$ `now()` + latency
4: Compute the "normal" receive timestamp of this message based on bandwidth and latency
5: **if** `torank` does not exist in S **then**
6: Add triple (`torank`, `now()`, `receive timestamp`) to S
7: **else**
8: Δ = `destination timestamp` - `now()` - latency
9: Increase receive timestamp of this message by Δ
10: Increase `source timestamp` in `torank`'s entry of triple to `now()`
11: Increase `destination timestamp` in `torank`'s entry of triple to the new receive timestamp
12: **end if**
13: Schedule `DataEvent` to logical process of `torank`, `dt` simulation time units in future, where `dt` = receive timestamp - `now()`

B. Efficient Implementation of Virtual Barrier and Virtual Collectives

An often misunderstood aspect is that native barriers of the real system cannot be employed as-is to simulate a (virtual) barrier of the simulated system. It is incorrect for every virtual

MPI process to simply invoke a native barrier implementation to realize its virtual barrier functionality. Such blocking on native calls interferes with simulation time advances. *In effect, it 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. Software-level collectives become necessary, implemented using timestamped events.

We employ an algorithm that is specifically tuned for PDES, and is distinct from traditional optimizations performed for native MPI collectives [24]. Our approach exploits low-cost local shared memory operations without polluting simulation time with wall clock time, and minimizes inter-node event communication. Every virtual MPI process executes this algorithm as the implementation of virtualized synchronization and collectives such as `MPI_Barrier()` and `MPI_Allreduce()`.

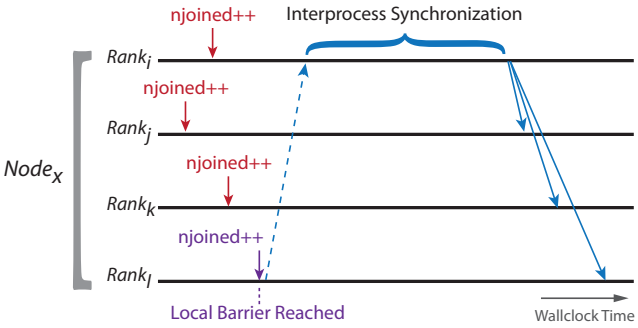

Fig. 3. Optimized event-based implementation of virtual barrier operation

1) Optimized Virtual Barrier Algorithm: The key to the optimized collective algorithm is the use of a variable called `njoined` that is globally visible to all virtual MPI processes mapped to a core (each core contains its own instance of this variable), initialized to zero. Every virtual MPI process increments this variable upon entry into the barrier. Let λ denote the number of virtual MPI processes per core. The virtual MPI processes mapped to a core are referred to as local ranks, and the first virtual MPI process on any core is that core's *leader* rank (relative to the communicator being used). 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 λ, and realizes it has joined last. No additional events are necessary to coordinate the "join" phase of the barrier among local ranks, and it can proceed with its leadership role representing all the local ranks. In the second case, with the leader arriving early (detected by `njoined` being less than λ), it proceeds to wait on an event reception. When the final local rank arrives last, it detects that it joined last and sends an event to the leader, completing the local "join" phase. This optimized virtual barrier is conceptually shown in Figure 3.

The remaining part of the synchronization is a direct mapping of traditional messaging for barrier (e.g., using a butterfly pattern), after aggregation at each node among all cores. The key difference is that messaging is again performed via timestamped simulation events, rather than native messages. Details are omitted here 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 algorithm has four important, independent beneficial factors: (a) The number of events of notification for the join phase of the barrier is dramatically reduced. In fact, there is at most one, and possibly zero, events generated for the entire join phase of local virtual MPI processes. When λ is large (e.g., 1024), thousands of events are eliminated at every core, reducing event overhead significantly (e.g., by up to three orders of magnitude per run for λ=1024). (b) The number of inter-core (and inter-node) events is dramatically reduced because the number of inter-core events is in effect decoupled from λ, and is only dependent on number of cores and nodes. (c) The number of context switches between threads that are hosting virtual MPI processes is also reduced. Since each virtual MPI process is a serial processing burden on each core, every event scheduled for a virtual MPI process incurs not only event cost but thread switching overhead as well. This cost, being significant on large values of λ, is nearly eliminated in the join phase. (d) While being efficient, this algorithm ensures the desired decoupling between wall clock time and simulation time, unlike other alternatives that can artificially introduce simulation time anomalies in favor of faster runtime.

This efficient template carries over well to other global collectives as well, such as `MPI_Allreduce()`, which we also implemented. The main difference between barrier and other collectives is that barrier does not need any data in the events, but other collectives need data fragmentation and reassembly with timestamped events. The optimized algorithm template is enhanced to accommodate data by storing the collected data in local shared-memory buffers during the join phase, and distributed via the same buffers in the release phase.

2) Events, Messaging, and Thread-Switching Analysis: Let n be the number of host nodes, c be the number of cores per host node, and λ be the number of virtual MPI processes mapped to each core. Let $R = n{\cdot}c{\cdot}\lambda$ be the total number of virtual MPI processes. For each barrier, traditional butterfly across all virtual MPI processes gives a time complexity of $\log_2 R$ time steps per virtual MPI process, or $\lambda \log_2 R$ steps per core, $\lambda \log_2 R$ thread switches per core, and total events simulated as $R \log_2 R$ in the system, of which $c\lambda \log_2 c\lambda$ are intra-node events per node, and $c\lambda \log_2 R - c\lambda \log_2 c\lambda = c\lambda \log_2 n$ inter-node events per node. Our improved algorithm reduces the complexities to $\lambda + c + \log_2 n + c + \lambda$ time steps per virtual MPI process (worst case), $\lambda + c + \log_2 n + c + \lambda$ thread switches per core, and total events equal to $c + cn + n \log_2 n + cn + \lambda c$ in the system, of which $\log_2 n$ are inter-node events per node, and $2c + \frac{\lambda c}{n}$ are intra-node events on average. The two most significant gains are in terms of thread switches and inter-node events. Compared to the straightforward implementation with a butterfly across the complete virtual MPI process space, thread switches are reduced by $\lambda \log_2 nc\lambda - 2\lambda - 2c - \log_2 n = \lambda(\log_2 n - 1) + \lambda \log_2 c\lambda - 2\lambda - 2c$ per core, and inter-node events are reduced by $(c\lambda - 1)\log_2 n$ per node. When λ, c and n are large, the performance difference becomes appreciable (e.g., inter-node events are reduced by 110,583 for a typical configuration with $\lambda = 1024$, $c = 12$ and $n = 512$).

An important additional influence on the performance is the amount of imbalance that may be present in simulation time horizon relative to the lookahead window. While in a perfectly balanced, synchronous execution, the preceding complexity analysis holds well, the performance can become dominated by synchronization cost if and when imbalance in timestamps of events gets introduced by the application, which can initiate a large amount of synchronization messaging by the simulation engine. Such cost is unavoidable in a conservative parallel execution, in which case, the observed performance serves as a lower bound on performance, which can only improve with either larger lookahead or more balanced workloads. In fact, the interaction between the performance of the actual program and the performance of its simulation is an extremely interesting aspect of simulating at large scale in which such effects get amplified.

III. Performance Study

All empirical evaluations were performed on a Cray XT5 system in which each node has two hex-core AMD Opteron 2435 (Istanbul) 2.6GHz processors with 16GB of memory. Communication is supported by Cray's SeaStar 2+ router of the Cray XT5. Compilation was performed via the Portland Group (pgi) compiler 2.2.73 with `-O3 -fast` flags.

A. Validation

The ping test benchmark is used to measure bandwidth and latency between pairs of communicating MPI processes. This ping test has virtual MPI processes arranged in a naturally-ordered ring topology. The sender sends data to the next higher virtual MPI processes while receiving data from the lower virtual MPI processes. If the virtual MPI process number is even, it performs a blocking send followed by a blocking receive. The order of operations is reversed for odd-numbered virtual MPI processes. These operations are timed via calls to `MPI_Wtime()` for bandwidth and latency measurement.

These operations are iterated successively from 8 bytes to the maximum specified test message size, where the length of each message is doubled for each trial until the maximum limit is reached. For the validation tests, the maximum test message size was 16 MB.

For the MPI ping test validation, the latency and bandwidth were measured for the Cray XT5 system using the MPI ping test itself. The average across three measurements at each data point is assigned as the delay metric for a particular sized data chunk that is sent through virtual MPI communicators within $\mu\pi$. The highest observed bandwidth served as the maximum virtual bandwidth. Due to the hierarchy of communication involved between SMP nodes, $\mu\pi$ can accept two-levels of latencies and bandwidths to reflect a shared memory intra-node communication tier and a network inter-node communication tier. Shared memory was timed by exercising ping test across all cores within one node. Network metrics were gathered by executing ping test across an equivalent number of cores, but only with one core per node. A "flat" timing model is also measured and $\mu\pi$ is fed with a single level of latencies and bandwidth observed across a test scenario of multiple nodes exercising all cores.

Figure 4 and Figure 5 show validation results for $\mu\pi$ across both 1008 and 16128 MPI processes. Note that for measured data, the data represents real MPI processes while for $\mu\pi$, the data represents total virtual MPI processes. Since the ping test does not perform any computation, the $\mu\pi$ charging API was set to ignore CPU time accumulated in non-MPI routines. It is observed that the simple network model without any complexities associated with full-blown network link models provides close-to-measured two-way transfer times. As expected, although an exact match of the simulated model to the real measured results is impossible to obtain due to other concurrently running jobs, operating system noise, network traffic and unpredictable node allocation, the timing model provides a very good approximation of the real behavior at a fairly large fraction of the system at 16128 processors.

B. Scaling Experiments

We implemented the aforementioned algorithms in $\mu\pi$ and tested them on the Cray XT5 system. For testing, commonly available MPI examples in source form have been successfully executed over $\mu\pi$, such as `matmul`, `deadlock_fix`, `mpiping`, `picalc`, `ring_blocking` and `ring_nonblocking`. For performance testing, we exercised the collectives with two benchmarks, `barriertest` and `allreducetest`. In `barriertest`, every MPI process 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 MPI process advances simulation time by one millisecond to model a relatively coarse-grained computation. The `allreducetest` operates similarly, except that (a) `MPI_Allreduce` is used instead of barrier, and (b) the virtual computation time charged between reductions is randomized across MPI processes, in order to model staggered arrival times at the reduction. To exercise data payload effects, a vector of double precision values is offered (using the sum operator) for every reduction.

The timing scenarios are chosen to represent some of the most severe execution constraints on the simulator, stress-testing capabilities such as: (a) ability to instantiate and advance millions of virtual MPI processes on simulation time axis (b) test performance under very tight coupling among MPI processes, especially with very low inter-MPI process latencies in the virtual interconnection network, and (c) exercise high levels of multiplexing for maximum efficiency (i.e., largest values of λ reasonably sustained).

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` and reductions by `allreducetest`, are repeatable across runs.

While our prior results [22] on the Cray XT5 was limited to λ=128, with the aforementioned scalability optimizations, we were able to increase the multiplexing efficiency, thus enabling: (1) increased number of total virtual MPI processes simulated, and (2) a reduction in the number of processors needed to simulate a similar virtual job size. Accordingly, we

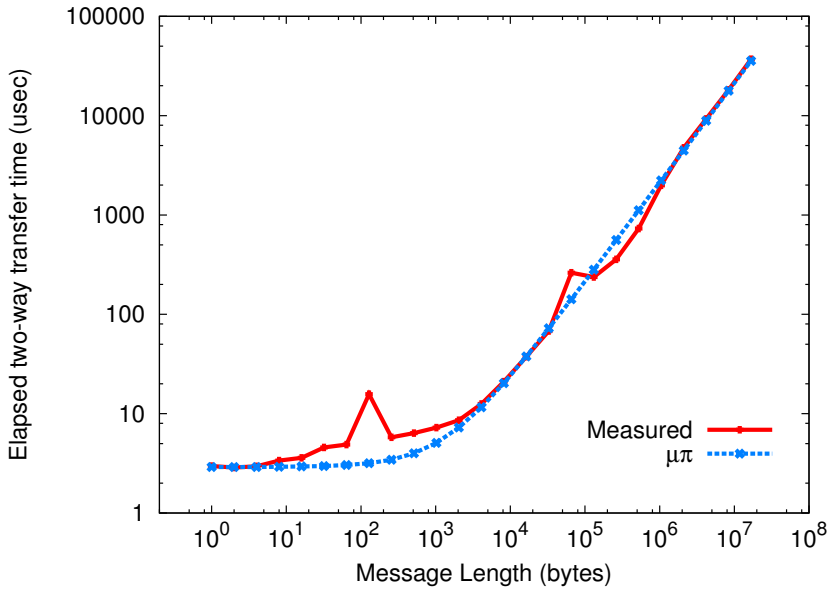

Fig. 4. Cray XT5 MPI Ping Test Validation: 1008 Real or Virtual MPI Processes

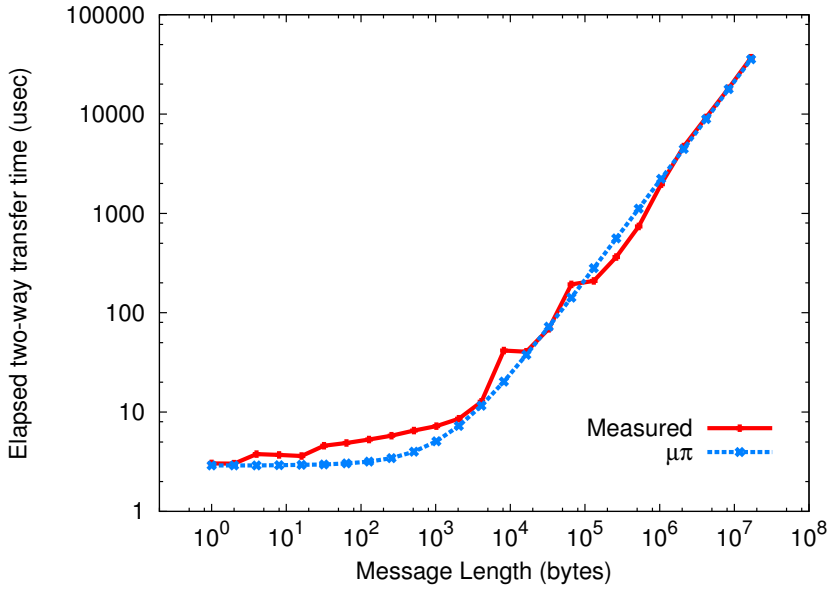

Fig. 5. Cray XT5 MPI Ping Test Validation: 16128 Real or Virtual MPI Processes

experimented with multiple λ values, and chose the highest value of λ=1024 beyond which the aggregate system memory becomes insufficient to represent the simulated system.

Figure 6 and Figure 7 show elapsed time (lines) and corresponding remote event counts (bars) for the `barriertest` and `allreducetest` benchmark with increasing number of virtual MPI processes. Two different scenarios of the virtual network are simulated, simply to exercise the simulator with different dynamics (β denotes simulated bandwidth and δ denotes simulated latency) – virtual network latency critically determines the amount of lookahead (a lever of concurrency), available in the parallel simulation. At 216,000 processors and λ=128, there were 27,648,000 virtual MPI processes simulated with a virtual barrier taking 4.63 wallclock seconds and reduction of 1024 double values taking 13.92 wallclock seconds to complete. At 216,000 processors and λ=1024, there were 221,184,000 virtual MPI processes taking 154.80 wallclock seconds to complete a virtual barrier and reduction of 1024 double values taking 408.93 wallclock seconds. As expected, it is observed that λ=128 runs faster than λ=1024, due to reduced multiplexing load per core, for the same number of virtual MPI processes being simulated. However, it is also encouraging to note that nearly an order of magnitude fewer processors can be employed to simulate the same number of virtual

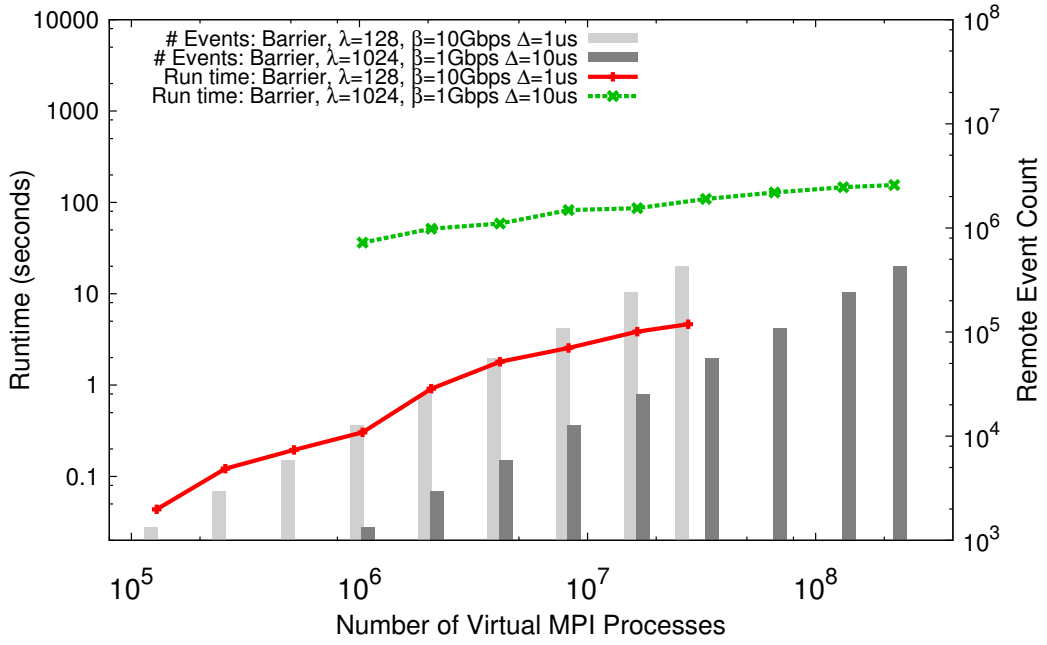

Fig. 6. Virtual Barrier Performance

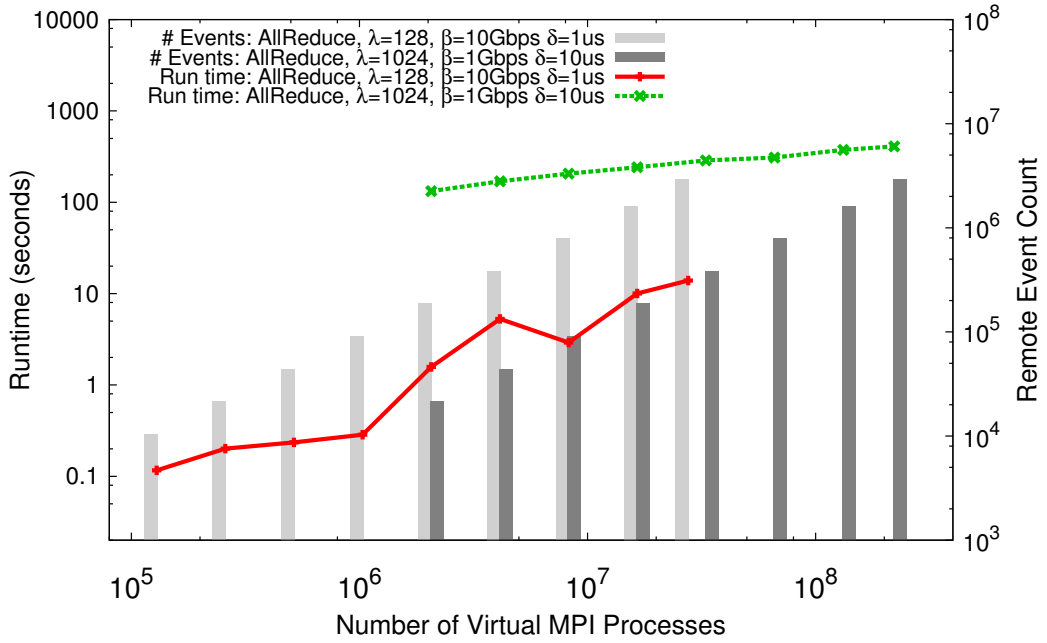

Fig. 7. Virtual AllReduce Performance

MPI processes, as a cumulative result of our performance improvements. It is also seen that the remote event count scales very well (note logarithmic abscissa).

The main point of the performance chart is that it shows the feasibility to perform fully time-controlled, software-level experimentation with very large number MPI jobs. *This performance data represents the most scalable simulation of unmodified MPI programs, under stringent global operations such as virtual barrier/reduction simulated in just a few seconds.* The number of virtual MPI processes has been pushed to hundreds of millions primarily to test the scalability of the approach and the actual implementation. Clearly, normal jobs with fewer MPI processes can be expected to perform very well.

Overall, the results strongly indicate a sweet spot that is worth further exploration by the parallel computing community in the experimentation tradeoff between scale, fidelity, speed and control on accuracy. It is conceivable that new enhancements and extensions to massively parallel computation (e.g., fault tolerance, non-blocking collectives) can be debugged, analyzed, tested, and evaluated with actual software virtually at scale.

IV. Summary and Future Work

With large-scale computing initiatives moving towards exascale computing, software-level experimentation is crucial in helping design and develop future systems. Also, parallel software development in general stands to benefit from simulations that help better prepare codes in anticipation of next levels of scalability. Large-scale top-down simulation capabilities are a step in that direction.

Here, we addressed some of the key issues arising at scale, and presented solutions with the net outcome of being able to achieve hundreds of millions of virtual MPI process executions much more efficiently than before. *Although our implementation focused on MPI, the concepts apply to any simulation of massively parallel communicating sequential process systems.*

Admittedly, different design problems warrant different frameworks and systems for software-level experimentation. Nevertheless, here we have reported the feasibility of sustaining very large top-down parallel program simulations in time-controlled, user-specified scenarios. The approach appears promising, yet much additional work is needed to carry the results to more complex codes and couple them with hardware and system-level simulators to enhance fidelity.

As may be expected, new issues and challenges arise in simulating future large-scale applications, and we are potentially only scratching the surface. With the introduction of accelerators such as GPUs into large-scale computing installations, the concurrency increases even more dramatically, with millions of threads being executed on each GPU. A future extension to the scaling is the incorporation of GPU thread models, and the ability to incorporate accelerator code into the simulation just as the MPI code is incorporated directly by $\mu\pi$ into the simulation. Since the offered load by such an extension magnifies the discrete event simulation load by an additional factor of $10^6 - 10^8$ (corresponding to the GPU-level launch of millions of threads per kernel invocation at each node), scalability of the simulator will be further stressed, which in turn will require additional optimizations. Also, the interconnection network(s) within the parallel system play a crucial role in the overall parallel program performance, and hence, a more detailed model may need to be incorporated for uncovering runtime dynamics (such as congestion) that may be missed from simple point-to-point model. Similarly, parallel programs that have intensive input/output behaviors will need incoporporation of file system and disk operation models.

References

[1] "Top 500 supercomputer sites." [Online]. Available: http://top500.org

[2] S. Pllana, I. Brandic, and S. Benkner, "Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art," in *Proceedings of the First International Conference on Complex, Intelligent and Software Intensive Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 279–284.

[3] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: Virtual prototyping of parallel computers," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993, vol. 21, pp. 48–60.

[4] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. pp. 50–58, 2002.

[5] P. Zheng and L. M. Ni, "Empower: a scalable framework for network emulation," in *International Conference on Parallel Processing*, 2002, pp. 185–192.

[6] E. Grobelny, D. Bueno, I. Troxel, A. D. George, and J. S. Vetter, "Fase: A framework for scalable performance prediction of hpc systems and applications," *Simulation*, vol. 83, no. 10, pp. 721–745, 2007.

[7] S. Pllana, S. Benkner, F. Xhafa, and L. Barolli, "Hybrid performance modeling and prediction of large-scale computing systems," in *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, March 2008, pp. 132–138.

[8] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama, "Warpp: a toolkit for simulating high-performance parallel scientific codes," in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, ser. Simutools '09, ICST, Brussels, Belgium, 2009, pp. 1–10.

[9] C. Minkenberg and G. R. Herrera, "Trace-driven co-simulation of high-performance computing systems using omnet++," in *2nd International Workshop on OMNeT++*, 2009.

[10] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel computer architectures," *International Journal of Distributed Systems and Technologies*, vol. 1, no. 2, pp. 57–73, 2010.

[11] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, March 2011.

[12] S. Prakash, E. Deelman, and R. Bagrodia, "Asynchronous parallel simulation of parallel programs," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 385–400, 2000.

[13] T. Wilmarth, G. Zheng, E. J. Bohm, Y. Mehta, N. Choudhury, P. Jagadishprasad, and L. V. Kale, "Performance prediction using simulation of large-scale interconnection networks in pose," in *Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[14] G. Zheng, G. Kakulapati, and L. Kale, "Bigsim: a parallel simulator for performance prediction of extremely large parallel machines," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004.

[15] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kale, "Simulation-based performance prediction for large parallel machines," *Intl. J. of Parallel Programming*, vol. 33, no. 2, pp. 183–207, 2005.

[16] T. Hoefler, T. Schneider, and A. Lumsdaine, "Loggopsim: simulating large-scale applications in the loggops model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. New York, NY, USA: ACM, 2010, pp. 597–604.

[17] P. Dickens, P. Heidelberger, and D. M. Nicol, "Parallelized direct execution simulation of message-passing programs," *IEEE Trans. on Par. and Dist. Systems*, vol. 7, no. 10, pp. 1090–1105, 1996.

[18] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Feedback-directed virtualization techniques for scalable network experimentation," University of Utah, Technical Report, 2004.

[19] J. Liu, Y. Yuan, D. M. Nicol, R. S. Gray, C. C. Newport, D. Kotz, and L. F. Perrone, "Simulation validation using direct execution of wireless ad-hoc routing protocols," in *18th Workshop on Parallel and Distributed Simulation*. ACM, 2004.

[20] J. Liu, D. Nicol, B. Predmore, and A. Poplawski, "Performance prediction of a parallel simulator," in *13th Workshop on Parallel and Distributed Simulation*, 1999, pp. 156–164.

[21] K. S. Perumalla, R. Fujimoto, P. Thakare, S. Pande, H. Karimabadi, J. Driscoll, and Y. Omelchenko, "Performance prediction of large-scale parallel discrete event models of physical systems," in *Winter Simulation Conference*. Orlando, FL: IEEE, 2005.

[22] K. S. Perumalla, "$\mu\pi$: A scalable and transparent system for simulating mpi programs," in *Proceedings of the 3rd International Conference on SIMUTools*, 2010.

[23] ——, "μsik - a micro-kernel for parallel/distributed simulation systems," in *Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[24] K. S. Perumalla and A. J. Park, "Improving multi-million virtual rank mpi execution in $\mu\pi$," in *Proceedings of the 19th International Symposium on MASCOTS*, 2011.