μsik – A Micro-Kernel for Parallel/Distributed Simulation Systems

Kalyan S. Perumalla
www.cc.gatech.edu/fac/kalyan
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, USA

Abstract

A novel micro-kernel approach to building parallel/distributed simulation systems is presented. Using this approach, a unified system architecture is developed for incorporating multiple types of simulation processes. The processes hold potential to employ a variety of synchronization mechanisms, and could even alter their choice of mechanism dynamically. Supported mechanisms include traditional lookahead-based conservative and state saving-based optimistic execution approaches. Also supported are newer mechanisms such as reverse computation-based optimistic execution and aggregation-based event processing, all within a single parsimonious application programming interface. The internal implementation and a preliminary performance evaluation of this interface are presented in ***μsik****, which is an efficient parallel/distributed realization of the micro-kernel architecture in C^{++}.*

1. Introduction

High-performance parallel and distributed discrete event simulation (PDES) systems have traditionally been built from the ground up, one for each major variant of various PDES techniques. However, it is desirable to have the freedom to add new techniques without having to develop entirely new systems from scratch for each variant. To this end, we are interested in isolating the core invariant portion of PDES techniques, and in providing a generalized framework for building traditional as well as newer techniques on top of the core. The core constitutes the micro-kernel, and the traditional implementations (conservative or optimistic) form the system services on top of the micro-kernel. This permits the incorporation of newer techniques on top of the core, as well as optimization of existing system services, without the need for system-wide changes.

The PDES micro-kernel approach is based on analogy with operating systems[1]. In operating systems that are based on micro-kernel architecture, a very basic set of services is provided by the operating system core (e.g., process identifiers and address spaces). Using such primitive services, the rest of the system services are in fact built outside the core (e.g., file systems and networking). We borrow this approach in our system. A micro-kernel operating system provides an easy and safe way of adding new system/kernel services, such as new network protocols and file systems. Similarly, a PDES micro-kernel provides an easy way to add new types of simulation processes without the need for an overhaul of the entire PDES system implementation.

Our micro-kernel approach is experimental in nature to test the feasibility of developing such a system that can accommodate multiple synchronization techniques and endure additions over time, while at the same time maintaining high-performance execution without undue performance penalty.

The rest of the document is organized as follows. Section 2 presents the motivation and background for the design and development of the micro-kernel approach. The micro-kernel concepts for PDES are introduced in Section 3. Implementation details of the micro-kernel interface are described in Section 4. A preliminary performance study of our microkernel implementation on a distributed platform is presented in Section 5. Finally, current status and future work are presented in Section 6.

2. Motivation and Background

In some of our current projects in collaboration with modeling experts in physical sciences, we are pursuing development of physics simulation models (e.g., of Earth's magnetosphere). These physics simulations are complex, and employ fine-grained events. It is unknown as to which synchronization method works best for these models, hence a specific synchronization scheme cannot be chosen *a priori*. More ideally, the models can benefit from a single engine that not only semi-transparently supports multiple synchronization approaches, but also entails execution with sufficiently low overheads. A generalization of the goals is for the simulation system to allow simulation processes the freedom to adopt any event processing scheme, or freely switch between schemes at runtime. Additionally, since our focus is on very large-scale simulations, especially of physics models in our current projects, we need scalable parallel/distributed execution capabilities.

2.1. Traditional vs. New Systems Approach

The method of prevalence in building PDES systems is to build the system specifically for one synchronization method (e.g., one conservative algorithm, or one optimistic variant). This tradition has two fallouts. First, additions to the underlying framework involve major overhauls. Secondly, modelers need to either determine and stick to one mechanism, or re-code their models to switch to a new mechanism. Such a limitation is deplorable: the PDES research community has developed a host of techniques for high-performance execution; yet, an elegant *systems* framework is lacking for incorporating the multitude of techniques in an incremental, modular fashion.

Our thesis is that a large number of PDES techniques can be *transparently* supported in a single unified framework, with a small set of fundamental primitives. Based on this premise, we develop a unified application program interface (API) that encompasses most, if not all, synchronization approaches. Using this interface, simulation models can be written in a manner that is resilient to changes and optimizations.

2.2. Related Work

The High Level Architecture (HLA)[2] defined by the US Department of Defense provides services for integrating a wide variety of simulator implementations, including space and/or time parallel (conservative, optimistic) discrete event simulations, and time-stepped continuous simulations. However, the architecture has been designed for interoperation of coarse integration entities, such as distributed programs communicating over the network. As such, it is not well-suited for integration of fine-grained entities, as in the hosting of multiple event-oriented logical processes and/or threads within a single UNIX process. In particular, primitives to facilitate efficient process scheduling are not addressed in the standard; such primitives turn out to be the key to efficient execution of fine-grained autonomous entities.

A more closely related work is by Jha and Bagrodia[3] in which a unified framework is presented to permit optimistic and conservative protocols to interoperate and alternate dynamically. (A variation of Jha and Bagrodia's protocols is later discussed in [4], but in the context of VLSI applications). High-level algorithms are presented in [3] that elegantly state the problem along with their solution approach. However, they do not address system implementation details or performance data. Their treatment provides proof of correctness, but lacks an implementation approach and a study of runtime performance implications. Our work differs in that we are interested in defining the interface in a way that guarantees efficient implementation, and we describe details of a high-performance implementation of such a unified interface. Some of our terms share their definitions with analogous terms in their work, but our interface uses fewer primitives and diverges in semantics for others. For example, our interface does not require the equivalent of their Earliest Output Time (EOT). Also, in contrast to their need for lookahead, we do not require that the application always specify a non-zero lookahead. Their related PARSEC system supported an API for processes to dynamically switch between optimistic and conservative modes, but we differ in our systems approach in implementing similar functionality. Another related work is by Rajaei, Ayani and Thorelli [5] on a hierarchical system to combine Time Warp with conservative execution; this work overlaps in goals with our work, but differs in approach.

SPEEDES[6] is a commercial optimistic simulation framework that is capable of distributed execution; however, we were unable to find evidence on its large-scale parallel performance capabilities for fine-grained applications. GTW[7] and ROSS[8] are representative of high-performance implementations of optimistic simulators, but they are restricted to parallel execution on symmetric shared memory multiprocessor (SMP) platforms. The SMP-only constraint sometimes limits the user's choice of hardware as well as scalability. An exception is the WARPED simulator[9], a shared-memory time warp system extended to execute on distributed memory platforms, but it has only been evaluated on relatively small hardware configurations. We are interested in scalable execution on large-scale computing platforms, such as large clusters (hundreds) of quad-processor SMP machines typically available in supercomputing installations for academic research. The cluster-of-SMPs platform is appealing since it is relatively less expensive as compared to a comparable SMP system for large number of processors.

We note that, while the possibility of switching between types of protocol is not entirely new, our parsimonious API and our high-performance implementation approach are novel.

3. PDES Micro-Kernel Concepts

In this section, we introduce some terminology and concepts, and provide high-level descriptions of important micro-kernel operations. It is assumed that PDES models are written in terms of simulation processes that exchange events. Multiple simulation processes (also called logical processes) hosted on each processor. Operationally, one operating system process (e.g., a UNIX process) hosts several simulation processes.

In the PDES micro-kernel system view, simulation processes are fully autonomous entities. They are free to determine for themselves when and in what internal order they would process their received events. The micro-kernel does not process events in and by itself – it only acts as a router of events. In particular, it does not generate, consume or buffer any events. It does not examine event contents, except for the event's header (source, destination and timestamp). The micro-kernel does not distinguish between regular events, retraction events, anti-events or multicast events. It also does not perform event buffer management (memory reuse, fossil collection, etc.), in contrast to traditional parallel/distributed simulation engines. The distinctions among event types and their associated optimizations are deferred to protocol-specific functionality of services outside the kernel proper. The responsibility of a micro-kernel is restricted to only providing services to the simulation processes such that the processes can efficiently communicate events with each other, and collectively accomplish "asymptotic" time-ordered processing of events.

3.1. *Core Services*

The micro-kernel core consists of *naming*, *routing* and *scheduling* services, as follows:

- **Naming**: The micro-kernel provides a uniform way for simulation processes to locate and refer to each other, within and across processors in a parallel/distributed execution setting. A list of valid identifiers is maintained to map identifiers to processes and vice versa.
- **Routing**: The routing services ensure that events are transparently forwarded to the receiver process, regardless of whether the sender and receiver are co-located or distributed across processors. This is coupled with a guarantee that no event timestamp is overlooked in global timestamp-ordered processing.
- **Scheduling**: The micro-kernel takes care of allocating CPU cycles among multiple simulation processes in a manner that best promotes simulation progress, and ensures absence of livelock or deadlock.

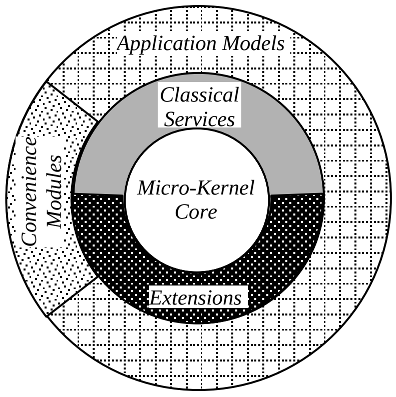
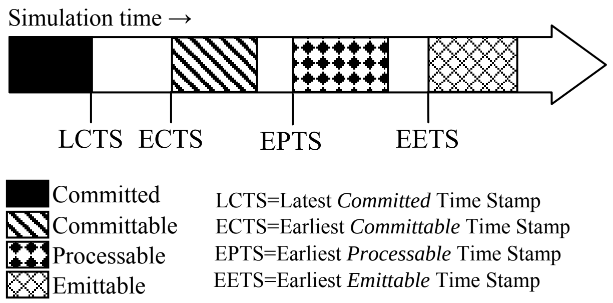


Figure 1: Elements of the micro-kernel architecture, and their inter-relationships.

A wide variety of PDES mechanisms can be built around this parsimonious set of core services, as outlined in Figure 1. Classical services include support for conservative and optimistic processing – event processing/commitment, rollback support and lookahead specification services. They also include kernel process support for remote communication, retractions and multicast (group) communication. Extensions are placeholders for newer techniques in the future, such as "aggregate event processing", "constrained out-of-order execution" and the like (to appear in later publications). Convenience services include routines such as initialization, timers, and reversible random number generation.

3.2. *Event Lifecycle and Categories*

Events can be considered to go through different stages in their life cycle. First an event is allocated and scheduled by a sender simulation process. Next, the receiver simulation process performs initial processing of the event. This stage includes executing application (model) code associated with that event type. Eventually, in a following stage, final actions associated with the event are committed. Finally, the memory used by the event is released and recycled.



Simulation time →
LCTS
ECTS
EPTS
EETS
Committed
Committable
Processable
Emittable
LCTS=Latest *Committed* Time Stamp
ECTS=Earliest *Committable* Time Stamp
EPTS=Earliest *Processable* Time Stamp
EETS=Earliest *Emittable* Time Stamp

Figure 2: Illustration of the simulation timeline and important event categories in each simulation process. The relation LCTS≤ECTS≤EETS always holds.

Based on the disposition of event lifecycle stages, at any given snapshot moment during simulation, all events belonging to a simulation process can be categorized into four distinct classes – committed, committable, processable and emittable. The first set of events (committed set) is those that have been processed, committed and whose memory has been released for reuse. The second set (committable set) consists of those that have been processed but are waiting to be committed. The third set (processable set) consists of events received by this simulation process that are waiting to be processed. The final set (emittable set) is a logical set that comprises those events that are potentially schedulable by this simulation process to other simulation processes (excluding itself) during the processing of its current set of committable and processable events. Event categories and their mutual ordering are illustrated in Figure 2.

In purely conservative processes, all application code executes during "commit" stages of events. In optimistic processes, revocable portions (slices) of code execute during the "process" stage, while irrevocable portions are done in the "commit" stage.

A Lower Bound on Time Stamp (LBTS) value is defined as a distributed snapshot[10, 11] of the least EETS value among all processes in the simulation. It is essentially a guarantee on the value of the smallest timestamp receivable by any process in future.

When LBTS advances to/beyond the timestamp of a committable event, examples of actions performed when committing the event include, but are not limited to, the following:

- **State vector release**: Release of state vectors, if any, used for state saving during optimistic processing of the event.
- **Input/Output**: Operations such as conservatively printing output to the terminal, or reading from a file.
- **Memory allocation/release**: Finalizing the effect of dynamic memory operations initiated by the application while processing the event.

3.3. Determining Event-Category Times

For classical services, assume that the events in a process are *logically* stored in two data structures: FEL and PEL. The Future Event List (FEL) consists of events in the process' processable event set. Processed Event List (PEL) consists of events in the process' committable event set. For a simulation process *i*, let FEL_i^{top} be the minimum timestamp in FEL_i (infinity if FEL_i is empty) and PEL_i^{top} be the minimum timestamp in PEL_i (infinity if PEL_i is empty). Note that PEL_i^{top} is always infinity for conservative simulation processes.

The earliest time stamp for each event category is determined as follows:

1.	$ECTS_i$	=	*Min*(FEL_i^{top}, PEL_i^{top})
2.	$EPTS_i$	=	*infinity* if conservative FEL_i^{top} if optimistic
3.	$EETS_i$	=	*Min*(FEL_i^{top} + $Lookahead_i$, PEL_i^{top})

In the preceding equations, $EETS_i$ is defined rather simplistically, but could accommodate additional complexity if so desired. For example, if lookahead is highly variable across events, $EETS_i$ could be defined on a per-event basis: $EETS_i=min(E_j+LA_j)$ for each event E_j in FEL_i, and LA_j is the lookahead for event E_j. Similar refinements can be made based on limiting it by the set of destination processes of process *i*. Additional refinements can be made for optimistic processes as well. The main idea is that the event categories provide simple yet powerful abstractions that enable several types of synchronization.

3.4. Process Scheduling

On each processor, the scheduling algorithm proceeds by executing the code in Figure 3 within a loop:

```
1.  if( $ECTS_{min}$ < LBTS )
2.      $P_{ECTS\text{-}min}$.advance( LBTS )
3.  else
4.      $P_{EPTS\text{-}min}$.advance_opt( $EPTS_{min2}$ )
```

Figure 3: Micro-kernel's (simplified) scheduler loop.

$ECTS_{min}$ is the minimum ECTS among all processes on that processor. $Process_{ECTS\text{-}min}$ is the process with the minimum ECTS value. $Process_{EPTS\text{-}min}$ is the process with the minimum EPTS value. $EPTS_{min2}$ is the second least EPTS value among all processes on that processor. The method P.advance(T) conservatively processes all events of process P with timestamps less than or equal to time T. The method P.advance_opt(T) optimistically processes all events of process P with timestamps less than or equal to time T. Either method is a no-op if P is null. The operation of this loop will become clearer in the following two subsections.

The LBTS itself is computed as the minimum EETS among all processes across all processors. Any transient event (in transit across processors) is accounted for by the sender process' queues until the event reaches its receiver process. The LBTS computation can either be performed concurrently with the scheduler, or, periodically inside the scheduler loop just prior to each optimistic processing step (line 4).

3.5. Conservative Processing

During normal processing, the micro-kernel only schedules conservatively executable actions in increasing order of their committable timestamps. Only those processes whose ECTS values are less than or equal to the LBTS value are considered for conservative scheduling. The process with the least ECTS value is scheduled, and it is permitted to advance up to and including the current LBTS value. When that process is finished with its processing, the micro-kernel schedules the process with the next minimum ECTS value, and so on. Note that new events, if any, generated by the scheduled process will necessarily have timestamps greater than or equal to the current LBTS value.

If no process exists whose ECTS value is less than or equal to the current LBTS, then the micro-kernel initiates a new LBTS computation (if one is not already in progress). A new LBTS value typically takes time to be computed, due to communication latency across processors. It is this delay that induces blocking of conservative computation. This blocking period can be utilized as an opportunity to perform optimistic event processing. Hence, while a new LBTS value is being computed, the micro-kernel schedules those processes that are capable and willing to perform optimistic event processing, as described next.

3.6. Optimistic Processing

In optimistic mode, the micro-kernel schedules the process that has the least EPTS value. Recall that the EPTS value for conservative processes is infinity, and for optimistic processes it is equal to the minimum timestamp among unprocessed events (or, infinity if FEL is empty). Thus, if there are any optimistic processes, their EPTS values can make them schedulable for optimistic processing.

When at least one optimistic process exists for scheduling, optimistic execution is scheduled as follows: two processes with the minimum and the next minimum EPTS values (say, $EPTS_{m1}$ and $EPTS_{m2}$) are selected. If only one optimistic process exists, $EPTS_{m2}$ is set to infinity (in this case, this limit needs to be customized, if necessary, to throttle unbounded optimism). Then, the process with $EPTS_{m1}$ is allowed to optimistically process its events with timestamps less than or equal to $EPTS_{m2}$.

Initiating optimistic execution only when all conservative processing is blocked ensures that time spent in correct execution is maximized, and the potential for incorrect execution (in optimistic mode) is minimized.

4. Micro-Kernel Implementation

We now describe our implementation of the micro-kernel approach in a new software system named μsik (**micro si**mulation **k**ernel, pronounced "mew-seek"). μsik is written in C^{++}, linkable to an application as a library, and provides class hierarchies rooted at base classes corresponding to micro-kernel concepts.

A naïve implementation of the micro-kernel approach could entail significant overheads, as compared to the traditional monolithic simulator implementations. In a monolithic simulator, it is possible to optimize the implementation by employing centralized data structures such as event buffers, event lists and state vectors. On the other hand, in a micro-kernel, the key data structures are, by design, encapsulated inside simulation processes. The challenge is to find efficient ways of implementing the micro-kernel framework so as to minimize or eliminate overheads.

A key issue is the problem of always accurately tracking the ordering among processes with respect to their ECTS, EPTS and EETS values. For example, when a new event is sent from one simulation process to another, the receiver's ECTS, EPTS and EETS values can change. Similarly, a simulation process will have its values changed at the end of processing an event. Event retractions need to be dealt with appropriately, as they too alter timestamp ordering.

It is clear that the right choice of data structures determines the efficiency of micro-kernel operation. As its main components, the micro-kernel maintains a list of local user processes, a hash table for mapping process identifiers to processes, and a list of "kernel processes". For scheduler operations, three important priority queues are maintained. Each of these components is described next.

4.1. Naming Services

To provide naming services, the micro-kernel maintains a mapping of process identifiers to process instances. Process identifiers are specified as a pair of integers: (processor number, local process number). Simulation processes can be "kernel processes" or user processes. Kernel processes are used for internal implementation of services on top of the micro-kernel (see Section 4.3). User processes are part of application model.

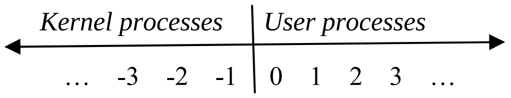


Figure 4: Every simulation process is assigned a locally unique identifier as soon as it is added to the simulation. User processes are assigned positive identifiers starting with 0, while kernel processes are assigned negative identifiers.

User processes are assigned local identifiers as positive integers, starting at 0, while kernel processes are assigned negative integers, as shown in Figure 4. The rationale behind this scheme is that it allows applications to rely on their processes being identified from 0 to *n*-1 (this is a common way in which models are written). Using negative identifiers for kernel processes makes them transparent to the application, and will not interfere with the traditional modeling methods. Special identifiers are also defined for specifying an invalid identifier, and to specify multicast destinations.

4.2. Scheduling Services

The scheduler is implemented as a loop inside a micro-kernel method.

Process Ordering

Three in-place *min-heaps* are used, one each for tracking the ECTS, EPTS and EETS values of simulation processes. Each heap maintains the minimum time-stamped process at the top. For example, the process with the least ECTS value is always available as the top of ECTS heap. The heaps are designed to rapidly update and readjust the elements when the key of an element is increased or decreased. This rapid update is essential to quickly keep the heaps consistent before and after every scheduling action by the scheduler (see also Section 4.4).

Readjusting Timestamp Orders within Scheduler

When events are sent or received by simulation processes, the processes' relative ordering can change with respect to their ECTS, EPTS, and EETS values. The heaps of the micro-kernel scheduler need to be readjusted to restore correct timestamp order. This readjustment is accomplished via a pair of before_dirtied() and after_dirtied() methods within the base simulation process. These methods keep track of whether any changes occurred to the key timestamps. If (and only if) any of the ECTS, EPTS or EETS values of an affected process changes, the corresponding scheduler heap is readjusted. The affected process that needs to be updated could be the active (sending) process that is currently scheduled, or, it could also be the set of processes to which the currently scheduled process generates new events.

Distributed Time Synchronization

To compute LBTS values, we employ the distributed snapshot algorithm described in [12]. We use the publicly available implementation [13] of this algorithm. Its current implementation includes two different modules: one is based on efficient global hierarchical reductions[12, 14], while the other is based on an optimized variant[15] of the Chandy-Misra-Bryant null message algorithm[16]. These are reported to have been tested by their authors on large-scale platforms, and demonstrated to scale very well, even up to supercomputing configurations of more than 1500 processors[14, 15, 17].

4.3. Routing Services

Local event exchange is trivially handled by enqueueing the event in the local destination process. Remote communication is implemented via a special delegation mechanism using kernel processes (see next). The micro-kernel itself never stores or buffers any events at any time. Every event routed through the micro-kernel is immediately delegated either to the destination process (if it is a local user process), or delegated to a local kernel process (if the destination is a remote process or a multicast group). We omit discussion of multicast communication due to space limitations.

Kernel Processes

Kernel processes are used to implement remote federate communication and multicast event exchanges. The reason they are implemented this way is that the functionality can be quite seamlessly implemented using the scheduling services provided by the micro-kernel core. This is fairly analogous to operating system micro-kernels. Services such as networking, file I/O, etc. are implemented as processes outside the micro-kernel core, which themselves utilize many of the services that normal user processes utilize.

A notion of kernel processes for PDES is introduced (for improving rollback efficiency) in ROSS[8]. Our concept of kernel processes and its usage is quite different and unrelated, serving a different notion and purpose.

Remote Event Communication

On each processor, one kernel process is instantiated for every other (remote) processor. These kernel processes for remote communication act as local representative proxies for the corresponding remote processors. This scheme operates as follows.

Let us denote by KP^i_m the *m*'th kernel process on processor *i*. When a user process on processor *i* attempts to send an event to a user process on a remote processor *m*, the micro-kernel on processor *i* delegates that event to its local kernel process KP^i_m. KP^i_m is then responsible for forwarding the event to KP^m_i, which is its peer kernel process on processor *m*. When KP^m_i receives that event, it forwards to the destination user process (guaranteed to be local) via the micro-kernel.

This scheme, despite its simplicity, affords elegant implementation of a wide range of features and optimizations studied in PDES literature. Sophisticated variants can be incorporated with few changes to the rest of the system. Here we briefly discuss a few possibilities:

Optimistic Sends: In this most common method, an event scheduled to a remote process is immediately sent over the wire to its corresponding remote processor. A downside with this scheme is that the network communication cost becomes a wasted overhead if the event is later retracted. The event retraction could be initiated either by the user (in conservative or optimistic processing) to take back a previously scheduled event, or by the kernel for event cancellation (anti-messages for secondary rollbacks in optimistic processing).

Lazy Sends: Instead of forwarding the event immediately over the wire to the remote processor, the event could be withheld within the kernel process for *dt* simulation time units, where $0<dt\leq(T_{event}-T_{now})$. Delaying the event longer will postpone the network communication cost, which is beneficial in case the event is retracted later. On the flip side, it might increase the event communication latency, and stall the receiving processor waiting to receive the event for its own progress. Adaptive schemes could be devised and implemented in the kernel process to exploit this "lazy send" optimization.

Non-aggressive Sends: The kernel process can also be used to easily implement non-aggressive sends – i.e., to send remote messages if and only if they cannot be retracted in the future. This is a well-known PDES variant in optimistic simulation to separate risk and aggressiveness [18], in which events are processed optimistically *locally*, but only "correct" events are propagated *across* processors. The kernel process adds the event in its FEL, and "processes" the events in a conservative fashion. The event is actually sent over the wire to the remote processor only when it is committed. Since events are committed only when they are guaranteed to be not retracted, non-aggressiveness is assured.

Message Bundling: To amortize the cost of network communication, it is possible to bundle multiple events into one message. The cost savings can be good especially when events are small in size, as compared to network message headers (e.g., TCP header size). Again, such bundling techniques can be incorporated into the kernel process responsible for remote communication.

All KP^i_m are responsible for maintaining a mapping from event identifiers to event buffers. Such a mapping is necessary in order to implement event retractions (during conservative and/or optimistic execution) and anti-events (to realize secondary rollback/cancellation in optimistic execution). The kernel process is also responsible for periodically flushing the hash table when events are committed and can no longer be retracted or canceled.

The beauty of this kernel process scheme lies in the fact that the kernel processes themselves are time-synchronized automatically (since they are simulation processes themselves). This fact can be exploited to easily and modularly implement the aforementioned variants.

4.4. *Optimized Queues and Lists*

The micro-kernel uses priority queue and list data structures in its implementation. The efficiency of these data structures is critical for keeping runtime overheads low. We define our own heap and list data types, to avoid overheads of dynamic memory allocation of conventional libraries. Our definitions are different from other standard library templates in that our definitions permit the same object to be linked into multiple instances of the same container type, without the need to allocate container headers to hold the elements. Standard template libraries are difficult to use or inefficient when the same element needs to belong to multiple instances of the same type of container *simultaneously*.

For example, in our micro-kernel, we need to link each simulation process into three different priority queues, all at the same time. This is to order the processes along their three basic timestamps: ECTS, EPTS and EETS. The key used for ordering in each queue is different, yet, the container data type is exactly the same (a min-heap priority queue).

5. Performance Study

We now turn to a study of runtime performance.

Platform: Our implementation currently runs on a network of shared-memory multiprocessors, and is portable across homogeneous configurations of Windows, Mac and Unix/Linux platforms. All performance data reported here are collected on a cluster of 8-way Intel SMP systems, each system with 8 Intel Xeon 550MHz processors and 4GB memory.

Applications: μsik is currently being used in multiple projects, exercising its conservative and optimistic execution modes, as well as experimenting with a few newer mechanisms. It has successfully been used as the engine for a conservative parallel execution (on up to 128 processors) of discrete event models of 1-dimensional particle-in-cell physics[19]. Another application of μsik is in parallel simulation of the nervous system, presented in [20]. Yet another application of μsik is in optimistic parallel execution of a plasma simulation model of spacecraft charging in outer space. Reverse computation is used for rollback in this application, and a performance study is reported in [21]. Here, we focus on performance study using a synthetic benchmark, namely, the classical PDES benchmark known as Phold[22].

In our Phold implementation, *NLP* simulation processes are evenly mapped to all available processors. A fixed population of events, *NLP*R*, is generated at initialization, with random destinations. *R*, an integer, is the ratio of number of events to number of processes. When a process receives an event, it schedules a new event into the future to another random destination (possibly to itself) with a minimum time increment called lookahead. With probability *L*, the destination is on the same processor as the source. We use a uniform random number generator (URNG) to randomly determine event destinations, and another URNG stream to determine time increment. Since Phold is fine-grained, with very little computation performed per event, it represents a worst-case scenario that can expose runtime overheads of the simulation engine.

In the following, we evaluate μsik for its sequential execution performance, its parallel time-synchronization costs, its optimistic execution performance and mixed conservative-optimistic performance.

5.1. *Sequential Performance*

Figure 5 shows the average time taken to process an event in Phold, for increasing number of simulation processes and events. The time per event includes send/receive costs, process scheduling costs, as well as random number generation costs.

Context Switching Cost: The process scheduling costs are accentuated when the event population is low. For example, when *R*=1, each simulation process has a single event to process on average, and holds a high probability that its next send is not to self. This forces a "context switch" from one process to another for each and every event. In a context switch, the micro-kernel is required to update the time queues for the scheduled process as well as the destination process for the newly scheduled event. When *R*=10, each process has ten events to process on average, which implies processing an average of ten events between two context switches.

It is seen that our micro-kernel implementation scales excellently with the number of simulation processes, without drastic overheads for the maintenance of ECTS, EPTS and EETS values. In the largest sequential configuration on one processor, we are able to simulate an event population of 10 million events and 100,000 simulation processes, with less than 10 microseconds per event.

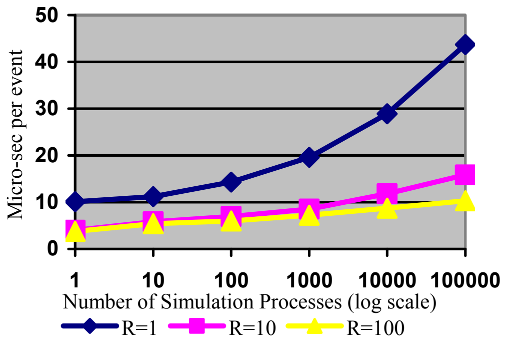


Figure 5: Performance of μsik on Phold, demonstrating scalability up to 10 million events and hundred thousand simulation processes on one processor.

5.2. *Parallel Time-Synchronization Cost*

Figure 6 shows the average processing time per event while the number of processors is varied. This experiment is intended to measure the cost of synchronizing simulation time across processors in isolation from remote event exchange costs. This is achieved (*L*=100%) by choosing random destinations only from among local processes (i.e., no event goes across processors). The entire distributed execution, however, is still time synchronized – LBTS computations are performed, and time advances of simulation processes are permitted only upon LBTS advances. For comparison, performance with a slight amount of inter-processor event communication is also plotted (*L*=99%).

While Figure 6 shows performance for relatively large simulation configurations (million LPs and events), Figure 7 is intended to show that significantly lower runtime costs can be observed when the number of context switches is low. This is done by using one million event population as in Figure 6, but using only 1000 LPs, resulting in a event/process ratio of R=1000, which in turn results in 1000-fold reduction in the average number of context switches.

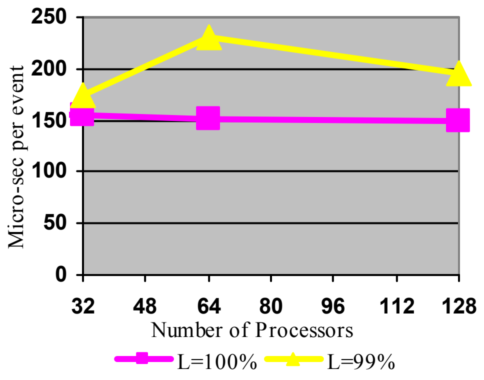


Figure 6: Conservative parallel performance of μsik on Phold with 1 million LPs and 1 million event population.

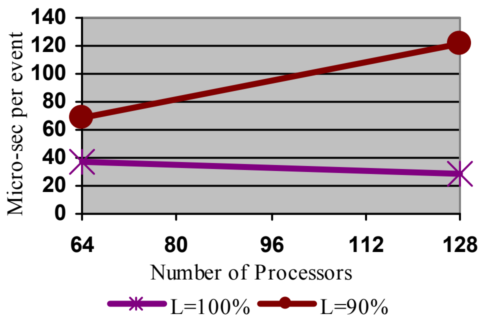


Figure 7: Conservative parallel performance of μsik on Phold with 1000 LPs and 1 million event population.

5.3. *Optimistic Simulation Performance*

In the optimistic configuration, each of the Phold processes executes its events optimistically ahead in time. Rest of the application is unmodified. In fact, the only source-code change between the conservative and optimistic executions is setting an optimistic execution flag in the simulation process.

Figure 8 shows the performance of optimistic parallel execution on a 1-million LPs & 1-million event configuration of Phold. This execution is intended to demonstrate the capability as a working proof-of-concept of our prototype. Further work is needed to reduce overheads in larger parallel executions, especially by incorporating flow-control mechanisms that can adaptively throttle over-optimistic execution. In particular, we are incorporating non-blocking TCP communication to send events across processors, to overcome potential deadlocks due to blocking sends.

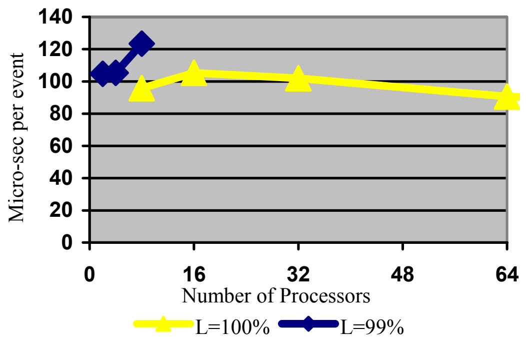


Figure 8: Optimistic parallel performance of µsik on Phold with 1 million LPs and 1 million event population.

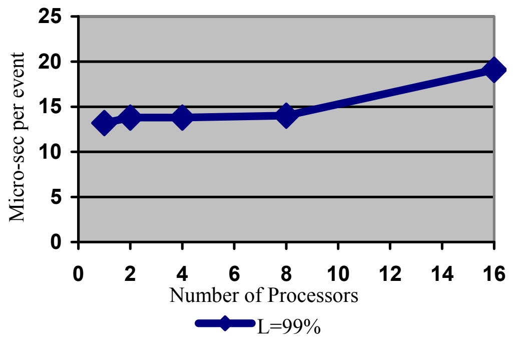


Figure 9: Optimistic parallel performance of µsik on Phold with 1000 LPs and 100,000 event population.

On smaller number of processors, the application is relatively well balanced, and optimistic mode delivers excellent performance. Figure 9 shows costs under 15 microseconds per event when operating entirely with shared memory communication (up to 8 processors), and under 20 microseconds per event even involving TCP communication across the 8-CPU SMPs nodes. It is worth noting that these experiments were executed on 550MHz CPUs, and can be expected to be significantly better on more recent platforms (e.g., 3GHz CPUs).

5.4. *Mixed Simulation Performance*

A simple change of the configuration yields an example in which every alternate process in Phold is conservative, and every other process is optimistic. This configuration is once again intended to serve as proof-of-concept demonstration of the micro-kernel approach that can accommodate both types of processes. Figure 10 shows the performance of such a mixed configuration executing in parallel on up to 128 processors.

5.5. *Memory Usage*

Figure 11 demonstrates that memory is recycled efficiently on large configurations. With active time-synchronized committing of events, events are committed (and hence freed) as early & aggressively as possible, in a scalable fashion.

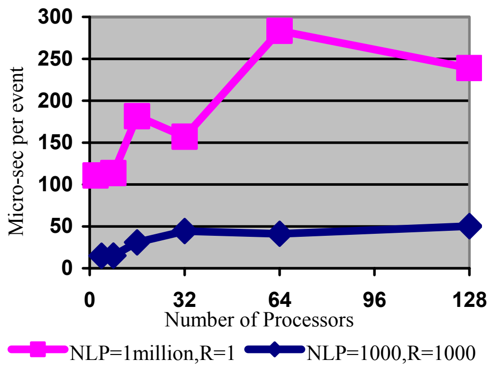


Figure 10: Parallel performance of µsik on Phold, with mixture of optimistic and conservative processes, one million event population, and localized communication.

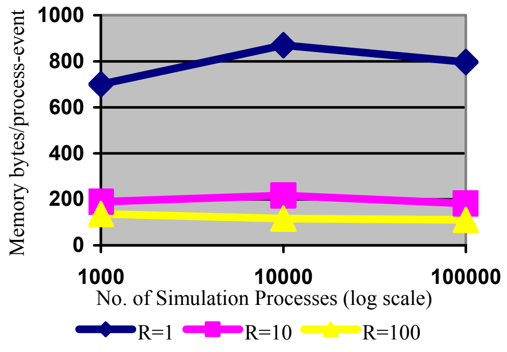


Figure 11: Memory usage of µsik in sequential execution of Phold.

Memory usage in bytes on the *y*-axis is computed as: $(M_{RP}-M_1)/(R*P+P)$. Here, M_{RP} is the memory usage for a Phold execution of *P* simulation processes and *R*P* event population. M_1 is the memory usage of 1 process and 1 event (fixed cost of the simulator). When *R*=1, processes and events are equally weighted in the average. When *R*=100, event memory usage dominates, showing that each event on the average consumes 110 bytes. This includes optimistic processing overheads, such as causal list pointers. The data shows that µsik's usage of memory per event is bounded in proportion to the actual number of events, and scales with the both number of events as well as the number of simulation processes.

6. Status and Future Work

µsik is a general-purpose parallel/distributed simulation kernel built upon a micro-kernel architecture consisting of autonomous simulation processes. Simulation processes hold and manage their own events, and can be optimistic or conservative in their event processing, or adopt other techniques such as aggregate event processing. The micro-kernel overhead is kept very low by design. µsik also uses the concept of kernel processes, which serve to push kernel-functionality to outside the micro-kernel.

The current implementation is portable across UNIX/Linux and Windows platforms. The micro-kernel source-code is compact (~4000 lines of C^{++} code). The µsik release includes the micro-kernel source code, example applications, and a user's manual. The most recent version can be obtained from www.cc.gatech.edu/fac/kalyan/musik.htm.

µsik currently supports: lookahead-based conservative execution; rollback-based optimistic execution with both state-saving and reverse computation; resilient computation (zero rollbacks) and any combination of them; per-process limits to optimism; user-level retractions; dynamic process addition/deletion; automated network-throttled flow control; shared-memory/distributed execution; and (conservative) process-oriented views based on POSIX threads). It is being successfully used in non-trivial applications with both conservative as well as optimistic modes.

Analogous to micro-kernel based operating systems, observed performance depends on process context switches. We are profiling the run-time performance to identify the most time-critical paths in execution, and envision further reduction in runtime overheads for process scheduling, and distributed synchronization on larger number of processors. We are also porting our system to supercomputing platforms for scalability to hundreds of processors. Further exercising generality and extensibility, we are exploring accommodation of Critical Channel Traversal[23] algorithms, Lookback[24], and Approximate Time[25] notions over the micro-kernel.

Acknowledgements

This work was supported in part by the National Science Foundation grant ATM-0326431.

References

[1] J. Liedtke, "On Micro-Kernel Construction," ACM Symposium on Operating Systems Principles, 1995.

[2] "IEEE Std. 1516: High Level Architecture," in *Institute of Electrical and Electronic Engineers*, 2000.

[3] V. Jha and R. Bagrodia, "A unified framework for conservative and optimistic distributed simulation," PADS, 1994.

[4] D. Lungeanu and C.-J. R. Shi, "Distributed simulation of VLSI systems via lookahead-free self-adaptive optimistic and conservative synchronization," IEEE/ACM International Conference on Computer-Aided Design, 1999.

[5] H. Rajaei, R. Ayani, et al., "The Local Time Warp Approach to Parallel Simulation," PADS, 1993.

[6] Metron, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation," 2004, www.speedes.com.

[7] S. R. Das, R. M. Fujimoto, et al., "GTW: A Time-Warp System for Shared Memory Multiprocessors," WSC, 1994.

[8] C. Carothers, D. Bauer, et al., "ROSS: A High-Performance, Low Memory, Modular Time Warp System," *Journal of Parallel and Distributed Computing*, vol. 62, 2002.

[9] G. D. Sharma, R. Radhakrishnan, et al., "Time Warp Simulation on CLUMPS," PADS, 1999.

[10] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transaction on Computer Systems*, vol. 3, 1985.

[11] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, vol. 18, 1993.

[12] K. S. Perumalla and R. M. Fujimoto, "Virtual Time Synchronization over Unreliable Network Transport," PADS, 2001.

[13] K. S. Perumalla, "libSynk Homepage," 2004, www.cc.gatech.edu/computing/pads/kalyan/libsynk.htm.

[14] K. S. Perumalla, A. Park, et al., "Scalable RTI-based Parallel Simulation of Networks," PADS, 2003.

[15] A. Park, R. M. Fujimoto, et al., "Conservative Synchronization of Large-scale Network Simulations," PADS, 2004.

[16] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, 1981.

[17] R. M. Fujimoto, K. S. Perumalla, et al., "Large-Scale Network Simulation -- How Big? How Fast?" MASCOTS, 2003.

[18] P. F. Reynolds, Jr., "A Spectrum of Options for Parallel Simulation," WSC, 1988.

[19] H. Karimabadi, Y. Omelchenko, et al., "A New Approach to Modeling Physical Systems: Discrete Event Simulations of Grid-based Models," Workshop on State-Of-The-Art in Scientific Computing (PARA), 2004.

[20] C. J. Lobb, Z. Chao, et al., "Parallel Event-Driven Neural Network Simulations Using the Hodgkin-Huxley Neuron Model," PADS, 2005.

[21] Y. Tang, K. S. Perumalla, et al., "Optimistic Parallel Discrete Event Simulations of Physical Systems using Reverse Computation," PADS, 2005.

[22] R. M. Fujimoto, "Performance of Time Warp Under Synthetic Workloads," SCS Multiconference on Distributed Simulation, 1990.

[23] Z. Xiao, B. Unger, et al., "Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation," PADS, 1999.

[24] G. Chen and B. Szymanski, K., "Four Types of Lookback," PADS, 2003.

[25] R. M. Fujimoto, "Exploiting Temporal Uncertainty in Parallel and Distributed Simulations," PADS, 1999.