Conservative Distributed Discrete Event Simulation on Amazon EC2

Kurt Vanmechelen, Silas De Munck and Jan Broeckhove
Department of Mathematics and Computer Science
University of Antwerp
Antwerp, Belgium
kurt.vanmechelen@ua.ac.be

Abstract—A discrete-event simulator’s ability to distribute the execution of a simulation model allows one to deal with the memory limitations of a single computational resource, and thereby increase the scale or level of detail on which models can be studied. In addition, distribution has the potential to reduce the round trip time of a simulation by incorporating more computational cores into the simulation’s execution. Such gains can however be voided by the overhead that time synchronization protocols introduce. These protocols are required to prevent the occurrence of causality errors under a parallel execution of a simulation. This overhead depends on the protocol used, characteristics of the simulation model, and the architecture of the computational resources used. Recently, infrastructure-as-a-service offerings in the cloud computing domain have introduced the possibility for flexibly acquiring computational resources on a pay-as-you-go basis. At present, it is unclear to what extent these offerings are suited for the distributed execution of discrete-event simulations, and how the characteristics of different types of resources impact the performance of distributed simulations. In this paper we investigate this issue, assessing the performance of different conservative time synchronization protocols on a range of cloud resource types that are currently available on Amazon EC2.

Keywords—PDES; cloud computing; EC2; performance; conservative time synchronization;

I. INTRODUCTION

Computer simulations have become an indespensible tool in multiple research domains for the empirical study of large-scale systems. This is mainly due to their availability, reproducability and cost-effectiveness. Although simulations often allow for a more timely delivery of results compared to real-world experimentation, it remains a challenge to perform simulations of large-scale systems (e.g. peer-to-peer networks, cloud computing systems) in a timely manner. Techniques to distribute and parallelize a computer program’s execution can be employed to tackle this challenge.

In the case of discrete-event simulations, parallel discrete event simulation (PDES) can be realized by partitioning the simulation domain and associating the resulting partitions to different event cores. These cores then concurrently process events that occur in the partitions. Parallel event processing introduces the need for algorithms that prevent causality errors from occurring during the simulation’s execution. A multitude of approaches for time synchronization in a PDES setting have been described in the literature [1]. Recently, the PDES domain has been given renewed interest, in part due to the architectural shift towards multi-core architectures that has taken place in processor design.

At the same time, a new paradigm has arisen for flexibly acquiring and configuring third-party IT resources and services. Cloud computing offerings in the infrastructure-as-a-service (IaaS) domain allow consumers to flexibly acquire resources on a pay-as-you-go basis with minimal administrative overhead and provider interaction [2]. To the users of simulation tools, adopting such offerings can be advantageous compared to the use of local infrastructure as:

- The flexibility to acquire and release resources on a pay-as-you-go basis gives users access to a broader range of QoS levels. It also introduces the possibility to align these QoS levels with the requirements of an individual user. Users can acquire large amounts of computational power that are immediately delivered, but allocated on a limited temporary basis. This flexibility is especially valuable for consumers that need simulation results in time w.r.t. a given deadline (e.g. paper submission, presentation, meeting,...).
- The flexibility with respect to the architectural properties of the resources that are acquired enables one to optimally align the cost and the architectural properties of the computational resources to achieve the desired goals. Depending on the characteristics of the simulation model, trade-offs can be made between cost and resource characteristics such as computation and communication performance, memory availability and availability of multiple CPU cores.

For researchers that tackle the algorithmic challenges in the PDES domain, IaaS can provide a wide range of computational platforms to test the robustness of their approach.

Nevertheless limited records currently exist of parallel and distributed discrete-event simulation on cloud computing infrastructures. In this paper, we present an approach for instantiating simulation models on such infrastructures and analyze the performance implications of the different virtual server offerings that are currently available on Amazon EC2 [2] for different time synchronization protocols.
II. RELATED WORK

Initial research on discrete event simulation dates back to the 1950s. Early work on parallel and distributed simulation was driven by the development of the null-message protocol, also referred to as the CMB protocols by Chandy and Misra [2], and independently developed by Bryant [3]. Many additions and improvements to the original conservative protocol were made thereafter to reduce the number of null-messages by sending them more intelligently [4]. The optimistic class of protocols was introduced by Jefferson in 1985 [5]. These protocols introduce a roll-back mechanism providing proper synchronization across event cores, by reverting to a previous state if a causality error occurs. Numerous extensions and improvements have been developed since then. A performance comparison of both optimistic and conservative time synchronization algorithms on a large-scale distributed cluster system has been performed in [6], demonstrating that both types of protocols are able to scale well to a large number of processors with a highly performant technology in parallel and distributed discrete-event simulation. 

A detailed overview can be found in [7], [1].

Fujimoto et. al have proposed an approach for using optimistic time synchronization protocols in cloud environments [9]. In [17], they propose an optimistic time synchronization protocol that is better tailored to execution platforms such as clouds that exhibit longer communication delays and higher external interference from other users' computations, as compared to tightly coupled local clusters. Their evaluation does not use real cloud offerings though, as they deploy the simulation on a locally controlled set of machines that belong to different LANs. D’Angelo has discussed the potential advantages of using cloud computing infrastructure for PDES and has proposed the use of entity migration techniques to improve the performance of simulations in heterogeneous and dynamic environments such as clouds [10].

To our knowledge, we are the first to investigate the performance of different time conservative time synchronization protocols for PDES in real cloud environments.

III. PARALLEL AND DISTRIBUTED DISCRETE-EVENT SIMULATION

In this section we briefly introduce the concepts and terminology in parallel and distributed discrete-event simulation. A detailed overview can be found in [7], [11].

A. Terminology

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. These messages are represented by events on the level of the LP. The state of the 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. In the discrete model of time progression, that we will adopt in the remainder of this paper, 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 or at a preconfigured finish time.

B. Approach

A discrete-event simulation core executes the LP by adopting 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 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 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, the event cores synchronize their time advances 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 [11], the results from a single core simulation are the same as those from a multi-core simulation. To realize this, a synchronization mechanism or protocol is required. In the remainder of this paper we will adopt algorithms from the conservative class of time synchronization protocols. These only allow an event core to process an event if it is guaranteed that no events with a smaller firetime can arrive in the event queue. In these protocols, each outgoing event e is associated with a send time $T_s(e)$ and a firetime
$T_f(e)$. An LP contains an incoming message first-in-first-out (FIFO) queue for each other logical process. 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 $\min_i(T(Q_i)) + l$ is reached. The lookahead $l$ is model-dependent and crucial to introduce parallelism in the simulation’s execution [11]. The minimum $\min_i(T(Q_i))$ is essentially a lower bound for the local virtual time (LVT) in all logical processes, also referred to as the lower bound time stamp (LBTS). If one of the input queues in an LP becomes empty, the LP must wait until new messages arrive in that queue. Events in other queues can be processed up to the time of the last processed event. This mechanism may result in a deadlock or memory overflow caused by a cycle of logical processes that are blocked by an empty queue. To avoid this, an LP must receive updates on the LBTS of the simulation, so that it can ensure no events before a certain time can arrive. Therefore, empty events called null-messages are sent that only carry a $T_s(e)$ timestamp.

An overview of an event core that operates according to the proposed design is depicted in Fig.1. 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 recalculates the LBTS. 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. For a full explanation of the design choices and further details concerning our parallel and distributed simulator implementation, we refer to [12] and [13].

C. Null-message sending strategies

Different strategies exist for reducing the overhead caused by the transmission of null-messages. Because of the higher costs of core-to-core communication in a distributed simulation context, their importance increases significantly compared to a non-distributed simulation setting. We analyse the performance of the following strategies:

- **Standard Chandy-Misra-Bryant protocol (STD):** This is an implementation of the protocol as described in [4]. Each time an event core advances its time, a null-message with firetime $LVT + l$ is generated and broadcasted to the other event cores. The null-message will only be sent to a specific core if the firetime $t_i$ of the last outgoing event was before the firetime of the null-message.

- **Timeout-based Null-message Sending (TIM):** This strategy reduces the number of null-messages by only sending them after a timer has expired. As null-messages are only needed when the normal process of event communication can’t provide enough time update notifications between cores, a timer is introduced for each outgoing communication channel. This timer is reset every time an event is transported through the channel. This technique is often referred to as delayed null-message sending.

- **Deadlock Avoidance Null-Message Sending (BLO):** In this strategy, a core broadcasts null-messages to all other cores when it is about to enter a blocked wait state, unable to advance its LVT until new events arrive.

- **On-demand Null-Message Sending (REQ):** In this strategy, an event core broadcasts a request for null-messages to the other event cores when the core is entering a wait state, waiting for events from other cores, and unable to advance. If an event core receives such a null-message send request, it responds by sending a null-message to the requesting core as quickly as possible. 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 more quickly because the other event cores respond to these request for null-messages.

- **Timeout Protocol with Deadlock Avoidance (BLOTIM):** A combination of the BLO and TIM protocols. For details, we refer to [12], [13].

- **Ideal Simulation Protocol (ISP):** The Ideal Simulation Protocol (ISP), introduced in [14], uses a trace of a previous simulation 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 thereby serves as a benchmark, indicating the optimal performance that can be achieved for a given simulation model w.r.t. time synchronization performance. Note that for all of these protocols, including STD, redundant null-messages waiting in the send-queue are filtered.

IV. DISTRIBUTED SIMULATION ON EC2

A. Infrastructure-as-a-Service

Cloud computing has become a catch-all term for various forms of computing in which third-party services are acquired in a flexible manner with low administrative overhead. The model’s recency and the hype surrounding cloud computing has precluded the broad acceptance of a single definition of what cloud computing exactly is, introducing ambiguity [15]. A more specific characterization of cloud computing is possible through a taxonomy on the type of services that are offered. In Software-as-a-Service, access to finished software products is provided to users under a flexible payment model or a freemium model. Example products in this domain are G-Mail (mail), Salesforce (CRM) and Google Docs (office suite). Platform-as-a-Service offerings allow the user to build and deploy applications on third-party infrastructure using a specific development framework that forms the core of an offering. The framework provides the developer with interfaces and library routines that partially automate the creation and efficient deployment of applications that perform (large-scale) processing on the provider’s infrastructure. Examples of PaaS offerings are Windows Azure and Salesforce.com. Infrastructure-as-a-Service offerings provide a consumer access to low-level resources such as virtual servers that are launched in the provider’s data center, scalable data storage facilities, and network connectivity. Many IaaS providers currently exist (e.g. Amazon, Rackspace, GoGrid, CloudSigma), each having their own price and product models. Nevertheless common denominators in the various offerings are the following:

- The operating system and software that runs on the virtual server can be fully customized by the consumer (restrictions can apply to the OS kernel). A user can create an image of such a software configuration that can be used to launch multiple servers.
- Resources in the data center are paid for in a fine-grained manner (e.g. hours of instance time used, GBs of data stored and transmitted to/from the cloud).
- Resources can be acquired and released in a fine-grained manner.
- To increase infrastructure utilization, the provider multiplexes the execution of multiple server instances (possibly created by different consumers) on a single physical server.

In the next section we will discuss the deployment of a distributed simulation on Amazon’s IaaS offering, the Elastic Compute Cloud (EC2).

B. Deployment of a distributed simulation on Amazon EC2

We have implemented the approach to distributed and parallel simulation outlined in section III in the Grid Economics Simulator (GES) [16]. GES has been primarily designed to study market-based resource allocation algorithms in Grid and Cloud computing settings.

In order to deploy a distributed simulation on EC2 the following steps have been made:

1) Image creation: A server instance is launched with a vanilla Ubuntu 11.10 image available at EC2. The following steps are taken to customize the image:

- Installation of Maven and SVN. Maven is used by GES to automatically download and install all library dependencies on a host machine, perform builds, and launch the simulator.
- Customization of the .bashrc file in order to initialize a number of environment variables.
- Two scripts for launching a head node or worker nodes are installed. The scripts automatically update the source code of the simulator and simulation models using SVN, launch a build using Maven and subsequently start the required service. Head nodes and worker nodes discover each other on the network using the Jini framework [17].
- The image creation is triggered through the EC2 Web management console and the image stored in Elastic Block Storage (EBS). In addition to a 32-bit and 64-bit image, a separate image for cluster instances is required in order to use all of the different instance types on EC2.

2) Elastic IP allocation: An elastic IP is allocated for the head node. A booted instance receives a new public and private IP by default when launched on EC2. Elastic IPs allow the user to attach a fixed IP address to a running instance.
3) **Head node/worker node launch**: A Java program that calls out to EC2 using the EC2 Web Service API launches a server instance of the specified instance type with the correct Amazon Machine Image (AMI). In the launch request we add a user data script that calls out to the script for launching the head or worker node. This user data script is automatically launched after the server instance is booted. The use of such scripts allow for the customization of a server instance irrespective from its AMI. As such, it reduces the administrative burden of keeping multiple AMIs up to date, and the costs associated with storing AMIs. After sending request for launching the head node instance, the Java program waits for the instance boot to finish, after which the elastic IP is associated to the instance. Subsequently, the required number of worker nodes are launched. The head node launches a simulation coordinator, whereas each worker node launches a simulation execution service and they both register their proxies with the Jini lookup service, enabling the services to discover each other. To execute a simulation run, a configuration file, containing model and simulator parameters, is submitted to the coordinator on the head node, which enqueues these configurations and launches the required amount of event cores in the worker nodes’ execution services when they are available. The coordinator is also responsible to set up the communication channels that allow for inter-core communication. An event core executing in the worker nodes also feeds back information to the controller in the head node. In order to enable inter-core communication, a security group is associated with the instances that defines the proper firewall configuration (by default EC2 instances aren’t allowed to accept any incoming traffic). Each server is launched with a user-specified key pair. The user can log in to the instances through the SSH protocol by using a private key. Each instance is also given a tag in order to properly identify the instance’s function in the AWS management console.

V. EXPERIMENTS

A. Simulation Models

Our experimental evaluation will use two simulation models that differ in terms of event arrival rates, communication patterns and computation to communication ratios. The first is a closed queuing network (CQN) [18], [1], [19], the second simulates an electronic auction for compute resources.

1) **Closed Queueing Network (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 that connects these entities has a constant delay of 100µs. The processing delay in the servers is a constant 10µs, while the processing delay in the switches is distributed using a uniform random distribution with discrete number ranging from 1 to 10µs. The CQN simulation runs for a fixed amount of simulated time (200ms). Initially, each switch inserts 10 packets into the system. The queuing network structure is depicted in Fig. 2.

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

2) **Electronic Auction**: 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, 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 AuctionConsumers which launch an AuctionBiddingAgent for the new auction if the consumer has more jobs to run and has budget left to place bids. The AuctionBiddingAgents 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 100 consumers and 10 providers per market in each event core. Each consumer has 10 jobs to run and joins auctions until all his jobs are finished. Additionally we add a number of external consumers per core that connect in a round-robin manner to all markets distributed over all cores, to enable remote interactions into the simulation. 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 25ms and the job runtime to 60seconds. A started auction launches with a uniformly distributed random start delay of 0 to 20 simulated seconds. This creates a scenario that is less than ideal to execute in a distributed
fashion, which is caused by the relatively low look-ahead and relatively large periods of inactivity in the simulation of this 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. The auction scenario is limited to execute for a fixed amount of simulated time (1000s).

B. Infrastructure

This section discusses the characteristics of the computational resources that were used to perform the experiments.

1) Instance types: We have executed the two simulation models discussed in the previous section on the EC2 instance types of which the characteristics are summarized in Table I. The only types that were excluded for our tests are the resource-limited t1.micro, the GPU based cg1.4xlarge and the most expensive cc2.8xlarge instance type. We have not launched multiple event cores per instance (even if the instance has multiple cores available). This was done to more clearly show the impact of the I/O and memory performance of the different nodes, and how the runtime of the simulation scales with the performance rating of an individual core. Note however, that multiple threads are started even when only a single event core is launched. The additional threads are used for sending and receiving messages and for garbage collection. In practice, this deployment setup leads to loads of 2.5 to 3 on a quad core instance. Optimally packing event cores on instance types, taking into account memory and I/O bandwidth limitations is non-trivial and is left for future work. As a consequence, we will focus on the performance delivered by the different instance types and refrain from comparing instances in terms of cost. In the I/O performance column, L stands for low, M for moderate, H for high and VH for very high. Memory is given in Gigabyte units. An Elastic Compute Unit (ECU) is an EC2-specific unit to express the computational performance of a CPU core. A core with 1 ECU exhibits a performance that is similar to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

<table>
<thead>
<tr>
<th>Name</th>
<th>ECU</th>
<th>Cores</th>
<th>I/O</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>M</td>
<td>1.7</td>
</tr>
<tr>
<td>m1.large</td>
<td>2</td>
<td>2</td>
<td>H</td>
<td>7.5</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>2</td>
<td>4</td>
<td>H</td>
<td>15</td>
</tr>
<tr>
<td>m2.xlarge</td>
<td>3.25</td>
<td>2</td>
<td>M</td>
<td>17.1</td>
</tr>
<tr>
<td>m2.2xlarge</td>
<td>3.25</td>
<td>4</td>
<td>H</td>
<td>34.2</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>3.25</td>
<td>8</td>
<td>H</td>
<td>68.4</td>
</tr>
<tr>
<td>cc1.4xlarge</td>
<td>4.19</td>
<td>8</td>
<td>VH</td>
<td>23</td>
</tr>
</tbody>
</table>

2) Local cluster: Aside from the EC2 offering, we also include performance figures for a local cluster called stewie. The cluster hosts 12 machines with 5 Dual Quad-Core AMD Opteron 2350 processors (listed at 2 GHz) and 7 Dual Quad-Core Intel Xeon L5335 processors (listed at 2 GHz) with at least 16 GB of memory. All these machines are connected in a gigabit ethernet network and run a 2.6 Linux kernel. All Java code was compiled and executed with the Sun Java 1.6.0.22 server distribution.

C. Methodology

We execute both simulation models on all different resource types described in the previous section. For each simulation, one machine is used as the head node, and ten machines are used as worker nodes. Each worker node runs one simulation core. The entities in the simulation are distributed evenly among the different event cores. For each combination of model, resource type, and null-message sending strategy, we execute six simulations in two batches. The allocation (and therefore the network location) of EC2 instances in the Amazon datacenter only changes in between batches and simulation runs for different instance types. That is, the instances are kept alive and new instances are only requested for simulation runs with a different instance type or for a different batch. For protocols where a choice of timeout value is required (TIM and BLOTIM), the results for the best possible timeout value are shown for each instance type.

D. Results

We commence our results discussion by comparing the performance of our local cluster with that of a set of EC2 cluster instances of type cc1.4xlarge for which the CQN model results are shown in Figure 3 (a). This instance type is specifically tailored to coupled HPC computations, offering increased network locality and I/O performance compared to other instance types. These properties are expected to significantly impact performance because event cores in a distributed simulation have a high frequency of message exchange. Our results show that this instance type can approximate the performance of our local cluster and that the relative performance of the different synchronization protocols is retained when launching the application in the cloud. The graph in Figure 3 (b) shows that these remarks also hold for the auction model. Note that the timeout protocol performs best in the CQN scenario, while the blocking protocol dominates in the auction scenario. Neither of both protocols perform well on both simulation models though. The combination of timeout and blocking protocols shows better robustness. For the CQN scenario, it closely approximates the optimal performance as denoted by the ISP runtime, and achieves comparable performance to the blocking protocol for the auction scenario. As the results for the STD protocol show, intelligent null-message sending
can greatly improve performance compared to the standard CMB protocol.

![Figure 3](image.png)

Figure 3. Results for the CQN (a) and Auction (b) models for different null-message protocols and instance types.

The data for m2.4xlarge instance type in Figure 3 also demonstrate that non-cluster instance types can provide a high level of performance. Interestingly, the m2.4xlarge instance type which has a lower quoted I/O and CPU performance, and an equal amount of cores compared to the cc1.4xlarge type (see Table I), performs significantly better than the cluster instance type. It thereby also surpasses the performance of the stewie cluster. We have established that this is not due to the higher amount of memory available on m2.4xlarge as for both scenario’s the amount of consumed heap space remains well below 1.7 GB, the smallest amount of memory available on any of the EC2 instance types.

![Figure 4](image.png)

Figure 4. Execution time and variance for the CQN (a) and Auction (b) model on different instance types using the BLOTIM protocol.

Figure 4 show the performance for the other instance types using the best performing protocol BLOTIM. These results show a performance degradation for the lower-cost instance types and a higher variance. Especially the m2.2xlarge, m1.small and m1.large show significantly higher variance. We have found that this higher level of variance is caused by the fact that on some of the simulation runs, time periods occur in which the simulation exhibits a very low event processing rate. This is illustrated in Figure 5 that depicts the amount of events processed per second over the course of the simulation’s execution time, for two subsequent simulation runs in the same batch. The first run shows a steady event processing rate, while in the second run the rate falls back to a near-zero level in the middle of the simulation’s execution. We have not been able to establish the root cause for this behavior as of yet, but suspect that a throttling mechanism is limiting the rate of messaging that is allowed on the instance. For the other instance types with high variance, the same remarks hold.

![Figure 5](image.png)

Figure 5. Event processing rate for the auction model in two individual runs within the same batch on a m2.xlarge instance.

A possible explanation for the higher variance in the other instance types is the external influence of other loads on the physical server to which the virtual machine of a given instance type is mapped. Indeed instances such as m2.4xlarge more than likely use an entire physical server. Therefore the simulation processes have exclusive access to the network interface and I/O channels of the server. Sharing this interface with other loads can lead to additional delays in communication and memory access.

Note that for the c1.xlarge instance, we were unable to obtain any results for the auction scenario. This scenario has a higher message intensity compared to CQN, and its execution consistently leads to failure as one of the worker nodes becomes unreachable.

![Figure 6](image.png)

Figure 6. Breakdown of the performance across different batches for the m2.xlarge instance.

Finally, Figure 6 shows the breakdown of the execution times across the three batches of distributed simulations executed on the m2.xlarge instance. The timings for the first and second batch illustrate that consistent performance can be attained within a single batch, even if the average
execution time differs highly between the two batches. As the third batch demonstrates however, runs within a batch can also show significant variance. As the results for the ISP protocol indicate, this variance can be attributed to the network and are not caused by varying CPU performance across runs in the batch.

VI. CONCLUSION

Parallel and distributed simulation techniques can improve the round trip time of simulations and the scale or level of detail at which models can be studied. Distributed discrete-event simulation is challenging however, as causality constraints make the problem non-trivial to parallelize. Consequently, null-message sending strategies in conservative time synchronization protocols play a determining role in the performance that can be attained on a distributed infrastructure. This paper presents the first results on how these strategies perform on real cloud infrastructure. Our results indicate that a combination of a timeout and blocking protocol shows the highest overall performance and that the relative performance of the different protocols is attained on cloud infrastructure. Apart from the Amazon EC2 cluster instance types which are specifically tailored to coupled HPC workloads, other high-end instance types also show high performance for our workloads. However, performance variability increases on lower-specced instance types that are likely to share I/O channels with other VMs. Our results show that on these instance types, sudden periods of low event processing activity cause this high level of variance, as opposed to an overall degradation in performance during the simulation.

ACKNOWLEDGMENT

This research has been partially supported by a AWS Research Grant.

REFERENCES


