Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A.M. Uhrmacher, eds

TUTORIAL: PARALLEL SIMULATION ON SUPERCOMPUTERS

Kalyan S. Perumalla
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831 USA

ABSTRACT

This tutorial introduces the typical hardware and software characteristics of extant and emerging super-computing platforms, and presents issues and solutions in executing large-scale parallel discrete event simulation scenarios on such high performance computing systems. Covered topics include synchronization, model organization, example applications, and observed performance from illustrative large-scale runs.

1 INTRODUCTION

This tutorial article introduces primary concepts and considerations in the intersection of supercomputing and parallel simulation, with a focus on implementation of parallel discrete event simulation techniques and applications. It serves as a primer on moving from small-scale parallel simulation to large-scale using supercomputing capabilities. It is intended for someone familiar with parallel simulation who is interested in exploring high-end simulations via supercomputing. Prior exposure to parallel computing and parallel discrete event simulation is assumed. Concepts in parallel discrete event simulation such as events and time-stamp ordering, global virtual time synchronization, lookahead, and optimistic vs. conservative simulation are assumed to be familiar to the reader. Similarly, parallel computing concepts such as parallel computing architectures, and algorithmic considerations such as deadlocks and live-locks, and termination detection are also assumed to be understood. An introduction to parallel simulation concepts can be found in the literature (Fujimoto 1990; Fujimoto 2000; Perumalla 2006).

What is Supercomputing? Supercomputing is a term commonly used to refer to installations of hardware and software systems that together offer the highest-end computing capabilities, in which a fairly large number of computing elements (processors, memory hierarchies, network, storage systems, reliability services) are housed and maintained in a tightly-coupled fashion, with close monitoring, dedicated administration and maintenance. The exact sizes of installations that constitute supercomputing varies with each passing year, but they always represent the very largest configurations of the day installable in a single physical site. Although future commodity computers eventually catch up to current supercomputing power, the supercomputers also themselves advance similarly rapidly and offer proportionately larger computing capacity compared to commodity computers of the day. For example, although the capability of a supercomputer of the 1970's may be equivalent to that of a single desktop installation of today, the supercomputers of today are four to five orders of magnitude more powerful than today's desktop installations. Thus, the term supercomputing is time-sensitive and is understood to be used in a relative sense.

Systems: Currently, systems at the highest end are offered by only a few vendors, such as Cray, IBM and Fujitsu. Cray's XT series (XT3 of 2005-07, XT4 of 2008-09 and XT5 of 2008-2011) represented a number of installations, with some systems containing over 200,000 processor cores in a single installation. The Blue Gene series (L of 2005-07, P of 2008-2011, and Q of 2011-now) series of offerings from IBM constitute another line of recent supercomputers that pushed peak capacity to very large scale at relatively lower energy cost. As of writing, the "K" supercomputer in Japan heads the list of top supercomputers (with 88,128 2.0 GHz 8-core SPARC64 VIIIfx processors), followed by the Tianhe-1A system in China (mix of 14,336 2.9GHz 6-core Intel Xeon X5670 processors and 7,128 NVIDIA Tesla M2050 accelerators), followed by the Cray XK6 system in the USA (with 37,360 2.6GHz 6-core AMD Opteron processors). The interconnect technologies also are in intense flux, spanning both propriety technologies (e.g, SeaStar and Gemini) and standards-based solutions (e.g., Infiniband), and specializing in different aspects such as high-bandwidth or low-latency operation.

Supercomputing vs. Clusters and Cloud Computing: In a concurrent development, the cluster and "cloud" technologies are offering medium-scale parallel/distributed computing solutions, which offer computing and storage capacities that are comparable to major fractions of those offered by supercomputing installations. However, the fundamental differences in terms of reliability, accessibility, and inter-node latencies make cluster and cloud platforms significantly different in terms of programming and application development methodologies. This distinction is much more pronounced in the case of parallel simulation, and even more so for parallel discrete event simulation (PDES): PDES requires fundamentally different simulation engines and frameworks for supercomputing than for clusters and cloud platforms.

Programming Systems: Supercomputing applications have been traditionally and predominantly developed in FORTRAN, C and C++ languages. Parallelism is most commonly realized using the Message Passing Interface (MPI), with most installations supporting MPI 1.1 or higher, and many supporting a large subset of MPI 2.0 (but omitting primitives such as process creation). In recent times, newer languages and programming models have been proposed and introduced in some of the supercomputing installations. These include Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC), Co-Array FORTRAN, X-10 and Chapel. They provide language-level support for declaring arrays that span distributed (processor) memories, such as a new array declaration by which a special dimension for the shared array specifies memory affinity of the array's blocks partitioned across memories.

The most common hardware view is a network of "nodes" in which each node contains one or more shared-memory multi-core processors. This hardware view is reflected in the most common execution model natively supported, which is a world of MPI tasks for each run, with at least one MPI task spawned on each node. A common variant of this view is the introduction of multi-threading subordinate to the task view. Thus, the framework most widely used and supported is a combination of MPI and multi-threading, in which one or more MPI tasks execute on each shared memory node, with one or more threads (e.g., pthreads-based threads) within each MPI task. For example, on a system with N nodes, each node containing C cores, one might design their application to execute with $P=NC$ MPI tasks (one task per core), or with $P=N$ MPI tasks containing t threads, $1 <= t <= C$, or with any other such combinations of P MPI tasks and t threads per task, where $N <= P <= NC$, P/N is an integer, and $1 <= t <= C$.

Access: Among the key design considerations of such supercomputing systems are performance (speed), power (energy used), reliability and resilience, which, at high levels, together contribute to a large installation and operating cost. Due to the immense cost of installing, operating and maintaining supercomputing systems, access and use to the installations typically follow formal allocation and accounting processes. Applications for access are reviewed and granted as projects with set limits on the total number of computing hours consumable by each project. Since the total available time is limited, it becomes important to carefully budget and utilize the allocated time for debugging, testing, fine-tuning and actually running large application runs. As of this writing, access is available, at various institutions across the world, in support of open science research. In the USA, projects may be proposed and approved for execution on supercomputers sponsored by the Department of Energy, National Science Foundation, and Department of Defense, at several national laboratories and supercomputing centers. Some industry-supported research is also possible via outreach initiatives from companies such as Google and Amazon.

Usage with Batch Submission: A common element of surprise to new users of supercomputing installations is the *batch submission* procedure for executing their applications. User submit their executable (with appropriate input files) as a *job* to a batch submission queue; the job then gets scheduled by a job scheduler some time later, upon which the application runs and generates its output to output files. Supercomputing facilities have continued to rely on batch submission as the primary way of execution because of a combination of three reasons: (1) the immense cost of the installation requires it to be shared resource, (2) the system must be operated at the highest possible utilization levels to amortize the large fixed cost in a few years, and (3) most scalable, parallel applications are written with a "clean computer" view that assumes that all the processors allocated to the application are dedicated only to that application.

The following code fragment lists an example batch submission script on a Cray XT series system. Lines prefixed by `#PBS` are parsed by the Portable Batch Submission (PBS) system for user-specified job parameters. The `-A ac00` option specifies use of project account `ac00` to account for consumed computing hours. The `-N testsim` option specifies test as the user-specified name of the job, the `-j oe` option combines both standard output and standard error streams of the job, the `-l walltime=1:30:00` specifies the maximum amount of wall clock time to be allocated to this job to be 1.5 hours (job would be terminated if it happens to exceed this amount of time), and `size=131072` specifies that 131,072 processor cores must be dedicated to this job. The non-directive commands are executed after the processors are allocated for the job. The `aprun` command actually spawns the job; in this example, the executable named `simulation` gets spawned with 131,072 MPI ranks. Submission scripts in other batch systems (e.g., IBM's Load Lever system) are analogous.

```
#!/bin/bash
#PBS -A ac00
#PBS -N testsim
#PBS -j oe
#PBS -l walltime=1:30:00,size=131072
cd /tmp/work/$USER
date
aprun -n 131072 ./simulation
```

2 PARALLEL DISCRETE EVENT SIMULATION (PDES)

With the dawn of multi-petascale (and future exascale) computing, parallel simulation applications are now faced with the reality of the availability of hundreds of thousands of processor cores for their execution on a single computing installation. PDES applications constitute an important class of parallel applications that can stand to benefit from the unprecedented scale. An increasing number of critical applications are evolving to warrant high end computing capabilities. Emergency planning & response, global/local social phenomena prediction & analysis, defense systems operations & effects, and sensor network-enabled protection/awareness systems are all representative application areas. Simulation-based applications in these areas include detailed vehicular and behavioral simulations, high-fidelity simulation of communication effects in large sensor networks, and complex models in social behavioral/agent-based simulations, to name just a few.

In their "next generation" levels of operation, these applications include scenarios characterized by their scale (several millions of simulated entities), their need for fast simulation of multiple scenarios (thousands of alternatives explored as fast as possible for reasoning, understanding and refining the models or solutions), and their (re)configuration based on large-sized dynamic data. When the application scenarios are scaled to large configurations of interest, they warrant the use of parallel discrete event simulation (PDES). For example, regional- or national-scale micro-simulation of vehicular traffic (e.g., for accurate computation of evacuation time or computation of energy consumption) involve the simulation of 10^6-10^7 road intersections and 10^7-10^8 vehicles. Many potential scenarios (e.g., evacuation routes, or alternative energy incentives) are to be evaluated. All such considerations together motivate the need for highly scalable PDES execution capabilities on high performance platforms.

The common core among most of these applications is their use of discrete event-based modeling approaches. Discrete event execution evolves the states of the underlying entities of the model in an asynchronous fashion, in contrast to time-stepped execution. The challenge in moving a parallel discrete event simulator from a few processors to a supercomputing-scale configuration is that new issues arise whose effects are not so pronounced in small-scale parallel execution. For example, synchronization costs that were negligible or tolerable on small numbers of processors can become a dominating cost when scaled to thousands of processors. The simulation may experience a slow down instead of speed up when the number of processors is increased, even if the model affords sufficient concurrency. Event and entity-state data structures, if not properly designed for scalability, can consume memory that exceeds the available memory on a node. New needs, such as flow control and deadlock resolution, arise when the simulator is attempted to be executed over high-end interconnects. The sheer volumes of event traffic across processors due to large numbers of communicating logical processes simulated on each processor core requires additional debugging and optimization methodologies to test and improve the simulator at supercomputing scale. Also, new algorithms are needed to effectively exploit high-end communication technologies only available on supercomputing platforms.

Note that all PDES simulations are single parallel runs, in contrast to replicated independent runs employed by other parallel simulation approaches such as Monte Carlo methods. PDES is far from being embarrassingly parallel. Each simulation run utilizes all available processors, with non-trivial synchronization operations among all processors.

Implementation of PDES engines incurs the complexity and challenges that arise with any parallel computing application. In addition to the traditional issues in the implementation of parallel discrete event simulation, a few issues arise that are largely specific to supercomputing system requirements and characteristics. Some of the issues – synchronization, 1-sided communication, and flow control – are discussed next.

2.1 Synchronization Problem

One of the main challenges is the synchronization of virtual time to enable global timestamp-ordered event execution across all processors. Fine-grained event processing adds significantly to the challenge; event computation can consume little wall clock time soon after which synchronization of event timestamps is logically required among the rest of the events in the parallel system. For example, on current-day processors, application-specific model code inside event computation for Internet Protocol "packet handling" models can be performed in as little as 5-10μs for Internet simulations, and similarly, in 10-50μs for updating vehicular states in microscopic models of vehicular transportation simulations. Correctness of results requires that events are executed to preserve global timestamp order among all events. While several synchronization algorithms have been proposed, only a few have been shown to be inherently scalable, and even fewer have been tested on very large parallel platforms.

Runtime engines for discrete event simulations need to deliver fast and accurate global timestamp-ordered execution across all the processors. Among the major challenges in scaling PDES is increasing the speed of global virtual time computation which directly determines the speed of advancement of the virtual time at all processors.

At the crux of the problem, synchronization of processors for global time ordering of events involves efficiently accounting for events "in flight" between processors. Since all events have to be accounted for in the global time guarantees (else, events in flight can arrive later and potentially result in violation of time ordered processing), the number of potential events in flight increases with the number of processors, and hence, time synchronization cost grows with the number of processors. To keep this cost low, the algorithm must be scalable. A few such scalable algorithms exist, some that exploit the underlying communication support to perform scalable reductions (mostly in synchronous mode), and others that build a simulator-level reduction operation on top of the point-to-point messaging of underlying communication layers. Both approaches have been found feasible; the former using the efficient reduction primitives (e.g., `MPI_Allreduce`) implemented in MPI on IBM Blue Gene systems, and the latter using asynchronous messaging (e.g., `MPI_Iprobe`, `MPI_Testsome`, or `MPI_Testany`) in MPI on Cray XT systems. On Cray XT5 systems, results using both approaches have been reported with PDES runs on up to 216,000 processor cores. On Blue Gene/L and /P systems, the former has been reported in PDES experiments performed on up to 32,768 processor cores. Fundamentally, however, it is clear that asynchronous computation of global virtual time is the only way to keep PDES execution scalable, since the alternative (blocking-collectives) suffer from the high overheads arising from inherent jitter in computation across processors in PDES, and from quashing much of the optimistic processing of events in optimistic parallel simulation.

2.2 1-sided vs. 2-sided Communication

While most parallel simulators are traditionally developed over the two-sided communication, operation over one-sided communication layers is uncommon. However, one-sided communication is a primary mode of inter-processor data transfer on many extant supercomputing systems. Direct memory access engines and interfaces form the native mechanism for data transfer across memory spaces (nodes), with some systems even supporting a single global addressing memory model. Standard application programming interfaces such as MPI are implemented over the direct memory support.

Although one-sided communication has been used in other parallel applications, its use in PDES is different in that it mixes the static nature of inter-processor messaging for global virtual time computation with the dynamic inter-processor interaction for event exchange among logical processors mapped to different processors. PDES being a highly latency-bound parallel application, the low latency exchanges offered by 1-sided communication support can greatly improve the overall speed of simulation.

Parallel discrete event simulation can expect to exploit one-sided communication by achieving a separation of the channels over which synchronization messages (of fixed inter-processor connectivity and fixed volumes) and event messages (of dynamically changing inter-processor messaging topology and variable volumes) are sent.

2.3 Flow Control

When using a large number of processors communicating over a single interconnect, in order to maintain stability and uniform progress across all processors, the amount of network load offered by each processor must be carefully controlled. Without such control, for example, one processor may induce more event (messaging) traffic than can be accepted and processed by the receiving processor in a given amount of time, and the offered load on the entire network may overwhelm the capacity of the network.

As an analogy, the vehicular traffic on a highway in a city gets slower, due to congestion effects, as the number of entry points and the number of vehicles increases. Solutions such as ramp meters are introduced at highway entry points, to carefully regulate the introduction of new vehicles onto the highway at peak traffic times. The ramp meters act as flow controllers that dynamically vary and limit the offered load on the network. The control is designed to find the best tradeoff between the apparently conflicting goals of maximizing the highway utilization and minimizing the average vehicle travel time.

Similarly, the inter-processor event messaging traffic in simulations must be carefully controlled at each processor's network device interface to dynamically optimize the latency and bandwidth use by the simulation. Due to model-level and system-level dynamics, the offered load by each processor onto the network changes dynamically at runtime. This introduces highly time-varying volumes of event messages at each network entry point, which must be buffered at the source, transmitted at the most appropriate time, acknowledgement(s) received and permission given back to the simulator to proceed with additional network activity.

The determination of the optimal flow control algorithm is difficult, as it depends heavily on the properties of the interconnect system of interest and the behavior of the parallel discrete event model. Nevertheless, one can err on the side of safety and correctness, and use a control scheme that prevents deadlocks and minimizes network congestion. To illustrate, a simple scheme is described here that works on any system and any model, but provides a parameter for the user to tune the scheme to minimize latency without overloading the network or destination processors.

Simple Flow Control: Since the send-receive model is in itself synchronous, a simple source-based flow control mechanism would suffice. The basic idea behind this mechanism is to limit, on the sender's side, the number of messages that reach the receiver processor before the receiver gets a chance to absorb them. This is ensured as follows: given any sender-receiver pair, for every *k* messages sent for that pair, the sender awaits an explicit acknowledgement from the receiver indicating reception of all the *k* sends. It is only upon receiving the acknowledgement that the sender initiates a send of the *k*+1th message. The algorithm is given in Figure 1. In the algorithm, the acknowledgement is referred to as a *FlowControlSync* message. Based on empirical observation, a value of *k* in the range of 1 to 10 works sufficiently well.

```
Flow control data structures:
    Constant Integer k; /*Customizable*/
    Integer sendcount[NPE], recvcount[NPE];
    Boolean ready[NPE];
    List ppslist; /*Postponed sends of events*/
PDES-Init()
    For all i: sendcount[i]=0; recvcount[i] = 0; ready[i] = true;
    ppslist = empty-list;
PDES-Send(msg,j) /*Send msg to PE j*/
    If( ready[j] )
        MPI_Send(j,…);
        sendcount[j]++;
        If(sendcount[j] mod k==0) ready[j] = false
    Else
        msg2 = deepcopy of msg
        ppslist.append(msg2)
    Endif
PDES-Poll()
    While( ppslist is not empty )
        MPI_Probe(…)
        If( Non-blocking receive is ready )
            MPI_Recv(Message)
            If(Message is FlowControlSync from j) ready[j] = true;
            Else If(Message is regular message from j)
                recvcount[j]++;
                If(recvcount[j] mod k==0) PDES-Send(FlowControlSync,j)
                Submit Message to engine
            Endif
        EndIf
        Attempt to send a message from ppslist;
    Endwhile
```

Figure 1: Simple algorithm executed at each processor for *flow control* of events sent to other processors

Without flow control, a large-scale execution of the simulation can suffer a slow-down by as much as an order magnitude. In some cases, the simulation may even deadlock (due to hold-and-wait on memory buffers arising from exhaustion of memory) and may not terminate. Thus, flow control can be very important when executing PDES applications on some supercomputing systems.

3 DISCRETE EVENT APPLICATION GUIDELINES

Here, we describe some of the salient features of PDES as distinguished from other supercomputing applications, and provide a few common guidelines on development and implementation of efficient PDES models at supercomputing scales.

3.1 Difference between other applications and PDES

In traditional supercomputing applications that are built on physical system models, the inter-processor communication tends to be largely invariant in structure and the bandwidth needs remain predictable. The time scales of the average inter-message computation tends to be in milliseconds (or greater). In contrast, PDES applications exhibit highly dynamic, time-varying inter-entity/inter-processor interactions, and the time scales of average event computation between two successive events sent to remote processor tend to be in the order of microseconds rather than milliseconds. The fine-grained nature of event computation, and the interludes of establishment of timestamp ordering requirement between any two events result in runtime dynamics that are vastly different from other traditional supercomputing-based simulations.

Another important distinction is in terms of the amount of linear algebra operations performed by the application. Many of traditional supercomputing applications for scientific computing and physical system simulations are such that their core computation gets transformed into linear algebra operations. For example, the model may consist of coupled differential equations which are evaluated using linear algebra methods after discretization. However, in a majority of PDES applications, the core computation is not dominated by linear algebra (in fact, in many cases, there is no linear algebra component at all in the models – e.g., in queueing network models). This distinction has strong bearing on the type of parallelism one can hope to exploit. Parallel speed up is obtained not from scaling single linear algebra primitives (such as matrix-matrix multiplication or matrix factorization), but by evolving the distributed system state over virtual time as fast as possible across all processors. Thus, traditional metrics from benchmarks such as LINPACK on a supercomputer are not a good indicator of performance of PDES applications on the same supercomputer. A more pertinent benchmark, perhaps, is the GUPS benchmark, which can better indicate how well a system would perform with PDES applications. Similarly, the metric based on the number of floating point operations performed per wall clock second is also an inappropriate for PDES, since a large fraction of PDES workload involves integer operations. It is clear that, qualitatively, PDES is an entirely different class of supercomputing applications.

3.2 Model Organization

When contemplating an execution of a PDES application at supercomputing levels, it is important to carefully consider several aspects of the application, such as concurrency, locality, event granularity, lookahead and timestamp distributions.

As a primary consideration, the source of model-level concurrency must be first identified; for example, the number of logical processes available in the modeled scenario must be ascertained to ensure that at least one logical process can be mapped to each processor core (to prevent idling/non-usage of processors). This occurs in different forms in different applications, and creativity may be needed to uncover the concurrency that may be hidden in a naïve/initial modeling exercise. For example, in a national scale model, the choice of whether cities or states are modeled as individual logical processes determines how many of them can be safely delinked and partitioned across processors.

Another important consideration is the amount of messaging locality, which is indirectly determined by the amount of event exchange that occurs across processors. The greater the locality of event exchange (e.g., LPs within the same node communicate more often among themselves than with LPs across nodes), the better the runtime performance.

As a related concept, the amount of computation involved within each event processing unit can play a major role in scalability to supercomputing sizes. If the computation per event is too small, then the runtime can be dominated by the event scheduling and synchronization overheads, since those overheads are large at supercomputing scales. On the other hand, if the computation per event is too large, then the amount of concurrency can be diminished if synchronization can be performed relatively faster than that event time; in this case, some processors will not have events to process in timestamp order, and thereby efficiency suffers. Additionally, the conventional wisdom of parallel simulation, namely, the importance of obtaining the largest values of lookahead possible in the model, applies even stronger in supercomputing scales (Carothers and Perumalla 2010).

3.3 Partitioning

Algorithm Complexity: Given that the model has been divided into logical processes, the logical processes will need to be mapped to processors. This assignment of logical processes to processors becomes a much harder problem at supercomputing scales because, often, the optimal partitioning algorithms become computationally infeasible to compute as the number of elements increases (in fact, have exponential time complexity). Heuristics that reduce the complexity to linear or nearly-linear time are clearly the only choice, since even quadratic time complexity becomes too large when dealing with 10^5-10^6 processors.

Virtual Time Imbalance: To add to the runtime complexity, another issue that is unique to PDES arises, namely, the need to define a variable that accurately captures inter-processor communication cost. Due to global timestamp-based ordering, imbalances in virtual time arise dynamically across processors, which are hard to capture as simple weights to be assigned to the edges of a graph which is to be partitioned across processors.

Computational Time Imbalance: Additionally, depending on the model, the simulation may have multiple event types, with each event type requiring a different amount of computation as part of event processing (e.g., vehicles in a road network may require much more computation for modeling transfer across intersections than movement along a road). This translates to *dynamic* computational load imbalance, which must be incorporated into the partitioning heuristics appropriately.

Although some past work exists, partitioning PDES models at supercomputing scales remains an open area of research, without very many existing solution approaches that are effective.

3.4 Event Data Transfer

In any parallel implementation, one is faced with the unavoidable issue of deciding on the format of inter-processor data transferred at runtime. While several techniques exist, such as Interface Description Language (IDL), most are inapplicable for use in PDES because of the small event granularity. In other words, since event computation often takes only a few microseconds, the overhead of encoding the data for transfer across memories must be much less than a few microseconds. Format changes must therefore be avoided to keep messaging overheads low. Fortunately, most supercomputing installations provide homogeneous set of nodes with respect to architecture, so often direct byte transfer works well and no conversions or encodings are needed. In the µsik engine, for example, events (class instances) are byte copied from one processor memory to another without any serialization and de-serialization overheads. This provides for the fastest way possible to implement event exchanges across processors, giving a good parallel runtime performance. However, the caveat with this approach is that all data within an event data structure must be byte-contiguous, and should not contain data scattered across memory via pointer-based linkage. This often is not a problem for models designed for supercomputing use from the ground up, but can be a thorny issue for existing simulations built with reliance on sequential processing techniques.

3.5 Debugging and Performance Tuning

A major impediment in porting parallel discrete event engines to supercomputing scales is the immense difficulty of debugging a range of issues: (1) correctness of engine code (2) correctness of application model code (3) performance of engine code, and (4) performance of model code.

While there are a very small number of graphical and automated tools to help the user with initial debugging, the problem of accurate determination, isolation and fixing of correctness or performance bugs remains a highly niche task. Tools such as `TotalView` are available for debugging at relatively small scale (a few thousands of cores), they become unwieldy for effective use at the largest processor counts. It should not come as a surprise that many of the users still rely on and resort to simple instrumentation of their own code to print various status lines and data values to the standard output of their programs. To facilitate this type of debugging that relies on generation of user-inserted output, a separate file for printing the standard output per MPI task/rank is very useful. Also, barriers may need to be inserted liberally into various places in the code during debugging, but to be turned off for production runs. The output needs to be explicitly flushed by the programmer to ensure the last actions are in fact logged and available for inspection in backtracking towards a bug.

With respect to performance tuning, one of the most common optimizations that is always suggested (and probably good practice) is the pre-posting of message sends and receives before they are actually received or sent respectively. This can be very important for good performance on some of the largest installations. More fundamentally, pre-posting is unavoidable for good performance of PDES because it is not possible for the system to automatically predict the dynamically changing inter-processor event exchanges in the model. Small-scale debugging tools such as `Vampir` and `PAPI` are also useful to debug performance problems.

As a general performance-friendly insight, it is useful to limit as much as possible the number of different destinations to which each processor sends messages; try to arrange so that the set of destinations does not vary too rapidly. This reduces the overhead of the interconnect to set up and tear down communication channels dynamically, which can make an order of magnitude difference in observed performance.

4 STATE-OF-THE-ART

As a result of research over the past three decades, PDES has now reached a stage where the runtime engines of the most scalable PDES simulators are able to gainfully execute on 100s of thousands of processor cores in a single simulation run. Roughly estimating, the scalability of PDES engines seems to have increased by almost an order of magnitude every 3 years, starting with about 16 processors in the year 1997. The advancements were demonstrated using benchmarks such as PHOLD or in the context of large-scale network simulations. The largest number of processors to which PDES runs were scaled is 216,000 processor cores with the µsik PDES engine on a range of benchmarks. Overall, several PDES applications have now been scaled to supercomputing levels; a few representative ones are listed below.

- Epidemiological simulations – Models of epidemic outbreaks were modeled at the level of individuals, with probabilistic timed transitions for disease evolution inside each individual (EpiSims and EpiSimdemics simulations reported in (Barrett, Bisset et al. 2008; Bisset, Jiangzhuo et al. 2011) scaled to 10^3 processors, and the results in (Perumalla, Park et al. 2011; Perumalla and Seal 2011) demonstrated scaling to nearly a billion individuals simulated using 10^5 processors.
- Radio signal propagation simulations A discrete event model of radio wave propagation was parallelized and simulated using optimistic (Time Warp) methods in which reverse computation was used for rollback, and scaled to several thousands of Blue Gene processors (Bauer, Carothers et al. 2009) and also to over 127,500 processors of Cray XT5 (Seal and Perumalla 2011)

- High-performance interconnect simulations – A parallel discrete event model of a futuristic, billion-node torus network was simulated using 32,768 cores of a Blue Gene/L system (Liu and Carothers 2011).
- MPI simulations –A purely discrete event execution-based simulation of MPI programs at massive scales are achieved on up to 216,000 processor cores of a Cray XT5 machine. The μπ simulation system was exercised with several million virtual ranks in MPI benchmarks (Perumalla 2010; Perumalla and Park 2011).
- Internet simulations – In some of the pioneering work on extending the scalability of discrete event simulations, detailed packet-level models of Internet traffic was simulated with millions of network hosts and routers using high-fidelity software models of TCP protocols (Fujimoto, Perumalla et al. 2003; Park, Fujimoto et al. 2004). These represented some of the largest PDES application results of the day, scaling to the 1,536 processors of the largest supercomputer of that time.
- Hardware circuit simulations – Recently, massively parallel discrete event execution on several thousands of processors has been extended to simulating very large numbers of circuits in detailed hardware simulations of microprocessors (Gonsiorowski, Carothers et al. 2012)
- Social behavior simulations – Very large-scale simulations of certain cognitive/social behavioral phenomena (Naming Game model for consensus formations) were performed on Blue Gene platforms (Lu, Korniss et al. 2008). These are some of the largest social behavioral simulations executed on supercomputers, with a variety of complex inter-entity connectivity (such as scale-free networks and small world networks), exercising complex inter-processor communications.

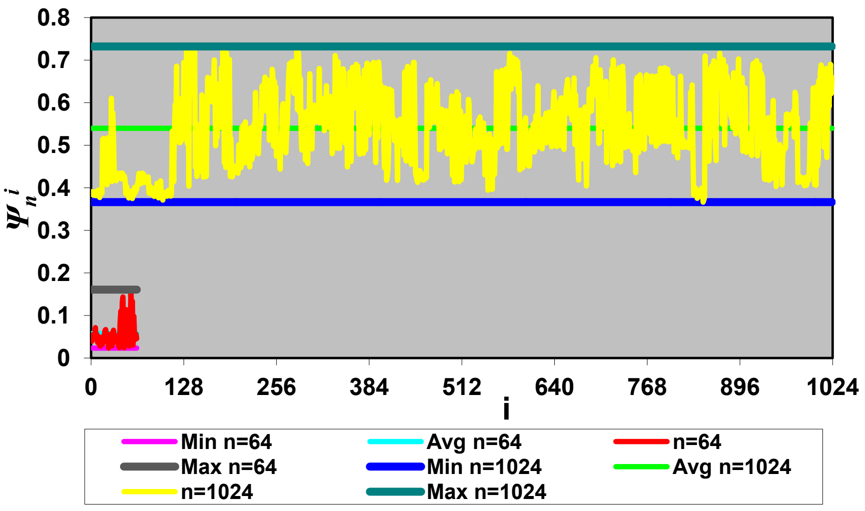
5 TIME STEPPED SIMULATION ON SUPERCOMPUTERS

Time-stepped simulation is the other major class of parallel simulations that are very commonly used for large-scale execution on supercomputers. While time-stepped simulation is relatively easier to implement, it quickly encounters scalability problems when supercomputing scale execution is attempted. In time-stepped simulation (see Figure 2), the algorithm's correctness crucially depends on the barrier operation that is need to ensure all processors complete a time step before entering the next time step. This barrier operation is relatively efficient on smaller parallel systems; however, the blocking nature of the barrier primitive induces two problems: (1) the slowest processor holds back the rest of the processors, making all of them idle, (2) the overall parallel execution becomes sensitive to dynamic variations in the processor usage, such as from operating system jitter, and (3) the runtime cost of barrier algorithm increases with larger number of processors, making each time step take longer to execute. As the scale is increased, these effects become extremely detrimental.

```
$t_{now}$ = 0
While $t_{now}$ < $t_{end}$
    barrier() /*Blocking synchronization with rest of processors to start new time step*/
    $t_{now}$ += timestep /*Advance current simulation time*/
    For all (i on this processor)
        advance( i, $t_{now}$ ) /*Advance all entities' state to next time step*/
    For all (i on this processor having off-processor neighbors):
        /*Send copy of new state to all neighboring entities*/
        send( state(i), neighbor-processors(i) )
```

Figure 2: Time-stepped algorithm for parallel execution

Let C_i be the time spent by processor i (MPI task) while waiting/processing inside collective communication calls. Then, $C^n=\sum C_i$ is total collective call time consumed by application on n processors. Let T^n be the total execution time of application on n processors. Then, the average loss of efficiency is C^n/nT^n, and $\Psi_n^i=C^i/T^n$ is the fraction of processor i's capacity lost to collective communication when the application is executed on n processors. With time-stepped simulation, the fraction Ψ_n^i can grow dramatically with n depending on the amount of imbalance across processors in the application. This is illustrated in Figure 3 with runtime data from actual time-stepped applications.



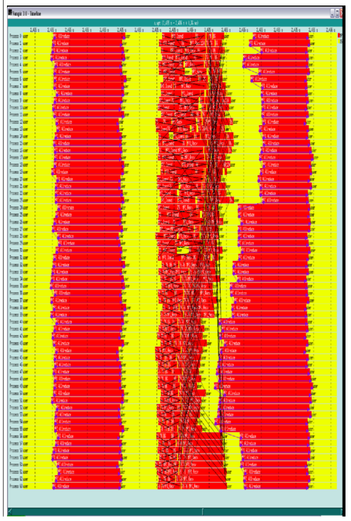


Figure 3: Efficiency is lost due to imbalance across processors in a time-stepped simulation when scaled from **n**=64 processors to **n**=1024 processors (left); visualization of example blocked time in red (right)

One method to solve this problem is to alter the model to permit some computation-communication overlap, so that some computation (either computation of a part of the next time step, or a pre-computation that can be used in the next time step) is performed while the barrier (or similar collective, such as all-to-all reduction) is executed. To enable this at the communication interface level, non-blocking variants of collective communications are needed. Extensions to standards such as MPI are becoming available to include support for non-blocking collectives (Hoefler, Lumsdaine et al. 2007; Hoefler and others 2007).

ACKNOWLEDGMENTS

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

Barrett, C. L., K. R. Bisset, et al. (2008). EpiSimdemics: An Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks. Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Austin, Texas, IEEE Press.

Bauer, D. W., C. D. Carothers, et al. (2009). Scalable Time Warp on Blue Gene Supercomputers. Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation, IEEE Computer Society**:** 35-44.

Bisset, K., C. Jiangzhuo, et al. (2011). Interaction-based HPC modeling of social, biological, and economic contagions over large networks. Proceedings of the 2011 Winter Simulation Conference (WSC).

Carothers, C. D. and K. S. Perumalla (2010). On deciding between conservative and optimistic approaches on massively parallel platforms. Proceedings of the 2010 Winter Simulation Conference (WSC).

Fujimoto, R. M. (1990). "Parallel Discrete Event Simulation." Communications of the ACM **33**(10): 30-53.

Fujimoto, R. M. (2000). Parallel and Distributed Simulation Systems, Wiley Interscience.

Fujimoto, R. M., K. S. Perumalla, et al. (2003). Large-Scale Network Simulation -- How Big? How Fast? Modeling, Analysis and Simulation of Computer and Telecommunication Systems.

Gonsiorowski, E., C. D. Carothers, et al. (2012). Modeling Large Scale Circuits Using Massively Parallel Discrete-Event Simulation. IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. Washington, D.C., IEEE Computer Society.

Hoefler, T., A. Lumsdaine, et al. (2007). Implementation and performance analysis of non-blocking collective operations for MPI. Proceedings of the 2007 ACM/IEEE conference on Supercomputing. Reno, Nevada, ACM**:** 1-10.

Hoefler, T. and others (2007). "Optimizing a Conjugate Gradient Solver with Non-blocking Collective Operations." Parallel Computing **33**(9): 624-633.

Liu, N. and C. D. Carothers (2011). Modeling Billion-Node Torus Networks Using Massively Parallel Discrete-Event Simulation. Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation, IEEE Computer Society**:** 1-8.

Lu, Q., G. Korniss, et al. (2008). "Naming games in two-dimensional and small-world-connected random geometric networks." Physical Review E **77**(1).

Park, A., R. M. Fujimoto, et al. (2004). Conservative Synchronization of Large-scale Network Simulations. Workshop on Parallel and Distributed Simulation.

Perumalla, K. S. (2006). Parallel and Distributed Simulation: Traditional Techniques and Recent Advances. Winter Simulation Conference, Monterey, California, USA, INFORMS.

Perumalla, K. S. (2010). µπ:: A Scalable and Transparent System for Simulating MPI Programs. Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques. Torremolinos, Malaga, Spain, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering)**:** 1-6.

Perumalla, K. S. and A. J. Park (2011). Improving Multi-million Virtual Rank MPI Execution in [MUPI]. Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on.

Perumalla, K. S., A. J. Park, et al. (2011). GVT algorithms and discrete event dynamics on 129K+ processor cores. High Performance Computing (HiPC), 2011 18th International Conference on.

Perumalla, K. S. and S. K. Seal (2011). "Discrete event modeling and massively parallel execution of epidemic outbreak phenomena." Simulation **Online-First**(2011).

Seal, S. K. and K. S. Perumalla (2011). "Reversible Parallel Discrete Event Formulation of a TLM-Based Radio Signal Propagation Model." ACM Trans. Model. Comput. Simul. **22**(1): 1-23.

AUTHOR BIOGRAPHIES

KALYAN S. PERUMALLA founded and leads the High Performance Discrete Computing Systems team at the Oak Ridge National Laboratory. He is a Senior R&D Staff and Manager in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory, and an Adjunct Professor at the Georgia Institute of Technology. He holds a PhD in Computer Science (1999, Georgia Institute of Technology). His email address is perumallaks@ornl.gov.