Research Repository UCD

Title	Synchronisation for Dynamic Load Balancing of Decentralised Conservative Distributed Simulation
Authors(s)	Bragard, Quentin, Ventresque, Anthony, Murphy, Liam, B.E.
Publication date	2014-05-21
Publication information	Bragard, Quentin, Anthony Ventresque, and Liam Murphy B.E. “Synchronisation for Dynamic Load Balancing of Decentralised Conservative Distributed Simulation.” Association for Computing Machinery, May 21, 2014. https://doi.org/10.1145/2601381.2601386.
Conference details	Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Denver, Colorado, USA, 18 - 21 May, 2014
Publisher	Association for Computing Machinery
Item record/more information	http://hdl.handle.net/10197/7146
Publisher's statement	© Bragard, Quentin; Ventresque, Anthony; Murphy, Liam | ACM} {2014}. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation, Denver, Colorado, USA, 18 - 21 May, 2014, http://dx.doi.org/10.1145/10.1145/2601381.2601386.
Publisher's version (DOI)	10.1145/2601381.2601386

Downloaded 2026-05-02 00:28:00

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

Synchronisation for Dynamic Load Balancing of Decentralised Conservative Distributed Simulation

Quentin Bragard, Anthony Ventresque, and Liam Murphy
Lero@UCD, School of Computer Science and Informatics,
University College Dublin, Ireland
Dublin, Ireland
quentin.bragard@ucdconnect.ie, anthony.ventresque@ucd.ie, liam.murphy@ucd.ie

ABSTRACT

Synchronisation mechanisms are essential in distributed simulation. Some systems rely on central units to control the simulation but central units are known to be bottlenecks [10]. If we want to avoid using a central unit to optimise the simulation speed, we lose the capacity to act on the simulation at a global scale. Being able to act on the entire simulation is an important feature which allows to dynamically load-balance a distributed simulation. While some local partitioning algorithms exist [12], their lack of global view reduces their efficiency. Running a global partitioning algorithm without central unit requires a synchronisation of all logical processes (LPs) at the same step.We introduce in this paper two algorithms allowing to synchronise logical processes in a distributed simulation without any central unit. The first algorithm requires the knowledge of some topological properties of the network while the second algorithm works without any requirement. The algorithms are detailed and compared against each other. An evaluation shows the benefits of using a global dynamic load-balancing for distributed simulations.

Keywords

Dynamic load-balancing; Synchronisation; Distributed simulation

1. INTRODUCTION

Distributed simulation is an important tool in many scientific fields. It has been designed to answer to problems such as processing power requirement, volatility or stability. However, using distributed systems instead of a single processor (running a non distributed simulation) leads to issues such as dividing and balancing efficiently the load, minimising the communication cost and synchronising LPs. A lot of work has been done in this area. For instance, [16, 32, 34] present algorithms - respectively called METIS, QuadTree and SParTSim - used to partition a simulated environment. QuadTree partitions the environment into smaller regions according to spatial information. METIS tries to simplify a weighted graph before dividing it, then it projcts the partitions back on the original graph. SParTSim uses traffic characteristics, such as the importance of the roads, the number of vehicles, the number of neighbours per region, to partition road networks. Various work have tackled the problem of maximising the efficiency of the communication in general distributed systems. Message Passing Interface [31] is one of the most known communication system while some work try to develop their own library [22], especially in graph processing [7, 24, 21]. Synchronisation between the different parts of a distributed system is critical and synchronisation mechanisms can be classified in two categories [11, 20, 3, 19]: conservative or optimistic. The former does not allow any causality violations, i.e., any two events have to be processed in the same order in any LP where they are present, while the latter allows LPs to process events even if they may raise contradictions and need to be roll-backed. There is no strict consensus on which synchronisation mechanism is the best.

Once these three characteristics (workload partitioning among the LPs, communication, synchronisation) are set for a distributed simulation, there is still one element that can hamper the efficiency of the system: the system evolves and the load apply to each LP may vary. In that case, some LPs will have more work to compute while the others will be waiting, and the overall simulation will be slower. It is generally accepted that dynamic load-balancing is necessary for distributed simulations [12, 9, 37]. However, those work focus on local dynamic load-balancing as none of them uses a central unit to monitor the distributed simulation. While they offer fair results, they cannot compete with global load-balancing due to their lack of global information. The problem now is that centralised distributed simulations are not always a good solution - as they generate single points of failure for instance.

In this paper we address the question of the feasibility of overall synchronisation for dynamic load balancing of decentralised conservative distributed simulations. In short, (i) LPs can process a step at a time, and wait for their neighbours to finish their steps before processing the next one (*conservative*); (ii) there is *no central entity* to orchestrate/monitor the simulation; and (iii) the load in the system tend to be *imbalanced* after some time. This scenario is common in distributed simulations [37] and to the best of our knowledge there is no adequate solution that addresses it. We propose two novel algorithms that allow to provide an

.

overall synchronisation[1] of the system at a particular step, i.e., stop all the LPs at the same step, so that some global process can happen (repartitioning to dynamically load balance the system in our case): TaSyn and GenSyn. TaSyn requires some topological information on the distributed simulation, such as eccentricity of each node, while GenSyn does not require any additional information and only floods the graph of LPs to figure out the best step to stop the simulation at. We show also in this paper that dynamic load-balancing mechanisms (either using TaSyn of GenSyn) can improve the processing time of a distributed simulation. In particular we find that the time saved using dynamic load-balancing, depending on the sensitivity of the algorithms to load imbalance in the system, is comprised between 7.4% and 15.1% (over 200 steps of simulation).

In the remainder of the paper we explain with more details the requirements of a global synchronisation mechanism (Section 2 and in general what distributed simulation are (Section 3). Section 4 details the two algorithms and compares them while Section 5 shows the impact of a dynamic global load-balancing on a distributed simulation. Finally, Section 6 concludes this paper and presents some future work.

2. MOTIVATION

Several approaches to synchronise distributed systems exist depending on the scale (LANs, WANs) or the kind of system (peer-to-peer, data-centre, clouds computing) [30, 25, 1]. However, those mechanisms are complex and deal with issues such as time-shift or fault-tolerance. While those issues are real at a physical level, they are not relevant in time-discrete distributed simulation, where time is measured logically by the number of steps executed by a LP. We can classify distributed simulations in two categories: distributed simulations with a central unit and distributed simulations where LPs share information locally. At the end of each step, LPs will report their activities to the central unit which will take decisions for the next step for the whole simulation. In this case, synchronising and stopping every LPs at the same step seems straightforward as the central unit knows the state of every LPs and can easily find the first step to stop at. However, distributed simulations with central unit are not always a good solution as the central unit can be a bottleneck. When information are shared locally, each LP communicates directly with its neighbours and LPs do not have a full knowledge of the network.

As workload is transferred through the distributed simulation, some LPs might receive more load than others and, therefore, take more time to complete their work. Each LP depending on the outputs of those slow ones will in turn be slowed down and eventually the whole simulation slows down. In order to prevent this issue, the simulation must find a way to globally balance the load over the LPs. However, without a central entity, this operation can be complex. First of all, a LP has to understand that the distributed simulation could have faster results. Different solutions can be imagined to evaluate a slowdown. The first way can be through synchronisation messages: each time the LPs synchronised, they send a message with their actual load and compare the value with their neighbours or with the average load. It would be possible for a LP to periodically broadcast a message in order to share information about its load. However, if we want this solution to be more efficient than local dynamic load-balancing, we have to broadcast the message far away from its origin which will considerably increase the communication on the network. An ideal solution would be for a LP to be able to evaluate itself without sharing information with the rest of the simulation. While a LP alone cannot evaluate how much slower it is compared to its neighbour, it is possible for a faster LP to evaluate how much faster it is. After a step - or few steps with optimistic synchronisation mechanism - a LP requires to synchronise with its neighbours. Measuring the time between the end of its step and the last neighbour synchronising would give an idea about how faster the LP is. An evaluation can be conducted during the simulation without perturbing the processing just by measuring the slowest response time and compare it with a threshold. If during a couple of consecutive synchronisations, the latency is higher than usual, the problem might come from an unbalanced state of the simulation. Thus, the LP detecting the unusual latency is able to trigger an overall synchronisation.

[1]In this paper we use 'overall synchronisation' or 'synchronisation' for the same process of stopping all LPs at the same step.

3. RELATED WORK

Distributed simulation is the representation of the evolution of a system using multiple LPs working together. They can be used to simulate work such as adaptive mesh refinement [18], car crashes [29], behaviour simulation [5, 8], graph functions [6] and others. To respond to those requirements, distributed simulations can take a lot of forms: continuous or discrete time/space, time-step or event-driven, local or global communication (through a central unit). However, they all share the same basic requirements: a partitioning algorithm to distribute as evenly as possible the load among the LPs, a communication mechanism to allow LPs to exchange information and a synchronisation mechanism to allow LPs to organise themselves when needed.

3.1 Partitioning Algorithms

For the distributed simulation to be efficient, it is important to distribute the load as fairly as possible between each LP. As shown section 5, disparity among the load can have important consequences on the overall simulation time. A lot of research has been conducted to address this problem, leading to numerous algorithms including uniform and nonuniform space partitioning [33, 23, 2, 32], graph partitioning [16, 17] (some designed for specific tasks including road traffic partitioning [34, 36]), car crashes analysis [29] and others.

3.2 Communication Mechanism

The aim of the communication mechanism is to ensure that LPs share information throughout the simulation. It is a critical element as the very idea of LPs working together is based on their capacity to communicate. If the communication library is not efficient enough, it can lead to an important slow-down of the overall simulation. The most used communication library is MPI [31] as it is well-known to be efficient, simple to use and portable. Extensions of MPI have been developed to improve its efficiency. For instance, MPICH-G2 [15] has been designed to handle heterogeneous communication environments where commu-

nication latencies are nonuniform. To minimise the impact of the communication, some work try to load-balance the communication while partitioning the environment [34, 27].

3.3 Synchronisation Mechanism

Synchronisation mechanism is also required for an efficient distributed simulation. It ensures that messages are handled in the correct order of their timestamps. This law is called local causality constraint. Fujimoto [11] categorises synchronisation mechanisms in two classes: conservative synchronisation and optimistic synchronisation, dependant on the distributed simulation and no consensus seems to have emerged on wether to use one or the other.

3.3.1 Conservative Synchronisation

Conservative synchronisation is designed so that the local causality constraint cannot be violated. For instance, conservative time-stepped distributed simulations have to synchronise all LPs after each simulation step. When, and only when, a LP is sure that no message will arrive with a smaller timestamp, it proceeds the next step. In order to improve the communication, work such as [37] have their LPs synchronising only with their neighbours, allowing some time flexibility without violating the local causality constraint.

3.3.2 Optimistic Synchronisation

On the other side, optimistic synchronisation allows LPs to break the local causality constraint. In order to optimise the distributed simulation, LPs limit their synchronisation to every few steps instead of every step. Therefore, it is possible to receive a message which should have been handled at the step t while being already at the step $t+1$. Rollback mechanisms are present to be able to go back to step t and handle the message. However, rollbacks have a heavy cost as the simulation has to reload its previous state and re-start the steps.

4. SYNCHRONISATION

Synchronising a simulation can be a critical task if a LP, or the user, wants to perform an action which can impact the whole distributed simulation. For instance, in distributed simulations, maintaining a good load-balance between LPs is important to get optimal results which force the simulation to balance itself during the simulation. While it is, in some cases, possible to foresee a change and therefore plan a modification of the partitioning before running the simulation, unpredictable load changes require dynamic partitioning. Load-balance can be done locally by exchanging, for instance, vertices or agents with its neighbours [12]. However, the risks are that it takes a while to correct the load-balance and that you increase the number of messages exchanged. Synchronising all LPs would offer the possibility to globally partition the simulation in one shot, saving time and network resources. In order to globally synchronise a time-stepped distributed simulation without using a central entity, two options are possible. Either moving forward in time until a chosen step or rollback until the latest checkpoint. Performing rollback - by state saving, or reverse computation - is a common operation in distributed simulations [28, 35]. However, this operation requires to periodically save the state of the LPs - or the operations executed in case of rollback by reverse computation - which is time-consuming. Moreover, when the simulation has performed a rollback and is load-balanced again, it has to recompute steps which have already been done before. In addition, if rollbacks are indeed useful in distributed simulation using an optimistic synchronisation mechanism, it is not necessary with conservative synchronisation. On the other solution is, for some or all LPs, to go few steps forward. The first algorithm presented in this paper has been designed to quickly propagate the first safe step to stop at, without any global information, based on graph theory properties. The second algorithm presented propagates requests to all LPs in order to gather graph information and find the first possible step to stop at.

4.1 Graph Theory and Distributed Simulation

It is generally accepted that, as distributed simulations - and more generally a distributed systems - are composition of LPs linked together by communication channels, their structure can be compared to a graph with logical processes equivalent to vertices and communication channels equivalent to edges. The similarities can be extended to graph theory concepts such as:

- **Eccentricity** of a vertex is the maximum distance between a vertex and any other vertex. By extension, the eccentricity of a LP will be the maximum distance between a LP and any other LP. For instance, on 1, the eccentricity of F is 3 because it is at a maximum distance of 3 from every LP.
- **Diameter** and **Radius** are respectively the maximum and the minimum eccentricity. On 1, the maximum eccentricity is hold by A, E and H with a value of 4, therefore, the diameter of the graph is 4. In the same way, the minimum eccentricity is hold by C with a value of 2, thus, the radio of the graph is 2.
- **Peripheral vertices** are the vertices with an eccentricity equal to the diameter. In the same way, **central vertices** are the vertices with an eccentricity equal to the radius. The vertices left are called **pseudo-peripheral**. In this paper, we will use the vocabulary **peripheral LPs**, **central LPs** and **pseudo-peripheral LPs** to talk about the vertex equivalents in distributed system.

4.2 TaSyn: Topologically-aware Synchronisation

TaSyn, which stands for Topologically-aware Synchronisation, works using the eccentricity of a graph to find a step at which LPs can synchronise. Time-stepped distributed simulations using local conservative synchronisation allow a small time-flexibility between a LP and its neighbours. For instance, if the LP n_j has just completed the step $t-1$, its neighbour n_i will receive the information required to compute the step t even if n_j is stuck at the end of $t-1$. At the end of n_i's step, its neighbour, n_k will have enough information to compute the step $t+1$ even if n_j has not moved from the end of the step $t-1$. By transition, the difference of steps between two LPs in the simulation is logically smaller or equal to the distance which separate those two LPs. Therefore, the maximum difference of steps between any two LPs is smaller or equal to the eccentricity of those LPs. When computing the eccentricity of a LP, we know the maximum difference of steps between itself and any other LP

in the distributed system. If the eccentricity is known before running the distributed simulation, a LP can safely request a general stop at the following step:

$$synchroStep = e(n_t) + s(n_t) \quad (1)$$

with n_t the LP triggering TaSyn, $e(n_t)$ the eccentricity of n_t and $s(n_t)$ the current step of n_t. If we know the topology of the distributed simulation, algorithms such as Johnson's algorithm [14] offer the possibility to process the eccentricity of each node in the graph.

Algorithm 1: TriggerLP for TaSyn

1 **if** *triggered is false & synchronising is false* **then**
2   synchronising ← true;
3   synchroStep ← eccentricity + currentStep;
4   sendToAll("SYN "+synchroStep);
  `// The LP can continue until reaching synchroStep`

Algorithm 2: Regular LP for TaSyn

1 $\overrightarrow{parents}$ ← new array;
2 **foreach** *msg in msgBox* **do**
3   **if** *msg.startWith("SYN") is true & synchronising is false* **then**
4   synchronising ← true;
5   tempStep ← msg.split(" ")[1];
6 **if** *tempStep < synchroStep* **then**
7   synchroStep ← tempStep;
8     $\overrightarrow{parents}$.add(msg.sender);
9 **foreach** $n \in$ *neighbours* **do**
10   **if** *n is not* $\in \overrightarrow{parents}$ **then**
11   n.send("SYN "+synchroStep);
  `// The LP can continue until reaching synchroStep`

An important question to ask at this stage is: what happens if two LPs trigger the algorithm? If two LPs are at the same step and have the same eccentricity, they will send the same message. If a LP receives two messages at the same time, it will just assume that the two LPs are its parents. If one message comes later, it will just be discarded. When the value carried by the messages is not the same, the LP will select the lowest value as by definition, both values correspond to a safe step to stop at. Figure 1 presents a case of collision between LP F which has an eccentricity of 3 and is at the step 9 and LP H which has an eccentricity of 4 and is at step 10.

TaSyn is simple and straightforward but does not compute the first step where it is possible to stop because the triggering LP does not know at which step the other LPs are. It only knows that they are within the boundary of the constraint.

4.3 GenSyn: General Synchronisation

When it becomes important to stop the distributed simulation as soon as possible, GenSyn, which stands for General

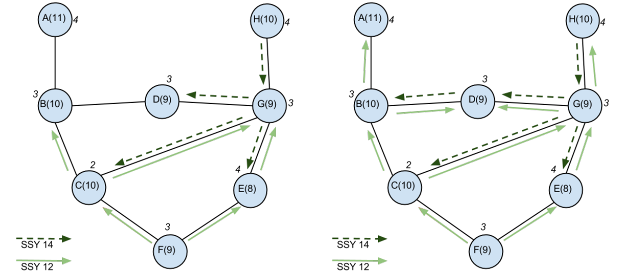


Figure 1: Collision in case of simultaneous trigger by *F* and *H*.

Synchronisation, can find the most ahead LP, and thus the first step to which it is safe to synchronise everyone. The algorithm works in three phases:

- `Propagation (PRP).` The first phase consists in propagating a message through the distributed simulation to find the LP with the highest step. Each LP receiving one or more messages from its neighbours will compare its step with the step contained in each message. The LP will place the highest step in a message and forward it to all its remaining neighbours while the sender(s) id(s) will be stored. If the LP receives a message after it has propagated to its neighbours, the message is discarded. Figure 2(a) presents the propagation of the message if the LP F triggers GenSyn.

- `Answer (ANS).` If after removing its parents, a LP has no other neighbour to propagate the message to, the second phase starts and the LP sends back to its parents an answering message containing the highest step stored. Each LP waits for an answer from all its children before forwarding back its own answer. Figure 2(b) shows how the answer will be sent back to the root LP F.

- `Confirmation (CNF).` Once the triggering LP has received answers from all its neighbours, it sends a confirmation message with the highest step of the distributed simulation. Each neighbour stores the step contained in the confirmation message and continues its work until it reaches this step. Figure 2(c) presents the propagation of the confirmation message.

Algorithm 3: Leader LP for GenSyn

1 **if** *triggered is true & synchronising is false* **then**
2   synchronising ← true;
3   leader ← true;
4   sendToAll("PRP "+currentStep+" "+id);

In case of collisions, the LP arbitrary selects the smaller id between the LPs triggering the algorithm.

4.4 Efficiency of GenSyn and TaSyn

Information about the eccentricity of each LP used by TaSyn makes it far less complex than GenSyn. However, despite the fact that this information is not always available,

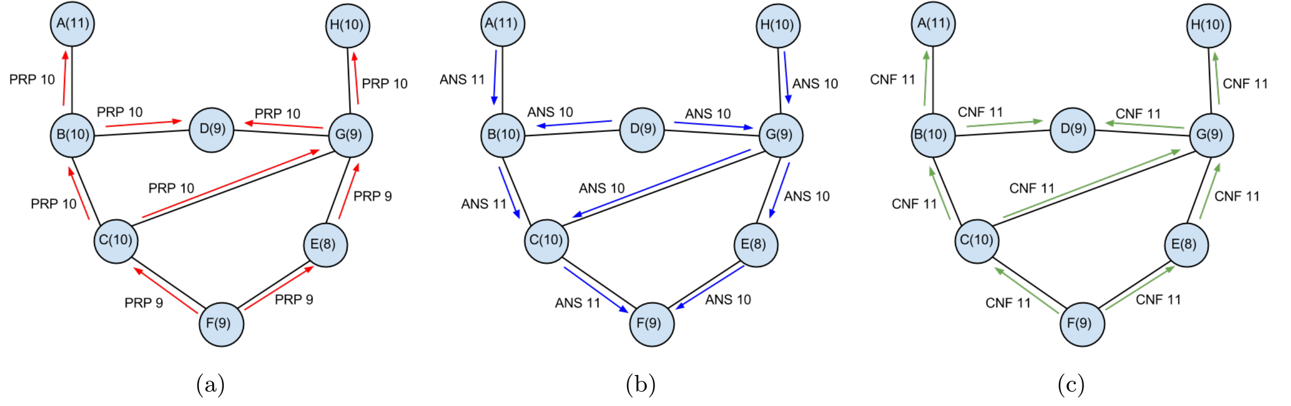


Figure 2: Evolution of the three steps of GenSyn. Each circle represents a LP and is labelled with LP id and latest processed step (in parenthesis).

TaSyn's simplicity does not make it more efficient in every case. To compare the , we consider that each LP has to wait until the global synchronisation is finished to process the remaining steps. In other words, the communication phase and the processing phase will be executed sequentially - in practice, intermediate LPs can start processing steps if they know they are at a lower step compared to the potential maximum (i.e., they received propagation messages with a higher step than their actual one).

To reach the furthest neighbours, TaSyn requires a number of steps equal to the eccentricity of the triggering LP. The remaining step(s) to process for a LP will be the difference between the triggering LP current step added to its eccentricity and the current step of the LP. Therefore, TaSyn communication time ($CommTime_{TaS}$) and processing time ($StepTime_{TaS}$) requirements are:

$$CommTime_{TaS} = t_u \times e(n_t) \tag{2}$$

$$t_s \times e(n_t) \leq StepTime_{TaS} \leq t_s \times (e(n_t) + s(n_t) - min(s(n))) \tag{3}$$

As the communication phase of GenSyn consists in crossing the network 3 times, its communication phase is 3 times longer than TaSyn. The maximal number of steps required before reaching the synchronisation step is given by the difference between the highest step and the lowest step present on the simulation. From that point, GenSyn communication time ($CommTime_{GenS}$) and processing time ($StepTime_{GenS}$) requirements are defined as:

$$CommTime_{GenS} = t_u \times 3 \times e(n_t) \tag{4}$$

$$0 \leq StepTime_{GenS} \leq t_s \times (max(s(n)) - min(s(n))) \tag{5}$$

with $CommTime_{TaS}$ and $CommTime_{GenS}$ respectively TaSyn and GenSyn's communication times, $StepTime_{TaS}$ and $StepTime_{GenS}$ respectively TaSyn and GenSyn's processing times, t_u the time required to send a message to a neighbours, n_t the LP triggering the algorithm, $e(n)$ the eccentricity of the LP n, t_s the time required to process a step and $s(n)$ the step of LP n. The minimum value for the processing time happens when the LPs are all close to be at the same step which means $max(s(n)) - min(s(n)) = 0$, while the maximum processing time occurs when the LPs are at completely different steps and then $max(s(n)) - min(s(n)) = max(e(n))$. For the comparison, we will assume that we are in the worst case, $max(s(n)) - min(s(n)) \rightarrow max(e(n))$ but the process is identical for the case 1 and for the intermediate values. By adding equations 2 to 3 and equations 4 to 5, we obtain the total time required respectively by TaSyn and GenSyn. As we want to compare them according to the topology of the graph, the communication time and the processing time, we can write:

$$TotalTime_{GenS} = TotalTime_{TaS}$$

$$t_u \times 3 \times e(n_t) + t_s \times max(e(n)) = t_u \times e(n_t) + t_s \times 2 \times e(n_t)$$

$$t_u \times (2 \times e(n_t)) = t_s \times (2 \times e(n_t) - max(e(n))) \tag{6}$$

In graph theory, it is proven than $radius \leq diameter \leq 2 \times radius$ [4]. As explained before, the diameter is the maximum eccentricity so $dia = max(e(n))$ and the radius the minimum eccentricity $rad = min(e(n))$. In the case of $dia = 2rad$, if n_t is a pseudo-peripheral LP - $rad \leq e(n_t) \leq dia$ - from equation 6, the relation between the communication time and the processing time is given by:

$$t_u = \frac{t_s \times (2 \times e(n_t) - dia)}{2 \times e(n_t)} \tag{7}$$

If n_t is a central LP - $rad = e(n_t)$ -, as $dia = 2 \times radius$, the equation 7 can be reduced to:

$$t_u = \frac{t_s \times (dia - dia)}{2 \times e(n_t)}$$

$$t_u = 0 \tag{8}$$

Finally, if n_t is a peripheral LP - $dia = e(n_t)$ -, for the same reason, we can reduce equation 7 to:

$$t_u = \frac{t_s \times (2dia - dia)}{2 \times dia}$$

$$t_u = \frac{1}{2} \times t_s \tag{9}$$

Equation 8 proves that, if the LP triggering the algorithm is a central node, TaSyn will always be faster than GenSyn.

Algorithm 4: GenSyn LP for TaSyn

```
1  parents⃗ ← new array;
2  foreach msg in msgBox do
3      tempRootId ← msg.split(" ")[2];
4      if rootId == null then
5          rootId ← tempRootId;
6      else if tempRootId < rootId then
7          rootId ← tempRootId;nbPrp ← 0;nbAns ← 0;
8  foreach msg in msgBox do
9      if msg.startWith("PRP") is true then
10         nbPrp ← nbPrp + 1;
11         tempStep ← msg.split(" ")[1];
12         triggered ← true;
13         parents⃗.add(msg.sender);
14         if synchronising is true then
15             synchronising ← true;
16             currentMaxStep ← step;
17         if tempStep > currentMaxStep then
18             currentMaxStep ← tempStep;
19         if nbPrp == neighbours⃗.size() then
20             foreach p ∈ parents⃗ do
21                 if n is ∈ parents⃗ then
22                     n.send("ANS "+currentMaxStep);
23             parents⃗.clear();
24         else
25             prpToSend ← true;
26     else if msg.startWith("ANS") is true then
27         nbAns ← nbAns + 1;
28         currentStepMax ← msg.split(" ")[1];
29         if leader is true & nbAns == neighbours⃗.size()
           then
30             stoppingStep = currentStepMax;
31             sendToAll("CNF "+stoppingStep);
32         else if nbPrp+nbAns == neighbours⃗.size()
           then
33             foreach p ∈ parents⃗ do
34                 p.send("ANS "+currentMaxStep);
35             parents⃗.clear();
36     else if msg.startWith("CNF") is true &
       confirmation is false then
37         confirmed ← true;
38         stoppingStep ← msg.split(" ")[1];
39         parents⃗.add(msg.sender);
40         foreach n ∈ neighbours⃗ do
41             if n is not ∈ parents⃗ then
42                 n.send("CNF "+stoppingStep);
43 if prpToSend is true then
44     foreach n ∈ neighbours⃗ do
45         if n is not ∈ parents⃗ then
46             n.send("PRP "+currentMaxStep);
```

However, if the triggering LP is a peripheral node, GenSyn will be faster than TaSyn if the communication time is less than twice the processing time. Identically, if the LP trigger is a pseudo-peripheral node, the relation between the communication time and the processing time is given by the equation 7.

Using the same way, if we are in the case of $dia = rad$, all LPs have the same status and the equation resulting is:

$$t_u = t_s \quad (10)$$

Which means that no matter which LP triggers the algorithm, GenSyn will outperform TaSyn if the communication time is smaller than the processing time.

If we are in the case where $rad < dia < 2rad$, by declaring $dia = 2rad - x$ with $0 < x < rad$, we obtain the following equations:

$$t_u = t_s \times \frac{x}{2rad} \quad (11)$$

$$t_u = \frac{t_s}{dia} \quad (12)$$

$$t_u = t_s \times \frac{dia}{2 \times e(n_t)} \quad (13)$$

for a triggering LP respectively a central node 11, a peripheral node 12 and a pseudo-peripheral node 13.

This comparison shows that neither TaSyn nor GenSyn outperforms the other. Therefore, the user can select which algorithm is more suitable for its distributed simulation depending on the following features: communication time, processing time, network topology and the position of the node if known.

5. EXPERIMENTATION

To test their efficiency, we implement our algorithms into the simulator of distributed system Peersim [26] and simulate the behaviour of the distributed simulation.

5.1 PeerSim

PeerSim has been developed to simulate and test distributed (peer-to-peer) systems. Its main features are its scalability, which allows it to run algorithms on hundreds of thousands of nodes quickly, its modularity as all the components are configurable and its graph abstraction, useful to import networks as graphs and use topological information. PeerSim also implements two types of simulations, event-based and cycle-based. We choose to implement our algorithm with the cycle-based simulation as we are studying time-stepped simulations. For each PeerSim cycle, LPs are randomly selected and executed one after another.

5.2 Scenario

We measure the impact of dynamic load-balancing by using a distributed simulation implementing our algorithms against a reference distributed simulation. Our distributed simulation will be able to perform a global dynamic load-balancing but not the reference. We simulate a 200 steps-long stepped-based distributed simulation using Peersim. Each Peersim cycle, the LPs increase a value until they reach their maximum load. It is a simple task representing the LPs doing their part of the simulation. When a LP reaches it maximum load, it sends to its neighbours an

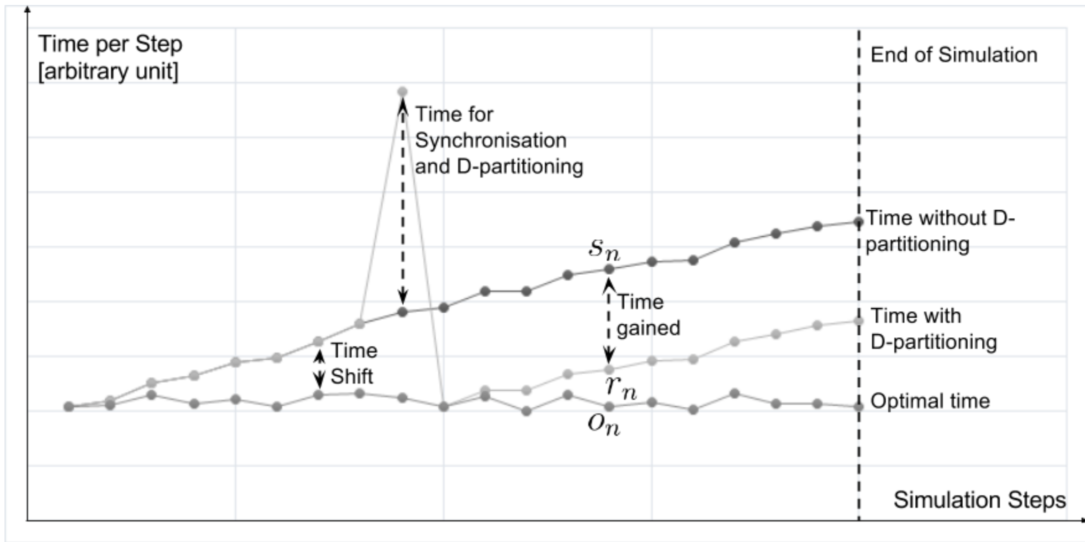


Figure 3: Expected synchronisation impact over the simulation.

end-of-step message and waits until all its neighbours send back the same message. Once the end-of-step messages are gathered, the LP calls a method to increase or decrease its maximum load of 1%, depending of a random value. This phase represents the fact that a distributed simulation seldom stays at the optimal state of a perfect load-balance. Then, the LPs will proceed to the next step with more load, or less load, than its neighbours. It is very likely that, after a while, some LPs will have a lot more, or a lot less, work than others and will slow down the simulation, or have to wait a long time. From this point, the simulations using GenSyn and TaSyn will try to synchronise the distributed simulation in order to perform a dynamic partitioning which will bring-back the load-balanced state. On the other side, the reference simulation will have to continue with its unbalanced state. It will allow us to compare the effect of the dynamic-load balancing against its absence and measure its impact in term of simulation time.

In order to trigger the global synchronisation algorithms, and therefore the dynamic partitioning, a LP will compare the time it requires to simulate a step against the time it has to wait its neighbours. We will set 4 different thresholds in order to compare their impact. We have chosen as thresholds: $StepTime = \{1/2, 1/4, 1/8, 1/16\}\ WaitingTime$. For the scenario of our algorithm, we did not implement any specific partitioning algorithm to run when the LPs are synchronised which could, in some cases, take a certain time. Following the same idea, our LPs communicate by shared memories which cannot be compared, in term of time, with messages sent in the network. However, we will present our results being aware of those two missing constraints.

5.3 Metrics

The first observation we expect is a global impact such as represented by the Figure 3. Optimal time means the time to process the simulation if the system is perfectly load-balanced. As the simulation is losing its load-balanced state, the time want measure the average time required by all the LPs to move to the next step, which means both the processing-time and the waiting-time. This measure will show us the impact of the load-balancing on each step of the simulation and on the overall simulation. The time gained over the whole simulation will be obtained from:

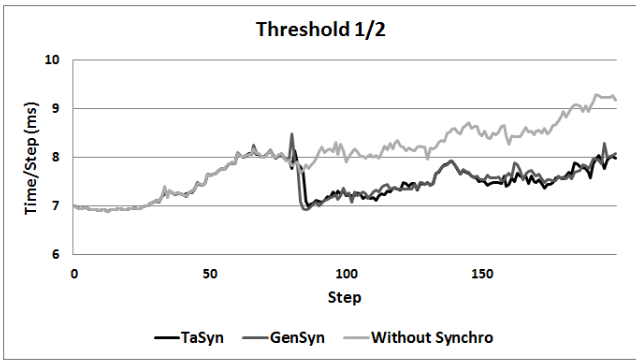
$$TotalTimeGained = \sum_{i=0}^{n} \left(\frac{\sum_{j=0}^{m}(s_j)}{m} - \frac{\sum_{j=0}^{m}(r_j)}{m}\right) \quad (14)$$

where n is the number of steps, m the number of LPs, $s_i \in S$ the values of the simulation without the synchronisation and $r_i \in R$ the values of the simulation including a synchronisation. The different measurements according to each threshold will allow us to evaluate the efficiency of a load-balancing methods regarding the unbalanced state. In order to obtain the efficiency per synchronisation, we will divide the percentage of time gained by the number of synchronisations. The efficiency of the synchronisation will allow us to discuss over the time we can spend running a real partitioning algorithm. Lastly, we will measure and compare the reduction of waiting time obtained thanks to the load-balancing. We will trigger our algorithms using the four different thresholds introduced above.

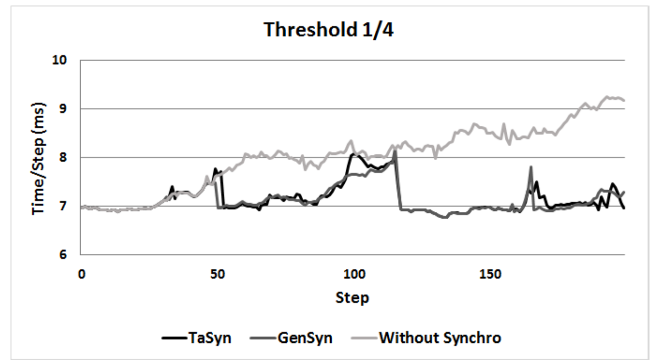
5.4 Results

Our first concern was to observe the behaviour expected from Figure 3 and all the measurements confirm our expectations. Figure 4(a) presents this behaviour around step 80 while figure 4(b) presents it three times around steps 50, 115 and 165. Both figure 4(c) and (d) show this behaviour several times along the simulation. These are a series a great figures that show exactly what we expected: the system diverge to a imbalanced state and both GenSyn and TaSyn allows it to come back to a balanced state, more or less quickly depending on the sensitivity of the algorithms (i.e., threshold).

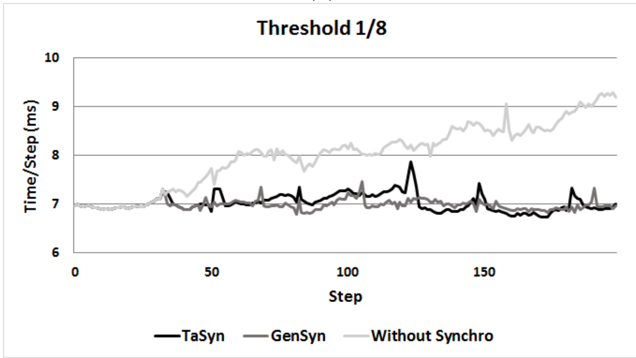
Figures 5(a-d) present the overall time required to execute the 200 steps of the simulation. As expected, a divergence is observed on all figures. Moreover, while the sensitivity of the synchronisation trigger increases, the gap between the reference simulation and the two other keeps increasing. It



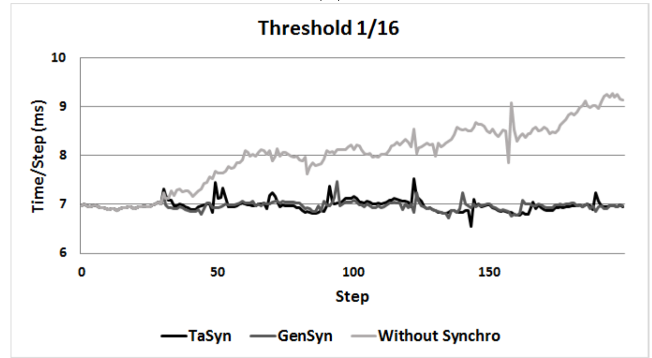
(a)



(b)

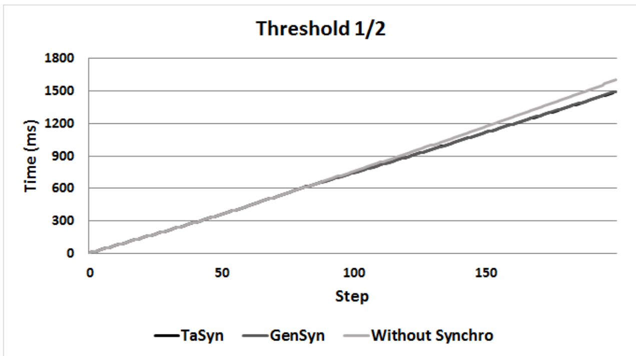


(c)

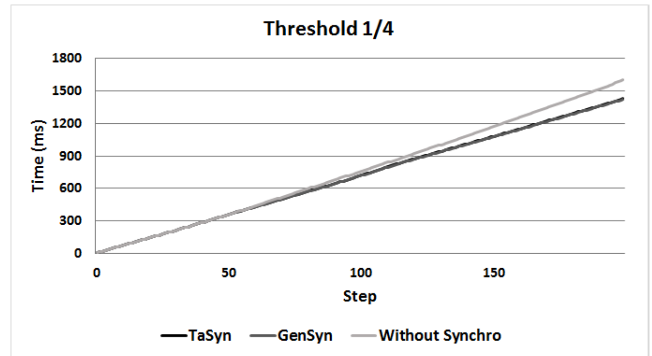


(d)

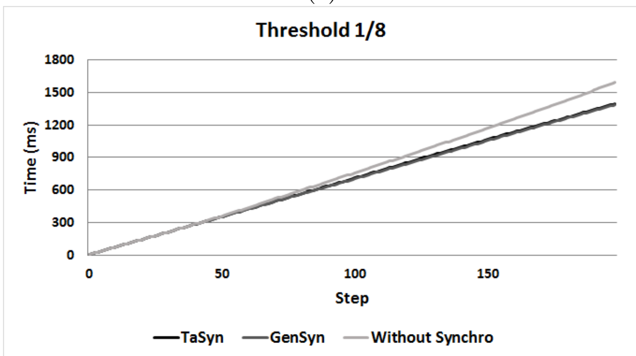
Figure 4: Evolution of the time required per step over the simulation. The lower the better.



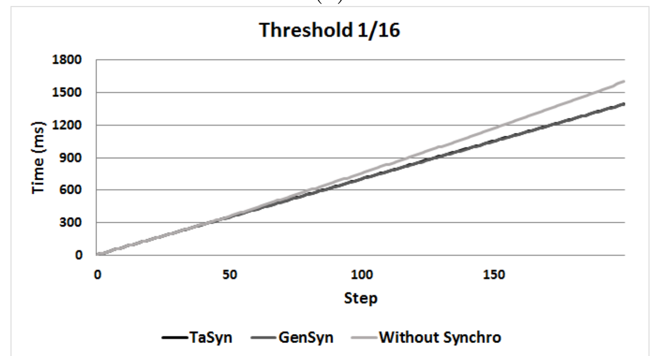
(a)



(b)



(c)



(d)

Figure 5: Simulation time for the whole simulation. The lower the better.

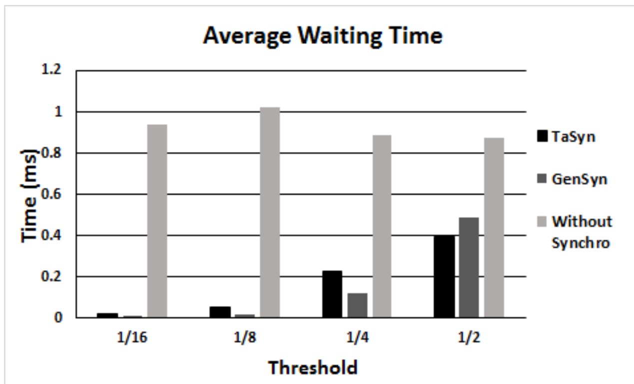


Figure 6: Average waiting time of each case over the four thresholds.

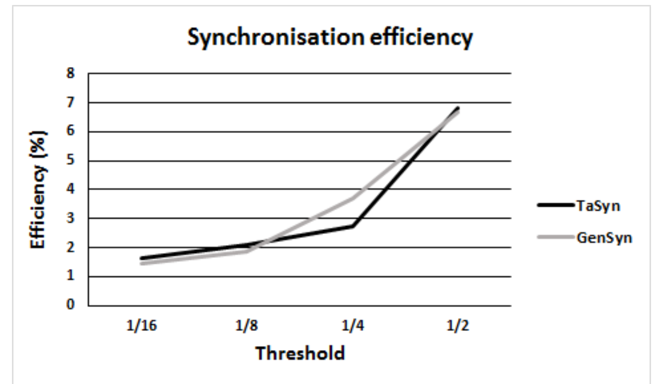


Figure 7: Efficiency of a dynamic partitioning according to the threshold of the trigger.

results from the number of synchronisations as the more the simulation synchronises, the more the load-balance will be brought back closer to it optimal state. We can note as well that the curves are linear which leads to think that the longer the simulation will be, the more time will be gained. We find that the gain is very similar for TaSyn and GenSyn: from 7.4% for a TaSyn and threshold of 1/2 (7.6% for GenSyn), to 15.1% for GenSyn and a threshold of 1/16 (14.9% for TaSyn) - 13.5% for GenSyn and a threshold of 1/4 (13.2% for TaSyn), 14.9% for GenSyn and a threshold of 1/8 (14.5% for TaSyn). This is a very good results as processing time is often the critical metric for distributed simulations.

Figure 6 presents the time lost in average by each LP at each step (we compare to the average time for a simulation without imbalance). At the optimal state, each LP will finish its step at the same time and the waiting time will be limited to the time required to exchange end-of-step messages. While the reference simulation misuses between 0.9ms and 1ms per step, the average waiting-time decreases as we increase the sensitivity of the trigger. If the synchronisation is triggered more often, the simulation will be brought closer to the optimal load-balance state more often.

Efficiency of the synchronisation is an important measure. As we did not implement any time-consuming load-balancing algorithm - what they usually are [13] - we need a way to evaluate how many times we have at our disposition to add the time required for the load balancing and the exchange of synchronisation messages. Figure 7 presents the efficiency of a synchronisation in our different test-cases. On a simulation of 200 steps, the efficiency of a synchronisation would be close to 7% for both TaSyn and GenSyn with a trigger set at 1/2. However, the efficiency decreases rapidly at 3.5% for GenSyn and 2.8% for TaSyn with a trigger set at 1/4. It seems to stabilise around 2% and 1.8% for a trigger respectively at 1/8 and 1/16. In a real situation, we would also have to take care of the efficiency of the load-balancing algorithm used to determine how much time we can allow to it and how many times we can run it during the distributed simulation. However, those results are the proof that there is certainly room for improvement.

6. CONCLUSION

This paper introduces two algorithms TaSyn and GenSyn which are able to synchronise globally a time-stepped conservative distributed simulation. This global synchronisation makes global actions possible, such as a global-scale load-balancing in an environment where information are only shared locally or unpredictable requirement to dump the memory because of imminent failure. GenSyn and TaSyn complexities have been compared and we have proven that their efficiency depends on the structure of the distributed simulation as well as the communication latency and the time required to perform a step. We have also compared, by simulation through PeerSim, a distributed simulation able to perform a global dynamic load-balancing against a distributed simulation which cannot load-balance itself. The results show that dynamically load-balancing a distributed simulation is an important operation which offers great improvements on the simulation time.

As future work we would like to compare our fully decentralised approach and high level architectures (HLA) and investigate whether TaSyn/GenSyn can still be applied (e.g., replace the synchronisation management offered by RTI). We are also interested in implementing our algorithms in distributed simulations (e.g., [5]) to evaluate the overhead of synchronisation and load-balancing on a real system. Eventually, our main target now is to compare global and local load-balancing on the system.

7. ACKNOWLEDGEMENT

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

8. REFERENCES

[1] K. Arvind. Probabilistic clock synchronization in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 5(5):474–487, May 1994.

[2] J. W. Barrus, R. C. Waters, and D. B. Anderson. Locales and Beacons: Efficient and Precise Support For Large Multi-User Virtual Environments. In *VRAIS*, pages 204–213, 1996.

[3] D. W. Bauer Jr., C. D. Carothers, and A. Holder. Scalable time warp on blue gene supercomputers. In *PADS*, pages 35–44, 2009.

[4] B. Bollobás. *Modern Graph Theory*. Springer, Heidelberg, corrected edition, 1998.

[5] Q. Bragard, A. Ventresque, and L. Murphy. dsumo: Towards a distributed sumo. In *First SUMO conference*, Berlin, Germany, 2013.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, Apr. 1998.

[7] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *SIGMOD*, pages 1123–1126, 2010.

[8] I. D. Couzin, J. Krause, N. R. Franks, and S. A. Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433(7025):513–516, 2005.

[9] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2–3):133 – 152, 2005.

[10] A. Ferscha and S. K. Tripathi. Parallel and distributed simulation of discrete event systems. 1998.

[11] R. M. Fujimoto. Parallel and distributed simulation. In *WSC*, pages 122–131, 1999.

[12] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2–4):485 – 500, 2000.

[13] S. Iqbal and G. F. Carey. Performance analysis of dynamic load balancing algorithms with variable number of processors. *Journal of Parallel and Distributed Computing*, 65(8):934 – 948, 2005.

[14] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, Jan. 1977.

[15] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[16] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[17] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[18] B. Kirk, J. Peterson, R. Stogner, and G. Carey. libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3-4):237–254, 2006.

[19] S. Lin, X. Cheng, and J. Lv. Micro-synchronization in conservative parallel network simulation. In *PADS*, pages 195–202, 2008.

[20] J. Liu and R. Rong. Hierarchical composite synchronization. In *PADS*, pages 3–12, July 2012.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[22] E. Lusk and K. Yelick. Languages for high-productivity computing: the darpa hpcs language project. *Parallel Processing Letters*, 17(01):89–102, 2007.

[23] M. R. Macedonia, M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Barham. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. In *IEEE Virtual Reality Annual International Symposium*, pages 2–10, 1995.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[25] D. L. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[26] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 99–100, 2009.

[27] L. Nyland, J. Prins, R. Yun, J. Hermans, H.-C. Kum, and L. Wang. Modeling dynamic load balancing in molecular dynamics to achieve scalable parallel execution. In *Solving Irregularly Structured Problems in Parallel*, volume 1457, pages 356–365. 1998.

[28] K. Perumalla. Parallel and distributed simulation: Traditional techniques and recent advances. In *WSC*, pages 84–95, 2006.

[29] S. Plimpton, SteveAttaway, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Parallel transient dynamics simulations: Algorithms for contact detection and smoothed particle hydrodynamics. *J. Par. Distrib. Computing*, 50:50–1, 1998.

[30] P. Ramanathan, K. Shin, and R. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, Oct 1990.

[31] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[32] A. Steed and R. Abou-Haidar. Partitioning Crowded Virtual Environments. In *VRST*, pages 7–14, 2003.

[33] D. J. Van Hook, S. J. Rak, and J. O. Calvin. Approaches to Relevance Filtering. In *Workshop on Standards for the Interoperability of Distributed Simulations*, pages 26–30, 1994.

[34] A. Ventresque, Q. Bragard, E. S. Liu, D. Nowak, L. Murphy, G. Theodoropoulos, and J. Q. Liu. Spartsim: A space partitioning guided by road network for distributed traffic simulations. In *IEEE/ACM DS-RT*, 2012.

[35] X. Wang, S. Turner, M. Low, and B.-P. Gan. Optimistic synchronization in hla based distributed simulation. In *PADS*, pages 123–130, May 2004.

[36] Y. Xu and G. Tan. An offline road network partitioning solution in distributed transportation simulation. In *IEEE/ACM DR-ST*, pages 210–217, Oct 2012.

[37] T. Zou, G. Wang, M. V. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White. Making time-stepped applications tick in the cloud. In *2Nd ACM Symposium on Cloud Computing*, pages 20:1–20:14, 2011.