Distribution of Parallel Discrete-Event Simulations in GES : Core Design and Optimizations

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

Computer simulations have become an indispensable tool for the empirical study of large-scale systems. The timely simulation of these systems however, is not without its challenges. Simulators have to be able to harness the full computational power of modern architectures through parallel execution and overcome the memory limitations of a single computer. In this paper we investigate techniques for distributed and parallel execution of the Grid Economics Simulator. We present the design of a parallel and distributed simulation core that uses a conservative time synchronization protocol and describe the optimizations we performed to improve the performance of the simulator. We analyze the performance of the distributed simulation setup through two different application scenarios. Our results demonstrate how the presented techniques contribute to attain significant speedups on a distributed system consisting of multi-core machines and commodity networking hardware.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Performance

Keywords

parallel, distributed, discrete event simulation, performance analysis, scalability, optimization

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) [19] 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 entities connected in a communication network.

We previously designed and analyzed to what extent a parallel discrete event simulation core for GES, based on proven techniques and mechanisms for the implementation of parallel and distributed simulators, can result in satisfactory performance on modern multi-core commodity hardware [7]. This was motivated by the current evolution towards multi-core processors and consequently the need for a concurrent mode of execution of a discrete event simulation to fully benefit from the continued increase in computing power.

More complex and larger scale systems, require simulations with a large number of simulated entities and a high level of detail (e.g. cloud computing platforms or peer-to-peer networks). Consequently, scalability in terms of memory is an additional requirement for current simulation frameworks. At the same time, the bundling of computing power in a networked environment using grid computing, cloud computing or high performance cluster computing, together with the appearing limitations of single machine execution, drives the demand for application execution in a distributed environment.

In this paper, we describe how we extended the existing parallel discrete event simulation core in GES [7] to support distributed execution and we present a number of optimizations that were necessary to reach a satisfactory level of performance.

1.1 Simulation Basics

This section gives a brief introduction concerning the terminology throughout this paper [8, 10]. 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 in-

SIMUTOOLS 2011, March 21-25, Barcelona, Spain

DOI 10.4108/icst.simutools.2011.245500

teract 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.

1.2 Simulator 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). The control loop in the event simulation core 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 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, ensures consistent execution of events in the right time order.

In order to parallelize the execution of the simulation, the simulator runs multiple event cores in the simulator 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 exchanging events. To ensure correct execution across event cores running in parallel, these independent event cores synchronize their time, using a 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 that postdates 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* [9], 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. The current implementation follows the conservative time synchronization protocol, allowing an event core to process an event only if it is guaranteed that no events with a smaller firetime can arrive in the event queue. The choice of a conservative time synchronization protocol implies the exploitation of *look-ahead* [7, 9] in the simulation model to introduce parallelism in the execution of the simulation. The synchronization protocol we applied is based on the conservative approach often referred to as the Chandy-Misra-Bryant (CMB) protocol [15]. This protocol introduces the concept of a *lower bound time stamp* (LBTS) as the minimum timestamp an individual event core can safely advance to. Additionally, *null-messages* are broadcasted by an event core to inform the other cores of its current LVT, to ensure correct LBTS calculation avoiding dead-lock situations [4]. For a full explanation of potential problems, design choices or details concerning our parallel simulator implementation, we refer to [7].

1.3 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 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 suspension 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 [12, 13].

GES facilitates communication between simulation entities using an object oriented RPC-style mechanism, akin to Java RMI. 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 the calling `Process` continues execution. 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.

A graphical representation of this calling mechanism, for local single-core and parallel multi-core execution, is shown in the top part of Fig. 1. In the picture, an entity A calls an `@AsynchronousNetworkMethod` on entity B, passing a reference to itself as an argument. In case both entities are associated with the same event core (e.g. Core 1), the call triggers the creation of an event encapsulating the real function call on B. This event is then inserted into the EVL of the same event core (Core 1), with a firetime that incorporates the network delay between the two entities. At the event's firetime, the event and consequently the encapsulated call on

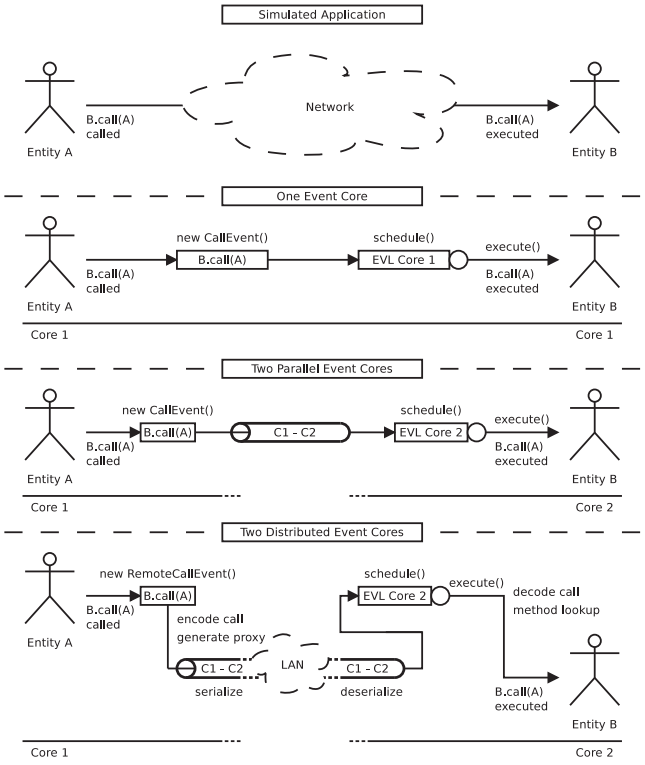

Figure 1: Simulated Entity Method Calling Mechanism

B is executed. Alternatively, if the entities are associated with two different local event cores, running on the same machine, a slightly different procedure occurs. The call is also encapsulated in an event, but instead of being directly inserted into the EVL, the event is sent through a channel to another (local) event core and subsequently inserted into the EVL of the other event core.

2. DISTRIBUTED EXECUTION

Distributed execution of the event core requires a number of changes compared to the previously described multi-core implementation. The shift from a single-host, multi-threaded shared-memory execution to a multi-host, multi-threaded distributed memory approach requires changes to the event core communication infrastructure, the entity distribution system and the simulation start-up and coordination subsystem.

2.1 Communication Channels

For a distributed simulator setup, separate event cores running on different machines need to exchange events and time synchronization information. Therefore the event communication module was refactored to allow different types of *communication channels* between event cores.

The communication channel represents a one way communication link that the event core uses to send and receive events from other cores. Currently, there is an implementation for these communication channels to connect local event cores for the multi-core setting, using a local blocking queue, and one to connect event cores on different hosts using Jini/RMI [16] technology, currently developed within the

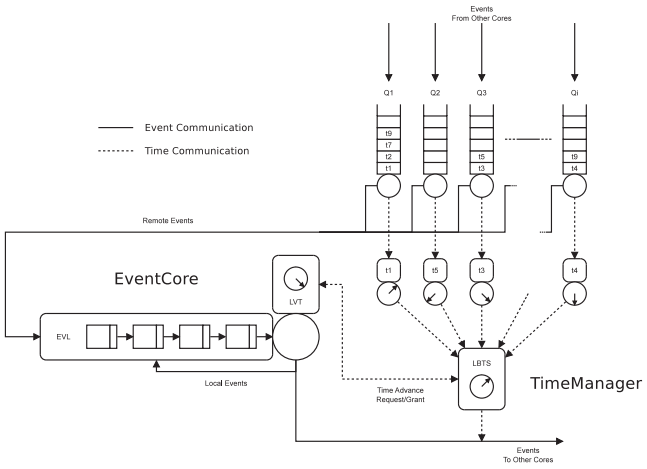

Figure 2: Event Core Design

Apache River project [2]. The re-design of the communication subsystem also alters the way event cores implement the time synchronization compared to the approach described in [7]. Instead of sending out-of-band *time updates* previously, we now use a new type of events, called `TimeEvents` that are sent in-band through the same channels used for sending the other simulation events. The time synchronization protocol, as described in [7], did not change.

An overview of an event core that operates according to the proposed design is depicted in Fig. 2. Events with timestamps t_i received from other cores appear in the input queues Q_i. Based on the timestamps of the last received event for each queue, the `TimeManager` re-calculates LBTS (Sect. 1.2). The event core requests to advance its LVT and the request is granted or not based on the LBTS. When the event core is not allowed to advance the LVT and if there are no more events to process in the EVL, new events are pulled from the input queues.

2.2 Coordination

The distributed execution of a simulation requires a coordination service to manage the simulator cores running on different machines. For this purpose, we developed a Jini service, called the `DistributionController`. The main tasks of this service are bootstrapping the event cores that run on separate hosts and managing exported and therefore remotely accessible simulation entities.

The simulation bootstrap process consists of multiple stages, that are synchronized across all event cores. After launch, all event cores use Jini lookup services to find the controller and each other. When this discovery process is completed, the event cores register themselves with the controller. A notification from the controller triggers the cores to enter the next start-up stage, namely the communication channel set-up between cores. If the communication links are ready, the real simulation is started and remote entities can be registered and looked up using the the `EntityManager` in the `DistributionController`.

The controller is also responsible for the monitoring and collection of runtime statistics. An event core collects more than 40 runtime parameters (e.g. event duration, core idle time, communication overhead, ...). These statistics can be sent to the controller during the run or at the end of the

simulation, where they are processed and written to disk, to a database or to the console for further analysis. This information is useful for optimization and debugging of the simulator code as well as application code. Finally, the distribution takes care of synchronizing the core shutdown procedure when the simulation ends successfully or organizes a graceful shutdown if an error occurred in one of the event cores.

2.3 Entity Distribution & Communication

Simulation entities are created and bound to a specific event core when they are created. Currently it is not possible to migrate entities from one event core to another.

In the simulation it is essential that entities in different cores can communicate using the same RPC-style (Sect. 1.3) call mechanism. Therefore, calls to local and remote entity methods annotated with `@SynchronousNetworkMethod` and `@AsynchronousNetworkMethod` must be handled transparently to the simulated application, whether or not the simulation is run in a distributed fashion. Consequently an *entity proxy* is required, that represents an entity residing in a remote core. As before, the calls to annotated methods on entities are wrapped into events and sent in serialized form through a communication channel to the remote event core, where they are executed at their pre-set firetime. If the simulated call is a synchronous network call, a return value is sent back to the originating core, where the calling process is then resumed, incorporating the correct simulated network delays. The described calling mechanism combined with the use of these entity proxies enables transparent redirection of calls to a remote entity residing on another host.

Note that due to technical limitations concerning AspectJ code weaving, it is not possible to transport standard call-events, that use AspectJ *around* advice, over the network using standard object serialization. To solve this, we create remote call-events that include a string-encoded representation of the called method signature, additionally including serialized arguments. On the remote side, the signature is decoded and the corresponding method called using Java reflection on the real entity. The calling mechanism in a distributed setting is depicted in the lower part of Fig. 1.

Entities can be made available remotely to other cores in two ways. First, they can be explicitly exported and registered in the `DistributionController`, so that they can be found using various search criteria, e.g. a name or an entity group name. If an entity is exported, an entity proxy is automatically created and stored in the entity manager in the distribution controller, allowing it to be found by other cores. Alternatively, entities can be passed as an argument or return value in a method call to a remote entity. The call redirection mechanism automatically replaces the entity that leaves the local environment by its proxy. This procedure allows the application to design its own entity lookup and discovery functionality instead of using the available entity manager in the `DistributionController`.

3. SCALABILITY & EFFICIENCY IMPROVEMENTS

In this section we discuss and evaluate the performance improvements effected to increase the efficiency and scalability of our simulator for distributed execution. The impact of these improvements will be demonstrated using two scenarios. The first is a simple closed queuing network (CQN) [3, 10, 20], the second simulates an electronic auction for compute resources.

The closed queuing network consists of 64 *queue*s in each event core ($Q1$ to $Q64$), with 16 *servers* per queue. Each queues is connected to a *switch* ($S1$ to $S64$) that randomly decides, using a uniform distribution, which queue will be the next one a packet is sent to. These queue destinations are chosen from all queues in the system, including those that reside in other event cores. The simulated network interconnects between the entities (queues, servers and switches) have a constant delay. The CQN simulation runs for a fixed amount of simulated time. In this scenario, we will scale the queuing network with the number of cores used. For example, a simulation with 3 event cores will have 3 switches connected to 3×64 queue's. The queuing network structure across cores is depicted in Fig. 3.

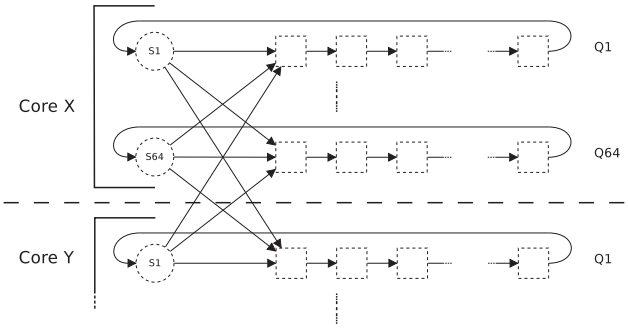

Figure 3: Closed Queueing Network

To examine the behavior with a totally different application scenario, we implemented a simulation of a distributed electronic auctioning system. In this environment, consumers bid for computational resources that are offered by providers through English auctions. Providers, implemented as an `AuctionProvider` entity, launch an `Auctioneer`, managing the auction for the provider's resources. Consumers are represented as an `AuctionConsumer` entity, running an `AuctionBiddingAgent` for each auction they join. If an `Auctioneer` starts an auction, it is advertised on the `AuctionMarket`, notifying the `AuctionConsumer`s which launch an `AuctionBiddingAgent` for the new auction if the consumer has more jobs to run and has budget left to place bids. The `AuctionBiddingAgent`s of multiple consumers then compete for the resource by bidding, given a starting price and limited by the available budget. The `AuctionConsumer` starts one `AuctionBiddingAgent` per auction and at most one per job left to run. In this test we simulate 2 markets with 125 consumers and 10 providers in each event core. Each consumer has 10 jobs to run and joins auctions until all his jobs are finished. Furthermore, we can configure an additional number of consumers joining a market in an external event core, different from their own. The scenario scales the number of entities with the number of active cores and in each core 20% of the consumers connect to an external market in another core.

Important in this scenario is that we set the constant delay in the simulated network to $25\,ms$ and the job runtime to $60\,seconds$. This creates a scenario that is less than ideal to execute in a distributed fashion. This is caused by the relatively low look-ahead in this scenario and relatively large

periods of inactivity in the simulation, the event cores are inevitably waiting for each other to advance most of the time. As this is an auction scenario where the consumers have a fixed amount of work to be done, the level of activity constantly decreases during the run of the simulation when more and more consumers finish their jobs.

These scenarios were tested on a heterogeneous cluster of 12 machines with 5 Dual Quad-Core AMD Opteron 2350 processors ($2\,Ghz$) and 7 Dual Quad-Core Intel Xeon L5335 processors ($2\,Ghz$) with at least $16\,GB$ of memory. All these machines are connected in a gigabit ethernet network. Cluster nodes all run a 2.6 Linux kernel. The tests were compiled and executed with Sun Java 1.6.0.16 Server VM. In the next subsections, we present and evaluate several optimizations to enhance the performance in distributed execution mode. For each of them, we selected the best performing configuration to show the isolated effect on performance of each individual optimization. In our experiments, the relative standard deviation of the measured runtimes is in most cases below 2%, with some rare peaks at 5%.

3.1 Null-Message Reduction

As established in our previous performance evaluations [7] and mentioned elsewhere [15, 17], the standard null-message protocol for conservative time synchronization sends a large amount of redundant null-messages. This is the case because an event core broadcasts null-messages to each other core when it advances the LVT to avoid deadlock situations. In a local multi-core execution this is not a big issue because only references to events are exchanged and the transport of events between cores is a relatively cheap operation. In a distributed setting however, this becomes an issue as the overhead created by these messages is considerably higher. To transport events to a remote event core, they have to be serialized, send over over the network and go through a deserialization process at the remote end. We present three techniques to send null-messages more intelligently resulting in a significant reduction in the number of broadcasted null-messages.

The first null-message reduction technique reduces the number of null-messages by only sending them after a certain timer was expired. Null-messages are only needed when the normal process of event communication can't provide enough time update notifications between cores. For this reason, we introduce a timer for each outgoing communication channel that is being reset every time an event is transported through the channel. Null-messages are sent through the channel only when this timer has expired, effectively reducing the number of null-messages significantly while still providing frequent time update notifications to remote event cores. This technique is often referred to as delayed null-message sending. For now, we choose the timeout value to be $20\,ms$, resulting in acceptable performance, but further analysis is required to determine the optimal value for a specific scenario.

As a second method to reduce the null-messaging overhead, we added *on-demand* null-message sending [15]. With this technique, an event core will broadcast a *request* to send null-messages to the other event cores when the core is about to enter an idle state, waiting for events from other cores. More details about these core states can be found in [7]. If an event core receives such a null-message send request, it will immediately respond by sending a null-message to the requesting core. This causes a recalculation of the LBTS in the requesting core, allowing it to advance its LVT.

Another reduction technique consists of broadcasting null-messages, instead of requests, to all connected cores when the event core is about to enter the idle state. This effectively avoids deadlock situations where the simulation cannot advance without null-messages and the event cores are blocked in their idle state, waiting for incoming remote events.

Fig. 4 to 7, Tbl. 1 and 2 display the runtime measurements in both test scenarios in a parallel as well as a distributed setting for different combinations of these three reduction methods. Due to time limitations, the auction scenario was only tested up to 10 cores and stopped at $10.000\,seconds$. Also note that in this test we disabled *null-message filtering* (Sect. 3.2) to show the effective impact of these reduction methods more clearly.

From the parallel CQN scenario runtime graph, as shown in Fig. 4 and Fig. 5, it is clear that the use of a timeout based null-message reduction algorithm brings a runtime performance improvement up to 10% when running with 8 cores compared to the standard CMB-based null-message algorithm. The blocking and request based null-message reduction techniques are reducing the amount of synchronization messages to much, increasing the amount of idle time, which results in a significantly longer runtime. The timeout based reduction method, reduces the number of messages while still sending enough of them so that the idle time does not increase.

The distributed execution of this scenario, as depicted in Fig. 6 and Fig. 7, indicates that the large amount of null-messages in the standard protocol has more a negative effect on runtime compared to the parallel version because of the higher communication costs. Here, the experiments with a timeout based reduction method also result in the shortest runtime. The use of a combination of reduction methods does not produce better performance results for this application.

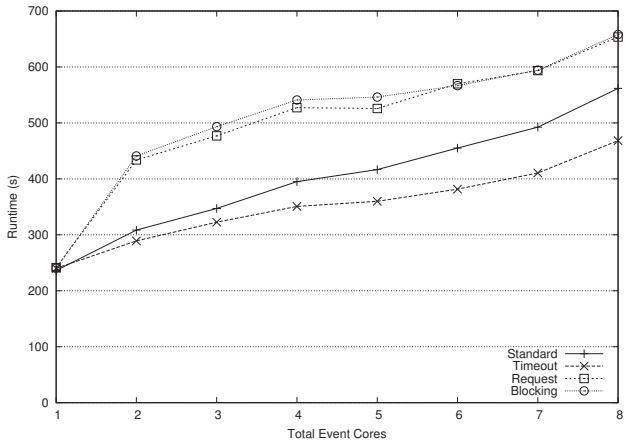

Figure 4: Parallel execution with various null-message reduction techniques for the CQN scenario

The runtimes for the auctioning scenario reveal a different outcome as shown in Tbl 1 and Tbl. 2. A combination of the timeout and blocking techniques yields the best overall runtime performance in the parallel execution mode, where

Cores	Standard	Timeout	Request	Blocking	Tim.+Blo.	Blo.+Req.	Tim.+Req.	Tim.+Blo.+Req.
1	159 s	158 s	157 s	157 s	159 s	158 s	159 s	159 s
2	514 s	> 10.000 s	301 s	280 s	268 s	279 s	289 s	291 s
4	1.106 s	> 10.000 s	2.940 s	848 s	471 s	3.027 s	2.973 s	3.104 s
8	2.244 s	> 10.000 s	5.014 s	763 s	771 s	4.919 s	4.810 s	4.568 s

Table 1: Runtime for parallel execution with various null-message reduction methods for the auctioning scenario

Cores	Standard	Timeout	Request	Blocking	Tim.+Blo.	Blo.+Req.	Tim.+Req.	Tim.+Blo.+Req.
2	2.897 s	> 10.000 s	419 s	348 s	376 s	430 s	416 s	413 s
4	4.466 s	> 10.000 s	2.265 s	540 s	546 s	2.210 s	2.278 s	2.359 s
8	8.061 s	> 10.000 s	5.826 s	1.209 s	1.260 s	5.814 s	5.923 s	6.236 s
10	9.788 s	> 10.000 s	7.299 s	1.549 s	1.585 s	7.166 s	7.525 s	7.469 s

Table 2: Runtime for distributed execution with various null-message reduction methods for the auctioning scenario

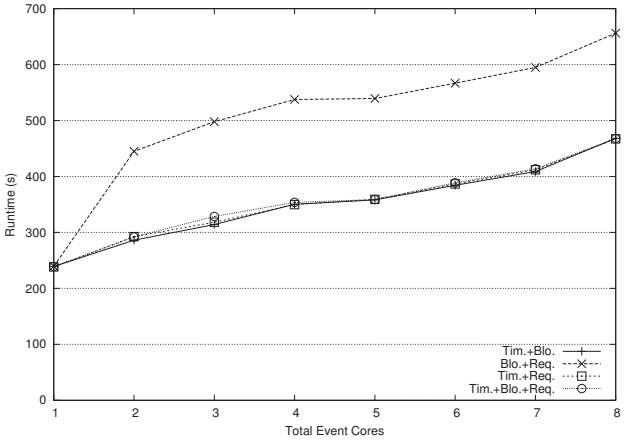

Figure 5: Parallel execution with combined null-message reduction techniques for the CQN scenario

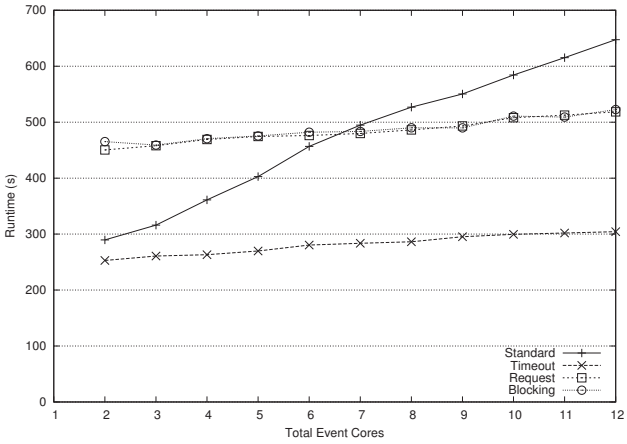

Figure 6: Distributed execution with various null-message reduction techniques for the CQN scenario

simulation finishes in at least 45% less time compared to the standard null-message protocol for this scenario.

For the distributed execution, the blocking reduction technique produces the best runtime performance with an improvement over the standard protocol of at least 85%. The use of only the timeout algorithm does not perform well in this scenario. The requirement to enable the blocking send method in order to achieve good performance is caused by the nature of this auction scenario which has a high level of activity at the beginning of the simulation run, but this activity gradually decreases towards the end where the activity appears in short bursts with longer periods of inactivity in between them. In periods of high activity the simulation is able to advance without null-messages, while in the periods with reduced activity, the necessary broadcast of null-messages is triggered when the cores are entering the idle state by the blocking reduction algorithm. If only the timeout based reduction is used, the LVT moves very slowly in these periods of low activity. In this situation, a lower timeout value would have been appropriate. The null-messages triggered by the blocking algorithm however, counteract the badly configured timeout. In the future, we will conduct experiments with dynamic timeouts to address these issues.

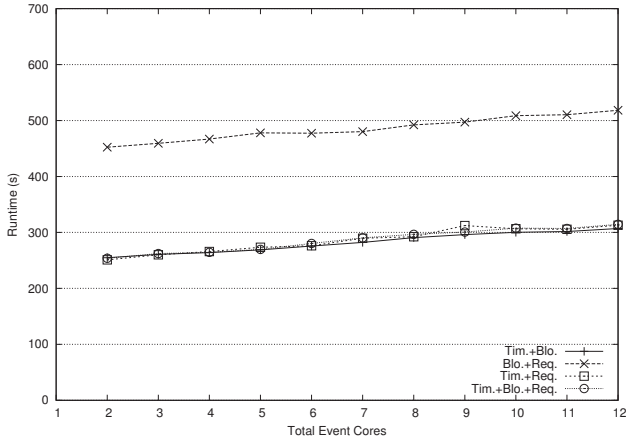

Figure 7: Distributed execution with combined null-message reduction techniques for the CQN scenario

Although the combined methods result in shorter runtimes compared to the standard protocol, the attained results are very similar to the simple reduction methods. Nevertheless, these combination of reduction methods might be useful in other applications with more variability in the ac-

tivity.

3.2 Null-Message Filtering

A technique called null-message filtering is applied to additionally reduce the number of null-messages required for synchronization. The algorithm tries to filter redundant null-messages that are still waiting in the send-queue of a core's communication channel. A null-message is redundant and can be removed if it is directly followed in the channel by another null-message with a higher firetime.

Cores	without filter	with filter	% improvement
2	376 s	354 s	6 %
4	546 s	495 s	9 %
8	1.260 s	829 s	34 %
10	1.585 s	983 s	38 %

Table 3: Runtime for distributed execution with and without redundant null-message filter in the auctioning scenario

If the other null-message reduction techniques discussed in this paper are used, only a limited number of redundant messages occur however. In both application scenarios, less than 1% of the remaining null-messages is actually filtered. In the CQN scenario this does not yield a measurable performance improvement. Due to the properties of the auction scenario (e.g. a large amount of idle time), the extra filtering does show a runtime improvement of up to 38% for a 10 core simulation as shown in Tbl. 3. The amount of runtime improvement increases with the number of cores used for the simulation execution.

3.3 Asynchronous Communication

The communication between event cores uses Jini/RMI behind the scenes. A communication channel has two end-points of which the receiving end has a `Remote` interface, including a `transport` function with one event as an argument (`void transport(EventInterface event)`). The receiving or destination end-point is exported as a `Remote` object that is discoverable using the Jini lookup and discovery services [18]. The source end-point in the other event core performs a lookup for the required channels and receives a remote reference. This establishes the connection and allows the transport function to be called at a later time to transfer events between the two cores. This procedure is repeated for all communication channels. In the initial implementation, calls to this transport function at the remote end were synchronous, because the Jini and RMI mechanism does not support an asynchronous calling mechanism. In order to improve the simulation performance, we implemented an asynchronous calling mechanism that uses a blocking queue and an additional thread that empties the events from the queue and sends them to the remote core. Note that redundant null-message filtering is not possible in synchronous communication mode because then there is no outgoing message queue (Sect. 3.2).

Fig. 8 and Tbl. 4 show the difference between the two methods. Asynchronous sending results in a runtime improvement from 47% to 67% for the auction scenario and from 13% up to 33% in the CQN scenario.

3.4 Entity Proxies

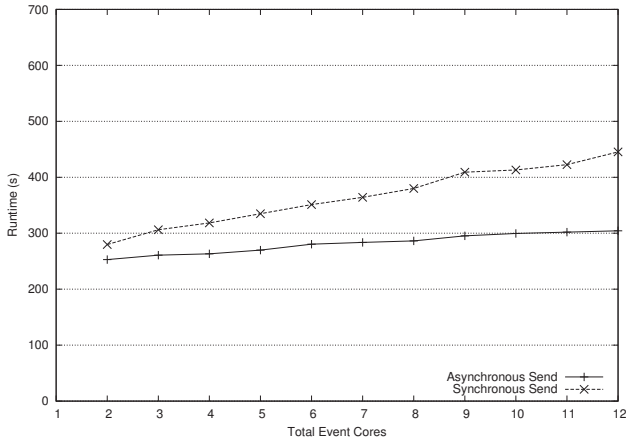

Figure 8: Runtime for distributed execution with synchronous vs. asynchronous send method in the CQN scenario

Cores	Synchronous	Asynchronous	% improvement
2	667 s	354 s	47 %
4	1.340 s	495 s	63 %
8	2.430 s	829 s	65 %
10	2.954 s	983 s	67 %

Table 4: Runtime for distributed execution with synchronous vs. asynchronous send method in the auctioning scenario

The entity call mechanism wraps method calls to network methods (`@SynchronousNetworkMethod` or `@AsynchronousNetworkMethod`) into call-events that are inserted in the EVL and executed in the event core at a later time (Sect. 1.3). This works well and is efficient when all event cores are running in one Java virtual machine, but problems arise when a method is called on an entity in a remote event core. As explained before, an entity is represented by a proxy in a remote core. In the distributed execution mode, the same call interception mechanism is used, but results in a remote call-event with the remote function signature encoded in the event, including serialized arguments or entity proxies. This event is then sent over to the remote event core, which will then execute this encoded call using a Java reflection-based mechanism, at the time the event is executed.

Initially, the proxy call mechanism was built using standard Java dynamic proxies (`java.lang.reflect.Proxy`) [14]. A problem arises with this approach when a proxy appears in the core where the real referred entity resides. This can for example occur when the entity manager lookup mechanism is used to retrieve an entity reference. In that case, a call on the proxy should be redirected and locally executed on the real entity for optimal efficiency. Using the standard Java dynamic proxies, this can be realized by installing a custom `InvocationHandler` that redirects all calls on the object to a handler method that invokes the method on the real entity. As a reflection-based calling mechanism incurs an additional runtime cost, we use bytecode generation to improve performance. The proxies are created using the Java Programming Assistant (Javassist)

bytecode manipulation framework [1]. In general, an improved function call performance of more than a factor of 10 can be attained compared to a reflection-based invocation using the Javassist bytecode generated proxies [5]. When required, a new proxy class is generated that implements the same interface as the real entity. The implementation redirects all method calls to the real entity if the proxy resides in the same core as the real entity. Additionally, these new proxies are more efficient to transport over the network, as they are about 30% smaller than regular reflection based proxies.

In the queuing network simulation, the queues, switches and servers are connected at the beginning of the simulation. There is no communication of remote entity references during the simulation run, which is exactly where this optimization has its effect. In the auction scenario, proxies do get transported and used for call-backs during the simulation. In this scenario, the use of the optimized proxy type yields a small runtime improvement due to reduced communication overhead. The effects of the improved proxy call invocation performance is less noticeable due to limited ratio of calls through local proxies. The results for the auction scenario are presented in Tbl. 5 and show an improvement of up to 5%.

Cores	Java proxy	Javassist proxy	% improvement
2	365 *s*	354 *s*	3 %
4	502 *s*	495 *s*	1 %
8	869 *s*	829 *s*	5 %
10	1.004 *s*	983 *s*	2 %

Table 5: Runtime for distributed execution using Javassist generated proxies and standard Java dynamic proxies in the auctioning scenario

3.5 Method Cache

A call between entities in two different cores on separate hosts triggers the creation of an event with an encoded method signature that will be executed in the event core of the destination entity. The decoding of these signatures requires the search for a compatible method using reflection, incorporating method overloading and argument type covariance. To avoid these costly lookups we added a method cache, mapping the encoded signature to the right method when the method is called for the first time.

Cores	without cache	with cache	% improvement
2	377 *s*	354 *s*	6 %
4	551 *s*	495 *s*	10 %
8	920 *s*	829 *s*	10 %
10	1.074 *s*	983 *s*	8 %

Table 6: Runtime for distributed execution with and without remote method lookup cache for the auctioning scenario

For the CQN scenario, there is no difference in runtime whether the method lookup cache is enabled or not. The reason for this is that the queuing network classes only have a few methods, making the reflection based method lookup's as fast as the cached lookups. On the other hand, the results for the auction scenario show a reduction in runtime of up to 10%, as illustrated by Tbl. 6. The effects of the cache decrease with a higher number of cores in the scenario because the relatively higher idle time of the event cores.

4. CONCLUSION

In order to address the challenges inherent to the simulation of large-scale systems, simulators can be developed to execute discrete-event simulations in a parallel and distributed fashion. We have described the changes required to the conservative parallel discrete-event core of the Grid Economics Simulator to support distributed execution. Several performance optimizations were presented and evaluated using two different applications: namely, a closed queuing network and an electronic auction for resources. In both scenarios, asynchronous communication and null-message reduction improves performance, although the optimal reduction method differs between the scenarios. The implementation of asynchronous communication also has a significant positive effect on performance. In the auction scenario, an even shorter runtime can be attained by enabling redundant null-message filtering, and a remote method invocation lookup cache. The use of bytecode generated proxies also results in performance gains. In the queuing network scenario, the gains for these additional optimizations are negligible.

The experiments illustrate that through the addition of several optimizations, distributed execution of a simulation becomes feasible in a wider range of applications than before. However, further research will have to be conducted on how to dynamically determine the optimal selection of optimizations to attain the best performance for a specific application.

5. REFERENCES

[1] Javassist project, 2006. Java Programming Assistant, JBOSS Community.

[2] Apache river project, 2008. The Apache Software Foundation.

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

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

[5] Chiba, S., and Nishizawa, M. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering* (New York, NY, USA, 2003), GPCE '03, Springer-Verlag New York, Inc., pp. 364–376.

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

[7] De Munck, S., Vanmechelen, K., and Broeckhove, J. Design and performance evaluation of a conservative parallel discrete event core for ges. In *SIMUTools '10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques* (ICST, Brussels, Belgium, Belgium, 2010), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–10.

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

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

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

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

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

[13] Laddad, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[14] Microsystems, S. Dynamic proxy classes.

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

[16] Newmarch, J. *Foundations of Jini 2 Programming*. Apress Apress, Berkeley, California, 2006.

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

[18] SUN. Jini technology core platform specification, version 2.0., June 2003. Jini Specification by Sun Microsystems, Inc..

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

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