Title	Global dynamic load-balancing for decentralised distributed simulation
Authors(s)	Bragard, Quentin; Ventresque, Anthony; Murphy, Liam, B.E.
Publication date	2014-12-10
Conference details	2014 Winter Simulation Conference (WSC), Savanah, Georgia, USA, 7 - 10 December, 2014
Publisher	Institute of Electrical and Electronic Engineers (IEEE)
Item record/more information	http://hdl.handle.net/10197/7141
Publisher's statement	© © 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Publisher's version (DOI)	10.1109/WSC.2014.7020207

Proceedings of the 2014 Winter Simulation Conference
A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.

GLOBAL DYNAMIC LOAD-BALANCING FOR DECENTRALISED DISTRIBUTED SIMULATION

Quentin Bragard
Anthony Ventresque
Liam Murphy

Lero@UCD,
School of Computer Science and Informatics,
University College Dublin, IRELAND

ABSTRACT

Distributed simulations require partitioning mechanisms to operate, and the best partitioning algorithms try to load-balance the partitions. Dynamic load-balancing, i.e. re-partitioning simulation environments at run-time, becomes essential when the load in the partitions change. In decentralised distributed simulation the information needed to dynamically load-balance seems difficult to collect and to our knowledge, all solutions apply a local dynamic load balancing: partitions exchange load only with their neighbours (more loaded partitions to less loaded ones). This limits the effect of the load-balancing. In this paper, we present a global dynamic load-balancing of decentralised distributed simulations. Our algorithm collects information in a decentralised fashion and makes re-balancing decisions based on the load processed by every logical processes. While our algorithm has similar results to others in most cases, we show an improvement of the load-balancing up to 30% in some challenging scenarios against only 12.5% for a local dynamic load-balancing.

1 INTRODUCTION

Simulations are common tools in the scientific community, for instance to observe complex behaviours (Wang, Salles, Sowell, Wang, Cao, Demers, Gehrke, and White 2010). These simulations often require so much computing power that they have to run on several interconnected computers, and we then call them *distributed simulations*. To implement distributed simulations, developers need to divide the simulated environment in partitions and give them to some Logical Processes (LPs) - one partition per LP. Each LP runs its share of the simulated environment and regularly communicates with its neighbours to exchange the position of borderline individuals or individuals that are crossing the borders between partitions. Beside the difficult problems of communication and synchronisation (Snir, Otto, Walker, Dongarra, and Huss-Lederman 1995, Lusk and Yelick 2007, Liu and Rong 2012, Bauer Jr., Carothers, and Holder 2009, Lin, Cheng, and Lv 2008), load-balancing distributed simulations is another major challenge, i.e. giving to each LP an even amount of load to process, so that each LP takes about the same time to process its load.

This has been the focus of an extensive literature in the context of a priori and/or static partitioning (Karypis and Kumar 1995, Zhang, Jiang, and Li 2009, Ventresque, Bragard, Liu, Nowak, Murphy, Theodoropoulos, and Liu 2012). However, while the simulation is running, load can change and the system can reach un-balanced states, where some LPs have more load to process than others. There is in these situations a need for a *dynamic load-balancing* to ensure that each LP has a fair share of the global simulation and that the performance is maintained. In a previous work (Bragard, Ventresque, and Murphy 2014) we showed that the time required per step in decentralised distributed conservative simulations tend to increase naturally as the load on each LP varies over-time.

Another distinction between two classes of distributed simulations is between those using a central entity to manage the simulation and those that do not. Central entities are useful as they can collect information from all the LPs and make decision based on a global view of the system, but they can also a bottleneck and make the system not fault tolerant. Suzumura and Kanezashi (2012), Suzumura, Kato, Imamichi, Takeuchi, Kanezashi, Ide, and Onodera (2012) show for instance that increasing the number of LPs in their centralised system put pressure on the central entity (master node) and increases the synchronisation overhead. In this paper, we consider it is a better option not to have a central entity, which increases the scalability and resilience of our distributed simulations: we are in the context of *decentralised distributed simulations*. The main challenge here is that getting a global view of the distributed system and performing any global process on the distributed system is difficult. This explains why, to the best of our knowledge, all the solutions in the literature focus on local dynamic load-balancing for decentralised distributed simulations (Hendrickson and Devine 2000, Devine, Boman, Heaphy, Hendrickson, Teresco, Faik, Flaherty, and Gervasio 2005, Zou, Wang, Salles, Bindel, Demers, Gehrke, and White 2011). We believe in this paper that these mechanisms do not perform the best load-balancing for decentralised distributed simulations, and that it is possible to obtain a *global dynamic load-balancing for decentralised distributed simulation*.

We focus in this paper on time-stepped conservative distributed simulations, i.e. time is divided in units of time during which each individual in the simulation perform a limited number of actions in the simulation (typically only one action). Synchronisation between LPs is basically done at the end of each step, or after a predefined number of these steps. Even in this context there can be a shift between LPs (one LP processing its steps faster than another LP having too much load, and being ahead in time), hampering the overall simulation performance. In a previous work (Bragard, Ventresque, and Murphy 2014) we described two algorithms to perform a dynamic global synchronisation within a decentralised time-stepped conservative distributed simulation. As we can dynamically synchronise every LPs without central entity, it becomes feasible to gather global information and perform a global dynamic load-balancing techniques.

In this paper, we present a global dynamic load-balancing algorithm for decentralised distributed time-stepped simulations. After having dynamically synchronised every LP using the work described in (Bragard, Ventresque, and Murphy 2014), the LP which has triggered the synchronisation performs a virtual global load-balancing producing an optimised matrix of load exchanges. Then, the triggering LP propagates messages containing local load-exchange. As soon as a LP has performed its exchanges, it can continue the simulation. Results show that while common local dynamic load-balancing algorithms optimise a distributed simulation up to 12.5%, our algorithm can improve the overall simulation performance up to 30%.

In the remainder of the paper we describe with more details the requirements of a global dynamic load-balancing mechanism and the work done in the field of distributed simulation (Section 2). We then present an example of a time-stepped distributed simulation of flock of birds (Section 3). Section 4 details our load-balancing algorithm while sections 5 and 6 shows the impact of our algorithm on our distributed simulation. Finally, Section 7 concludes this paper together with a presentation of some future work.

2 MOTIVATION AND RELATED WORK

Load-balancing, static or dynamic, is a key feature for distributed simulation. In general, it is assumed that the time required to execute a simulation is given by the time required by the slowest LP to process its load. In decentralised distributed simulations, the time-flexibility between each LP allows LPs to catch up a little bit and minimises the impact they had at the previous step. However, the slowest LP at each step is still slowing down the overall simulation and simulation would be faster if all LPs have the same processing time. Static load-balancing algorithms have been created to ensure that the load is balanced over the LPs when the simulation starts. As those algorithms are executed before the simulation starts, it is trivial to provide general information about the overall environment which allows a very efficient load-balancing. On the other hand, dynamic load-balancing is meant to be executed at runtime during a distributed simulation. As decentralised distributed simulations do not use central entities, algorithms can

only rely on local shared information to perform load-balancing which can lead to shift each LPs' view of the global state of the simulation and thus, diminish the efficiency of the load-balancing.

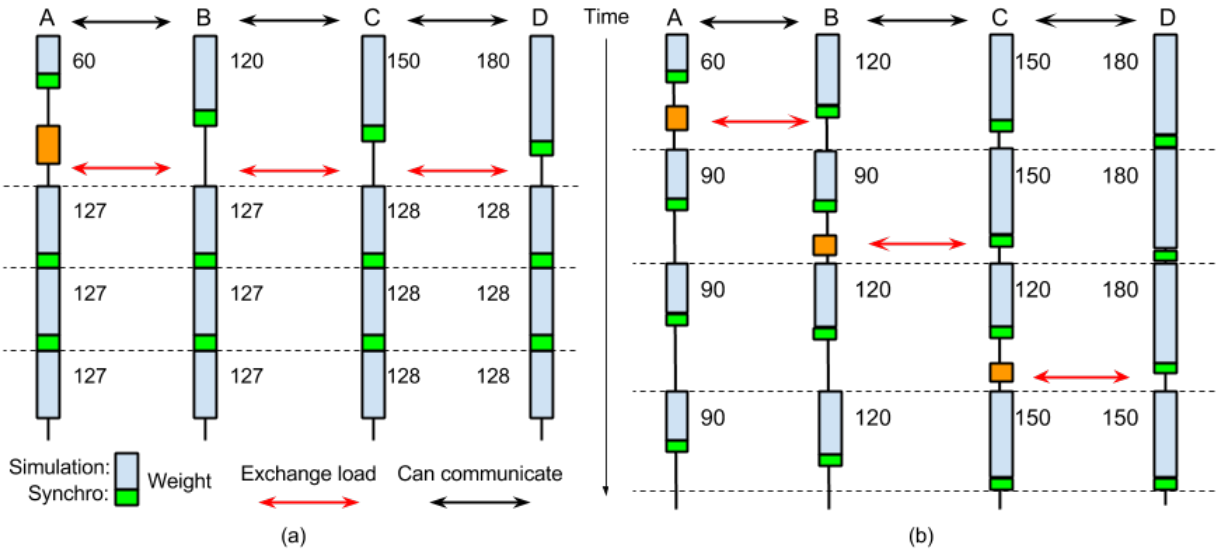

Figure 1: Theoretical impact of global load-balance (a) and local load-balance (b) techniques on a decentralized distributed simulation composed of four LPs.

Figures 1(a) and 1(b) show the theoretical impact that could have global information in comparison to only local information. Both distributed simulations are in an unbalanced state with *A* really under-loaded and *C* and *D* overloaded. However the limits to trigger the load-balancing is reached when the difference between a LP and one of its neighbours reaches 60. Figure 1(b) shows that after 4 steps, the load-balancing is not triggered anymore but the distributed simulation is still unbalanced. On the other hand, as shown on Figure 1(a), global load-balancing techniques tack a more time to gather global information and process them but reaches a load-balanced state after one step. From this figure, the importance of global information over local information is straightforward.

Unlike distributed simulation using local dynamic load-balance algorithms, our decentralised distributed simulation can rely on a global dynamic load-balancing without having to suffer from bottleneck induced by a central entity. While other dynamic load-balancing algorithms gather information locally, the dynamic global synchronisation introduced in our previous work (Bragard, Ventresque, and Murphy 2014) allows to gather of global information and to perform a more accurate load-balancing.

3 DBOIDS

We developped a distributed simulation of flock of birds called dBoids to test different dynamic partitioning algorithms. As birds' behaviour lead them to fly as a group, it induces potential important changes of load when a flock of birds migrates from a LP to another. Therefore, the distributed simulation looses its balance and slows down if no dynamic partitioning is provided.

3.1 Birds Simulation

Behaviour simulation is common in the scientific community to study the evolution of behaviours according to different parameters. Many papers use simulation flock of birds to study their social behaviour (Thompson, Vertinsky, and Krebs 1974), for instance, when predators attack or when then have to manoeuvre around

obstacles, or as benchmark for algorithms (Zou, Wang, Salles, Bindel, Demers, Gehrke, and White 2011). Their behaviour is modeled through three methods:

- The first method is called *Alignment*. When flying in a flock, birds change their own direction according to the directions of their neighbours which makes the birds in the flock fly in a similar direction.
- *Cohesion* is the second method: it makes the birds try to fly close to their neighbours in the flock.
- The last method, *Separation*, makes the birds keep a safety distance between them.

3.2 Distributed Simulation

We choose to develop a time-stepped decentralised distributed simulation of flocks of birds (dBoids) using conservative synchronisation. No central unit is present as it would create an important bottleneck. Each LP communicates only with its neighbours. At the end of each step, LPs send the birds present on their borders to their neighbours as they are required to process the next step. Each LP loads those birds as temporary birds, will not move them and will only keep track of them for this step. These temporary birds cost synchronisation time but no processing time. Another solution would have been to duplicate the borders. We chose to send the temporary (borderline) birds at each step in order to have a processing time dependent only on the number of birds present on each LP as our load-balancing algorithms do not balance the communications but only the load.

3.2.1 Partitioning

The partitioning method used in our work is derived from grid-based partitioning techniques. The algorithm starts by dividing the space into smaller cells. Then it selects cells, based on the number of birds present on them, which will become the seeds of each partition. One after another, the seeds will aggregate a neighbouring cell until there is no more cell available to be aggregated. From that point, the different regions will trade some of their cells in order to increase the load-balancing. A space indexation based on the hierarchical Quadtree data structure is used to traverse quickly the space from a cell to its neighbours.

3.2.2 Border Management

Borders management is not a trivial task and it is critical in distributed simulations. Two main approaches can be found in the literature: duplicate the border (Bragard, Ventresque, and Murphy 2013) or requesting information from its neighbours (Zou, Wang, Salles, Bindel, Demers, Gehrke, and White 2011). Duplicating the border reduces the amount of data exchanged on the network as there is no need to request required information about other agents at every step. However, it implies that agents at the border are processed by more than one LP which reduces the overall performance. The other possibility is for each LP to query its neighbours at every turn to get the information needed. In this situation, each agent is processed by one and only one LP but more data is sent through the network at each step. The choice of one or another type of border management is strongly scenario-dependant. In our distributed simulation, we chose to implement the second solution and send the information about the birds present at the border at each step.

4 DYNAMIC PARTITIONING

Before being able to perform a Global Dynamic Load-balancing, we need to perform a beforehand global synchronisation. In this section we briefly introduce our previous work on Global Dynamic Synchronisation and detail our Global Dynamic Load-balancing algorithm.

4.1 Previous work

Global Dynamic Synchronisation consists in making sure that every LP stops at the same step, without planning it before running the simulation. In our previous work (Bragard, Ventresque, and Murphy 2014) we presented two algorithms, TaSyn and GenSyn, designed to perform GDS. Using those algorithms, when a LP finds out that it is significantly faster than its neighbours, it triggers the most suitable algorithm according to the situation.

4.1.1 TaSyn

TaSyn, stands for Topologically-aware Synchronisation. It is a Global Dynamic Synchronisation algorithm which uses graph properties to ensure that every LP stops at the same step. Decentralised time-stepped simulations allow a difference of one step between a LP and its neighbours as any LP starts and processes its step *n* only after receiving data about the end of step *n-1* from all its neighbours. By transition, the maximum difference, in terms of steps, between any two LPs is equal to the distance between those two LPs. From that, we can deduce that at any time during the simulation, the maximum divergence of steps in the system will be lesser or equal to the eccentricity of the LP triggering the global synchronisation.

Therefore, when a LP triggers a synchronisation, it sends a message to its neighbours requesting a global stop at the step $e(n)+s(n)$ with $e(n)$ and $s(n)$, respectively the eccentricity and the current step of the triggering LP. A message is then be propagated by the neighbours until everyone in the simulation stops. Once everyone has stopped, LPs are able to send back information regarding their current load and which are their neighbours to the triggering LP, giving him global information about the simulation (see below section 4.2).

4.1.2 GenSyn

GenSyn (General Synchronisation) on the other hand, does not require any information to perform a GDS. GenSyn works in three phases:

- The first step is called **Propagation**. The triggering LP sends to its neighbours a message containing its current step. Then, its neighbours compare their current step with the one contained in the message and put the highest number into a new message which is disseminated to their neighbours,etc.
- When the propagation message has reached every LPs, they send back an answer containing the highest step found in their neighbourhood. Messages are sent back until they reach the triggering LP, which then knows the highest step in the whole system. This is the second phase called **Answer**.
- The last phase, **Confirmation**, consists in sending a confirmation message containing the highest step to everyone. Each LP stores the highest step as the step it has to reach and stops at, sends the confirmation message to its neighbours, and then continues to process its part of the simulation normally, until reaching the stopping step.

While sending back their maximum step through the *answer* messages, LPs also add information regarding their load and their direct neighbours in order to give the triggering LP global information about the simulation (see below section 4.2).

4.2 Global Dynamic Load-Balancing

Once every LP in the distributed simulation is synchronised at the same step, we can perform a global dynamic load-balancing. Our algorithm requires load and topological information which can be gathered during the Global Dynamic Synchronisation. While the answers go back to the LP which triggered the synchronisation, each LP adds its id, its neighbourhood and its load to the message. Therefore, when the triggering LP receives the messages, it knows the load of every LP as well as their connections and is able to reproduce the topology of the distributed simulation and optimise the load balance. Because static

Algorithm 1: Global Dynamic Partitioning.

```
1  loads.sort();
2  average ← loads.getAverage();
3  while not isEquilibrate(loads) do
4      biggest ← loads.getBiggest();
5      smallest ← loads.getSmallest();
6      if biggest-average > average - smalles then
7          absorption(smallest,average - smallest,loads);
8      else
9          diffusion(biggest,biggest-average,loads);
10 foreach <from,to> ∈ loadToExchange do
11     if <to,from> ∈ loadToExchange then
12         value1 ← loadToExchange.get(<from,to>);
13         value2 ← loadToExchange.get(<to,from>);
14         if value1 > value2 then
15             loadToExchange.put(<from,to>, (value1-value2));
16             loadToExchange.remove(<to,from>);
17         else
18             if value1 < value2 then
19                 loadToExchange.put(<to,from>, (value2-value1));
20                 loadToExchange.remove(<from,to>);
21             else
22                 loadToExchange.remove(<from,to>);
23                 loadToExchange.remove(<to,from>);
```

partitioning algorithms can be planned before running the simulation, processing time is not an important requirement. However, global dynamic load-balancing needs to be fast in order to lose as minimum time as possible. Establishing new connections between the LPs requires a non-negligible amount of time which is why we want to keep the same network topology. Our global dynamic load-balancing algorithm (See Algorithm 1) works in three phases: virtually load-balancing the distributed simulation, optimising the number of exchanges and propagating the orders of exchange to apply the load-balancing. To load balance the distributed simulation, the algorithm orders the different loads. Then it takes the farthest load from the average, either the smallest or the biggest, and performs respectively either an *absorption* of load or a *diffusion* of load. The *diffusion* (See Algorithm 2), executed by the LP with the biggest load, consists in sending load to their direct neighbours to reduce their overload. On the other hand, the *absorption* of load (which works the same way as *diffusion*), executed by the LP with the smallest loads, consists in taking load from their direct neighbours to compensate their lack of load. The loads are taken, or sent, according to a weight calculated from the actual LP's load. For instance, if an overloaded LP has 2 neighbours, one under-loaded and the other overloaded, it will send its extra load to the under-loaded neighbour. After a diffusion or an absorption, the new loads are calculated and the LPs re-ordered according to their load. The algorithm continues until it reaches an arbitrary level of load-balancing. At the end of this phase, we obtain a matrix of exchanges which leads to an overall load-balanced state. The second phase tries to simplify the matrix of exchanges in order to speed-up the process. The simplification is made by going through the matrix of exchanges and each time we find two, or more, exchanges concerning the same two

LPs, we simplify them to obtain only one exchange. Once the matrix of exchanges reaches its simplest state, the orders are propagate trough the distributed simulation and each LP applies its exchanges.

Algorithm 2: Diffusion of load.

```
1  weight ← new array;
2  sum ← 0;
3  foreach n ∈ neighbours do
4      val ← biggestLoad - n.load; weight.add(val); sum+=val;
5  foreach n ∈ neighbours do
6      currentValue ← n.load;
7      quantityDiffused ← quantityToDiffuse/sum * weight.get(n);
8      newValue ← n.load+ quantityDiffused;
9      loads.put(n,value);
10 loads.sort();
```

5 EXPERIMENTATION

We present in this section the local dynamic load-balancing algorithm (used as baseline) as well as the different scenarios and the metrics chosen.

5.1 Environment

The environment of our distributed simulation is composed of 8 LPs processing a total of 60,000 birds. The space is a square map of $1km^2$, with the border North in connection with the border South, and the border East in connection with the border West, to create an unbounded world. The space is divided in 4,096 cells of $32m^2$ spread over the 8 LPs. Each bird as a speed of $5m.s^{-1}$ and a field of vision with a radius of $10m$.

5.2 Local Dynamic Partitioning

We use a common local dynamic partitioning algorithm (see Algorithm 3) as a baseline (Hendrickson and Devine 2000).

Algorithm 3: Local Dynamic Partitioning.

```
1  if tWaited > threshold then
2      send("RL "+currentLoad, slowestNeighbours);
3  ...
4  data = socket.receive();
5  if data.startWith("RL ") then
6      loadToShare ← (myLoad - data.load)/2;
7      load = getLoadToSend(loadToShare,data.sender);
8      send("NL "+load, data.sender);
```

When a LP spends an abnormally long time waiting for its neighbours to finish their steps, it triggers the local dynamic load-balancing. The method consists in sending a message to the slowest neighbour, requesting it to share its load. The message contains the load processed by the triggering LP. The receiver

of the message compares its current load with the one contained in the message and decides how much load has to be sent in order to locally re-establish the load-balancing. Finally, a message is sent back with the extra-load and both LPs will have similar amount of load to process. In order to select which cells will be sent, the algorithm *getLoadToSend* (see Algorithm 4) goes through all available cells starting with the border between the two partitions. When selecting a cell, we have to ensure that transferring it has no impact on the topology of the distributed simulation.

Algorithm 4: Method getLoadToSend.

```
1  cellsToSend ← new array;
2  available ← new array;
3  foreach cell ∈ borderWith(slowestNeighbours) do
4      if cell.connectedToRegions.size = 1 then
5          cellsToSend.add(cell);
6          available.add(cell);
7  if weighOf(cellsToSend) > loadToShare then
8      return cellsToSend;
9  foreach cell ∈ available do
10     foreach cellNeighbour ∈ getNeighbours(cell) do
11         if cellNeighbour.connectedToRegions.size = 1 then
12             cellsToSend.add(cellNeighbour);
13             available.add(cellNeighbour);
14     if weighOf(cellsToSend) > loadToShare then
15         return cellsToSend;
16     available.remove(cell);
17 return cellsToSend;
```

5.3 Scenarios

We compare the different load-balancing algorithms over 4 scenarios: (i) a stable evolution of the simulation; (ii) a slow increase of the load; (iii) an abrupt increase of the load and (iv) a new target that the birds have to move towards.

For the first scenario, called *Normal*, we leave the simulation running with no modification. The birds fly aimlessly, trying to form flocks but it only results in small changes of the load. The second scenario *Smooth* consists in adding 100 birds on one LP every step for the 20 firsts steps, increasing slowly the load. In the third scenario, *Rough*, we add 2,000 (around 30% of the actual load) birds on one LP at step 20, increasing drastically the load. Finally, in the last scenario, called *Target*, we order all the birds to gather at the centre of the map generating important flocks of birds crossing the border and a drmatic increase of the load for the region managing the centre of the map.

5.4 Metrics

The values we measure are the sum of the maximum and the sum of the average processing time and waiting time per step. The processing time represents the time required to move all the birds on the simulation. The

waiting time, on the other side, represents the time a LP wastes waiting for its neighbours to finish their steps, send their information and the time required to process the information. As we run our distributed simulation on the different cores of our server, the time required to send the messages is negligible. The last metric used is the evenness of the overall simulation. The evenness is computed using the Simpson diversity index (Simpson 1949) and represents the load-balancing of the simulation. The index is computed using the formula:

$$Si = \frac{1}{\sum_{i=1}^{n}(p_i/P)^2 \times n} \tag{1}$$

where n is the number of LPs, p_i the population of the LP i and P the total population. The value obtained is situated within the interval [0,1] with 1 representing the most load-balanced state possible and 0 the worst.

6 RESULTS

In this section we use the words *None*, *Local-LB* and *Global-LB* to refer, respectively, to the simulation without dynamic load-balancing, to the one using local dynamic load-balancing and to the one using global dynamic load-balancing. We also refer to the scenarios by their names *Normal*, *Smooth*, *Rough* and *Target* as defined in section 5. Table 1 and Table 2 gather the final results of each simulation regarding the different scenarios. The first noticeable thing is that there is no mechanism outperforming the others in all scenarios and over the 4 metrics.

Table 1: Sum of the maximum processing time (a) and waiting time (b) per step in millisecond.

Scenario	None	Local-LB	Global-LB
Normal	41,388	39,812	43,533
Smooth	52,174	46,708	52,868
Rough	66,670	62,071	68,802
Target	174,807	149,087	127,961

(a)

Scenario	None	Local-LB	Global-LB
Normal	59,498	88,670	67,895
Smooth	53,732	81,024	63,881
Rough	71,812	86,235	76,652
Target	229,270	206,004	163,708

(b)

Table 2: Sum of the average processing time (a) and waiting time (b) per step in millisecond.

Scenario	None	Local-LB	Global-LB
Normal	30,796	28,878	30,918
Smooth	33,386	30,656	33,929
Rough	34,294	31,890	34,050
Target	52,837	49,817	49,626

(a)

Scenario	None	Local-LB	Global-LB
Normal	28,431	45,249	31,029
Smooth	27,475	43,369	28,226
Rough	41,259	46,770	38,523
Target	167,369	147,064	98,841

(b)

For the first three scenarios, *Local-LB* outperforms *None* and *Global-LB* in term of processing time for the maximum values and the average ones (Tables 1 (a) and 2 (a)). However those performance as an important cost as *Local-LB* is behind the other algorithms in term of waiting time for both values (Tables 1 (b) and 2 (b)).

Target is a real challenge for the three algorithms as suggested by the values in the tables. Processing time and waiting time increase drastically as birds gather on one partition, generating for *Normal* an increase of around 250% while other partitions witness the diminution of their load of approximately 60%. The load-balancing algorithms prove their performance over this scenario as *Local-LB* improves the simulation time by 12.5% when *Global-LB* improves the simulation time by 30%.

When using dynamic load-balancing algorithms, it is important to find a balance between performance and complexity. As shown earlier, *Local-LB* shows better performance than *Global-LB* in terms of processing time but at a cost of a greater waiting time. Every time *Local-LB* or *Global-LB* are triggered, they modify the overall landscape of a region by exchanging small parts of a neighbouring region following an important rule: they cannot change the topology of the distributed simulation. This rule makes the choice of exchangeable cells tricky and it creates regions with complex shapes and long borders. While both algorithms use the same method to find the exchangeable cells, *Global-LB* tries to optimise as much as possible the exchanges of cells which limits their number. On the other hand, *Local-LB* is triggered more often and produces more complex regions increasing the time required to transfer birds. It is only when the scenario becomes complex enough that the gain of performance of *Local-LB* overcomes its complexity. On the other hand, *Global-LB* keeps its complexity low and thus produces acceptable results on simple scenarios while offering better results than *Local-LB* on complex scenarios.

To conclude with Tables 1 and 2, we can see that there is a little overhead in using our load-balancing techniques (both local and global) for simple scenarios while processing time is greatly improved when load changes a lot and simulation is very unbalanced (*Target* scenario), especially with the global load-balancing which outperforms the local load-balancing. This tells us that there is an advantage in using our global load-balancing technique in general, given that the simulation can be slightly slowed down in simple cases. An obvious solution against this overhead would be to add an adaptation module to the load-balancing system, triggering load-balancing only in some specific cases where it is greatly needed.

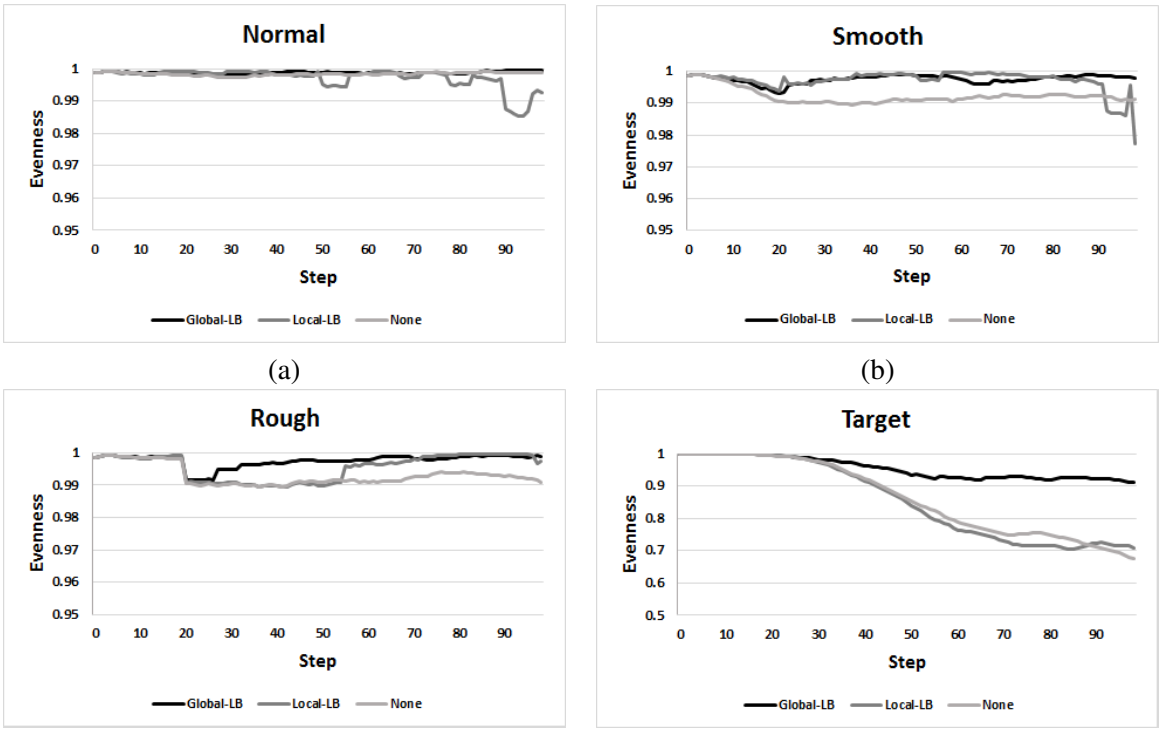

Figure 2: Evolution of the evenness of the simulation.

The evenness of each algorithm in each scenario is presented in Figure 2. Figure 2 (a) shows the results for the first scenario. As expected the evenness stays very high for all algorithms. A small glitch appears at the end of *Local-LB* due probably to a failure to improve the evenness. Figures 2 (b) and 2 (c) show a good response from *Local-LB* and *Global-LB* while *None* stays unbalanced. While *Local-LB* responds to the *Smooth* scenario slightly faster and better than *Global-LB*, it requires far more steps to

load-balance the *Rough* scenario. Its lack of global knowledge limits the optimisation of its performance as it cannot dilute the overloaded LP further than its direct neighbours. We notice again a small glitch at the end of *Local-LB* after a long period when the simulation is load-balanced. Again *Target* offers a real challenge to the algorithms. *Local-LB* has trouble performing at the same time a reduction of the size of the region where all birds are gathering and an extension of the regions the birds are leaving. It results in a load-balancing slightly better than *None*. However, *Local-LB* still outperforms the latter in terms of simulation time because the simulation complexity is exponential and the most crowded region of *Local-LB* is clearly less overloaded than *None*'s most crowded region. *Global-LB* offers really good performance by keeping the evenness above 0.9 while the others go down to 0.7.

7 CONCLUSION

We propose in this paper a global dynamic load-balancing algorithm for time-stepped decentralised distributed simulation. This is an unusual solution to the dynamic load balancing problem in decentralised distributed simulations. In a previous work, we described a global dynamic synchronisation mechanism (Bragard, Ventresque, and Murphy 2014), ensuring that all LPs stop at the same step. The work in the present paper leverages on this property and our algorithm then collects load information about the LPs in the network and decides the loads and the LPs to involve in the load balancing process.

We compare our algorithm again a distributed simulation implementing a basic local load-balancing method and a distributed simulation without dynamic load-balancing. The results show that our algorithm can optimise the load-balancing of a distributed simulation up to 30% for some scenarios, while traditional dynamic local load-balancing achieves 12.5% of optimisation.

As future work, we would like to improve our algorithm by adding two new features: (i) goal-driven load-balancing (as presented in (Wang, Lees, and Cai 2012)) and (ii) communication-balancing , both with the idea to minimise the number of birds crossing the borders. We would also like to perform tests of our global dynamic load-balancing algorithm on event-driven distributed simulations starting by adapting our global synchronisation algorithm GenSyn to this type of simulation. Another interesting direction we plan to follow is the use of techniques inspired from Distributed Interactive Simulation dead reckoning in order to either solve small load unbalance issues or to track global load values in the system.

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)

REFERENCES

Bauer Jr., D. W., C. D. Carothers, and A. Holder. 2009. "Scalable Time Warp on Blue Gene Supercomputers". In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, 35–44: IEEE Computer Society.

Bragard, Q., A. Ventresque, and L. Murphy. 2013. "dSUMO: Towards a Distributed SUMO". In *First SUMO conference*. Berlin, Germany.

Bragard, Q., A. Ventresque, and L. Murphy. 2014. "Synchronisation for Dynamic Load Balancing of Decentralised Conservative Distributed Simulation". PADS.

Devine, K. D., E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. 2005. "New challenges in dynamic load balancing". *Applied Numerical Mathematics* 52 (23): 133 – 152.

Hendrickson, B., and K. Devine. 2000. "Dynamic load balancing in computational mechanics". *Computer Methods in Applied Mechanics and Engineering* 184 (24): 485 – 500.

Karypis, G., and V. Kumar. 1995. "METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0". Technical report.

Lin, S., X. Cheng, and J. Lv. 2008. "Micro-Synchronization in Conservative Parallel Network Simulation". In *PADS*, 195–202.

Liu, J., and R. Rong. 2012, July. "Hierarchical Composite Synchronization". In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, 3–12.

Lusk, E., and K. Yelick. 2007. "Languages for high-productivity computing: the DARPA HPCS language project". *Parallel Processing Letters* 17 (01): 89–102.

Simpson, E. H. 1949. "Measurement of diversity". *Nature* 163 (4148).

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

Suzumura, T., and H. Kanezashi. 2012. "Highly Scalable X10-Based Agent Simulation Platform and Its Application to Large-Scale Traffic Simulation". In *DS-RT*, 243–250.

Suzumura, T., S. Kato, T. Imamichi, M. Takeuchi, H. Kanezashi, T. Ide, and T. Onodera. 2012. "X10-based massive parallel large-scale traffic flow simulation". In *ACM SIGPLAN X10 Workshop*, 31–34. New York, NY, USA: ACM.

Thompson, W. A., I. Vertinsky, and J. R. Krebs. 1974. "The survival value of flocking in birds: a simulation model". *The Journal of Animal Ecology*:785–820.

Ventresque, A., Q. Bragard, E. S. Liu, D. Nowak, L. Murphy, G. Theodoropoulos, and J. Q. Liu. 2012. "SParTSim: A Space Partitioning Guided by Road Network for Distributed Traffic Simulations". In *Conference Distributed Simulation - Real Time*. Dublin, Ireland.

Wang, G., M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White. 2010. "Behavioral simulations in mapreduce". *VLDB*:952–963.

Wang, Y., M. Lees, and W. Cai. 2012, Dec. "Grid-based partitioning for large-scale distributed agent-based crowd simulation". In *Proceedings of the 2012 Winter Simulation Conference*, 1–12. New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Zhang, D., C. Jiang, and S. Li. 2009. "A fast adaptive load balancing method for parallel particle-based simulations". *Simulation Modelling Practice and Theory* 17 (6): 1032–1042.

Zou, T., G. Wang, M. V. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White. 2011. "Making Time-stepped Applications Tick in the Cloud". In *Symposium on Cloud Computing*, 20:1–20:14.

AUTHOR BIOGRAPHIES

QUENTIN BRAGARD is a PhD student in the Performance Engineering Laboratory at University College Dublin, Ireland, under the supervision of Liam Murphy. His main research topics are Road Traffic Partitioning and Distributed Simulation. His email address for these proceedings is quentin.bragard@ucdconnect.ie.

ANTHONY VENTRESQUE, PhD, is a research fellow since 2012 in the Performance Engineering Laboratory of the School of Computer Science & Informatics at University College Dublin, Ireland and in Lero, the Irish Software Engineering Research Centre. He received a PhD in Computer Science in 2008 from University of Nantes, France. He was researcher in INRIA, France, in 2009 and a research fellow in Nanyang Technological University, Singapore, from 2010 till 2011. Dr. Ventresque has expertise in distributed systems: performance, optimisation and testing. His email address for these proceedings is anthony.ventresque@ucd.ie.

LIAM MURPHY received a B.E. in Electrical Engineering from University College Dublin in 1985, and an M.Sc. and Ph.D. in Electrical Engineering and Computer Sciences from the University of California, Berkeley in 1988 and 1992 respectively. He is currently a Professor in Computer Science & Informatics at University College Dublin, where he is Director of the Performance Engineering Laboratory. His email address for these proceedings is liam.murphy@ucd.ie.