Best effort strategy and virtual load for asynchronous iterative load balancing

Raphael Couturier, Arnaud Giersch, Mourad Hakem

Best effort strategy and virtual load for asynchronous iterative load balancing

Raphaël Couturier, Arnaud Giersch*, Mourad Hakem

FEMTO-ST Institute, Univ Bourgogne Franche-Comté, Belfort, France

Abstract

Most of the time, asynchronous load balancing algorithms are extensively studied from a theoretical point of view. The Bertsekas and Tsitsiklis' algorithm [1] is undeniably the best known algorithm for which the asymptotic convergence proof is given. From a practical point of view, when a node needs to balance a part of its load to some of its neighbors, the algorithm's description is unfortunately too succinct, and no details are given on what is really sent and how the load balancing decisions are made. In this paper, we propose a new strategy called *best effort* which aims at balancing the load of a node to all its less loaded neighbors while ensuring that all involved nodes by the load balancing phase have the same amount of load. Moreover, since asynchronous iterative algorithms are less sensitive to communication delays and their variations [2], both load transfer and load information messages are dissociated. To speedup the convergence time of the load balancing process, we propose *a clairvoyant virtual load* heuristic. This heuristic allows a node receiving a load information message to integrate the future virtual load (if any) in its load's list, even if the load has not been received yet. This leads to have predictive snapshots of nodes' loads at each iteration of the load balancing process. Consequently, the notified node sends a real part of its load to some of its neighbors, taking into account the virtual load it will receive in the subsequent time-steps. Based on the SimGrid simulator, some series of test-bed scenarios are considered and several QoS metrics are evaluated to show the usefulness of the proposed algorithm.

1. Introduction

Load balancing algorithms are widely used in parallel and distributed applications to achieve high performances in terms of response time, throughput and resources usage. They play an important role and arise in various fields ranging from parallel and distributed computing systems to wireless sensor networks (WSN). The objective of

*Corresponding author.

Email addresses: raphael.couturier@univ-fcomte.fr (Raphaël Couturier), arnaud.giersch@univ-fcomte.fr (Arnaud Giersch), mourad.hakem@univ-fcomte.fr (Mourad Hakem)

load balancing is to orchestrate the distribution of the global load so that the load difference between the computational resources of the network is minimized as much as possible. Unfortunately, this problem is known to be **NP-hard** in its general form and heuristics are required to achieve sub-optimal solutions but in polynomial time complexity.

In this paper, we focus on asynchronous load balancing of non negative real numbers of *divisible loads* in homogeneous distributed systems. Loads can be divided in arbitrary *fine-grain* parallel parts size that can be processed independently of each other [3, 4, 5]. This model of divisible loads arises in a wide range of real-world applications. Common examples, among many, include signal processing, feature extraction and edge detection in image processing, records search in huge databases, average consensus in WSN, pattern search in Big data and so on.

In the literature, the problem of load balancing has been formulated and studied in various ways. The first pioneering work is due to Bertsekas and Tsitsiklis [1]. Under some specific hypothesis and *ping-pong* awareness conditions (see section 4 for more details), an asymptotic convergence proof is derived.

Although Bertsekas and Tsitsiklis describe the necessary conditions to ensure the algorithm's convergence, there is no indication nor any strategy to really implement the load distribution. Consequently, we propose a new strategy called *best effort* that tries to balance the load of a node to all its less loaded neighbors while ensuring that all the nodes involved in the load balancing phase have the same amount of load. Moreover, most of the time, it is simpler to dissociate load information messages from data migration messages. Former ones allow a node to inform its neighbors about its current load. These messages are in fact very small and can often be sent very quickly. For example, if a computing iteration takes a significant time (ranging from seconds to minutes), it is possible to send a new load information message to each involved neighbor at each iteration. The load is then sent, but the reception may take time when the amount of load is huge and when communication links are slow. Depending on the application, it may or may not make sense for the nodes to try to balance a part of their load at each computing iteration. But the time needed to transfer a load message from one node to another is often much longer than the time needed to transfer a load information message. So, when a node is notified that later it will receive a data message, it can take this information into account in its load's queue list for preventive purposes. Consequently, it can send a part of its predictive load to some of its neighbors if required. We call this trick the *clairvoyant virtual load* transfer mechanism.

The main contributions and novelties of our work are summarized in the following section.

2. Our contributions

- We propose a *best effort strategy* which proceeds greedily to achieve efficient local neighborhoods equilibrium. Upon local load imbalance detection, a *significant amount* of load is moved from a highly loaded node (initiator) to less loaded neighbors.

- Unlike earlier works, we use a new concept of virtual loads transfer which allows nodes to predict the future loads they will receive in the subsequent iterations. This leads to a noticeable speedup of the global convergence time of the load balancing process.

- The SimGrid simulator, which is known to handle realistic models of computation and communication in different types of platforms was used. Taking into account both loads transfers' costs and network contention is essential and has a real impact on the quality of the load balancing performances.

The reminder of the paper is organized as follows. Section 3 offers a review of the relevant approaches in the literature. Section 4 describes the Bertsekas and Tsitsiklis' asynchronous load balancing algorithm. Section 5 presents the best effort strategy which provides efficient local loads equilibrium. In Section 6, the clairvoyant virtual load scheme is proposed to speedup the convergence time of the load balancing process. In Section 7, a comprehensive set of numerical results that exhibit the usefulness of our proposal when dealing with realistic models of computation and communication is provided. Finally, some concluding remarks are made in Section 8.

3. Related works

In this section, the relevant techniques proposed in the literature to tackle the problem of load balancing in a general context of distributed systems are reviewed.

As pointed above, the most interesting approach to this issue has been proposed by Bertsekas and Tsitsiklis [1]. This algorithm, which is outlined in Section 4 for the sake of comparison, has been borrowed and adapted in many works. For instance, in [6] a static load balancing (called DASUD) for non negative integer number of divisible loads in arbitrary networks topologies is investigated. The term *"static"* stems from the fact that no loads are added or consumed during the load balancing process. The theoretical correctness proofs of the convergence property are given. Some generalizations of the same authors' own work for partially asynchronous discrete load balancing model are presented in [7]. The authors prove that the algorithm's convergence is finite and bounded by the straightforward network's diameter of the global equilibrium threshold in the network. In [8], a fault tolerant communication version is addressed to deal with average consensus in wireless sensor networks. The objective is to have all nodes converging to the average of their initial measurements based only on nodes' local information. A slight adaptation is also considered in [9] for dynamic networks with bounded delays asynchronous diffusion. The dynamical aspect stands at the communication level as the links between the network's resources may be intermittent.

Cybenko [10] proposes a *diffusion* approach for hypercube multiprocessor networks. The author targets both static and dynamic random models of work distribution. The convergence proof is derived based on the *eigenstructure* of the iteration matrices that arise in load balancing of equal amount of computational works. A static load balancing for both synchronous and asynchronous ring networks is addressed in [11]. The authors assume that at any time step, one token at the most (units of load) can be transmitted along any edge of the ring and no tokens are created during the balancing phase.

They show that for every initial token distribution, the proposed algorithm converges to the stable equilibrium with tighter linear bounds of time step-complexity.

In order to achieve the load balancing of cloud data centers, a LB technique based on Bayes theorem and Clustering is proposed in [12]. The main idea of this approach is that the Bayes theorem is combined with the clustering process to obtain the optimal clustering set of physical target hosts leading to the overall load balancing equilibrium. Bidding is a market-technique for task scheduling and load balancing in distributed systems that characterize a set of negotiation rules for users' jobs. For instance, Izakian et al [13] formulate a double auction mechanism for tasks-resources matching in grid computing environments where resources are considered as provider agents and users as consumer ones. Each entity participates in the network independently and makes autonomous decisions. A provider agent determines its bid price based on its current workload and each consumer agent defines its bid value based on two main parameters: the average remaining time and the remaining resources for bidding. Based on JADE simulator, the proposed algorithm exhibits better performances in terms of successful execution rates, resource utilization rates and fair profit allocation.

Choi et al. [14] address the problem of robust task allocation in arbitrary networks. The proposed approaches combine a bidding approach for task selection and a consensus procedure scheme for decentralized conflict resolution. The developed algorithms are proven to converge to a conflict-free assignment in both single and multiple task assignment problem. An online stochastic dual gradient LB algorithm, which is called DGLB, is proposed in [15]. The authors deal with both workload and energy management for cloud networks consisting of multiple geo-distributed mapping nodes and data Centers. To enable online distributed implementation, tasks are decomposed both across time and space by leveraging a dual decomposition approach. Experimental results corroborate the merits of the proposed algorithm.

In [16] a LB algorithm based on game theory is proposed for distributed data centers. The authors formulate the LB problem as a non-cooperative game among front-end proxy servers and characterize the structure of Nash equilibrium. Based on the obtained Nash equilibrium structure, they derive a LB algorithm to compute the Nash equilibrium. They show through simulations that the proposed algorithm ensures fairness among the users and a good average latency across all client regions. A hybrid task scheduling and load balancing dependent and independent tasks for master-slaves platforms are addressed in [17]. To minimize the response time of the submitted jobs, the proposed algorithm which is called DeMS is split into three stages: i) communication overhead reduction between masters and slaves, ii) task migration to keep the workload balanced iii) and precedence task graphs partitioning.

Several LB techniques, based on artificial intelligence, have also been proposed in the literature: genetic algorithm (GA) [18], honey bee behavior [19, 20], tabu search [18] and fuzzy logic [21]. The main strength of these techniques comes from their ability to seek in large search spaces, which arises in many combinatorial optimization problems. For instance, the works in [22, 23] have proposed to tackle the load balancing problem using the multi-agent approach where each agent is responsible for the load balancing of a subset of nodes in the network. The objective of the agent is to minimize the jobs' response time and the host idle time dynamically. In [24], the authors formulate the load balancing problem as a non-cooperative game among users.

They use the Nash equilibrium as the solution of this game to optimize the response time of all jobs in the entire system. The proposed scheme guarantees an optimal task allocation for each user with low time complexity. A game theoretic approach to tackle the static load balancing problem is also investigated in [25]. To provide fairness to all users in the system, the load balancing problem is formulated as a non-cooperative game among the users to minimize the response time of the submitted users' jobs. As in [24], the authors use the concept of Nash equilibrium as the solution of a non-cooperative game. Simulation results show that the proposed scheme offers near optimal solutions compared to other existing techniques in terms of fairness.

4. Bertsekas and Tsitsiklis' asynchronous load balancing algorithm

In this section, a brief description of Bertsekas and Tsitsiklis' algorithm [1] is given, using its original notations. A network is modeled as a connected undirected graph $G = (N,A)$, where N is a set of processors and A is a set of communication links. The processors are labeled $i = 1,...,n$, and a link between processors i and j is denoted by $(i, j) \in A$. The set of processor i's neighbors is denoted by $V(i)$.

The load of processor i at time t is represented by $x_i(t) \geq 0$. Each processor i has an estimate of the load of each of its neighbors $j \in V(i)$ denoted by $x_j^i(t)$. This estimate may be outdated due to asynchronism and communication delays.

When a processor sends a part of its load to one or to some of its neighbors, the transfer takes time to be completed. Let $s_{ij}(t)$ be the amount of load that processor i has transferred to processor j at time t and let $r_{ij}(t)$ be the amount of load received by j from i at time t. Then the amount of load of processor i at time $t+1$ is given by:

$$x_i(t+1) = x_i(t) - \sum_{j \in V(i)} s_{ij}(t) + \sum_{j \in V(i)} r_{ji}(t) \tag{1}$$

The asymptotic convergence is derived based on the *ping-pong* awareness condition which specifies that:

$$x_i(t) - \sum_{k \in V(i)} s_{ik}(t) \geq x_j^i(t) + s_{ij}(t) \tag{2}$$

for any processor i and any $j \in V(i)$ such that $x_i(t) > x_j^i(t)$.

This condition prohibits the possibility that two nodes keep sending loads to each other back and forth, without reaching equilibrium.

Nevertheless, we think that this condition may lead to deadlocks in some cases. For example, let us consider a linear chain graph network of only three processors in which processor 1 is linked to processor 2 which is also linked to processor 3, but in which processors 1 and 3 are not neighbors.

Now consider that we have the following load values at time t:

$$x_1(t) = 10$$
$$x_2(t) = 100$$
$$x_3(t) = 99.99$$
$$x_3^2(t) = 99.99$$

Owing to the algorithm's specifications, processor 2 can either send a part of its load to processor 1 or to processor 3. If it sends its load to processor 1, it will not satisfy condition (1) because after that sending it will be less loaded than $x_3^2(t)$. So we consider that the *ping-pong* condition is probably too strong.

In spite of this, a weaker condition can be conjectured to exist since there does not seem to be any scenario that does not lead to convergence, even with load-balancing strategies that are not exactly fulfilling the authors' own conditions.

Even though this approach is interesting, several practical questions arise when dealing with realistic models of computation and communication. As reported above, the algorithm's description is too succinct and no details are given on what is really sent and how the load balancing decisions are made. To our knowledge, the only first attempt for a possible implementation of this algorithm is investigated in [8] under the same conditions. Thus, in order to assess the performances of the new *best effort*, it seemed natural to compare it with this previous work. More precisely, algorithm 2 from [8] will be used and, throughout the paper, will be referenced under the original name *Bertsekas and Tsitsiklis* for the sake of convenience and readability.

Here is an outline of the main principle of the borrowed algorithm. When a given node i has to take a load balancing decision, it starts by sorting its neighbors by non-increasing order of their loads. Then, it computes the difference between its own load, and the load of each of its neighbors. Finally, taking the neighbors following the order defined before, the amount of load to send s_{ij} is computed as $1/(|V(i)|+1)$ of the load difference. This process is iterated as long as a node is more loaded than its considered neighbors.

5. Best effort strategy

In this section, a new load-balancing strategy that is called *best effort* is described. First, the general idea behind this strategy is given, and then some variants of this basic strategy are presented.

5.1. Basic strategy

The description of our algorithm will be given from the point of view of a processor i. The principle of the *best effort* strategy is that each processor detecting itself to be more loaded than some of its neighbors, sends part of its load to its less loaded neighbors, doing its best to reach the equilibrium between the involved neighbors and itself.

More precisely, at each iteration of the load balancing process, processor i proceeds as follows.

1. First, the neighbors are sorted in non-decreasing order of their known loads $x_j^i(t)$.
2. Then, this sorted list is used to find its largest prefix such as the load of each selected neighbor is smaller than:
   - the load of processor i, and
   - the mean of the loads of the selected neighbors and processor i.

   Let $S_i(t)$ be the set of the selected neighbors, and $\bar{x}(t)$ be the mean of the loads between the selected neighbors and processor i which is given as follows:

$$\bar{x}(t) = \frac{1}{|S_i(t)|+1}\left(x_i(t) + \sum_{j \in S_i(t)} x_j^i(t)\right)$$

   so that the following properties hold:

$$\begin{cases} S_i(t) \subset V(i) & \\ x_j^i(t) < x_i(t) & \forall j \in S_i(t) \\ x_j^i(t) < \bar{x} & \forall j \in S_i(t) \\ x_j^i(t) \leq x_k^i(t) & \forall j \in S_i(t), \forall k \in V(i) \setminus S_i(t) \\ \bar{x} \leq x_i(t) & \end{cases}$$

3. Once this selection is done, processor i sends to each selected neighbor $j \in S_i(t)$ an amount of load $s_{ij}(t) = \bar{x} - x_j^i(t)$.

   In this way we obtain:

$$\begin{cases} x_i(t) - \sum_{j \in S_i(t)} s_{ij}(t) = \bar{x} & \\ x_j^i(t) + s_{ij}(t) = \bar{x} & \forall j \in S_i(t) \end{cases}$$

5.2. Leveling the amount of load to move

With the aforementioned basic strategy, each node does its best to reach the equilibrium with its neighbors. However, one question should be outlined here: how to handle the case where two (or more) node initiators might concurrently send some loads to the same least loaded neighbor? Indeed, there is a risk that a node will receive loads from several of its neighbors, and then might temporary go off the equilibrium state. This is particularly true with strongly connected applications.

In order to reduce this effect, the ability to level the amount of load to send is added. The idea, here, is to make as few steps as possible toward the equilibrium, such that a potentially unsuitable decision pointed above has a lower impact on the local equilibrium. A weighting system parameter k is introduced to orchestrate the right balance between the topology structure and the computation to communication ratios (CCR) values of the deployed application. Indeed, to speedup the convergence time of the load balancing process, one is faced with a difficult trade-off to choose an appropriate amount of load to send between node neighbors upon load imbalance detection. On the one hand, if k is small, faster convergence times are expected for sparsely connected applications and large CCR values. On the other hand, for strongly

connected applications and small CCR values, a large value of k will enable us to better balance the load locally and therefore minimize the number of iterations toward the global equilibrium. In the experiments section (Section 7.3), it can be observed that choosing k in $\{1,2,4\}$ leads to good results for the considered CCR values and the targeted topology structures. So the amount of data to send is then $s_{ij}(t) = (\bar{x} - x_j^i(t))/k$.

6. Virtual load

In this section, the new concept of *virtual load* is presented. It aims at improving the global convergence time. For that purpose, both load transfer messages and load information messages are dissociated. More precisely, a node wanting to send some amount of its load to one (or more) of its neighbors can first send a load information message about the load it will send, and later it can send the load message containing data to be transferred. Load information messages are in fact short and will be received soon. In contrast, load transfer messages are often larger ones and thus require more time to be transferred.

The concept of *virtual load* allows a node receiving a load information message to integrate (virtually) the future load it will receive later in its load's list even if the load has not been received yet. Consequently, the notified node can send a (real) part of its load to some of its neighbors when needed. By and large, this allows a node on the one hand, to predict the load it will receive in the subsequent time steps, and on the other hand, to make suitable decisions when detecting load imbalance in its closed neighborhoods. Doing so, faster convergence times are expected since nodes can take into account the information about the predictive loads even if these have not yet been received.

7. Implementation with SimGrid and simulations

In order to test and validate our approach, a simulator using the SimGrid framework [26, 27] was written. This simulator, which consists of about 2,700 lines of C++, allows to run the different load-balancing strategies under various parameters, such as the initial distribution of load, the interconnection topology, the characteristics of the running platform, etc. Then several metrics were considered to assess and compare the behavior of the different strategies.

The simulation model is detailed in the next section (7.1), and the experimental contexts are described in section 7.2. Then the results of the simulations are presented in section 7.3.

7.1. Simulation model

In the simulation model the processors exchange messages which are of two types. First, there are *control messages* which only carry the information exchanged between processors, such as the current load, or the virtual load transfers if this option is considered. These messages are rather small, and their size is constant. Then, there are *data messages* that carry the real load transferred between processors. The size of a data

message is a function of the amount of load that it carries, and it can be pretty large. In order to receive the messages, each processor has two receiving channels, one for each type of messages. Finally, when a message is sent or received, this is done by using non-blocking primitives of SimGrid[1].

During the simulation, each processor concurrently runs three threads: a *receiving thread*, a *computing thread*, and a *load-balancing thread*, which will be briefly described hereafter.

For the sake of simplicity, a few details were voluntary omitted from these descriptions. For an exhaustive presentation, interested readers are referred to the actual source code that was used for the experiments[2], and which is available at `http://info.iut-bm.univ-fcomte.fr/staff/giersch/software/loba.tar.gz`.

7.1.1. Receiving thread

The receiving thread is in charge of waiting for incoming messages, either on the control channel, or on the data channel. Its behavior is sketched by Algorithm 1. When a message is received, it is pushed in a buffer of received messages, to be later consumed by one of the other threads. There are two such buffers, one for the control messages, and one for the data messages. The buffers are implemented with first-in, first-out queues (FIFO).

Algorithm 1: Receiving thread

```
Data: ctrl_chan, data_chan: communication channels (control and data)
      ctrl_fifo, data_fifo  : buffers of received messages (control and data)
while true do
    wait for a message to be available on either ctrl_chan, or data_chan;
    if a message is available on ctrl_chan then
        get the message from ctrl_chan, and push it into ctrl_fifo;
    end
    if a message is available on data_chan then
        get the message from data_chan, and push it into data_fifo;
    end
end
```

7.1.2. Computing thread

The computing thread is in charge of the real load management. As outlined in Algorithm 2, it iteratively runs the following operations:

- if some load was received from the neighbors, get it;

[1] That are `MSG_task_isend()`, and `MSG_task_irecv()`.

[2] As mentioned before, our simulator relies on the SimGrid framework [27]. For the experiments, we used a pre-release of SimGrid 3.7 (Git commit 67d62fca5bdee96f590c942b50021cdde5ce0c07, available from `https://github.com/simgrid/simgrid`)

- if there is some load to send to the neighbors, send it;
- run some computations, whose duration is a function of the processor's current load.

Practically, after the computation, the computing thread waits for a small amount of time if the iterations are looping too fast (for example, when the current load is near zero).

Algorithm 2: Computing thread

```
Data: data_fifo : buffer of received data messages
      real_load: current load
while true do
    if data_fifo is empty and real_load = 0 then
        wait until a message is pushed into data_fifo;
    end
    while data_fifo is not empty do
        pop a message from data_fifo;
        get the load embedded in the message, and add it to real_load;
    end
    foreach neighbor n do
        if there is some amount of load a to send to n then
            send a units of load to n, and subtract it from real_load;
        end
    end
    if real_load > 0.0 then
        simulate some computation, whose duration is function of real_load;
        ensure that the main loop does not iterate too fast;
    end
end
```

7.1.3. Load-balancing thread

The load-balancing thread is in charge of running the load-balancing algorithm, and exchanging the control messages. As shown in Algorithm 3, it iteratively runs the following operations:

- get the control messages that were received from the neighbors;
- run the load-balancing algorithm;
- send control messages to the neighbors, to inform them about the processor's current load, and possibly the future virtual load transfers;
- wait a minimum (configurable) amount of time, to avoid iterating too fast.

Algorithm 3: Load-balancing

```
while true do
    while ctrl_fifo is not empty do
        pop a message from ctrl_fifo;
        identify the sender of the message, and update the current knowledge of
         its load;
    end
    run the load-balancing algorithm to make the decision about load transfers;
    foreach neighbor n do
        send a control messages to n;
    end
    ensure that the main loop does not iterate too fast;
end
```

7.2. Experimental contexts

In order to assess the performances of our algorithm, simulations with various parameters have been achieved out, and several metrics are described in this section.

7.2.1. Load balancing strategies

Several load balancing strategies were compared. Experiments with the *best effort*, and with the *Bertsekas and Tsitsiklis* strategies have been compared. First the *best effort* was tested with parameter $k = 1$, $k = 2$, and $k = 4$. Then, each strategy was run in its two variants: with, and without the management of *virtual load*. Finally, each configuration with *real*, and with *integer* load values was considered.

To summarize the different load balancing strategies, we have:

strategies: *Bertsekas and Tsitsiklis*, or *best effort* with $k \in \{1,2,4\}$

variants: with, or without virtual loads

7.2.2. End of the simulation

The simulations were run until the global equilibrium threshold was reached.

More precisely, the simulation stops when each node holds an amount of load at least inferior to 1% of the load average.

7.2.3. Platform

In order to make our experiments, an heterogeneous grid platform description was created by taking a subset of the Grid'5000 infrastructure[3], as described in the platform file `g5k.xml` distributed with SimGrid. Note that the heterogeneity of the platform here only comes from the network topology. Indeed, processors are considered to be homogeneous for the sake of simplicity. However, this situation is easily extendable to

[3] Grid'5000 is a French large scale experimental Grid (see `https://www.grid5000.fr/`).

the case of heterogeneous platforms by scaling the processor's load by its computing power [28]. The processor speeds were normalized, and we arbitrarily chose to fix them at 1 GFlop/s. Each type of platform with four different numbers of computing nodes: 16, 64, 256, and 1024 nodes is built in a similar way.

7.2.4. Configurations

The distributed processes of the application were then logically organized along three possible typologies: a line, a torus or an hypercube. Tests were divided into two groups on the basis of the initial distribution of the global load: i) some tests were performed with the total load initially on only one node, ii) and other tests were performed for which the load was initially randomly distributed across all the participating nodes of the platform. The total amount of load was fixed to a number of load units equal to 1,000 times the number of node. The average load is then of 1,000 load units.

For all the previous configurations, the computation and communication costs of a load unit are defined. They were chosen so as to have two different CCR, and hence characterize two different types of applications:

- mainly communicating, with a CCR of $1/10$;
- mainly computing, with a CCR of $10/1$.

7.2.5. Metrics

In order to evaluate and compare the different load balancing strategies, several metrics were considered. Our goal, when choosing these metrics, is to have something tending to a constant value, i.e. to have a measure which does not change once the convergence state is reached. Moreover, the goal is to have some normalized values, in order to be able to compare them across different settings. With these constraints in mind, the following metrics are defined:

average idle time: that is the total time spent, when the nodes do not hold any share of load, and thus have nothing to compute. A smaller value is better.

average convergence time: that is the average of the times when all nodes reached the final balanced load distribution. Times are measured as a number of (simulated) seconds from the beginning of the simulation.

maximum convergence time: that is the time when the last node reached the final stable equilibrium. A smaller value is better.

7.3. Experimental results

In this section, the results for the different simulations are presented, and our observations are explained.

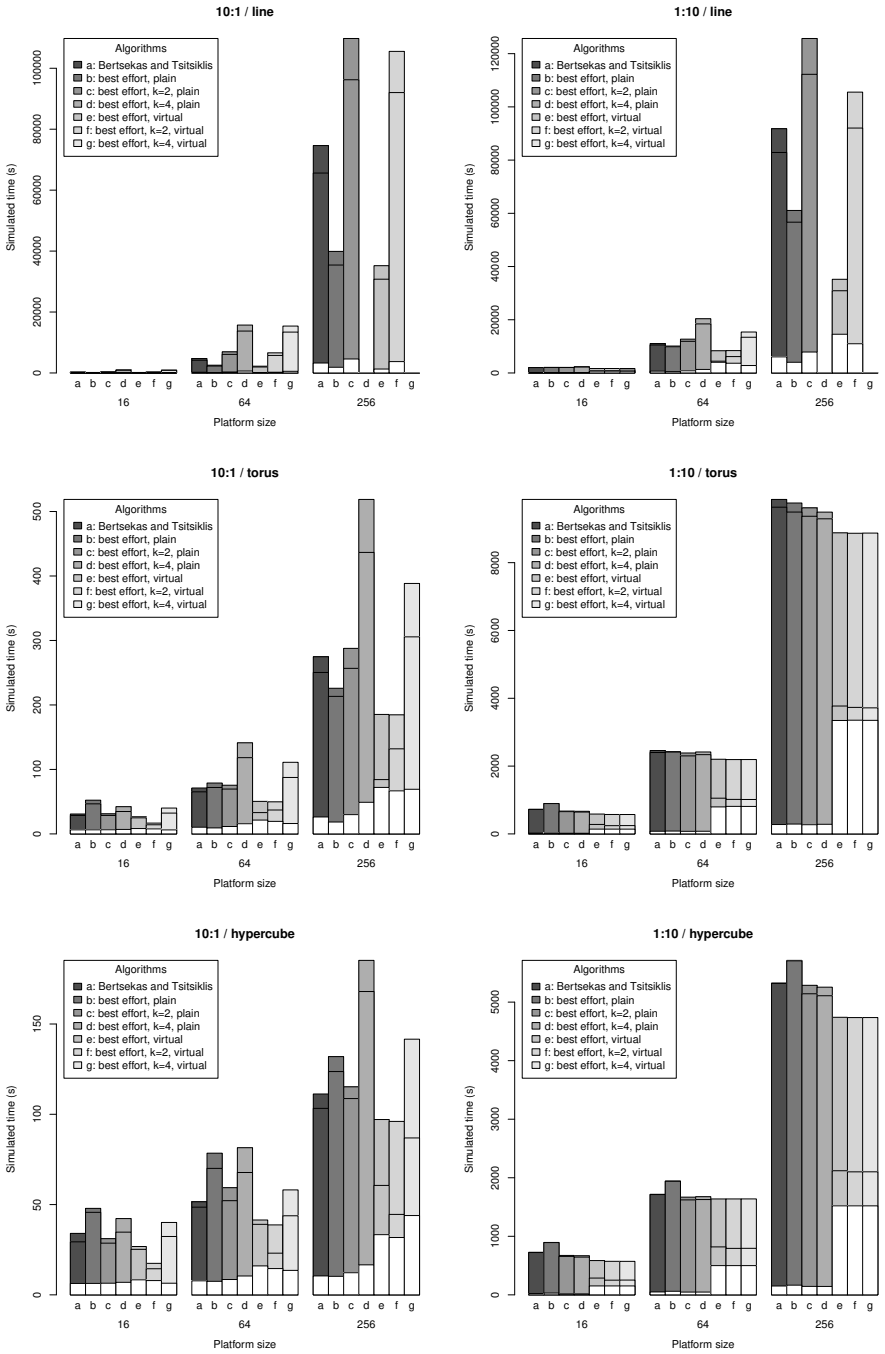

Figure 1: Real mode, initially on an only mode, CCR = $10/1$ (left), or $1/10$ (right). For each bar, from bottom to top starting at $t = 0$, the first part represents the average idle time, the second part represents the average convergence time, and then the third part represents the maximum convergence time.

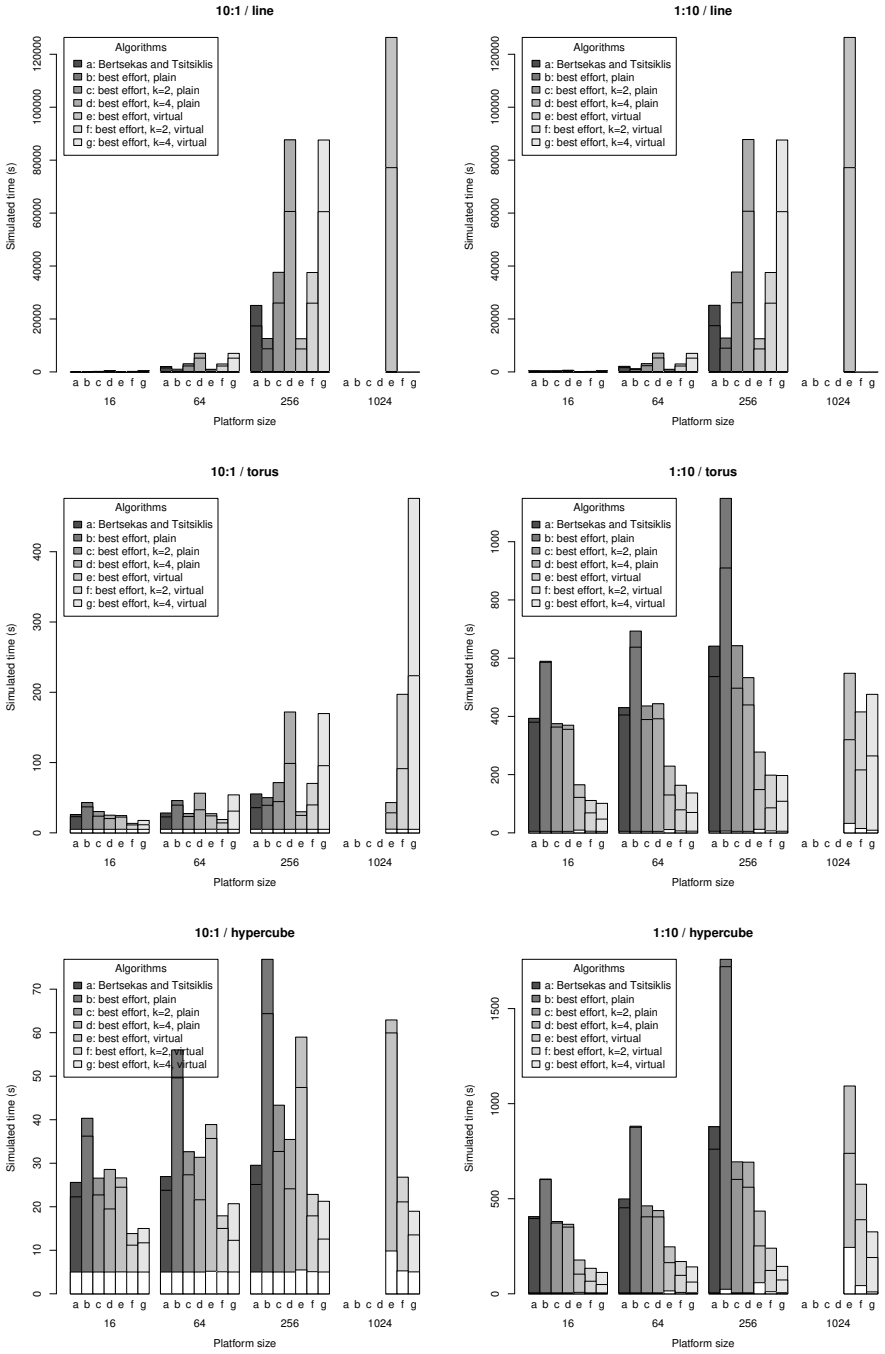

Figure 2: Real mode, random initial distribution, CCR = $10/1$ (left), or $1/10$ (right).

7.3.1. Main results

The main results for our simulations on grid platforms are presented in Figures 1 and 2. The results in Figure 1 are when the load to balance is initially on only one node, while the results in Figure 2 are when the load to balance is initially randomly distributed over all nodes. In both figures, the CCR is $10/1$ on the left column, and $1/10$ on the right column. In each Figure, 1 and 2, the results are given for the process topology being, from top to bottom, a line, a torus or an hypercube.

Finally, the vertical bars show the measured times for the evaluated metrics. These measured times are, starting at $t = 0$ and from bottom to top, the average idle time, the average convergence time, and the maximum convergence time (see Section 7.2.5). The measurements are repeated for the different platform sizes. Some bars are missing, especially for large platforms. This is because the algorithm did not manage to reach the convergence state in the allocated time.

7.3.2. The best effort *and* Bertsekas and Tsitsiklis *strategies without virtual load*

The *simple* (*plain*) version of each strategy is defined as the load balancing algorithm without virtual load's transfers. For each strategy, we compare the simple version (without virtual load) and the improved one (with virtual load). Each algorithm is evaluated in terms of achieved idle time and convergence time.

Before looking at the different variations, we will first show that the simple *best effort* strategy is valuable, and may be as good as the *Bertsekas and Tsitsiklis* strategy. In Figures 1 and 2, these strategies are respectively labeled "b" and "a".

We can see that the relative performance of these strategies is mainly influenced by the application topology structure. It is for the line topology that the difference is the most important. In this case, the *best effort* strategy is really faster than the *Bertsekas and Tsitsiklis* strategy. This can be explained by the fact that the *best effort* strategy tries to distribute the load fairly between all the nodes and is in a good agreement with the line topology since it is easy to load balance the load efficiently.

In contrast, for the hypercube topology, the *best effort*' performances are lower than the *Bertsekas and Tsitsiklis* strategy. In this case, the *Bertsekas and Tsitsiklis* strategy, which tries to give more load to a small number of neighbors, reaches the equilibrium faster.

For the torus topology, for which the number of links is between the line and the hypercube, the *Bertsekas and Tsitsiklis* strategy is slightly better but the difference is more nuanced when the initial load is only on one node. The only case where the *Bertsekas and Tsitsiklis* strategy is really faster than the *best effort* strategy is with the random initial distribution when communications are slow.

Generally speaking, the number of interconnection is very important. Indeed, the more numerous the interconnection links are, the faster the *Bertsekas and Tsitsiklis* strategy is because it quickly distributes significant amount of load, even if the distribution may be unfair, between all neighbors. However, the *best effort* strategy distributes the load fairly when needed and is better for sparse connected applications.

7.3.3. With virtual load

The impact of virtual load schemes is, most of the time, really significant compared to the simple version of the algorithm with the same configuration. For instance, as can

be seen from Figure 1, when the load is initially on one node, it can be noticed that the average idle times are generally longer with the virtual load than the simple version. This can be explained by the fact that, with a virtual load, processors will exchange all the load they need to exchange as soon as the virtual load has been balanced between all the processors. As a consequence, they cannot compute from the beginning. This is especially noticeable when the communication are slow (on the left part of Figure 1).

When the load to balance is initially randomly distributed over all nodes, we can see from Figure 2 that the effect of virtual load is not significant for the line topology structure. However, for both torus and hypercube structures with CCR = 1/10 (on the left of the figure), the performance of virtual load transfers is significantly better. This is explained by the fact that for small CCR values, high communication costs play quite a significant role. Moreover, the impact of communication becomes less important as the CCR values increase, since larger CCR values result in smaller communication times. The impact of CCR values were also tested on the performance of each algorithm in terms of idle times. From Figures 1 and 2, virtual load schemes can be seen to achieve really good average idle times, which is quite close to both its own simple version and its direct competitor *Bertsekas and Tsitsiklis* algorithm. As expected, for coarse grain applications (CCR =10/1), idle times are close to 0 since processors are inactive most of the time compared to fine grain applications.

Taken as a whole, the results illustrated in Figures 1 and 2 clearly show that our proposal outperforms the Bertsekas and Tsitsiklis algorithm. These results indicate that local load balancing decisions have a significant impact on the global convergence time achieved by the compared strategies. This is because, upon load imbalance detection, assigning an amount of load in an unfair way between neighbors will severely increase the total number of iterations required by the algorithm before reaching the final stable distributions. The reason of the poorer performance of the *Bertsekas and Tsitsiklis* algorithm can be explained by the inconvenience of the iterative load balance policy adopted for load distribution between neighbors. Neighbors are selected in such a way that the *ping-pong* condition holds. Therefore, loads are not really assigned to neighboring processors which would allow them to be fairly balanced.

Unlike the *Bertsekas and Tsitsiklis* algorithm, our approach is not really sensitive when dealing with realistic models of computation and communication. This is due to two main features: i) the use of "virtual load" transfers which allows nodes to predict the load they receive in the subsequent iterations steps, ii) and the greedy neighbors selection adopted by our algorithm at each time step in the load balancing process. The involved neighbors are selected in such a way that the load difference between the computational resources is minimized as much as possible.

Comparing the results of the extended version (with virtual load) to the results of the simple one, it can be observed in Figs. 1 and 2 that the improved version gives the best performances. It always improves both convergence and idle times significantly in all figures. This is because, with virtual load transfers, the algorithm seeks greedily to ensure a certain degree of load balancing for processors by taking into account the information about the predictive loads not received yet. Consequently, this leads to optimizing the final convergence time of the load balancing process. Similarly, the extended version achieves much better results than the simple one when considering

larger platforms, as shown in Figs. 1 and 2.

We also find in Figs. 1 and 2 that the performance difference between the improved version of our proposal and its simple version (without virtual load) increases when the CCR increases. This interesting result comes from the fact that larger CCR values reveal that we are dealing with intensive computations applications in grid platforms. Thus, in order to reduce the convergence time of the load balancing for such applications, it is important to make suitable decisions upon local load imbalance detection. That is why we added *virtual load* transfers scheme to the *best effort* strategy to perfectly balance the load of processors at each step of the load balancing process.

Finally, it is worth noting from Figures 1 and 2, that the algorithm's convergence time increases together with the size of the network. We also see that the idle time increases together with the size of the network when a load is initially on a single node (Figure 1), as expected. In addition, it is interesting to note that when the number of nodes increases, there is no substantial difference in the increase of the convergence time, compared to the simple version without virtual load. This is explained by the fact that the increase in the convergence time is already absorbed by the virtual load transfers between processors being in line with the network's size.

7.3.4. The k parameter

As explained previously when the communication are slow the *best effort* strategy is efficient. This is due to the fact that it tries to balance the load fairly and consequently a significant amount of the load is transferred between processors. In this case, it is possible to reduce the convergence time by using the leveler parameter (parameter k). The advantage of using this solution is particularly true when the initial load is randomly distributed on the nodes with torus and hypercube topologies and slow communication. When a virtual load scheme is used, the effect of this parameter is also perceptible in the same conditions.

7.3.5. With non negative integer load values

In addition to the first tests devoted to the case of non negative real load values, further experiments were also carried with integer load values to assess the performance of our proposal. As expected, the obtained results globally have the same behavior, that is why similar figures do not appear in this paper. The most interesting result is that the virtual mode allows processors in a line topology to converge to the uniform load balancing state. Without the virtual load, most of the time, processors converge to what is called the "stairway effect", that is to say that there is only a difference of at most one unit load between any pairs of neighbor nodes, i.e. the load difference between each processor and its neighbors is within one unit load (for example with 10 processors, we obtain 10 9 8 7 6 6 7 8 9 10 instead of 8 8 8 8 8 8 8 8 8 8).

To summarize, the simulation results led us to show that, with a few exceptions (without virtual load), our proposal is superior to the *Bertsekas and Tsitsiklis* algorithm in all the tested scenarios. The illustrated results indicate that network size, CCR values and initial load distribution have a significant impact on the algorithm's performances. Thus, this experimental study corroborates the usefulness of our algorithm, and confirms that when dealing with realistic model platforms, both *best effort* strategy

and *virtual load* transfers play an important role on the achieved idle and convergence times.

8. Conclusion

In this paper, a new asynchronous load balancing algorithm for non negative real numbers of divisible loads in distributed systems was presented. The proposed algorithm, which is called *best effort strategy*, seeks greedily for loads imbalance detection and tries to achieve efficient local load equilibrium between neighbors. Our proposal is based on *a clairvoyant virtual loads' transfer* scheme which allows nodes to predict the future loads they will receive in the subsequent iterations. This leads to a noticeable speedup of the global convergence time of the load balancing process. Based on SimGrid simulator, it was shown that, when dealing with realistic models of computation and communication, our algorithm exhibits better performances than its direct competitor from the literature. This makes it a viable choice for load balancing of both non negative real and integer divisible loads in distributed computing systems.

Acknowledgments

This paper is partially funded by the Labex ACTION program (contract ANR-11-LABX-01-01). We also thank the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

References

[1] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Athena Scientific, 1997.

[2] J. M. Bahi, S. Contassot-Vivier, R. Couturier, Parallel Iterative Algorithms: from sequential to grid computing, Numerical Analysis & Scientific Computating, Chapman & Hall/CRC, 2007.

[3] V. Bharadwaj, T. G. Robertazzi, D. Ghose, Scheduling Divisible Loads in Parallel and Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[4] M. Drozdowski, Selected problems of scheduling tasks in multiprocessor computer systems, Ph.D. thesis, Poznan University of Technology, Poznan, Poland (1998).

[5] H. Casanova, A. Legrand, Y. Robert, Parallel Algorithms, 1st Edition, Chapman & Hall/CRC, 2008.

[6] A. Cortés, A. Ripoll, F. Cedo, M. A. Senar, E. Luque, An asynchronous and iterative load balancing algorithm for discrete load model, J. Parallel Distrib. Comput. 62 (12) (2002) 1729–1746.

[7] F. Cedó, A. Cortés, A. Ripoll, M. A. Senar, E. Luque, The convergence of realistic distributed load-balancing algorithms, Theory of Computing Systems 41 (4) (2007) 609–618.

[8] J. M. Bahi, A. Giersch, A. Makhoul, A scalable fault tolerant diffusion scheme for data fusion in sensor networks, in: Infoscale 2008, The Third International ICST Conference on Scalable Information Systems, Vico Equense, Italy, 2008, p. 10 (5 pages).

[9] J. M. Bahi, S. Contassot-Vivier, A. Giersch, Load balancing in dynamic networks by bounded delays asynchronous diffusion, in: High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers, 2010, pp. 352–365.

[10] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, J. Parallel Distrib. Comput. 7 (2) (1989) 279–301.

[11] J. Gehrke, C. G. Plaxton, R. Rajaraman, Rapid convergence of a local load balancing algorithm for asynchronous rings, Theor. Comput. Sci. 220 (1) (1999) 247–265.

[12] J. Zhao, K. Yang, X. Wei, Y. Ding, L. Hu, G. Xu, A heuristic clustering-based task deployment approach for load balancing using bayes theorem in cloud environment, IEEE Transactions on Parallel and Distributed Systems 27 (2) (2016) 305–316.

[13] H. Izakian, A. Abraham, B. T. Ladani, An auction method for resource allocation in computational grids, Future Generation Comp. Syst. 26 (2) (2010) 228–235.

[14] H. Choi, L. Brunet, J. P. How, Consensus-based decentralized auctions for robust task allocation, IEEE Trans. Robotics 25 (4) (2009) 912–926.

[15] T. Chen, A. G. Marques, G. B. Giannakis, Dglb: Distributed stochastic geographical load balancing over cloud networks, IEEE Transactions on Parallel and Distributed Systems 28 (7) (2017) 1866–1880.

[16] R. Tripathi, S. Vignesh, V. Tamarapalli, A. T. Chronopoulos, H. Siar, Non-cooperative power and latency aware load balancing in distributed data centers, Journal of Parallel and Distributed Computing 107 (2017) 76–86.

[17] Y. Liu, C. Zhang, B. Li, J. Niu, Dems: A hybrid scheme of task scheduling and load balancing in computing clusters, Journal of Network and Computer Applications 83 (2017) 213–220.

[18] R. Subrata, A. Y. Zomaya, B. Landfeldt, Artificial life techniques for load balancing in computational grids, Journal of Computer and System Sciences 73 (8) (2007) 1176–1190.

[19] P. V. Krishna, Honey bee behavior inspired load balancing of tasks in cloud computing environments, Applied Soft Computing 13 (5) (2013) 2292–2303.

[20] Y.-K. Kwok, L.-S. Cheung, A new fuzzy-decision based load balancing system for distributed object computing, Journal of Parallel and Distributed Computing 64 (2) (2004) 238–253.

[21] R. Salimi, H. Motameni, H. Omranpour, Task scheduling using nsga ii with fuzzy adaptive operators for computational grids, Journal of Parallel and Distributed Computing 74 (5) (2014) 2333–2350.

[22] J. Cao, D. P. Spooner, S. A. Jarvis, G. R. Nudd, Grid load balancing using intelligent agents, Future generation computer systems 21 (1) (2005) 135–149.

[23] X.-J. Shen, L. Liu, Z.-J. Zha, P.-Y. Gu, Z.-Q. Jiang, J.-M. Chen, J. Panneerselvam, Achieving dynamic load balancing through mobile agents in small world p2p networks, Computer Networks 75 (2014) 134–148.

[24] D. Grosu, A. T. Chronopoulos, Noncooperative load balancing in distributed systems, J. Parallel Distrib. Comput. 65 (9) (2005) 1022–1034.

[25] S. Penmatsa, A. T. Chronopoulos, Game-theoretic static load balancing for distributed systems, J. Parallel Distrib. Comput. 71 (4) (2011) 537–555.

[26] http://simgrid.org/ [online]. SimGrid Website.

[27] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, scalable, and accurate simulation of distributed applications and platforms, Journal of Parallel and Distributed Computing 74 (10) (2014) 2899–2917.
URL `http://hal.inria.fr/hal-01017319`

[28] R. Elsässer, B. Monien, R. Preis, Diffusion schemes for load balancing on heterogeneous networks, Theory of computing systems 35 (3) (2002) 305–320.