Design and Performance Evaluation of a Conservative Parallel Discrete Event Core for GES

Silas De Munck
University of Antwerp
Middelheimlaan 1
Antwerp, Belgium
silas.demunck@ua.ac.be

Kurt Vanmechelen
University of Antwerp
Middelheimlaan 1
Antwerp, Belgium
kurt.vanmechelen@ua.ac.be

Jan Broeckhove
University of Antwerp
Middelheimlaan 1
Antwerp, Belgium
jan.broeckhove@ua.ac.be

ABSTRACT

The empirical study of large-scale distributed systems often calls for the use of computer simulations as real-world experimentation is too costly or simply infeasible. Computer simulations can also provide results on a much shorter timespan, increasing productivity. Nevertheless, large-scale system simulation can prove to be non-responsive on modern computers, especially when the modeled system has a high level of complexity or when detailed and compute intensive models are used. In order to fully harness the computational power of modern multi-core computer architectures, computer simulations need to execute in a parallel fashion.

In this paper we investigate the potential of parallelizing the execution of the Grid Economics Simulator (GES), a Java-based discrete-event simulator that is targeted towards the simulation of distributed systems in general, and economic forms of resource management in grids in particular. We present the design of a parallel continuation-based simulation core that uses a conservative time synchronization protocol. We analyze the performance of the parallel simulation core through synthetic benchmarks. The results of our performance evaluation give a clear insight in the impact of simulation model properties such as event arrival rates, computational workload, remoteness of events, and look-ahead size, on the speedup that can be achieved through parallel execution.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; I.6.8 [**Simulation And Modeling**]: Types of Simulation—*parallel, discrete event*

General Terms

Design, Experimentation, Performance

Keywords

Parallel discrete event simulation, performance analysis

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.

1. INTRODUCTION

Distributed and parallel processing techniques are common today in a wide range of applications. The increasing scale and complexity of distributed applications and systems necessitates research into more scalable and efficient algorithms and techniques for e.g. resource management and job scheduling. The evaluation of new algorithms on real testbeds is however impeded by their limited flexibility, controllability and availability. In addition, the costs of building and configuring large-scale testbeds are high. For this reason, researchers turn to simulation to evaluate new algorithms and techniques, especially during the initial phases of development. The Grid Economics Simulator (GES) [29] was developed for the evaluation of various economic approaches to resource management with regard to their ability to efficiently allocate and schedule tasks in a grid. The simulator consists of a time-stepped and a discrete event-driven simulation core. The discrete event core uses a process-oriented approach, and supports the simulation of interactions between various entities connected in a communication network.

In recent years, the rate of raw speed increase of individual processor units has been decelerating. Scaling up has become harder to do because of the physical limitations of the integrated circuits. First, the benefits of higher clock frequencies decrease as the memory speeds are not increasing as quickly as the logic speeds. Additionally, smaller and denser transistors, result in longer interconnection wires which introduce path delays that cancel the clock speed increase. Finally, the higher number of transistors packed together consume more power and produce more heat, causing cooling difficulties [17]. Because of these limitations, companies have searched for other techniques to create faster and more energy-efficient processors. These new developments have lead to the assembly of multi-core processors, which are essentially multiple processors joined together in a single processor socket and running in parallel. The change to a more parallel hardware design also requires a more parallel design of the software. However, this conceptual change to a parallel programming paradigm introduces new challenges and problems for software developers [27, 12, 20]. To fully benefit from the continued increase in computing power, application software must support a concurrent mode of execution on multiple processor cores.

In this paper, we describe the design of a parallel discrete event simulation core for GES. Our goal is to analyse to what extent an approach based on proven techniques and mechanisms for the implementation of parallel and distributed

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

simulators [7, 8, 9, 10], can result in satisfactory speedups on modern multi-core commodity hardware. In this regard, the presented results are a valuable reference point for more optimized implementations that are specifically tailored to multi-core and/or multi-processor hardware.

2. SIMULATOR DESIGN

2.1 Simulation Basics

This section gives a brief introduction concerning the terminology used in the field of simulation [7, 10] and throughout this paper. A simulation is a representation of a physical system evolving over time. This physical system or physical process is modeled by a *logical process* (LP). An LP consists of a number of virtual *entities* that are completing tasks or procedures, and that interact with each other by exchanging messages which are represented by *events* on the level of the LP. The state of these entities changes over time, consequently causing an evolution in the system. The process of these accumulated changes is driven by advancing the *virtual time* (VT) in the simulation. The modeling of the time progression is either continuous or discrete. Continuous time flow mechanisms represent the behavior of the system as a set of functions of time. In a discrete time simulation, state changes can only occur at certain discrete points in time. A discrete-event simulation advances simulation time to the execution time of the succeeding action, also referred to as an *event*. An event has an associated *firetime*, indicating the simulation time at which the event will occur. The execution of an event may create new events and the complete simulation finishes when all events have been processed. To summarize, a discrete-event simulation executes a sequence of events in time order and advances its time according to the event firetimes.

2.2 Single-core Design

A discrete event simulation core, which runs the LP, contains a control loop that continuously executes events performing operations, in firetime order, on the entities in the simulation. The main components of the discrete event core are the clock, keeping the virtual time value, and the event queue or event list (EVL), containing `Events` in the order of their firetime. The event queue is implemented using the Sun Java priority queue, which provides $O(log\, n)$ time insertions, linear time removal and constant time retrieval. The control loop of the logical process running in the `EventCore` pops the next event from the event queue for processing and then advances its time to the next event's firetime. The execution of an event may result in the creation of new events, which are again added to the event queue for later execution. The use of a priority queue combined with the condition that newly created events need to have a time stamp greater than or equal to the current virtual time, will ensure consistent execution of events in the right time order. The simulator core continues executing events until the event queue becomes empty. A schematic of this design is shown in Fig. 1

2.3 Entity Modeling & Communication

The discrete event simulator core of GES allows to declare certain objects as entities, representing real-world objects in the simulation [11]. Entities are able to communicate with other entities over a network link. Entities can be constructed in three ways: namely, by annotating the Java class

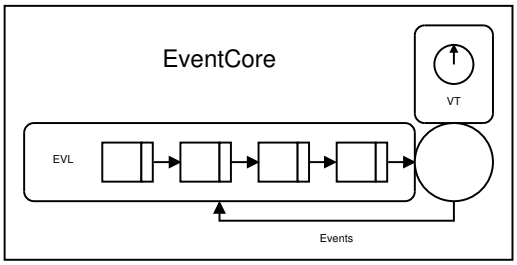

Figure 1: Single-core Design

with `@Entity`, by implementing the `EntityInterface` or by extending a `Process`. The first method creates only passive entities that can communicate with other entities but do not act on their own. A `Process` is an active thread-like entity that interacts with other entities on its own initiative. To the event core, a `Process` is essentially an `Event` whose execution can be suspended and resumed. The suspending of a `Process` causes the creation of a `ResumeEvent` that resumes the `Process` again at a later time, effectively simulating thread behavior. `Process`es are implemented using the JavaFlow library, that provides continuations [6]. Continuations provide an interesting alternative to threads for modeling concurrent behavior of entities in a simulation. They are lightweight structures that contain the stack contents and program counter. As such, they allow for the simulation of concurrent processes in a single system thread, and thereby give the programmer full control over the scheduling of simulated processes.

Methods in entities can be tagged with a `@ProcessMethod` annotation, causing the encapsulation of the method's execution in a `Process`, which is scheduled as an event and executed by the event core at a later time. This method encapsulation into a `Process` is performed by using AspectJ code weaving [16].

GES facilitates communication between simulation entities using an RPC-like mechanism. The simulator supports both synchronous and asynchronous remote method calls. A synchronous method call on another entity suspends the `Process`-context of the calling entity until the remote method returns, whereas an asynchronous method call is executed while continuing the calling `Process`. Remote entity methods are annotated with `@SynchronousNetworkMethod` or `@AsynchronousNetworkMethod`. The network calls are encapsulated into events, which are then rescheduled incorporating the appropriate network delay according to the network model used.

2.4 Multi-core Design

In order to parallelize the execution of the simulation, we install multiple event cores in the simulator that process events in parallel. The switch to a multi event core model requires several changes to the implementation of the simulator. First, the simulator must be able to support multiple event control loops in parallel, each in a separate thread and having its own *local virtual time* (LVT) and each running an LP. These different threads interact by adding events to each other's event queues. Second, these independent event cores need to synchronize their time, requiring a time management infrastructure. Both are the most important changes to the simulator to support execution on multi-cores.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

2.4.1 Simulation Core Driver

The bootstrap phase of the simulator consists of three steps. First, during the initialization of the `EventSystem`, multiple `EventCore`s are created. Then the simulation user code creates entities which are associated with an event core. Finally, the `EventCore`s enter the start-up phase of their control loop, effectively starting the simulation.

Multi-core execution increases the complexity of the `EventCore`, requiring a more advanced event core control loop. Events might be scheduled by another core, which introduces possible concurrent access to the event queue. Therefore, all access to the event queue needs to be protected by locking directives. Furthermore, determining when to start and stop the whole simulation becomes more complicated in the multi-core set-up. Instead of just executing events in a simple loop until the event queue is empty, the simulator core now consists of several program states:

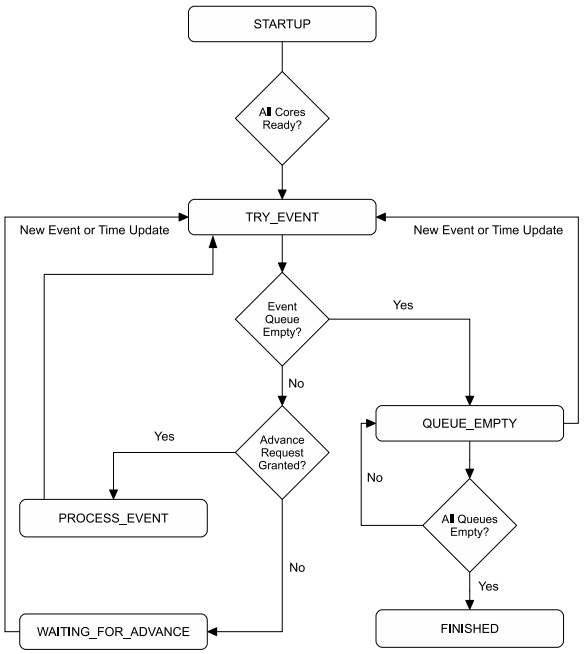

Figure 2: EventCore state transition diagram

STARTUP The start-up of the event core needs to be synchronized across all event cores. All cores have to enter this state before the simulation starts.

TRY_EVENT This core state is a dispatch state, as it will check certain conditions and cause a transition to another state. First, the event queue is checked for emptiness. If there are no events left in the queue, the core jumps to the EMPTY_QUEUE state. Considering there is at least one event in the queue, the core performs a time advance request for the next event's firetime to the core's time management service. The request returns the amount of time a core is allowed advance its time. If the next event's firetime is set before the allowed time, the core jumps to the PROCESS_EVENT state to process the event. When the firetime is after the allowed time, the event core switches to the WAITING_FOR_ADVANCE state and waits for a time update from the time management service.

PROCESS_EVENT In this state, the event core pops the next event from the event queue and then starts processing it. The execution of an event simulates state changes in the simulation model and possibly generates new events. On event completion, control is returned to the TRY_EVENT state.

QUEUE_EMPTY An empty event queue in a multi-core simulation does not necessary mean the simulation is finished. New events originating from other cores can still arrive in the queue. The simulation is only finished when all cores are in the QUEUE_EMPTY state and no events are running or in transit between cores. In that case, control transfers to the FINISHED state. If the simulation is not yet finished, the core waits using the Java `wait()` and `notify()`-mechanism. This blocks the event core thread until a new event arrives in its queue or a time update is received from another core.

WAITING_FOR_ADVANCE The core is waiting, until a request for time advance is granted or a new event has been scheduled in the event queue. The `EventCore` thread waits using a `wait()`-call and is blocked until a new event is scheduled or a time update from another core is received.

FINISHED The finished state ends the simulator core thread, hence finishing the simulation.

A state diagram of the control process running in each `EventCore` is depicted in Fig. 2.

2.4.2 Time Management Infrastructure

Generally, time stamp order execution in a simulation with a single logical process (LP) is ensured by the fact that an event that is being processed can only spawn new events with a firetime after its own firetime. Extending the simulation to multiple LP's must also ensure that all events, including events from other LP's, are processed in time order in each LP. If the LP's in all event cores comply with this condition, referred to as the *local causality constraint* [8], the results from a single core simulation are the same as those from a multi-core simulation. To realize this, the use of a synchronization mechanism or protocol is indispensable. There are two groups of protocols that ensure time stamp order execution: namely, conservative and optimistic time synchronization protocols [8]. An LP following the conservative synchronization protocol is allowed to only process an event if it is guaranteed that no events with a smaller firetime can arrive in the event queue. On the other hand, an optimistic synchronization protocol is less restrictive and tolerates the occurrence of causality errors, possibly violating the local causality constraint, but provides a roll-back mechanism to recover from these errors. The achievable parallelism in case of optimistic synchronization protocols is higher because all event cores run independently, rolling back their state if necessary. However, the roll-back mechanism requires the LP to keep previous simulation states which has implications

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

on performance and memory cost. In addition, for simulations with a high frequency of interactions between entities, roll-backs are common and performance gains from higher parallellism therefore limited. In this paper, we focus on the performance of a conservative time-synchronization protocol, leaving an analysis of the feasibility and performance of optimistic protocols for our simulation core to future work.

Look-ahead.

An event core running an LP processes both internal events, as well as external events originating from other cores, which may arrive in the event queue at any time. If logical process LP_a with LVT T_a sends an event to logical process LP_b with LVT T_b, then the firetime of this event cannot be smaller than T_b. Because the event is sent by LP_a and received in LP_b at the same virtual time, the local causality constraint requires that T_a must be equal to T_b, in which case only simultaneous events are executed in parallel. Therefore, the concept of *look-ahead* is crucial for conservative time synchronization protocols in order to introduce parallelism. Consider the example again where LP_b receives an event from LP_a, in other words, an entity living in LP_a sends a message to an entity residing in LP_b. As a physical system is being simulated, there is a delay inherent to the communication between these entities. In practice, if LP_b receives a message from LP_a, and the communication delay is d, the arrival time of this message in LP_b will be at least $T_a + d$. So that, LP_b is allowed to process events up to $T_a + d$, where in this case d is the look-ahead factor. In general, the minimum delay of a communication between entities can be used as the look-ahead l, effectively meaning that the lowest and the highest simulation time of a logical process in the simulation can differ at most by l time. Consequently, a logical process LP_p may process events with a firetime up to $T_p + l$. It is clear that the look-ahead value is intrinsically related to the details of the simulation model.

The behavior of the parallel LPs is shown in Fig. 3. Considering the look-ahead is equal to the minimum message delay d of a communication between entities. The event at t_1 in LP_b sends a message to LP_a, arriving at t_4 with a delay of d. LP_a can already process the event at t_2 when LP_b is still processing the event at t_1, because no event could arrive in LP_a from LP_b with a firetime smaller than t_4. Likewise LP_b can process the event at t_4 when LP_a is still processing t_2.

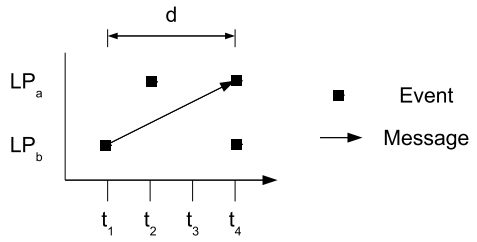

Figure 3: Look-ahead explained

Synchronization Protocol.

The synchronization protocol we applied is based on the conservative approaches often referred to as Chandy-Misra-Bryant (CMB) protocols [22]. These protocols associate each outgoing event e with a send time $T_s(e)$ and a firetime $T_f(e)$. A logical process (LP) contains an incoming message first-in-first-out (FIFO) queue for each other logical process. Events are sent with non-decreasing send time $T_s(e)$ to other LPs, which implies that the sequence of events arriving in each input queue will also have a nondecreasing order of sent timestamps $T_s(e)$. Each input queue i has a timestamp field $T(Q_i)$ associated with it, containing the time $T_s(e)$ of the queue-front or the last received message if the queue is empty. Then, the LP interleaves the execution of events from its own queue with those arrived in the incoming message queues, repeatedly processing events with the smallest timestamp. All pending events can be processed until time $min_i(T(Q_i)) + l$ is reached, where l is the look-ahead. The minimum $min_i(T(Q_i))$ is essentially a lower bound for the local virtual time (LVT) in all logical processes, also referred to as the *lower bound time stamp* (LBTS). However, if one of the input queues in an LP becomes empty, the LP must wait until new messages arrive in that queue. Events in other queues can be processed up to the time of the last processed event. This mechanism may result in a deadlock or memory overflow in the system. The problem is that a cycle of logical processes blocked by an empty queue can cause a deadlock situation, where the simulation cannot advance further, although there are several events that still have to be processed. To avoid this situation, an LP must receive updates on the LBTS of the simulation, so that it can ensure no events before a certain time can arrive. The solution to this issue is the concept of *null-messages*. These messages are empty events that carry only a $T_s(e)$ timestamp. The protocol sends null-messages after each event has been processed in an LP, guaranteeing that the LBTS of the simulation is updated accordingly. Other more optimized variants of this protocol exist, which for example incorporate a reduction of the amount of null messages by sending them more intelligently [22].

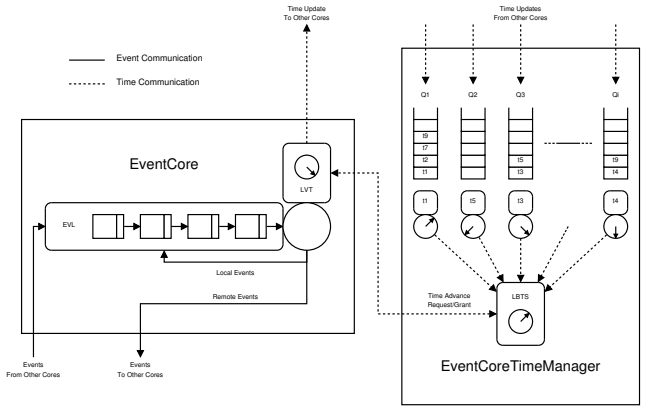

Figure 4: Multi-core Design

Implementation.

We have integrated the principles of this protocol in our event core to allow for parallel execution of events. Our single-threaded `EventCore` already contained a priority queue functioning as the event queue. We use this existing queue for both internal as well as external messages arriving from the other event cores. Note that this design potentially

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

raises a problem with lock contention on the queue, but measurements of the time an `EventCore` is waiting for a lock show that in most cases this is below 2% of the core runtime for 8 parallel `EventCore`s. The time management service is provided by the `EventCoreTimeManager` present in each `EventCore`. The time manager controls the time advances of the `EventCore` by keeping track of the LVT of the other event cores. In a traditional null-message protocol implementation, the event core has an event queue for each other core's external events. In our implementation, these separate queues are only used for time synchronization messages by the `EventCoreTimeManager`. The time manager has an input queue for time update messages for each other event core and an associated timestamp field for the last received time update, similar to the $T(Q_i)$ field described previously. When a `EventCore` advances the internal simulation time, the `EventCoreTimeManager` sends a time update message to the the other cores. This message arrives in the time manager's input queue for the sending core, and then the $T(Q_i)$ field is updated, based on the incoming time updates. After that, the LBTS estimation for the simulation is re-calculated in the `EventCoreTimeManager`. If the `EventCore` wants to advance its LVT, it performs a time advance request to the `EventCoreTimeManager`. The request is granted for time LBTS+l, with look-ahead l. Depending on the answer of the time management service, the `EventCore` advances its time or waits for new events. The described custom protocol acts as a distributed event core time synchronization protocol.

Fig 4 displays a graphical representation of this system. The `EventCoreTimeManager` on the right shows its input queues, containing time updates, and the $T(Q_i)$ field, containing the value of the last received time update, based on which it calculates the LBTS value. The `EventCore` depicted on the left, contains the LVT clock and the event queue. Event processing may spawn new internal or external events, that are rescheduled locally or sent to other cores respectively. The solid lines represent the flow of events, while the dashed lines represent the communication relevant to the time synchronization mechanism.

2.4.3 Data Distribution

Currently, the multi-core implementation of the GES simulator runs parallel cores using Java threads. These threads have a shared memory space, managed by the Java Virtual Machine (JVM). Consequently no real data distribution service is required, as all objects in memory are accessible to all event cores. The simulation entities and other objects can be created and used by any thread in the program. However it is still necessary to assign entities to a specific event core, to avoid concurrency problems when multiple cores execute events in parallel affecting the same entities and their datastructures.

Only entities with a communication delay in between them can be distributed among different event cores, as the minimum communication delay between entities provides the look-ahead necessary for achieving parallelism. A higher look-ahead value, results in a larger possible time difference between event cores and a higher degree of possible parallel execution of events, consequently improving the obtained speedup compared to a single-core execution. Entities that have a communication delay between them smaller than the look-ahead, or even zero, should be associated with the same event core. This may happen if a physical entity is modeled as a composition of several cooperating entities and processes.

The process of associating entities with an event core is managed by AspectJ code weaving. Entities that are created by another entity get associated with the same event core as their parent entity. Entities created elsewhere are currently assigned to an event core in a round robin fashion. Events also have an event core affiliation, which is established using the same parent-child relation, unless the event represents a network message and travels between entities in different event cores. If an entity A sends a message to another entity B in another event core, or entity A calls a network method on entity B, an event is created, and associated automatically with the event core of entity A. The AspectJ code actually performing the network call then re-associates the event with destination core B and schedules the event in the event core of entity B, where the event will be processed.

3. PERFORMANCE EVALUATION

This section presents a parallel performance evaluation of our multi-core discrete event system implementation. We investigate the impact of several event core and simulation scenario parameters on the efficiency and performance of the system using a synthetic test scenario. The results are obtained by averaging the data of 5 runs on a Linux cluster. Each cluster node consists of a dual socket motherboard containing two Intel Xeon 2 GHz quad core CPUs and 16 GB of memory. Cluster nodes all run a 2.6 Linux kernel. The tests were compiled and executed with Sun Java 1.6.0.16. In our experiments, the relative standard deviation of the measured runtimes used in the speedup calculations is in most cases below 1% and peaks at 3%. The relative wait time measurements have a standard deviation less than 1%.

In order to quantify the impact of different scenario parameters on the parallel performance of the event core, a synthetic test scenario was developed. This allows us to easily configure all the involved parameters. The test scenario consists of 100 entities of type `TestEntity`, that have a main loop with a tunable number of local and remote method calls. Local calls are `@AsynchronousNetworkMethod` calls to entities living in the same event core and generate events local to the core. Remote calls are `@AsynchronousNetworkMethod` calls to entities that reside in another event core and involve sending a remote event to this core. Both types of calls consist of two loops with a tunable number of iterations. One of these loops performs memory allocations and the other one is used to simulate computation time. The main loop of the `TestEntity` spawns events at a configurable rate using a Poisson distribution. We examine the impact of the event arrival rate, the remoteness of events, the look-ahead, the event duration and the amount of memory allocations on the achievable speedup, and on the amount of waiting time in the event cores. The results give an indication of the CPU usage efficiency and the overhead introduced by the multi-core discrete event simulation implementation under varying simulation model parameters.

3.1 Event Rate

The first test analyses the impact of the number of generated events per virtual time unit on the performance of the simulator. For this test we tune the local and remote

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

event rate of the Poisson process in our `TestEntity`. Generating events with a Poisson distribution makes sure events are spawned at random times but with an average rate over a longer time. This also ensures that the entities generate events distributed evenly over time, but not all at the same time.

We ran the test scenario ranging both local and remote event rate from 1 event per second to 1 event per millisecond (for each entity), while keeping the other simulation model parameters in the test fixed. Note that the `TestEntity` also runs a control loop that creates one resume event for each call made. This is the case because after each call (local or remote), the control loop `Process` suspends until the next time a method call is generated. There are 100 entities in the simulated scenario. The look-ahead is fixed to $100ms$, which corresponds to the network delay of the remote calls. We configure the events to have computational cost ranging from $0\mu s$ to $0.5ms$, generated by a uniform random number distribution.

Note that this computational cost, much like the event rate itself, is application-dependent. The lack of statistical data in this matter, compels us to resort to a distribution of computational times, something we would like to address in future work. In the simulation of market mechanisms for resource allocation in grids for example, the majority of the computational cost of the simulation is located in the bidding functions of the agents that participate in the mechanism, and in the functions that clear the market. Although the formulation of a bid can be done in a few microseconds, in the case of a simple English auction, we have determined that a more involved Gjerstadt-Dickhaut [13] bidding strategy in a continuous double auction setting, can easily lead to $50ms$ of computational load per bid that is placed. Similar considerations apply to the market clearing process. Whereas this cost is insignificant in case of an English auction, clearing a combinatorial auction can take minutes if not hours, depending on the size of the market and clearing algorithm used [26, 24].

Speedup.

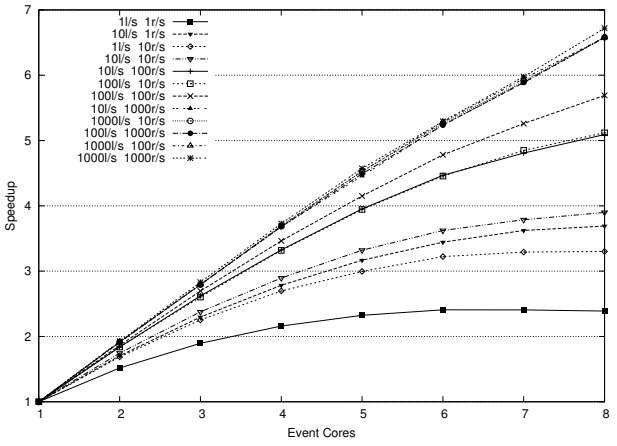

Figure 5: Speedup in relation to the local event rate (l/s) and the remote event rate (r/s)

Fig. 5 shows the test results indicating the impact of the event rate in the simulator on the speedup. At high event rates, the type of event, whether local or remote is not a determining factor for the speedup as there is only a small difference between curves where local events are dominant and curves where remote events prevail. The accumulated total event rate is the determining factor. With the 100 entities each spawning at least 100 events per second, or 1 event every 10 ms, the attained speedup is satisfying. Considering an event rate of 1 event every ms, or 1000 events per second, the speedup reaches almost 7 on the dual quad-core machine. At an even higher event rate (e.g. 2000 events per second) the speedup stays nearly the same. If the event rate is to low, e.g. 2 events per s, the maximum attainable speedup drops to 2.3. This is a consequence of the relatively higher time synchronization overhead in relation to the event process time at lower event rates. At lower event rates, we also note that the distribution of local versus remote events does have an impact on the speedup.

Event Core States.

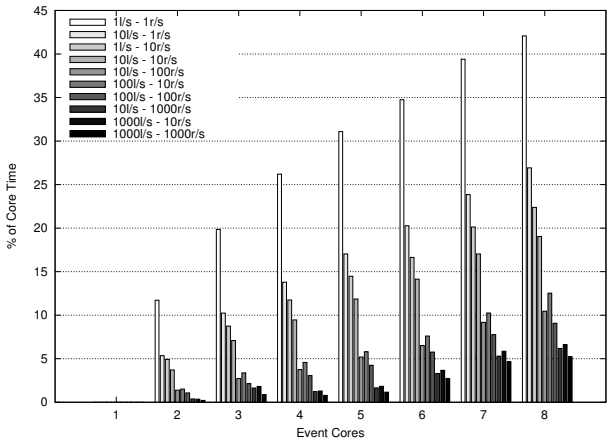

Figure 6: Event core waiting time in relation to the local event rate (l/s) and the remote event rate (r/s)

To analyse the event core state machine, we tested the amount of time the simulator core spends in each state. The tests revealed that the majority of time is spent in the PROCESS_EVENT and the WAITING_FOR_ADVANCE states of the event core, while the time spent in the other states is mostly constant and relatively insignificant across all of our tests. We therefore focus on the impact of the simulation model parameters on the percentage of execution time spent in the WAITING_FOR_ADVANCE state. Fig. 6 shows this percentage for varying local and remote event rates. If the rate of events is low (e.g. 2 events per s), the waiting time increases significantly with the number of event cores. In that case, the number of events in the system can't keep the CPU cores busy all the time, resulting in more waiting time for other cores, and a correspondingly higher time synchronization overhead. Higher event rates result in lower event core wait times. Event rates higher than 1000 events per second result in waiting times of about 5% for eight event cores, whereas the lower rate of 2 events per second already extends to 40% wait time. The increased wait time limits the useful CPU time and decreases the achievable speedup of the system. At event rates above or higher than 100 events per second, wait times are below 13% up to 8 event cores

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

3.2 Look-ahead

This section evaluates to what extent the value of the look-ahead influences the performance of the simulator. We test the look-ahead for values of $1ms$, $10ms$, $50ms$ and $100ms$, and keep the other parameters the same as in the previous test. We run this test also for different event rates, as this parameter changes the sensitivity of the system to different look-ahead values. In the network we are simulating, all entities have a fixed communication delay of $100ms$.

Speedup.

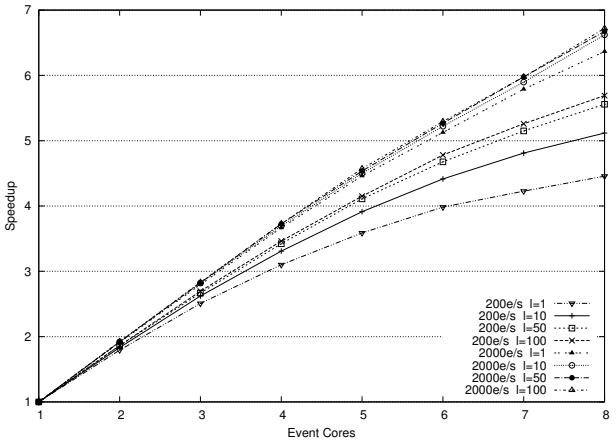

Figure 7: Speedup in relation tot the look-ahead for different event rates (e/s)

Fig. 7 displays the impact on the speedup of the look-ahead parameter for different event rates. The influence of the look-ahead is clearly visible in this synthetic test, where a lower look-ahead results in a reduced speedup. The graphs show that the effects of the changing value for the look-ahead decrease with higher event rates.

Event Core States.

A higher look-ahead results in a higher speedup, consequently the event core waiting time decreases with a higher look-ahead. This effect is illustrated in Fig. 8, where the event core wait time is shown for different look-ahead values and several event rates. As expected, the benefit of a higher look-ahead parameter is much higher when the event rates are lower. The look-ahead allows for higher virtual time differences between event cores, which is more important for achieving a high speedup at lower event rates.

3.3 Event Duration

In this section, we look at the impact of duration of events. To achieve maximum speedup, we have to make sure that all event cores are processing events all the time to limit the waiting time. By creating and processing events that have a high computational cost, taking a long time to process, the overhead resulting from the synchronization mechanism can be relatively reduced.

The following experiments evaluate these assumptions by running the simulation with various event durations. An event consists of a simple loop that is executed a variable number of times, resulting in a change in length of the event runtime. The loop is executed as a result of a local or remote

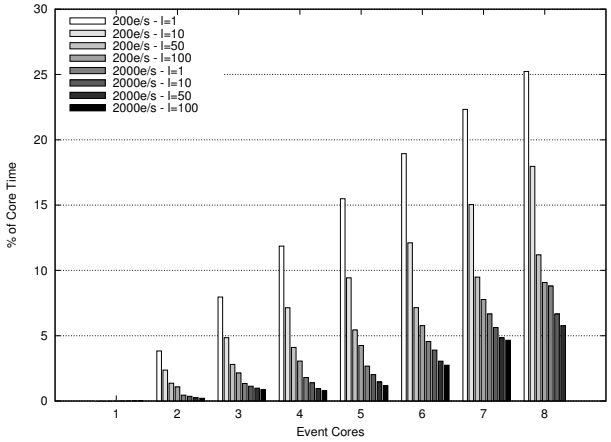

Figure 8: Event core waiting times in relation to the look-ahead for different event rates (e/s)

call on an entity, but there are still other events (e.g. the resume-events from the test driver `Process`, as described in Sect. 3). In this test, the event rate is set fixed at 200e/s, but the same effects are visible for other event rates. We test various upper limits for the uniform random distribution that determines the busy loop count, ranging from 0 to 750000 loop iterations, corresponding with a computation time ranging from 0 to $750\mu s$ on our test hardware.

Speedup.

As shown in Fig. 9, longer events result in a higher speedup, whereas the speedup drops significantly if the events are to short. The impact of the average event duration on the achieved speedup is most pronounced in the range of 10 μs to 500 μs.

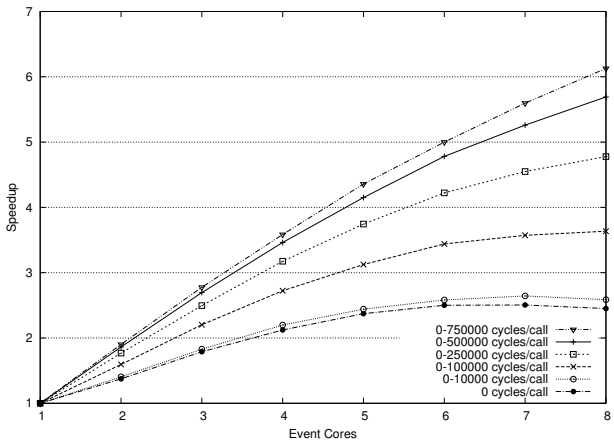

Figure 9: Speedup in relation to event duration

Event Core States.

The graph in Fig. 10 evaluates how the event core waiting times relate to the duration of an event. The overall impact of the event duration on the waiting times is rather limited, but shorter events result in a higher relative overhead of the time synchronization mechanism, the event core state ma-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

chine and more lock contention on the event queues, which explains the decreased speedup.

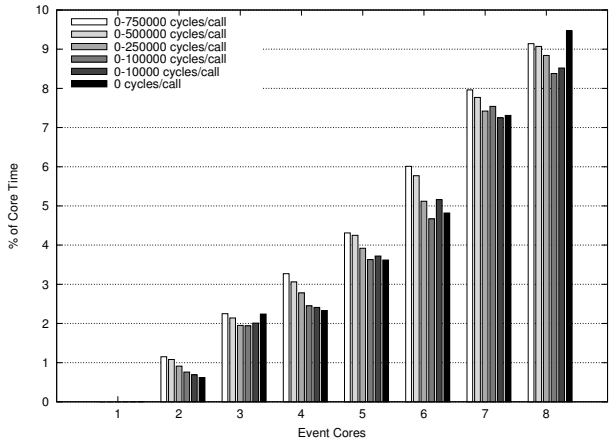

Figure 10: Event core waiting time in relation to the event duration

3.4 Memory Allocations

In this section, we analyse how the number of memory allocations performed during the processing of each event influences the parallel performance of the system. The loop that simulates memory allocation time during the processing of each event performs the allocation of a small string of 5 characters and an array of `double`s with size 10 for a configurable amount of iterations. The computational loop has a cycle count that is given by a uniform random number between 0 and 250000.

Speedup.

As shown in graph Fig. 11, the behaviour of the event core is different from previous tests. The graph illustrates that more allocations result in a lower achieved speedup, which is an opposite effect compared to the event duration test, where more computation time results in a higher speedup. The change in speedup caused by additional memory allocations becomes more important at higher event rates.

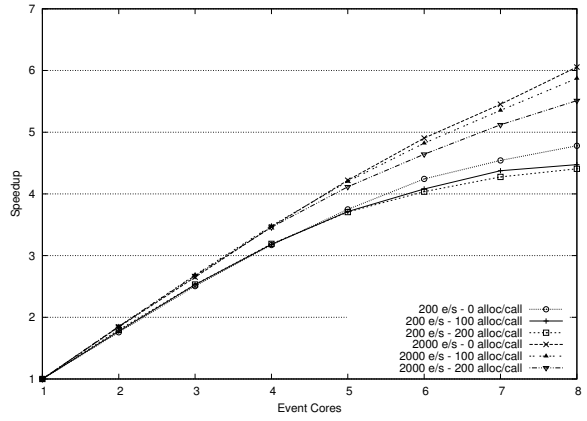

Figure 11: Speedup in relation to the number of allocations per event for different event rates (e/s)

Some additional experiments were run in order to investigate this behavior. We created a test code running a loop that calls a computationally intensive function and a function allocating an array of 10 `double`s. The loop runs a fixed number of times distributed across a variable number of threads from 1 to 8. We then compare the achieved speedup for different ratios of computation time versus memory allocation time. The graph in Fig. 12 shows that the amount of memory allocations in an application has a significant impact on the parallel performance. If 3% of the runtime is spent on memory allocations, a much lower speedup can already be observed. As the percentage of time spent on memory allocations reaches 100%, a speedup of two becomes unattainable irrespective of the number of cores used.

An identical experiment in C++ shows a different picture with significantly lower degradation of speedup under increasing intensity of memory allocation operations. The C++ results are shown in Fig. 13. Our tests have indicated that the JVM's garbage collector is not responsible for the speedup loss, as the garbage collection runtimes are negligible in relation to the total application runtime. However high allocation rates cause the garbage collector to add much more synchronization points in the JVM. The time lost reaching these so called synchronization *safepoints*, measured with Sun's `HotspotRuntimeMBean`, can reach up to 50% of the application runtime. The point at which these issues arise is dependant on the size of the young generation part of the JVM heap [21], which can be tweaked with the `NewSize` JVM parameter.

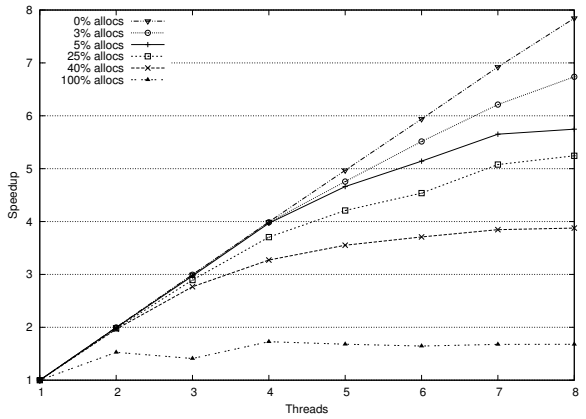

Figure 12: Impact of Java memory allocation on speedup (for different allocation time % per run, on a dual quad-core CPU)

Event Core States.

The graph shown in Fig. 14 depicts the time spent in the wait state of the event core for different allocation amounts and event rates. The event rate is a more important factor that determines the wait time. The number of allocations has little effect on the waiting time. In case of a higher allocation rate, the speedup loss is caused by the increased time spent in the JVM.

4. RELATED WORK

Initial research on discrete event simulation dates back

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

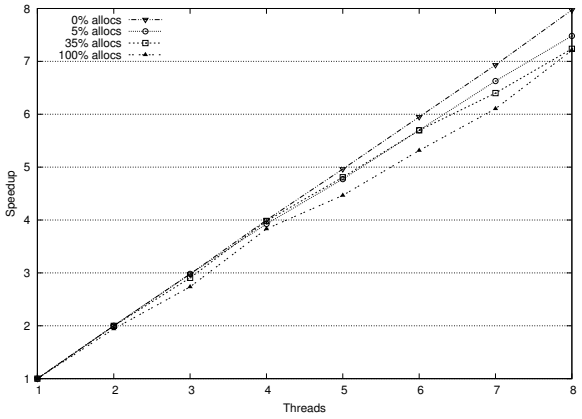

Figure 13: Impact of C++ memory allocation on speedup (for different allocation time % per run, on a dual quad-core CPU)

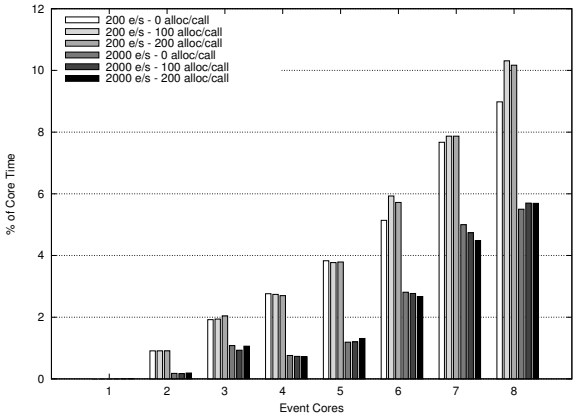

Figure 14: Event core waiting time in relation to number of allocations per event for different event rates (e/s)

to the 1950s. Early work on parallel and distributed simulation was driven by the development of the null-message protocol, also referred to as the CMB protocols by Chandy and Misra in 1978 [4], and independently Bryant in 1977 [3]. In the years thereafter, many additions and improvements to the original conservative protocol were designed. Some optimizations to reduce the number of null-messages by sending them more intelligently (e.g. on demand or delayed until some timeout occurs) were described in [22]. Another extension is the carrier null-message protocol [25], handling the synchronization more effectively by propagating more information in the null-messages. In contrast to the previous deadlock avoidance null-message algorithm, the deadlock detection and recovery algorithm [5] allows deadlocks to occur but providing a mechanism to detect and recover from them. Other more global oriented time synchronization mechanisms, where the basic null-message synchronization is decentralized, also became available (e.g. the bounded lag algorithm [19]). Simulations using a conservative time synchronization protocol have been evaluated using synthetic application benchmarks simulating queueing networks [2, 10, 30], communication networks [1] or electronic circuits [23, 1].

The publication of the Time Warp protocol by Jefferson in 1985 [15], introducing an optimistic synchronization protocol was another milestone in the history of parallel and distributed simulation. This protocol introduces a roll-back mechanism providing proper synchronization across event cores, by reverting to a previous state if a causality error occurs. Numerous extensions and improvements have been developed since then. For example, lazy cancellation reduces the number of roll-backs by only cancelling events that do not have the same outcome and lazy re-evaluation tries to keep as much state in formation as possible to allow a quick roll-back, at the cost of a higher memory usage. Furthermore, additional techniques to efficiently keep as much previous state information as possible with a minimal amount of memory (e.g. Cancelback [14], Artificial Roll-back [18], etc.) were developed. Parallel discrete event simulation performance evaluations using an optimistic synchronization protocol have been conducted using synthetic benchmarks simulating communication networks [1, 28] or electronic circuits [1].

Most of the parallel and distributed simulation basics and the conservative and optimistic optimization techniques are well described in [7] and [10].

Performance evaluation studies for both conservative and optimistic time synchronization have been limited to an evaluation of parameters specific to the simulation model, while our performance evaluation analyses the impact of more general discrete event simulation parameters such as event arrival rate, look-ahead, event duration and the remoteness of events.

5. CONCLUSION

In order to harness the power of multi-core architectures, simulators can be developed to execute discrete-event simulations in a parallel fashion. We have described the application of a conservative parallel discrete-event core design to the Grid Economics Simulator in this regard. Our performance evaluation shows to what extent the implementation of a conservative time synchronization protocol in our Java-based event core can result in a speedup of the simulation on a multi-core architecture with a dual quad-core setup. The results of our synthetic tests quantify the impact of various simulation model parameters on the attained speedup. As such, our results can be used to assess to what extent a particular simulation can benefit from parallel execution under a conservative time synchronization protocol.

We show that high parallel performance can be attained with the proposed parallel core, provided that a number of simulation model parameters reach adequate levels. The key parameters in this regard are the lookahead used, the average duration of events and the event arrival rates.

6. REFERENCES

[1] R. Bagrodia, R. Meyer, M. Takai, Y. an Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, 1998.

[2] R. L. Bagrodia and M. Takai. Performance evaluation of conservative algorithms in parallel simulation languages. *IEEE Trans. Parallel Distrib. Syst.*, 11(4):395–411, 2000.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637

[3] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.

[4] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, 1979.

[5] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[6] T. Curdt, K. Kawaguchi, and M. Cooper. Apache Commons Javaflow. `http://commons.apache.org/sandbox/javaflow`, 2008. [Accessed 03-10-2008].

[7] A. Ferscha. Parallel and distributed simulation of discrete event systems, 1995. Contributed to the: Handbook of Parallel and Distributed Computing, McGraw-Hill, 1995.

[8] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30 – 53, October 1990.

[9] R. M. Fujimoto. Parallel and distributed discrete event simulation: Algorithms and applications. In G. W. Evans, M. Mollaghasemi, E. C. Ruseel, and W. E. Biles, editors, *Proceedings of the 1993 Winter Simulation Conference*, pages 106 – 114, 1993.

[10] R. M. Fujimoto. *Parallel And Distributed Simulation Systems*. Wiley, 2000.

[11] J. M. Garrido. *Object-Oriented Discrete-Event Simulation with Java: A Practical Introduction*. Kluwer, 2001.

[12] P. Gepner and M. F. Kowalik. Multi-core processors: New way to achieve high system performance. In *PARELEC*, pages 9–13, 2006.

[13] S. Gjerstad and J. Dickhaut. Price formation in double auctions. *Games and Economic Behavior*, 22(1):1–29, January 1998.

[14] D. Jefferson. Virtual time ii: storage management in conservative and optimistic systems. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 75–89, New York, NY, USA, 1990. ACM.

[15] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. AkÅ§it and S. Matsuoka, editors, *Proceedings of ECOOP 1997*, volume 1241 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 1997. Springer-Verlag.

[17] G. Koch. Discovering multi-core: extending the benefits of moore's law. 2005.

[18] Y.-B. Lin and B. R. Preiss. Optimal memory management for time warp parallel simulation. *ACM Trans. Model. Comput. Simul.*, 1(4):283–307, 1991.

[19] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM*, 32(1):111–123, 1989.

[20] W. mei Hwu, K. Keutzer, and T. G. Mattson. The concurrency challenge. *IEEE Design & Test of Computers*, 25(4):312–320, 2008.

[21] S. Microsystems. Memory management in the java hotspot virtual machine, April 2006.

[22] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[23] N. O. Processes, E. Naroska, and U. Schwiegelshohn. Conservative parallel simulation of a large number of processes, 1999.

[24] T. Sandholm, S. Suri, A. Gilpin, C. Inc, and D. Levine. Cabob: A fast optimal algorithm for combinatorial auctions. *Management Science*, 51(3):374–390, March 2005.

[25] W. C. School, W. Cai, and S. J. Turner. An algorithm for reducing null-messages of cmb approach in parallel discrete event simulation. Technical report, 1995.

[26] J. Stößer and D. Neumann. GreedEx – A scalable clearing mechanism for utility computing. In *Proceedings of the Networking and Electronic Commerce Research Conference (NAEC) 2007*, 2007.

[27] H. Sutter and J. Larus. Software and the concurrency revolution:. *QUEUE*, September:54 – 61, 2005.

[28] Y.-M. Teo and S.-C. Tay. Performance evaluation of a parallel simulation environment. In *SS '99: Proceedings of the Thirty-Second Annual Simulation Symposium*, page 86, Washington, DC, USA, 1999. IEEE Computer Society.

[29] K. Vanmechelen, W. Depoorter, and J. Broeckhove. A simulation framework for studying economic resource management in grids. In *Proceedings of the International Conference on Computational Science (ICCS 2008)*, volume 5101, pages 226–235. Springer-Verlag, Berlin Heidelberg, 2008.

[30] E. Weingärtner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *ICC 2009: IEEE International Conference on Communications*, 2009.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8637
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8637