Revisiting Conservative Time Synchronization Protocols in Parallel and Distributed Simulation

S. De Munck*, K. Vanmechelen, J. Broeckhove

Silas De Munck, University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium

SUMMARY

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 multi-core architectures through parallel execution and overcome the memory limitations of a single computer. In this paper we evaluate the performance of a parallel and distributed simulator using several conventional time synchronization protocols executed on modern multi-core hardware. In addition, we comprehensively analyse a hybrid approach, combining two traditional protocols, increasing robustness and enabling improved performance in a wider range of simulation scenarios. Finally, an adaptive algorithm to automatically configure this hybrid protocol is introduced and evaluated, eliminating manual user intervention and further improving robustness with respect to varying simulation conditions.

Copyright © 2012 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: 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) [1] 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 discrete event-driven simulation core, that uses a process-oriented approach, and supports the simulation of interactions between entities connected in a communication network.

We previously 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 [2]. 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.

*Correspondence to: Silas De Munck, University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium

Copyright © 2012 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls [Version: 2010/05/13 v3.00]
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 cloud computing or cluster computing, together with the appearing limitations of single machine execution, drives the demand for simulation in a distributed environment.

Although the importance of efficient parallel and distributed simulation software is ever increasing, the number of recent performance evaluation studies concerning multi-core commodity hardware is limited. In addition, studies often omit details about the specific synchronization protocol used or only compare to the standard protocol as a baseline [3, 4]. The choice of a suboptimal synchronization protocol may have a significant negative impact on simulator performance. Additionally, the effect of this choice depends heavily on the simulation code and the execution environment. Therefore, synchronization protocol optimization is essential to achieve satisfiable speedup in a parallel and distributed simulation. Our contributions in this regard are:

- a detailed performance analysis of conventional conservative time protocol optimizations on modern platforms
- a thorough analysis of a hybrid protocol that combines two traditional protocols and results in enhanced robustness and performance gains in a wider range of simulation scenarios,
- a self-tuning version of this hybrid protocol that further improves robustness with respect to varying simulation conditions, eliminating the need for manual configuration.
- an evaluation of the attainable parallel speedup in a purely parallel, distributed and hybrid event core deployment.

The text is structured as follows: The remainder of this section introduces the key notions of parallel and distributed simulation (Sect. 1.1) and how this was implemented in our simulator (Sect. 1.2). We end this section with a number of core optimizations to support distributed execution (Sect. 1.3). Section 2 presents and evaluates, using two application scenarios, a number of traditional synchronization protocol optimizations. The section’s empirical results lead us to present and evaluate a hybrid time synchronization protocol, and its adaptive variant. Thereafter, the attainable speedup on a multi-core cluster is evaluated in Section 3. This is done for two application scenarios executing an a purely parallel, distributed or hybrid simulation core configuration. Finally, sections 4 and 5 discuss related work and summarize our findings.

1.1. Simulation Basics

This section gives a brief introduction on the terminology used throughout this paper. A more elaborate coverage can be found in [5, 6]. 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 that 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. 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.

In order to gain performance and increase model scale, parallelism is introduced in the simulation execution and the model is distributed across multiple machines. There are several options to achieve this in a discrete event simulation.

A first approach is to execute events that have the same firetime in multiple threads of execution in parallel [7]. This technique only requires one controlling LP whereas these execution threads and the associated simulation entities can be distributed across more than one physical machine. The reachable level of parallelism heavily depends on the availability of these simultaneous events.
Accuracy may be sacrificed to improve parallel performance by reducing the virtual clock resolution, thereby increasing the number of simultaneous events [11].

In time parallel simulation, the simulation timeline is partitioned instead of the state space. Separate intervals of the simulation timeline are executed in parallel. The main issue with this approach is to match the boundaries of these separate intervals to re-assemble the whole simulation timeline, possibly requiring corrective computations afterwards [6]. If this technique is applicable, which depends heavily on simulation model properties, massive parallelism can be achieved [8].

Another method to introduce parallelism relies on the execution of multiple synchronized simulation cores executing concurrently, 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 LPs must also ensure that all events, including events from other LPs, are processed in time order in each LP. If the LPs 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. There are two main groups of protocols that ensure time stamp order execution: namely, conservative and optimistic time synchronization protocols [9, 10]. 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 may be 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 on performance and memory cost.

A conservative time synchronization protocol implies the exploitation of look-ahead [2, 9] in the simulation model to introduce parallelism in the execution of the simulation. Note that optimistic time synchronization also requires some level of look-ahead to be available in the model to guarantee progress in a deterministic simulation execution [11]. The conservative protocol however, requires the look-ahead to be explicitly known instead of only being inherently available in the model. In this paper, we focus on the performance of conservative time-synchronization protocols, leaving an analysis of the feasibility and performance of optimistic protocols for our simulation core for future work.

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.

1.2.1. Time Management Infrastructure

We apply synchronization protocols that are based on the conservative approaches often known as Chandy-Misra-Bryant (CMB) protocols [12]. 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 non-decreasing order of sent timestamps $T_s(e)$. Each input queue $Q_i$ has a timestamp field $T(Q_i)$ associated with it, containing the time $T_s(e)$ of the queue-front or of 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 \(\min_i(T(Q_i))\) is reached. This 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 previously processed event that originates from the empty queue. 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 each time an LP advances its LVT, 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 \cite{12,16}. In Sect. 2 we compare several of these reduction algorithms to the standard protocol.

### 1.2.2. Implementation

Our simulator implements an LP as an event core, containing a priority queue as an EVL. Fig. 1 displays a graphical representation of this system.

![Figure 1. Event Core Design](image)

**Figure 1. Event Core Design**

A time manager controls the time advances of the event core by keeping track of the LVT of the other event cores. The communication system uses a channel abstraction, in accordance to the input/output FIFO queue model described in Sect. 1.2.1. All event cores have incoming and outgoing channel endpoints to each other event core. The time manager hooks into the processing of the input and output channel endpoints in an event core and has an associated timestamp field for the last received event time for each channel, which is the \(T(Q_i)\) field described previously. It also keeps track of the last outgoing timestamp \(T_f(e)\) of an event for each outgoing queue. The firetime \(T_f(e)\) is calculated in the destination core, where the event arrives, based on the current cores’ time and the network delay determined by the network model. Therefore an event object only contains the firetime, which we will refer to as \(t_i\). Based on the \(T(Q_i)\) field in each channel, the time manager performs an LBTS calculation.
The event core requests the time manager to advance its LVT. The request is granted if the requested
time is smaller than or equal to 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 all the input queues and
inserted into the EVL. This process changes the LBTS, allowing the event core to advance further
and execute the pending events in the EVL. If an event core is not allowed to advance and there are
no more pending incoming events, the event core enters a wait state until new events arrive at the
input channel endpoints.

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 [29] that are able to
communicate with other entities over a network link. Alternatively, a process is supported as an active
thread-like entity that interacts with other entities on its own initiative. Processes are implemented
using continuations from the JavaFlow library [30].

Entity Call-Mechanism

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. The network calls are encapsulated into events, which are
then rescheduled incorporating the appropriate network delay according to the network model used.
This method encapsulation into a executable simulator events is performed transparently for the
simulator using AspectJ code weaving [31, 32].

In a distributed context, simulation entities are bound to a specific event core when they are
created. It is essential that entities in different event cores can communicate using the same RPC-style
mechanism. Therefore, calls to local and remote entity methods must be handled transparently
with respect to the simulation code, 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.

Entities can be made available remotely to other cores in two ways. They can be explicitly exported
and registered in a centralized entity registry, 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, 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.

Coordination

The distributed execution of a simulation uses a centralized controller that
coordinates the simulator cores running on different machines. 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 intensity, etc.), which we use as a
data source for our experiments.

Simulation Core Communication

Separate event cores possibly running on different machines
need to exchange events and time synchronization information. Therefore different types of
communication channels exist to establish this interaction between event cores. Currently, there is
an implementation for these communication channels to connect local event cores for the parallel
multi-core setting, using a local blocking queue, and one to connect event cores on different hosts
using Jini/RMI [33, 34] technology.

1.3. Simulator Core Optimizations

In order to achieve acceptable performance in a distributed simulation setting, apart from optimizing
the time synchronization protocols, a number of low-level simulator core optimizations are necessary.
In this section, we briefly motivate and describe the optimizations that were introduced and evaluated
in [35]. They have been included in the implementations used in this work, whenever possible.
1.3.1. Remote 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, stored for subsequent lookups.

Runtime improvements up to 10\% were measured [35]. The advantage may be even higher, depending on the application scenario, more specifically the complexity of the entity classes and the volume of remote entity calls within the scenario.

1.3.2. Asynchronous Communication The communication between event cores uses Jini/RMI behind the scenes [36]. Because Jini/RMI only supports a synchronous calls, 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, in order to improve channel throughput. This technique yields a significant runtime improvement compared to the synchronous mechanism. We measured a decrease in runtime of up to 65\% in a specific 10 core simulation. The gains, although in most cases significant, are proportional to the amount of inter-core communication [35].

1.3.3. Null-Message Filtering We apply null-message filtering [23, 16] to reduce the number of null-messages. The algorithm filters redundant null-messages that are 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 message, a null-message or a regular event, with a higher or identical firetime. Note that redundant null-message filtering is not possible for the parallel multi-core configuration because then there is no outgoing message queue as events are inserted directly in the destination input queue of the receiving event core.

The use of this technique may result in no performance gain at all if the synchronization protocol already performs optimally. The more efficient the synchronization protocol, the fewer redundant null-messages will be filtered. Nevertheless, we measured performance gains of up to 40\% using this method [35].

1.3.4. Entity Proxy Optimization Initially, the proxy call mechanism was built using standard Java dynamic proxies [37]. A problem arises with this approach when a proxy appears in the core where the real referred entity resides. 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 invocation handler that redirects all calls on the object correctly. As the required 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 [38]. 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 [39]. These new proxies are also more efficient to transport over the network, as proxy instances are about 30\% smaller than regular reflection based proxies.

2. TIME SYNCHRONIZATION PROTOCOL OPTIMIZATION

The standard null-message protocol for conservative time synchronization sends a large amount of null-messages [2, 35, 12, 17]. This is the case because an event core broadcasts null-messages to each other core when it advances the LVT to avoid deadlock. 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 more important 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 the network and go through a deserialization process at the remote end.
In the following sections, we thoroughly evaluate and analyse a number of known conventional synchronization protocol variants that reduce the number of null-messages. Based on our findings, we then propose a hybrid protocol, that combines two traditional protocols, and a self-tuning or adaptive version of this hybrid protocol.

2.1. Evaluation Scenarios

We demonstrate the impact of these algorithms using two scenarios. The first is a simple closed queuing network (CQN) as used in many studies on parallel and distributed simulation performance [6, 24, 25]. The second simulates an electronic auction for compute resources.

2.1.1. Closed Queueing Network Scenario (CQN) The closed queuing network consists of 64 queues in each event core (Q1 to Q64), with 16 servers per queue. Each queue 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 of 100µs. The processing delay in the servers is kept constant at 10µs, while the processing delay in the switches is distributed using a uniform random distribution ranging from 1 to 10µs. The CQN simulation runs for a fixed amount of simulated time (500ms). In this scenario, we change the size of the network with the number of cores used, providing an evaluation based on weak scaling of the problem. For example, a simulation with 3 event cores has 3 switches connected to 3 x 64 queues, each containing 16 servers. Initially, the switch inserts 10 packets into the system. The queueing network structure across cores is depicted in Fig. 2.

![Figure 2. Closed Queueing Network](image)

2.1.2. Electronic Auction Scenario To examine the behaviour 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 launch an auctioneer, managing the auction for the provider’s resources. Consumers entities run a bidding agent for each auction they join. If an auctioneer starts an auction, it is advertised on a market, notifying the consumers. The bidding agents of multiple consumers then compete for the resource by bidding, given a starting price and limited by the available budget. The consumer starts one bidding agent per auction and at most one per job left to run. In this test we simulate 2 markets with 100 consumers and 10 providers per market in each event core. Each consumer has 10 jobs to run and joins auctions until all of its jobs are finished. Additionally we add a number of external consumers per core that connect in round-robin to all markets distributed over all cores, to enable remote interactions into the simulation. The scenario applies weak scaling, where the number of entities scales with the number of active cores. In each core 30% of the consumers connect to an external market in another core.
In this scenario we set the constant delay in the simulated network to 25\text{ms} and the job runtime to 60\text{seconds}. A started auction launches with a uniformly distributed random delay of 0 to 20 simulated seconds. This creates a scenario that is less than ideal to execute in a distributed fashion, as the relatively low look-ahead and relatively large periods of inactivity in the simulation of this scenario result in a inherently low degree of parallelism available in the scenario. During the run, 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.

2.1.3. Experiments

In both scenarios the load is evenly distributed across all the event cores. These experiments were performed on a heterogeneous cluster of 12 machines, named Stewie, with 5 Dual Quad-Core AMD Opteron 2350 processors (2 Ghz) and 6 Dual Quad-Core Intel Xeon L5335 processors (2 GHz) with at least 12 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. The Java VM maximum heap memory was constrained to 11 GB to avoid disk swapping.

In the next subsections, we present and evaluate several synchronization protocol variants to enhance the performance in both parallel and distributed execution mode. Additionally, we try to extract an optimal synchronization protocol that yields the overall shortest runtime for both scenarios in the majority of the configurations. The optimizations we reviewed in (Sect. 1.3) are enabled when possible in these experiments.

In the distributed scenario experiments, the relative standard deviation of the measured runtimes, is 1\% on average, with a maximum of 3\%. The parallel execution has slightly higher a relative standard deviation of 2\% on average, peaks up to 5\% in the CQN scenario and up to 7\% for the auction scenario executed with the less efficient protocol configurations. The measured runtime is an average of at least 3 runs for each data point.

2.2. Ideal Simulation Protocol (ISP)

The Ideal Simulation Protocol (ISP), introduced in [40], uses information from a trace of a previous run to calculate the LBTS and determine if an event core can safely advance. An execution with ISP is an ideal run without the synchronization overhead, e.g. null-message sending or other synchronization communication. With ISP we are able to compute the shortest possible execution time for a specific simulation model and simulator core. It allows us to analyse the parallel speedup exploited using the simulation protocol separately from the parallelism inherent in the simulation model. Evidently, the ISP protocol is not useful to execute a simulation in practice, as it requires a previously generated trace of an identical run.

To implement the ISP protocol, we created an ISP time manager that acts as a replacement for the default time management service implementation.

2.3. Traditional Synchronization Protocols

In an ideal conservative PDES simulation scenario, there is no need for an additional synchronization protocol. All LPs (or event cores) in the simulation are able to calculate the LBTS using the simulation messages they receive. They can advance virtual time seamlessly using this mechanism only. Because such scenarios are rare, PDES simulator implementations have to resort to sending null-messages to make sure no deadlock situations occur where LPs are waiting for each other and the simulation is unable to advance.

In the following sections we present a number of traditional conservative time synchronization protocols. These protocol variants are based on the well known CMB null-message protocols and use null-message reduction techniques to reduce synchronization overhead. We examine runtime in relation to the number of cores in both parallel and distributed parallel scenario configurations. In all of these experiments, the problem size scales with the number of cores, meaning that linear speedup is achieved when the runtime is equal, regardless of the number of cores. However, even with ISP the runtime increases with the number of cores, especially in the parallel scenarios, but also...
in the distributed settings. This is due to overhead caused by the changes of the problem size, e.g. network load, memory allocation rates and thread synchronization. This indicates that some part of the increased runtime is not caused by protocol overhead alone.

For both scenarios, runtime graphs for an increasing number of event cores are presented, in parallel as well as a distributed setting (Fig. 3 and Fig. 4), comparing the runtime performance of the different protocols. Additional graphs, categorizing the outgoing events generated by each event core in a distributed setting help to explain the behaviour of each protocol for both scenarios in more detail (Fig. 5b and 5a). The counterpart graphs for parallel execution mode were omitted because they generally show the same general trends.

![Graphs showing runtime performance](image)

Figure 3. Traditional synchronization protocol runtime comparison for the auction scenario

![Graphs showing runtime performance](image)

Figure 4. Traditional synchronization protocol runtime comparison for the CQN scenario

2.3.1. Standard Protocol We have implemented the principles of the standard Chandy-Misra-Bryant (CMB) protocol [12] as described in Sect. 1.2.1. When an event core advances its time, a broadcast of null-messages is triggered. At that moment a null-message with firetime $LVT + l$, with $l$ the look-ahead, is generated and broadcasted to the other event cores. The null-message is only sent to a specific core if the firetime of the last outgoing event is smaller than the firetime of the null-message about to be sent, eliminating duplicate null-messages. In case of a distributed simulation, we also apply the redundant null-message filtering technique, as described in Sect. 1.3.3, hence even the runtime results of this standard protocol already benefit from some optimization compared to the original CMB algorithm.
As seen from the graphs in Fig. 4, the standard null-message protocol is among the worst performers, in both the parallel and the distributed auction scenarios. The main reason for this is that it essentially generates a large amount of null-messages, creating significant extra processing as well as transportation overhead. The distributed auction scenario shows slightly better performance, because many generated null-messages are filtered by the redundancy checks. As seen in Fig. 5a, about 40% of the generated null-messages are redundant in the 10-core setting. As previously noted, redundant filtering is not available in the parallel execution mode.

With the CQN scenario (Fig. 4), the redundant null-message filtering has less effect, with less than 20% of null-messages filtered as shown in Fig. 5b. In case of the CQN scenario, less additional null-messages will be generated because events are much more evenly distributed over the virtual timeline due to the symmetric architecture of the queueing network, in contrast to the highly randomized auction scenario behaviour. This effect diminishes with an increasing number of cores and larger model scale. From 8 cores onwards, the standard protocol results in the worst performance of all algorithms in the distributed CQN setting and the second-worst performer in the parallel CQN execution.

2.3.2. Timeout-based Protocol

The first synchronization protocol variant reduces the number of null-messages by only sending them after a certain timer expires, and is therefore called the timeout-based null-message protocol [12]. Null-messages are only needed when the normal process of event communication is unable provide frequent time update notifications between cores to advance them properly. A timer is introduced for each outgoing communication channel, which is being reset every time an event (also a non-null-message event) is transported through the channel. This method effectively reduces the number of null-messages significantly while still providing frequent time update notifications to remote event cores. The timer operates with a µs-accuracy.

The main difficulty with a timeout based protocol is the choice of the specific timeout value for each scenario and execution environment. We conducted experiments to discover a range of optimal timeout values suitable for each of our evaluation scenarios in both parallel and distributed setting executed on the specified hardware platform. These experiments reveal how sensitive the runtime performance is to the choice of this timeout. Results are depicted in Fig. 6 for the auction scenario and Fig. 7 for the CQN scenario. We evaluated the scenario runtimes, for timeout values ranging from 10µs to 200µs.

In both the distributed and the parallel auction scenario experiments (Fig. 6), the runtime effects follow roughly the same trend. The runtime is stable up to a certain timeout. A slightly higher timeout results in an abrupt and steep increase in runtime. This behaviour is also visible in the other curves, only shifting the point where the steep slope starts in accordance to the number of cores. Generally, a higher number of cores in the simulation, executing a larger model, increases the degree
of event interleaving, which results in an improved ability to advance the simulation using regular events, hence reducing the need for additional null-messages generated by the timeout protocol. This explains the differences in runtime because of the model scale.

In the distributed auction, an optimal timeout value can be chosen without difficulty, in the range from 50\(\mu s\) to 300\(\mu s\). This is however impossible to do for the parallel auction setting, where the timeout optimum heavily depends on the scale of the model.

In terms of message-count, we observe (Fig. 5a) a large amount of generated null-messages, while about 20\% of them are filtered. The amount of null-messages that is transported after filtering is comparable to the standard protocol, resulting in similar runtime performance figures as shown in Fig. 3.

For the CQN scenario, we observe significantly different runtime timeout sensitivity behaviour compared to the auction scenario, as shown in Fig. 7. The shape of the curve in both the distributed and the parallel CQN scenario experiments is however comparable. In the distributed setting, the number of cores, nor the model scale have impact on the timeout sensitivity. However in the parallel setting, a smaller scale model shifts the position of the steep slope where the increase in runtime starts slightly to the right. The higher runtimes with lower timeouts are caused by an increased number of null-messages, resulting in more processing, filtering and transport overhead. The runtime increase at the higher end of the timeout range can be compared to the sudden and steep increase in runtime for the auction scenario, caused by a lack of incoming events or null-messages to drive the simulation.
at an optimal rate. The difference in the CQN scenario is however that the runtime increase is less steep at the higher end of the timeout range. This means that the CQN scenario is able to advance the simulation much better with relatively few null-messages compared to the auction scenario. Despite that, it is still necessary to add additional null-messages to reach an optimal performance level.

In general, for this scenario, the timeout sweet-spot can be found in the range of 10\text{ms} to 40\text{ms} in all settings (Fig. 7). As shown in Fig. 5b, the amount of transported null-messages is among the least, while still providing a near ISP level of runtime performance (Fig. 4).

Based on the timeout sensitivity graphs, we conclude that it is not possible to choose a timeout value that yields optimal performance for both simulation models, because of the great difference in optimal timeout ranges, whereas the choice of the optimal timeout value is essential to achieve good performance. The protocol comparison graphs for the CQN-scenario (Fig. 4) show an excellent performance result, near the ISP-level, in contrast to the auction scenario runtime results (Fig. 3) showing much worse results. The use of the timeout protocol does not necessarily yield best performance, even if the best possible timeout value is preconfigured. This protocol is useful to cover short fluctuations in the event communication between cores, but is unable to counteract the effects of larger low-activity periods properly. Note that in the protocol comparison graphs, we only depict the performance measurements resulting from the optimal timeout value.

2.3.3. Deadlock Avoidance Protocol

A second null-message reduction technique consists of only broadcasting null-messages to all connected cores, when the event core is about to enter a blocked wait state [26]. As described before (Sect. 1.2.2), this situation only occurs if there are no more events to process in the cores’ EVL or waiting in the input channel endpoint queues and if the core is not allowed to advance based on the current LBTS value. This technique generates a minimum amount of null-messages, only avoiding deadlock situations where the simulation cannot advance without additional time advance information. Therefore, we call this the deadlock avoidance or blocking protocol.

In case of the auction scenario, this blocking protocol is among the best performing protocols (Fig. 3). This can be explained by the fact that the blocking protocol is able to quickly respond and counter longer periods of inactivity, while the auction scenario itself is perfectly capable of advancing the simulation during the other high-activity bursts. Note that due to the properties of the auction scenario, the event core randomly oscillates between relatively short bursts of high-activity, e.g. during bidding, and longer low-activity periods e.g. during job execution.

On the other hand, the CQN scenario does not perform well with the blocking protocol (Fig. 3). In the small scale experiments it is even worse than the standard protocol. The reason for this is that the CQN scenario consists of a uniformly distributed amount of activity during the virtual timeline. In contrast to the timeout protocol, where a steady rate of synchronization information is ensured, counteracting fluctuations in the stream of events between cores, this protocol delays the generation of null-messages until an event core blocks completely. In that case, multiple event cores will become blocked and it takes additional time to recover from this situation, reducing performance level significantly compared to the timeout protocol.

In both scenarios, the blocking protocol sends the least amount of null-messages of all examined protocols, as seen in Fig. 5. Despite that, it only results in improved performance in case of the auction scenario, and has an opposite effect in case of the CQN-scenario. In short, the deadlock avoidance protocol can deal with long term dynamic activity of event communication between cores, but is unable to quickly counteract short term fluctuations of event communication rates.

2.3.4. On-demand Null-Message Sending

Another method to reduce the null-messaging overhead is to send null-messages on-demand [12]. With this technique, an event core broadcasts a request for null-messages to the other event cores when it is entering a blocked state, waiting for events from other cores, and unable to advance. On arrival of such a null-message request, an event core responds immediately, when the request is pulled from the input channel endpoint, 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. This technique is basically an extension of the deadlock avoidance strategy because
the requests are broadcast the same way the deadlock avoidance algorithm broadcasts null-messages. The difference is that a waiting event core potentially revives quicker because the other event cores respond more quickly to a request for null-messages. The request-generated null-messages adhere to the same rules as the standard sending technique by only sending relevant and non-redundant null-messages. The request-events themselves however, can not be filtered for redundancy.

The results concerning this protocol differ significantly between the parallel and distributed setting in case of the auction scenario, as shown in Fig. 3. In the distributed setting, the request protocol is the poorest performer of all protocols, even worse than the standard protocol. In contrast to the parallel setting, where it is the best performer. As illustrated in Fig. 5a, the total number of time synchronization messages, both requests and responses, slightly exceeds the standard protocol. Due to the network latency, the amount of requests that result in a useful response is much smaller than the total number of requests. Although these requests are triggered at the same time as the null-messages in the blocking protocol, the larger size and additional processing of the request-messages, results in much more generated messages compared to the blocking protocol. In the parallel setting, due to the significantly reduced latency, an irrelevant message size due to reference passing and the scenario inherent core idle time, the protocol is able to overcome the extra overhead caused by the request processing and generation and yields the best performance of these basic protocols.

The CQN-scenario gains some efficiency using on-demand null-messages, compared to the blocking protocol. As pictured in Fig. 4, performance lies between pure blocking and timeout protocols. Due to the less frequent blocking of the event core in this scenario, the amount of requests has a reduced negative impact on performance while the response null-messages make up for some of the short term fluctuations in simulation activity. These effects improve performance compared to the pure blocking protocol. The CQN-scenario shows a better ratio between requests and response messages, as shown in Fig. 5b, whereas the total amount of time messages is higher than those measured with the blocking protocol.

2.4. Advanced Synchronization Protocols

The previous section described various conventional null-message reduction techniques to limit the synchronization protocol overhead. The results indicate that the choice of the optimal algorithm heavily depends on the executed scenario. Although the performance of some of these basic techniques can be satisfactory, the user of a simulation framework still has to specifically configure the protocol for each experiment. Determining the optimal configuration is difficult and error-prone, as it depends on a number factors: the hardware platform, the scale of the simulation, the interactions patterns in the scenario and the entity distribution. In this section, we propose two hybrid protocols combining the properties of previous algorithms. The goal is to provide the best overall performance level, regardless of the scale or configuration of the simulation model and execution environment. In addition we want to make the protocol adaptive and minimize or eliminate the need for explicit configuration of parameters to attain the best performance.

2.4.1. Timeout Protocol with Deadlock Avoidance  As we concluded in previous Sect. 2.3, for distributed execution, the deadlock avoidance (blocking) protocol has the most optimal performance in case of the electronic auction scenario, whereas the CQN scenario performs best with the timeout based null-message reduction technique, given the right timeout. A combination of both algorithms may perform well for both settings.

Other combinations make less sense. A combination of the deadlock avoidance and on-demand protocol would only result in an equal amount or even more null-messages, because the requests are generated at the same time the blocking messages would be generated. Most of these regular null-messages will then be filtered due to redundancy, still generating at least as much requests as in the pure on-demand protocol. Furthermore, a combination of the on-demand and the timeout based protocol would perform worse or equal to the request protocol on its own for the same reason.

Given this combination, it is of course still necessary to choose an optimal timeout value. Therefore we performed experiments to determine the best timeout range for both our evaluation scenarios. The graphs of Fig. 8, illustrate the runtime timeout sensitivity for different simulation model scales in a
distributed setting for the hybrid protocol. The graphs for the parallel setting were omitted because they show similar behaviour.

Fig. 8a shows that for this protocol a timeout value starting from 10ms and above, has very limited or no influence on the runtime performance in case of the auction scenario. This is consistent with the fact that the blocking protocol has the greatest impact on performance with this scenario and the amount of generated null-messages is nearly identical to the message count of the blocking protocol (Fig. 5a).

On the other hand, the CQN scenario shows a different curve (Fig. 8b). Compared to the previous sensitivity graph (Fig. 7), the curve is flattened but still shows the same trends and an identical optimum range from about 10ms up to 45ms. Above 45ms, the runtime slowly converges to a runtime comparable with the blocking only protocol. We observe a steep increase in runtime for timeouts below 10ms, emphasizing the need to correctly tune the timeout value. The amount of generated null-messages is slightly higher compared to the pure timeout-based protocol, as shown in Fig. 5b.

Based on the timeout sensitivity experiments, a timeout value of about 20ms, results in good performance for both scenarios (Fig. 9 and 10). These graphs also show runtime results for the standard CMB protocol and the best protocol for each scenario in both parallel and distributed execution mode.
Note that in the parallel auction setting, the on-demand protocol is slightly faster than the basic blocking protocol and the new hybrid method. This on-demand algorithm however has significant negative effects on runtime in the other settings. This is caused by the worst-case properties of the auction scenario creating large amounts of idle time that diminish the effect of the extra overhead created by the request processing.

In summary, given the right timeout value is chosen, this combination of a timeout-based protocol with a deadlock avoidance strategy proves to result in a performance level near the best basic protocol for both our evaluation scenarios.

2.4.2. Adaptive Timeout-based Protocol with Deadlock Avoidance

Although the combination of the timeout and the deadlock avoidance algorithms shows good results in our experimental settings, with both the CQN and the auction scenario, the range of optimal timeout values may depend on both the evaluated simulation scenarios and the hardware platform. To overcome the limitation of having to choose a timeout value for each execution setting we introduce an adaptive algorithm that attempts to automatically configure the previously described hybrid timeout-based deadlock avoiding protocol, thereby maximizing the useful event throughput and thus minimizing the protocol overhead.

By only accounting for the throughput of useful events, excluding protocol specific events (e.g. null-messages), this algorithm searches for the optimum balance between null-message generation and processing overhead and used network-bandwidth.

Algorithm
The technique consists of a centralized two phase algorithm. Pseudo-code is show in Algorithm 1. After an initial startup delay, the algorithm enters the first phase, namely the discovery phase. In this stage an initial timeout is discovered by recursively subdividing an interval of timeout values into smaller pieces until the resulting interval is small enough to start the second adjustment phase.

We chose the initial interval to be in the range of 1ms to 200ms (upperLimit and lowerLimit respectively), a reasonably large range for current platforms. This leads to a minimal null-message rate of 5 events per second at the 200ms boundary. In each iteration, first three values are chosen in the interval following a power law, where the difference between each consecutive value grows exponentially. The reason for this is that the effect of identical change in timeout becomes smaller with higher timeouts, so that the larger the timeout value is, the larger the change in timeout has to be to observe an effect. Then, the algorithm measures which of these three values yields the highest useful event throughput. If the best timeout value is determined, new interval boundaries are obtained using the upper and lower neighbouring values out of the three calculated values and the previous interval boundaries. This sequence continues until the size of the new interval is smaller than 20% of the current optimal timeout value.
Algorithm 1 Protocol Tuning Algorithm

procedure INITIALIZE()
\begin{itemize}
  \item $T[0] \leftarrow \text{lowerLimit}$
  \item $T[5] \leftarrow \text{upperLimit}$
  \item $\text{delay} \leftarrow \text{sampleDelay}$
  \item $\text{bestIndex} \leftarrow -1$
  \item $\text{phase} \leftarrow \text{DISCOVERY}$
\end{itemize}

procedure \text{CALCULATE\_NEXT\_INTERVALS}(T, \text{bestIndex})
\begin{itemize}
  \item if $\text{phase} = \text{DISCOVERY}$
    \begin{itemize}
      \item if $\text{bestIndex} \neq -1$
        \begin{itemize}
          \item $T[0] \leftarrow T[\text{bestIndex} - 1]$
          \item $T[5] \leftarrow T[\text{bestIndex} + 1]$
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \item then $(T[1], T[2], T[3]) \leftarrow \text{POWERLAW\_DIVIDE}(T[0], T[5])$
  \item if $T[5] - T[0] < 0.20 \times T[\text{bestIndex}]$
    \begin{itemize}
      \item then $\text{phase} \leftarrow \text{ADJUSTMENT}$
    \end{itemize}
  \end{itemize}

if $\text{phase} = \text{ADJUSTMENT}$
\begin{itemize}
  \item $T[1] \leftarrow (1 - \text{PARETO\_RANDOM}) \times T[\text{bestIndex}]$
  \item $T[2] \leftarrow T[\text{bestIndex}]$
  \item $T[3] \leftarrow (1 + \text{PARETO\_RANDOM}) \times T[\text{bestIndex}]$
\end{itemize}

main
\begin{itemize}
  \item while \text{SimulationRunning}
    \begin{itemize}
      \item $\text{SLEEP}(\text{startupDelay})$
      \item \text{CALCULATE\_NEXT\_INTERVALS}(T, \text{bestIndex})$
      \item if $\text{MIN}(T) < 0.9 \times \text{lowerLimit}$
        \begin{itemize}
          \item or $\text{MAX}(T) > 1.1 \times \text{upperLimit}$
        \end{itemize}
      \item then \text{INITIALIZE}()
    \end{itemize}
  \end{itemize}

do \begin{itemize}
  \item for $i \leftarrow 1$ \text{to} 3
    \begin{itemize}
      \item do \begin{itemize}
          \item $R[i] \leftarrow \text{MEASURE\_RATE}(T[i])$
          \item $\text{SLEEP}(\text{delay})$
        \end{itemize}
      \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item if $\text{MIN}(R) < \text{threshold}$
      \begin{itemize}
        \item then $\text{delay} \leftarrow \text{delay} + 1$
      \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item $\text{bestIndex} \leftarrow j$ where $R[j] = \text{MAX}(R)$
  \end{itemize}
\end{itemize}

Consequently the adjustment phase of the algorithm is entered, from which point on generally small adjustments to the previously discovered timeout are made to keep the performance at an optimal level, incorporating changes in the simulated scenario or the execution environment. Therefore our algorithm applies simulated annealing, using a steeply shaped Pareto distribution ($\alpha = 5$), to determine upper and lower neighbouring values in each iteration. These calculated timeout values, in combination with the previous best one, are then evaluated again for optimal throughput. The use of a Pareto distribution ensures that the timeout changes are mostly localized, but allows for rare larger changes. This avoids the algorithm to strand in a local minimum. In order to make the algorithm more robust, the occurrence of a newly generated value that is more than 10% outside of the initial start interval causes a restart of the algorithm, switching back to discovery mode. Hence, a random restart might be triggered by an extreme value coming from the Pareto distribution. If the induced restart seemed unnecessary, the timeout converges quickly to the previous optimum, whereas
long term persistence in a suboptimal value is avoided. Additionally, it makes the algorithm more capable to respond to dynamic behaviour in the scenario or the execution environment.

The performance assessment in the algorithm is based on regular samples of the rate of outgoing events, excluding null-messages. Since these rates may fluctuate heavily, the rate is calculated per core based on the number of outgoing non-null-message events and then averaged over all cores in the controller. Due to the delayed observations of the changing timeout the sample interval is initiated at a relatively long time of 5 seconds (sampleDelay), which also averages out short term fluctuations. In case of a simulation scenario where the entities in each core communicate rarely, this sample interval might be still too small to calculate valid event rates, in which case our algorithm increases the sample delay and restarts the discovery phase until valid measurements can be extracted.

**Evaluation**

As observed in Fig. 8, the combination of a timeout based protocol and the deadlock avoidance strategy performs well in both evaluation scenarios if the timeout is configured with a value between 10 ms and 45 ms. Consequently, the adaptive algorithm has a relatively broad range of values to achieve good performance.

In order to further evaluate the potential of a dynamic tuning strategy, we executed the previously discussed CQN-scenario simulation on a different platform, namely on a cluster with Intel Pentium 4 3 GHz hyperthreaded CPUs, 1.5 GB memory, and a 100 Mbit interconnect. In Fig. 11a, the difference in the timeout sensitivity range for the two platforms is shown. Note that only 200 ms was simulated on the P4-cluster to result in a comparable runtime, as opposed to 500 ms in the other experiments. The results show that the optimum range of timeouts for the P4-cluster is shifted to the right to a value between 40 ms and 80 ms. Even though there still is a small overlap in the ranges, the difference indicates that the performance level of a specific timeout value cannot be generalized to all simulated scenario and/or execution environments. Fig 11b compares the fixed timeout hybrid method, with a timeout set in the optimum range, to the adaptive approach. The relative standard deviation in these experiments was slightly higher with a maximum of 6%. We also show the standard null-message protocol and ISP runtimes as a reference. The runtime with a suboptimal timeout setting of 10 ms is about 30% higher than the runtime with an optimal timeout (50 ms) or using the dynamic tuning algorithm in the 10-core experiment. In this alternative setting, the adaptive approach still performs as expected.

![Timeout sensitivity, comparison between two platforms](image1)

![Protocol runtimes on the P4-cluster](image2)

Figure 11. Timeout sensitivity and protocol runtimes for a distributed CQN scenario

The properties of the model also play a role in determining the optimal timeout range. Fig. 12a, compares multiple CQN scenario executions where the network delay, and consequently the look-ahead value, ranges from 5 µs to 100 µs. The graph shows that lower a look-ahead results in a smaller window of optimum timeouts. This is due to the fact that a lower look-ahead in the model requires more frequent time update notifications for good performance and therefore shows less tolerance for higher timeout values. Note that we reduced the amount of simulated time in these experiments...
where needed to shorten the runtime, therefore absolute runtime numbers are not relevant in these runs. If the look-ahead reaches a certain level, e.g. 50µs, this effect diminishes and the optimum timeout window stabilizes. The adaptive algorithm is able to produce runtimes close to these in the optimal timeout interval. Fig. 12b lists these scenarios with their configured finish times and the adaptive execution runtimes.

Figure 12. Timeout sensitivity for a distributed CQN scenario with various look-ahead values

Compared to the hybrid protocol, performance for the adaptive hybrid algorithm is equal in case of the auction scenario, as shown in Fig. 9. In case of the CQN scenario, we observe a slight increase in runtime, especially for the parallel scenario, as shown in Fig. 10. Even though our new dynamic algorithm introduces a number of additional algorithm inherent parameters (e.g. interval boundaries, sample interval, Pareto distribution parameters, etc.), the algorithm robustness is indicated by consistent runtime results. In practice, our dynamic algorithm reaches equal performance compared to optimal protocol in a wide range of simulation settings, while eliminating the requirement for manual configuration entirely.

3. EVENT CORE PACKING PERFORMANCE

In this section we evaluate the parallel performance in terms of speedup that can be obtained using the previously described CQN and electronic auctioning models, executed in a parallel, distributed and hybrid event core packing configuration. In contrast to previous experiments, strong scaling is used here. As such, our results show the amount of time that can be gained through parallelization and/or distribution, without increasing the scale of the problem. The model entities are distributed over a varying number of event cores running on one or more cluster nodes. The CQN scenario consists of 2560 event queues in total, with 16 servers each. The auction scenario consists of 40 markets with each 100 consumers, 10 providers and 10 jobs per consumer. We collected sample points for an entirely parallel core configuration (1 node), a fully distributed set-up with multiple nodes that each execute one event core and several hybrid runs with a number of parallel cores running on two or more cluster nodes. An example core distribution with 10 event cores running on 5 physical nodes, results in nodes executing two parallel event cores with each 256 queues with a CQN scenario and with each 8 markets for the auction scenario.

Figure 13 shows a grid where each crossing denotes a mapping of a number of event cores running on a number of physical cluster nodes. The intersection circles, with their respective speedup value, are scaled proportionally to the speedup value for the specific event core and node configuration of that sample point. The specific sample points on the grid, that correspond to core distributions, were chosen so that event cores as well as entities could be divided evenly among the available nodes and event cores respectively.
For the CQN scenario (Fig. 13a), we observe near linear speedup figures in case of a fully distributed setting with one event core per node. This is due to the fact that the increased communication and synchronization overhead for an additional event core is compensated by the added CPU power of an additional node. The one-node parallel configurations become less efficient the more the CPU on the nodes is loaded with the activity of event core threads. The unused CPU cores are used to handle communication to other event cores and by the JVM (e.g. for garbage collection). Additionally, as we observed before [2], the Java VM becomes less efficient with an increased number of event core threads that are creating and scheduling events at high rates.

Hybrid core distributions that run more than one event core per node utilize the available hardware better, at the cost of a slightly higher runtime. This shows that extra CPU cores can take on protocol overhead in parallel to event core processing, so that close to optimal speedups can be attained with regard to the number of nodes in the distributed cluster. For example, for a CQN experiment with 8 nodes and 1 event core per node, the runtime only increases by 20% compared to the run on only 4 nodes with 2 event cores per node.

The constant problem size in this experiment implies that the linear speedup trend should deteriorate when the model is subdivided into more and smaller partitions, causing the communication and synchronization overhead to predominate over useful event processing. This point is however not reached in this experiment for the CQN scenario that is executed on a 10 node cluster.

One can observe the impact of increasing overhead relative to useful event processing for each additional event core in the auction scenario (Fig. 13b). A growing number of nodes with one event core results in an increased speedup. The gain in speedup for each extra node however decreases with more nodes, as the additional overhead of additional event cores starts to dominate the useful event processing load. Due to the relatively high synchronization and communication intensity of the auction scenario, compared to the CQN model, the distributed auction scenario execution is able to attain a speedup of 3.1 and 2.9 in a 10 node and 4 node configuration respectively. This is despite the fact that core to core communication is significantly more costly in the distributed setting. In the purely parallel setting, speedup results for this scenario are not exceeding 1.8, regardless of the number of event cores executed on the node.

Interestingly, the maximum attainable speedup for a purely distributed execution is significantly higher compared to the pure parallel core configurations for both scenarios. Further investigation shows however that this is due to hardware limitations, e.g. memory bandwidth or CPU cache effects. Figure 14 shows the speedup results for the execution of the previous scenarios on a single node, comparing a single JVM parallel event core setup and a multiple JVM 'distributed' setup using the local machine loopback network interface to communicate. This shows that both the parallel and distributed communication subsystems perform on par up to 4 event cores. The additional overhead resulting from network communication in the multi-JVM deployment can be fully processed by the
other available CPU-cores. From 5 event cores on, this is no longer the case and the multi-JVM setup shows a degradation of performance. These results confirm, that the communication overhead between event cores in a single JVM, using reference passing, is significantly lower than between event cores in separate JVMs through a network connection. The increased performance for a fully distributed deployment in Fig. 13 is therefore not a consequence of the parallel or distributed implementation, but of hardware capabilities.

4. RELATED WORK

Initial research on discrete event simulation dates back to the 1950s. The development of the null-message protocol drove the early work on parallel and distributed simulation, see e.g. [14] in 1978 and [15] in 1977. In the years thereafter, many additions and improvements to the original conservative protocol were introduced. In [12, 16], the authors describe reductions of the number of null-messages by sending them more intelligently (e.g. on demand or delayed until some timeout occurs). Another approach is the carrier null-message protocol [17] that handles synchronization more effectively by including more information in the null-messages. In contrast to the deadlock avoidance null-message algorithms, the deadlock detection and recovery algorithm [13] allows deadlocks to occur but provides a mechanism to detect and recover from them. The bounded lag algorithm [18] and execution with barrier synchronization [19] are some of the other time synchronization mechanisms that were proposed. The conditional event algorithm [20] operates by determining conditional and definite events, eliminating the need for null-messages, but requiring detailed dependency information within the model.

In 1985, Jefferson [10] published the Time Warp protocol, a so-called optimistic synchronization protocol. It 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. The lazy cancellation reduces the number of roll-backs by only canceling events that do not have the same outcome and lazy re-evaluation tries to store as much state information as possible to allow a quick roll-back, at the cost of a higher memory usage. Several authors proposed techniques (e.g. Cancelback [21], artificial roll-back [22], etc.) to store as much state information as possible with a minimal amount of memory.

In Sect. 2.3 we describe and evaluate several conservative synchronization protocols: the delayed or timeout-based null-message sending, the on-demand or request-based protocols [12] and the
blocking or deadlock-avoidance algorithm also known as lazy null-message sending protocol [23]. In [23] the author also performs a theoretical analysis of null-message overhead for combination of the blocking and timeout algorithms. The authors have not investigated the protocol’s performance.

Simulations of communication networks [4, 27] and electronic circuits [4] were used to evaluate parallel discrete event simulation performance. Similarly, conservative time synchronization protocol performance was analysed using synthetic application benchmarks simulating queuing networks [24, 6, 25], communication networks [4] or electronic circuits [26, 4]. The study in [3] compares optimistic and conservative synchronization on a large scale platform and shows that both protocols scale well to a large number of processors.

In [5] and [6] one finds reviews of the parallel and distributed simulation basics, conservative, optimistic or hybrid approaches and optimization techniques. In [28], the author describes the common pitfalls in parallel and distributed simulation and gives suggestions on how to avoid them.

5. CONCLUSION

The computational challenge of simulation of large-scale systems requires discrete event simulators to be capable of executing on parallel, distributed or hybrid machine configurations. For discrete event simulators using conservative time synchronization, this implies having effective null message reduction techniques in the synchronization protocol, to optimize performance. Even when the exchange of null messages is minimized with respect to optimal binding of model entities to event cores and assignment of event cores to computing units, the synchronization protocol remains an important factor in determining performance.

We have evaluated a number of traditional reduction techniques with two application scenarios: a closed queuing network and an electronic auction for resources. We find that these techniques can offer performance improvement, provided the appropriate technique is matched to the application.

We have proposed a new protocol variant, combining two traditional techniques. It provides further performance improvement and does so for both application scenarios. The adaptive, self-tuning, version of the proposed protocol improves robustness against varying simulation conditions. It also eliminates the need to explicitly configure parameters to attain optimum performance for a specific model or execution machine configuration.

Further research will be conducted on the scaling of these techniques to a larger numbers of simulation cores and their characteristics on clusters with high-performance interconnects. We also intend to investigate the interaction of the adaptive protocol with load balancing in the simulator where the number of event cores and the binding of simulation entities to event cores is changed during execution to optimize memory footprint, CPU load or in response to changing communication patterns.

REFERENCES