Topology and affinity aware hierarchical and distributed load-balancing in Charm++

Emmanuel Jeannot, Guillaume Mercier, François Tessier

Topology and affinity aware hierarchical and distributed load-balancing in Charm++

Emmanuel Jeannot
Inria Bordeaux Sud-Ouest
Talence, France
Email: emmanuel.jeannot@inria.fr

Guillaume Mercier
Bordeaux INP
Talence, France
Email: guillaume.mercier@bordeaux-inp.fr

François Tessier
Inria Bordeaux Sud-Ouest
Talence, France
Email: ftessier@anl.gov

November 14, 2016

Abstract

The evolution of massively parallel supercomputers make palpable two issues in particular: the load imbalance and the poor management of data locality in applications. Thus, with the increase of the number of cores and the drastic decrease of amount of memory per core, the large performance needs imply to particularly take care of the load-balancing and as much as possible of the locality of data. One mean to take into account this locality issue relies on the placement of the processing entities and load balancing techniques are relevant in order to improve application performance. With large-scale platforms in mind, we developed a hierarchical and distributed algorithm which aim is to perform a topology-aware load balancing tailored for Charm++ applications. This algorithm is based on both LibTopoMap for the network awareness aspects and on TreeMatch to determine a relevant placement of the processing entities. We show that the proposed algorithm improves the overall execution time in both the cases of real applications and a synthetic benchmark as well. For this last experiment, we show a scalability up to one millions processing entities.

1 Introduction

Scientific applications needs for computing power are continuously increasing. In order to match these needs, larger and larger parallel computers are designed and built. Optimizing the use of the underlying physical resources is an essential objective in order to take advantage of the current complex multicore architectures of the nodes. One other issue stems from the fact that parallel applications have to deal with problems featuring irregular structures that can easily induce a load imbalance at the CPU level. Such an imbalance can in return yield counterproductive effects on the application performance. For instance, a process that has finished its work will become idle if it has to wait (synchronize) for another more loaded process still computing. In such a case, this idleness leads to a considerable waste of CPU cycles. This issue is even more prominent as the

number of processes grows. Therefore, balancing the load among the processes is an efficient and relevant optimization technique to decrease an application execution time.

However, load balancing can be detrimental to data locality since it involves processes/tasks movements and migrations across the computing platform. In case of internode movements, it implies network communications whilst in case of intranode movements, Non-Uniform Memory Access (NUMA) effects are likely to degrade performance. Since data locality is a key-factor for large-scale applications performance, it should also be taken into account by load balancers.

Improving or managing efficiently data locality is not an easy task, especially in the context of large clusters of multicore nodes. Indeed such computers feature intricate and hierarchical memory systems with several cache levels and banks physically scattered across the whole machine. This trend is going to gain momentum with the forthcoming advent of new memory technologies (e.g. NVRAM) because main memory will now be made of several levels of different memory types. Data locality can be managed in various fashions, one being to create a match between the physical topology (that is, the CPUs and memory hierarchy layout) and a virtual topology that describes the application bahaviour. For instance, an application communication pattern (in case of processes) or the pattern of memory accesses (in case of threads) are pertinent instanciations of virtual topologies.

We developed an algorithm, called TreeMatch [11] that computes such a matching between the physical entities delivering the computing resources and the virtual entities (e.g processes or threads or tasks) handling the data and performing the computation. TreeMatch uses a qualitative approach as it relies on the structure of the underlying physical topology. It has been successfully integrated within MPI libraries and runtimes in order to perform MPI process placement [9] and rank reordering [25]. Therefore, an intuitive idea is to expand load balancing algorithms and mechanisms with TreeMatch in order to make them data locality-aware.

For instance, the Charm++ parallel programming environment [19] natively features load balancing mechanisms. In Charm++, the load is divided among *chares* that are able to migrate across the platform. Hence, migration is used to balance the load, as opposed to work-stealing (for instance). Another interesting and useful feature of Charm++ is its ability to provide the user with various pieces of information during an application execution. For instance, information dealing with the workload evolution or the communication pattern can be exploited.

The contribution of this paper is a scalable parallel and distributed load balancer fully topology-aware (both at the nodes and at the network levels) that takes into account the computation as well as the communication cost of the application.

This paper is organized as follows: a state of the art is given in Section 2. The topology mapping problem is exposed in Section 3. The proposed load balancer is detailed in Section 4, while Section 5 describes the experiments carried out and the results obtained. Finally, Section 6 concludes this paper.

2 State of the Art

Charm++ [19, 14, 18] is a message-passing based programming environment that uses an object-oriented approach and relies on the C++ language. However, whilst MPI considers processes in its model (usually with a rather coarse granularity), Charm++ rather uses finer-grain objects and dispatches the computation onto small migratable tasks called *chares*. These chares, in addition to their assigned data and their ability to exchange messages asynchronously, are also characterized by their CPU load, their I/O communication volume and some other useful parameters. To manage

these processing entities, an adaptive runtime support is essential [17]. The Charm++ runtime system is able to handle the various physical resources, to enforce fault tolerance or manage load balancing [16, 15]. Regarding the load balancing aspects, Charm++ makes it possible to design, plug in and test load-balancers transparently without modifying the application code. Also the design of the Charm++ runtime system allows to perform load balancing at different frequencies during the execution of the application [5, 29].

Several generic load balancing mechanisms natively exist in Charm++ [28], but are tailored and give the best results for a specific optimization area. Common load balancing schemes that consider the CPU load on each processing unit have been extended in some works to also take into account the topology of the underlying architecture [2], [3]. For instance, Bhatelé and al. [3] show the gains obtained by applying a chare placement depending on the structure of a 3D torus in the context of an adaptive mesh refinement applications. Hierarchical load balancing (i.e. multilevel topologies) has also been investigated [31] [30] while other methods are designed to be distributed in order to achieve a satisfactory level of scalability on large-scale platforms [23]. NucoLB[1] [27] and HwTopoLB [26] apply a load balancing scheme based on a quantitative assessment of the topology links (latency and bandwidth values are mandatory). As a consequence, this type of strategy requires to gauge the target architecture communication performance with the appropriate tools before running any application. Our TreeMatch-based solution is more flexible and dynamic since we fully rely on a qualitative approach for our representation of the hardware topology. TreeMatch does not require to assess the performance of the system on which the application is running. We believe that this is a strong advantage, as gathering such information is error-prone, might be incomplete and is subject to inaccuracy.

We already worked on a topology-aware load balacing algorithm in Charm++ using TreeMatch [10] called TMLB_TreeBased. However, this load-balancer suffered from critical limitations (handled topology, scalability, load balancing techniques, etc.). The completely redesigned algorithm described in this paper is no longer limited to tree-shaped networks which is undeniably a more realistic approach. Most of the algorithm is distributed and hierarchical, improving significantly the capacity of scaling while the distribution of the work is parallelized in a master-worker fashion. Besides, we modifed the part of the algorithm in charge of the CPU load balancing into a tunable refinement approach.

3 Context and Problem Definition

We consider a Charm++ application with both load balancing and migration capabilities enabled. The load balancing phase is executed periodically by the Charm++ runtime system. The application is composed of multiple processes that are bound on their dedicated CPU cores. Each process executes a given set of Charm++ chares. The load balancer possesses the following pieces of information. (1) The load of the different chares (keep in mind that the number of chares is greater than the number of processes). This load actually represents the amount of CPU used by each chare since the last load-balancing phase. (2) The affinity between chares. This affinity is represented by a communication matrix which values are the amount of communication between a pair of chares since the last load-balancing phase. (3) The system topology. Here, we deal with two kinds of topologies: first, the node internal topology, where the memory hierarchy is represented by a tree

[1]Please note that NucoLB is designed for shared-memory machines.

and second, the network topology which forms an arbitrary graph where each vertex represents an allocated compute node for the application.

Both the load and the affinity of the chares are computed by the Charm++ runtime system and are available in the load-balancing code through the Charm++ API as a set of standard arrays or matrices. The node topology is retrieved thanks to the hwloc tool [4], following the assumption that all nodes feature a similar topology across the supercomputer (which is the case in most of today's top-end HPC systems). The network topology is obtained either through the batch scheduler or a standard system service that can be queried by the application to collect the graph of allocated resources (i.e. compute nodes). For instance, on the BlueWaters system, such information is obtained thanks to the `xtdb2proc` command, provided as part of the Cray environment of the machine.

Based on this information, our goal is to compute a new chares allocation for the processes so that the application overall execution time is decreased[2]. Experiments carried out in related works [27, 26, 10] have shown that taking solely into account the load to perform the load balancing leaves room for improvement. Indeed, due to both the network topology and the memory hierarchy, the communication costs between processes or chares have also to be considered. As a consequence, in order to tackle the issue of load balancing, this work considers not only the load but also both the affinity between the various computing entities of the application and the underlying target hardware topology.

4 Scalable Topology Aware Load Balancing at System Scale

4.1 Overview

In order to solve our problem we need to design a scalable load-balancing algorithm that takes into account the chares load, the chares affinity and the topology of the target machine. The chares distribution is computed at each load-balancing step. This computation is two-fold: globally at the nodes level and then inside each node, chares are assigned to cores in parallel. For the nodes step, we balance the CPU load at this level of hierarchy first and then we move groups of chares on nodes to reduce the communication costs. For the cores step, we first compute an affinity-aware chares placement then we apply a refinement algorithm to level the CPU load.

4.2 On the Need of a Global Load-balancing Phase

We first discuss the need for a global chare allocation and load-balancing step. One could devise a fully distributed algorithm with no global load-balancing phase (that is, chares would stay on their nodes and would only move from one core to an other). But the following theorem shows that the probability can be extremely high that at least one pair of two nodes has an imbalance factor greater than a given ratio.

Theorem 1 *Let m be the number of nodes of a run. Let n be the number of chares per node. Assuming that chares have a duration time uniformly distributed in $[a, b]$ then the probability q that*

[2]Our proposed algorithm and solution have been implemented in Charm++. However, this approach is generic and could find its place in any other runtime system featuring migration, provided the ability to gather the pieces of information described above.

there is two nodes which imbalance ratio is greater than α is

$$q = 1 - (1-p)^{\frac{m(m+1)}{2}}$$

where $p = 2 - 2\Phi\left(\alpha\frac{\sqrt{6n}}{2}\frac{b+a}{b-a}\right)$ *and* Φ *is the cumulative distributed function (CDF) of the standard normal distribution* $\mathcal{N}(0,1)$.

PROOF.

Let us consider that we have n chares per node. Let us suppose that the duration time of the chares is uniformly distributed between $[a,b]$. Hence, by definition of the uniform distribution, the mean duration time of the chares is: $\mu = \frac{b+a}{2}$ and the variance of the duration time of the chares is: $\sigma^2 = \frac{(b-a)^2}{12}$. On a given node, the sum of the duration times of the chares has a mean of $\mu_s = n\mu$ and a variance of $\sigma_s^2 = n\sigma^2$. If n is large enough[3], we can apply the central limit theorem to compute the distribution X_i of the sum of the duration times of the chares on node i. Assuming that the duration time of the chares is independent and identically distributed, then, X_i does not depend on i and, by the central limit theorem,

$$X_i \sim \mathcal{N}(\mu_s, \sigma_s^2) = \mathcal{N}\left(\frac{n(b+a)}{2}, \frac{n(b-a)^2}{12}\right)$$

where $\mathcal{N}$ is the gaussian (normal) distribution and $X \sim \mathcal{D}$ means that the random variable X follows distribution $\mathcal{D}$. Let us first answer the following question: what is the probability p that the ratio of duration times of all the chares of two nodes (i and j) differs by more than a given $\alpha \geq 0$? This can be written:

$$p = \mathbb{P}\left(\left|\frac{X_i - X_j}{n\mu}\right| \geq \alpha\right) = \mathbb{P}(|Z| \geq \alpha)$$

where

$$Z = \frac{X_i - X_j}{n\mu} = \frac{X_i - X_j}{\frac{n(b+a)}{2}}$$

From the rule of normal distribution difference, we have: $X_j - X_i \sim \mathcal{N}(0, 2\frac{n(b-a)^2}{12})$. Then, thanks to the scaling property of the gaussian distribution[4],

$Z \sim \mathcal{N}\left(0, \frac{2n(b-a)^2}{12}\frac{4}{n^2(b+a)^2}\right) = \mathcal{N}\left(0, \frac{2(b-a)^2}{3n(b+a)^2}\right)$.

Moreover, if we put: $Y = \frac{Z}{\frac{2}{\sqrt{6n}}\frac{b-a}{b+a}} \sim \mathcal{N}(0, \frac{2(b-a)^2}{3n(b+a)^2}\frac{6n(b+a)^2}{4(b-a)^2}) = \mathcal{N}(0,1)$, we can simplify the solution even further:

$$\begin{aligned} p &= \mathbb{P}(|Z| \geq \alpha) = \mathbb{P}\left(|Y| \geq \alpha\frac{\sqrt{6n}}{2}\frac{b+a}{b-a}\right) \\ &= 1 - \mathbb{P}\left(-\alpha\frac{\sqrt{6n}}{2}\frac{b+a}{b-a} < Y < \alpha\frac{\sqrt{6n}}{2}\frac{b+a}{b-a})\right) \end{aligned}$$

[3] in [8] Chap. 13, the central limit theorem is applied if $n > 30$ which is the case for all the runs we are doing

[4] If $X \sim \mathcal{N}(\mu, \sigma^2)$ then $\frac{X-\mu}{\sigma} \sim \mathcal{N}(0,1)$

Therefore, as Y is the standard normal distribution we have:

$$p = 1 - \left(\Phi\left(\alpha \frac{\sqrt{6n}}{2} \frac{b+a}{b-a} \right) - \Phi\left(-\alpha \frac{\sqrt{6n}}{2} \frac{b+a}{b-a} \right) \right)$$

Where Φ is the cumulative distributed function (CDF) of $\mathcal{N}(0,1)$. As $\mathcal{N}(0,1)$ is symmetric around 0 we have $\Phi(-x) = 1 - \Phi(x)$ therefore, $p = 2 - 2\Phi\left(\alpha \frac{\sqrt{6n}}{2} \frac{b+a}{b-a}\right)$ and $\alpha \geq 0$.

Interestingly, since Φ is increasing, we see that p is decreasing (as expected) with n and α. Note also that even if p depends on a and b that if $b = r \times a$ (b is a multiple of a then, $\frac{b+a}{b-a} = \frac{r+1}{r-1}$ is constant.

However, p is simply the probability that two nodes have a load ratio greater than α. If we call q the probability at least one pair of nodes (among the m used in the run) have a load ratio greater than α we have:

$$q = 1 - (1-p)^{\frac{m(m+1)}{2}}$$

Indeed we have $\frac{m(m+1)}{2}$ pairs of nodes and $(1-p)^{\frac{m(m+1)}{2}}$ is the probability that all pairs have a ratio lower than α. $\square$

Table 1 outlines the results for different realistic scenarios. We see that in most of the cases, there is a high probability that at least two nodes are highly imbalanced. This advocates for the use of a global balancing step where we allow to move any chare to any node. Note that the first line of the Table 1 corresponds to an experimental case of this paper.

Table 1: p: probability that a given pair of nodes has a load imbalance ratio greater than α when having m nodes, n chares per nodes and the load of the chares is uniformly distributed in $[a, b]$. q: probability that at least one pair of nodes has an imbalance ratio greater than α.

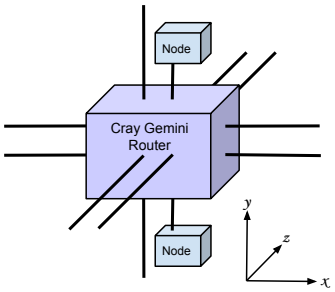
n	m	α	a	b	p	q
64	128	0.05	50	100	14,2%	100%
32	16	0.1	75	125	0.6%	94.8%
64	8	0.1	60	140	1.4%	40.4%
64	256	0.05	100	130	0.02%	99,7%

4.3 Computing the Chares Placement at Nodes Level

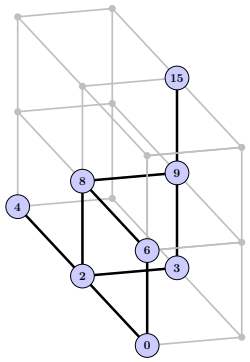
As shown above, global chare placement is often mandatory. Such a step is based on LibTopoMap which is able to compute a mapping on any arbitrary topology. To do so, we first need to compute the node topology (based on the job allocation) and then call LibTopoMap to compute the new allocation.

4.3.1 Computing the Allocated Nodes Topology

In order to balance the workload by taking into account both the network topology and the memory hierarchy and layout, we need to compute the topology formed by the nodes allocated by the batch scheduler. Then we apply to these nodes a local numbering scheme (starting from zero). In the case



(a) Cray Gemini router from a 3D torus. One link on the y-axis is reserved to connect the compute nodes.



(b) Extraction of the network topology of a sub-part of the Gemini 3D torus. Each vertex is a router to which computing nodes are connected.

Figure 1: Cray Gemeni topology management

of a fat-tree network, the nodes numbering is performed in topological order and the network spans the part of the tree used by the allocated nodes. If not, building the topology of the allocated nodes requires to extract the global coordinates of the nodes as well as their interconnections and then to renumber the nodes in a topological order. For instance, the BlueWaters Cray Gemini interconnect topology is a 3D torus. However, some nodes in the torus are only routers and compute nodes are connected to the routers as there are two compute nodes per router (c.f. Fig. 1a).

We proceed as follows: (1) Extraction with the Cray tools of the smallest 3D grid of routers including all the compute nodes allocated by the resource manager. Each router features a set of (x, y, z) coordinates in the 3D torus; (2) Renumbering of the routers in topological order (x, then y, then z); (3) Creation of an adjacency list of every routers of the grid; (4) Creation of the link between each compute node and its router; (5) Exclusion of the nodes not part of the resource allocation; (6) Output a file in a LibTopoMap-readable format.

As opposed to TreeMatch, LibTopoMap relies on a quantitative approach. To fulfill this requirement, we defined the weights of the topology edges as follows : the x-axis and z-axis weights are twice as large as the y-axis'. This choice is motivated by a particular feature of the Cray Gemini interconnect routers as illustrated by Figure 1a. All axes possess two interconnection links except for the y-axis where one link is devoted to the connection of compute nodes.

Figure 1b depicts how this step works. We can see a 3D grid which includes the allocated routers (blue vertices). The compute nodes are not shown but are indeed connected to these routers temporarily renumbered following the (x, y, z) order. Then, the corresponding adjacency list is given to LibTopoMap to compute the relevant placement.

4.3.2 Global chares allocation

As explained previously, the global chares allocation (i.e. assigning balanced groups of chares on nodes) is a two-step process.

From theorem 1 it is likely that at least two nodes are highly imbalanced. Therefore, we first smooth the load so that the imbalance factor α between two arbitrary nodes is lower than 5%. To do so, we perform a very simple set of refinements of the load by swapping the highest loaded chares from overloaded nodes for the lowest loaded one from underloaded nodes. After this step,

all the nodes roughly possess an equivalent load. Of course, this step is not performed if the load is already evenly distributed among the different nodes.

The second part consists in moving groups of chares by taking into account their affinity and the communication cost of the network. We use the LibTopoMap [6] library during this step because of its following pros: it works in a distributed fashion (as an MPI-based library) and it is able to find an efficient placement of MPI processes on a cluster of compute nodes regardless of the network topology (that is, not only tree-like topologies).

This library makes use of the virtual topologies management features of the MPI standard to determine this process placement. In the load-balancing algorithm presented in this paper, we use LibTopoMap in order to perform a first placement of the groups of chares on the nodes. Since Charm++ also uses MPI in its internal communication layer, we can access to the native MPI functions. As LibTopoMap only needs one MPI process per node to compute its network-aware placement, we select a rank on each node and create a dedicated communicator with `MPI_Comm_split`. LibTopoMap considers the *node communication matrix*. This matrix describes the affinity between groups of chares that are already assigned to a given node. LibTopoMap uses Metis [20] to assign groups of chares to the nodes according to the topology computed in Section 4.3.1 and the communication matrix.

4.4 Fine Placement of the Chares on the Cores

In the previous step, groups of chares are migrated from one node to an other according to their load, their affinity and the topology. In this step we assign chares to cores on each node, according to the same criteria but in a distributed fashion.

In this step, we need each core to be assigned an equal load and that the communication between chares is taken into account as well as the node topology. For taking into account the topology, we use the hwloc tool which provides a simple way to apprehend the memory and cache hierarchy as well as the core numbering. Remind that the affinity between chares and their load is provided through the Charm++ API.

4.4.1 Master Algorithm

Algorithm 1: The TreeMatch Parallel Load balancing Algorithm: Master Algorithm

```
   Input: m_global_chares The communication matrix between application chares
   Input: M current mapping of the chares to the cores
2  // In parallel on all cores of the master node
3  for every node i do in parallel
4  |  m_chares[i] ← extract_sub_matrix(m_global_chares, i, M) // The communication
   |     matrix between chares for node i
5  |  async.call result[i] ← compute_lb(m_chares[i], i);
6  for every node i do in parallel
7  |  when result[i] has arrived;
8  |    result ← aggregate_result(result[i]);
```

As each node performs its load-balancing step for its own set of chares, no global communication

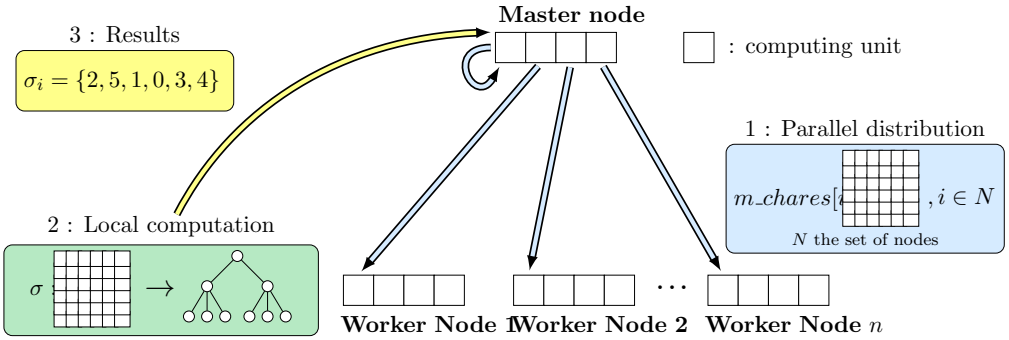


Figure 2: Master-Worker scheme of the Distributed Load Balancer

step is needed. Indeed, each node can perform its own step independently from the others and in parallel. To distribute the load among all the nodes, we use a master/worker scheme. The master algorithm is described in Algorithm 1 and features two phases: the first one is to decompose the global communication matrix into local communication matrices to be used by each node. To do so, we extract the element of the *m_global_chares* global communication matrix that corresponds to the chares of node i. This is done thanks to the knowledge of the current mapping of the chares $\mathcal{M}$. As a master node possesses in general several cores, we use these cores to perform this phase in parallel. To do so, we create a parallel OpenMP section that extracts the communication pattern of the group of chares of each compute node (lines 2 to 4). The second phase is to distribute the chares mapping of each node to the corresponding node. Therefore, each node computes the placement of its own set of chares. To be more specific, the call to `compute_lb` on line 5 is distributed thanks to Charm++ mechanisms and each node computes its chares placement in parallel. The call to `compute_lb` is done asynchronously. When each local result of node i is returned (line 7), it is aggregated to compute the final global placement of the chares (line 8). This global scheme is depicted in Fig 2.

4.4.2 Worker Algorithm

Each node receives the communication matrix corresponding to its own set of chares and computes the placement of these chares on its own cores. The worker algorithm is depicted in Algorithm 2 and is executed on each node, including the master one. Each worker knows the load and the affinity of the chares. The first step of the algorithm consists in getting the topology tree T of the node. Each leaf of T thus corresponds to a core. The problem is that generally we have much more chares than cores. We therefore need TreeMatch to perform the assignment with oversubscribing (line 3). To this end, we compute the ratio r between the number of chares (n) and the number of cores (c). We extend the tree T by adding a new level to the tree with r leaves to each core of T, leading to a tree T' with $c' = r * c$ leaves. TreeMatch then assigns each chare to a leaf of T'. In order to balance the number of chares to cores, we constraint the TreeMatch algorithm so that chares are evenly distributed.

Then the solution p (line 3) is such that chares assigned to leaves (of T') corresponding to the same core (of T) are actually assigned to this core. In this case we have at most r chares per core of the node. Then solution p is refined (line 4). The goal of this step is to take into account the load imbalance between different cores. This refinement step works as follows: chares are topologically

sorted according to the core they have been assigned to. Then cores are considered one after the other: if a considered core is the least loaded one, then it steals other chares starting from the closest leaf until it is no longer the least loaded one. In this way, we keep the chares that communicate a lot close to each other and the above refinement allows for a good repartition of the load among the cores.

Algorithm 2: Worker Algorithm. Implementation of the `compute_lb` function

Input: m_chares `The communication matrix between chares inside the considered node`
Input: l_chares `Array of the load of the Chares inside the current node`
Input: n `Number of chares to be assigned on the considered node`
Input: c `Number of cores of the considered node`
1 $T \leftarrow$ `hwloc_get_topology`();
2 $r \leftarrow \lceil n/c \rceil$`// Over subscribing ratio`
3 $p \leftarrow$`tm_oversubscribe_chares_to_cores` (m_chare, T, r);
4 return `refine_solution`(p,l_chares);

Every node returns its result back to the master so it can build a global view of the chares mapping and commit the chare migration using internal Charm++ features. No actual migration occurs before this point.

5 Experimental Validation

In this section, we present the results obtained with our load balancing algorithm. We chose three applications to evaluate our solution. The first one, called ChaNGa, is a cosmological application designed to perform collisionless N-body simulations. The second application, Ondes3D, is a simulator of three-dimensional seismical waves propagation. The third one named CommBench is a benchmark simulating irregular communications.

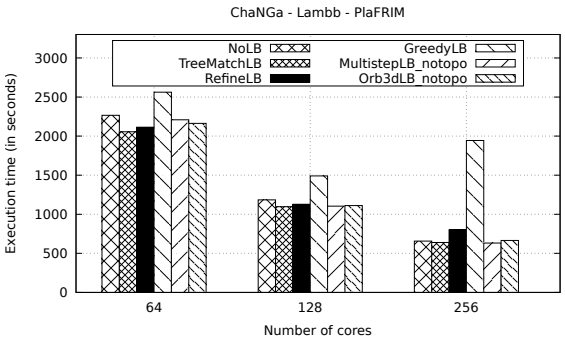
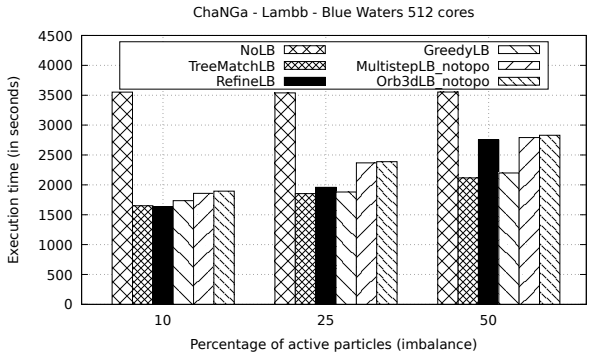
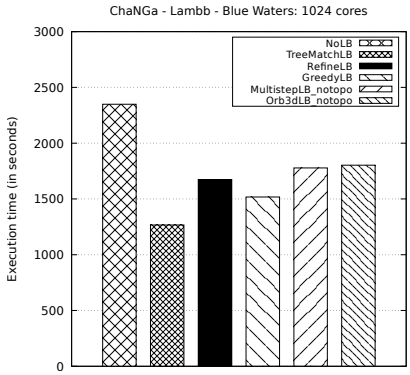


Figure 3: ChaNGa application walltime using different load-balancers for the lambb use-case when varying the number of cores on PlaFRIM.



(a) Application walltime using different load-balancers for the lambb use-case on for 512 cores (16 XE6 nodes) when varying the imbalance.



(b) Application walltime using different load-balancers for the lambb use-case on for 1024 cores (32 XE6 nodes).

Figure 4: ChaNGa Blue Waters experiments

5.1 ChaNGa

The first selected application is a cosmological simulator. Its main goal is to perform collisionless N-body simulations while using a Barnes-Hut tree to calculate gravity [12, 13, 24]. This application has been written in Charm++ in order to exploit the Charm++ features and particularly its ability to migrate chares to balance the CPU load. As a use case, we worked on *Lambb* [22] which is an 80 million particles representation of a 70 Mpc (Megaparcec) volume. According to the ChaNGa documentation, this simulation is used to calculate the mass function of dark matter halos in a dark energy dominated universe down to the scale of dwarf galaxies. The experiments on this real application were carried out both on the PlaFRIM cluster and the Blue Waters supercomputer. For each experiment, we compared our load balancing algorithm to an execution without load balancing (NoLB) and to other load balancers. RefineLB and GreedyLB are standard load balancers available within Charm++. RefineLB migrates chares from overloaded cores to underloaded cores to reach an average load. This strategy has the advantage to limit the number of chares migrations. GreedyLB re-assign all the chares by mapping the highest loaded chare to the least loaded core. Even if these load balancers were not designed for this particular use case, we choose to compare our solution with Orb3dLB_notopo (based on recursive bisection to find a balanced stat) and MultistepLB_notopo (considers a range of different timesteps to take its decision). Indeed, these strategies were developed for specific use cases of ChaNGa and help us to show that TreeMatchLB is a generic, yet relevant approach. It has to be noted that the load balancing time is included in each measurement of these experiments.

5.1.1 PlaFRIM

The PlaFRIM cluster is composed of nodes featuring two Quad-core-Intel Xeon Nehalem X5550 (2.66 GHz) processors. Each node offers 24 GB of 1.33GHz DDR3 RAM and the 8 MB of L3 cache are shared between the four cores of a CPU. Nodes are connected through an Infiniband fat-tree network. We ran experiments on 64, 128 and 256 cores (respectively 4, 8 and 16 nodes) of the

PlaFRIM platform.

The results we obtained on this platform are depicted in Figure 3. This figure shows the average execution time of one step of the *Lambb* case for each number of cores used. The first remark we can make is that TreeMatchLB yields a performance level that is at least on par with the other strategies. On 64 cores, TreeMatchLB is the fastest approach while for the 128 and 256 cores cases, it competes equally with Orb3dLB_notopo and MultistepLB_notopo. As far as the load balancing time is concerned, our solution is able to compute a chares placement in 600ms for 2048 chares distributed among 256 cores while the other solutions perform this in less than 50ms. In spite of this duration, the benefits in term of performance counterbalance this additional cost. Finally, it has to be noted that even if the results provided by GreedyLB are really subpar, we observed this behavior on this platform only but for several use-cases and applications.

5.1.2 Blue Waters

Blue Waters is a supercomputer from the University of Illinois at Urbana-Champaign providing a peak performance of 13.34 Petaflops. For these experiments, we used XE6 nodes made of two AMD 6276 Interlagos processors (16 cores each). Each node provides 64GB of main memory. The interconnection network is a Cray Gemini 3D torus as described in 4.3.1. Considering that the *Lambb* case is able to scale up to 1024 cores, we carried out experiments on 512 and 1024 cores (resp. 16 and 32 nodes) on Blue Waters.

Figure 4a shows the results obtained on 512 cores. For these experiments we carried out several versions of the *Lambb* case while varying the load imbalance which makes sense scientifically for ChaNGa. On the x-axis, this imbalance is measured by the percentage of active particles (an active particle generates CPU load). As for the experiments on PlaFRIM, we are able at worst to equalize the performances obtained with the best solution. Except compared to GreedyLB, the more imbalance the better performance for our load balancing algorithm. Another remark is that the gains obtained by RefineLB decrease as the the imbalance increases.

Figure 4b presents the execution time of the *Lambb* case on 1024 cores. In this use case, TreeMatchLB outperforms GreedyLB by 17%. Besides, it also has to be noted that the solution is computed considering eight chares per core, that is to say 8192 chares. This result shows the scalability of our algorithm and the possibility to adapt it to significant cases for coarse-grain paradigms.

5.2 Ondes3D

We also did run Ondes3D, a simulator of three-dimen-sional seismical wave propagation. This application has been ported on top of AMPI [7] in order to exploit the Charm++ features [21]. As a use case, we worked on a simulation based on the Mw6.6 2007 Niigata Chuetsu-Oki earthquake (Japan) [1]. The experiments on this real application were carried out on the PlaFRIM cluster.

In Fig 5, we compare TreeMatchLB with the standard Charm++ load balancers GreedyLB and RefineLB using 128 cores. The results show that these load balancers perform very poorly compared to our solution. Though in some cases, an execution without load-balancing step is comparable to a one using TreeMatchLB, we see that our approach is never outperformed and that its gain increases with the number of chares per core. Moreover, compared to previous results we see that the gain has increased with the size of the target platform.

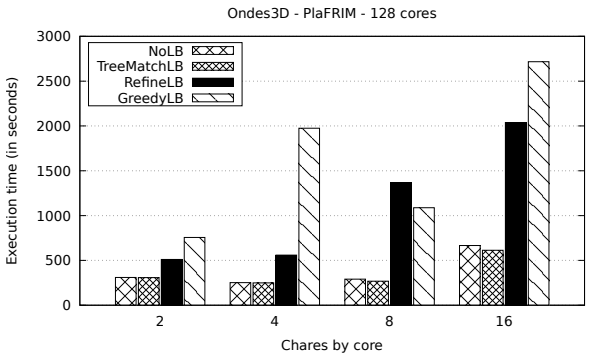


Figure 5: Ondes3D walltime on PlaFRIM when varying the number of chares per cores. 2, 4 and 8 chares/core: 16M cells. 16 chares/core: 64M cells

5.3 CommBench

The third test application, called CommBench, emulates a large amount of communication between chares. This benchmark is designed as follows: each chare sends a large chunk of data to two chares chosen randomly and the data size is heterogeneous. These communications are particularly irregular. This benchmark was tried out on 8192, 16384 and 32768 cores (resp. (256, 512 and 1024 XE6 nodes) on the Blue Waters platform.

Figure 6 shows the results obtained with this benchmark by using our load balancing algorithm TreeMatchLB. The plotted time measures correspond to the mean of ten executions. In order to show the algorithm scalability, we executed the application up to 32768 cores and 32 chares per core. It has to be noted that the standard native Charm++ load balancers (e.g. GreedyLB or RefineLB) did not work at this scale. This is why they are missing on the results. We compared the execution with TreeMatchLB to the one without load balancing. We can notice that the execution time is significantly improved while applying the TreeMatchLB algorithm. On the largest case, we see an improvement of 16.6%. The second remark is that our algorithm is able to scale for an important number of cores and a large amount of computing objects. On 32768 cores, we are able to load balance more than one million chares. This is due to the hierarchy and the distribution of TreeMatchLB. Adapted to a larger-grain paradigm, our algorithm should be able to deal with a substential number of tasks on very large-scale platforms.

5.4 Statistical comparison of the different strategies

Based on the above results, it appears that TreeMatchLB is in general the best strategy but that sometimes an other strategy provides similar performance. However, the competitive strategy varies depending on different parameters (Network topology, level of imbalance, problem size, etc.). An interesting question is therefore: is TreeMatchLB statistically better than the other load balancers? To have a statistical significance, of these experiments we measure, in addition of 5 lambb cases the performance of other use-cases: 4 different ChaNGa dwf[5] simulations and 8 Ondes3D runs (these later runs are excluded when we compare Multistep LB or Orb3dLB). All these runs have been chosen such that the amount of communication exchanged during execution

[5]dwf demonstrates how disk galaxies can form in a cosmological context.

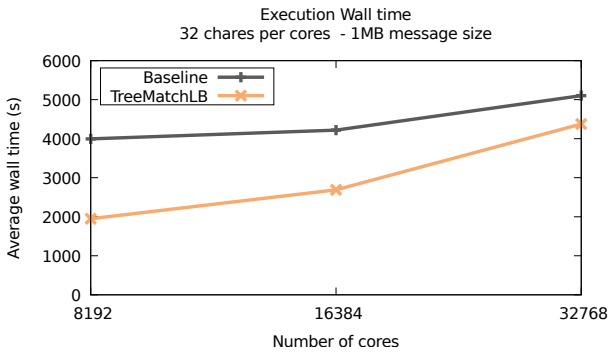


Figure 6: Average execution time of CommBench according to the number of cores with 32 chares per core and 1 MB per message. Carried out on XE6 nodes of the Blue Waters platform.

Statistical comparison of load balancers

TreeMatch LB	715.97 s / 1.45	343.41 s / 1.47	636.30 s / 1.86	269.77 s / 1.18	453.33 s / 1.34
[80 s, 1151 s]	No LB	−372.55 s / 1.01	−79.67 s / 1.28	−994.86 s / 0.62	−896.04 s / 0.68
[102 s, 585 s]	[−956 s, 211 s]	Refine LB	292.88 s / 1.27	183.50 s / 1.15	326.69 s / 1.28
[164 s, 1046 s]	[−821 s, 662 s]	[23 s, 563 s]	Greedy LB	154.89 s / 1.08	292.40 s / 1.18
[57 s, 483 s]	[−1416 s, −573 s]	[57 s, 310 s]	[−59 s, 369 s]	MultistepLB	118.14 s / 1.09
[271 s, 636 s]	[−1347 s, −445 s]	[106 s, 547 s]	[39 s, 546 s]	[19 s, 294 s]	Orb3dLB

Figure 7: Statistical comparison of TreeMatchLB with standard load balancers. Upper right panel: mean of the difference of the run duration. Lower left panel 95% confidence interval of the mean using the paired Student's t-test.

is very high.

Results are displayed in Figure 7. The figure is read as follows. On the diagonal (in shaded boxes), we have the different load-balancing strategies that we compare. On the upper right, each box is the mean of the difference of the runtime with the strategy on the row and the strategy on the column. For instance, we see that, runs using TreeMatchLB are, on average, 343.51s faster than the runs using RefineLB. On the lower left panel is the 95% confidence interval of the corresponding mean for the strategy on the row and the strategy on the column using the paired Student's t-test (see [8] chap. 13 p. 209). For instance, for TreeMatchLB vs. RefineLB, the interval is $[102s, 585s]$. The interpretation of the interval is the following. If the interval is positive there is a 95% probability that strategy on the row outperforms the strategy on the line. If the interval is negative then the strategy on the column outperforms the strategy on the row with probability 95%. Otherwise (the interval contains negative and positive values), we cannot conclude on which strategy is the best with 95% confidence.

Here, we see that TreeMatchLB outperforms all the compared strategies. This backup the intuition that even if in some cases other strategies may perform comparably with TreeMatchLB, on the average TreeMatchLB is the best choice for the type of use-case we are dealing with (i.e. highly communicating simulations). If we compare the other strategies (NoLB, RefineLB, GreedyLB, etc.) between them we see there that RefineLB is better than GreedyLB, Multi-

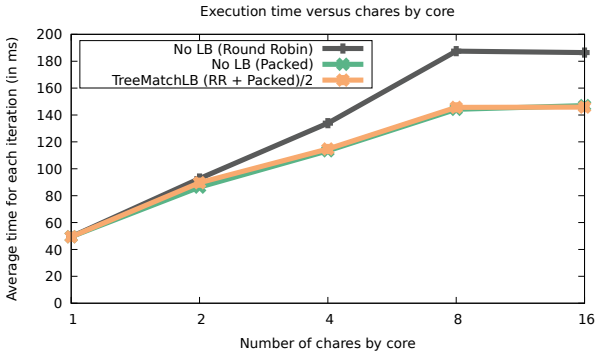


Figure 8: Evaluation of the initial placement at launch time for a balanced application whose optimal placement is known.

StepLB and Orb3dLB. We also see no statistical difference with NoLB and the 4 other ones. However if we reduce the confidence interval to 80%, the load balancers are almost totally ordered: TreeMatchLB $\succ$ RefineLB $\succ$ MultistepLB $\succ$ Orb3dLB $\succ$ GreedyLB $\approx$ NoLB.

5.5 Sensitivity to initial placement

The initial placement of the processing entities (here the chares) on cores actually impacts the application execution. An example is given on Figure 8 where we ran experiments on the eight cores of one PlaFRIM node. The application is balanced in terms of load: only communications impact the execution time. In this Figure, *No LB Round-robin* corresponds to an execution without a load-balancing algorithm where the groups of chares are affected on cores by applying the physical identity: group 0 of n chares on the physical core number 0, group 1 of n chares on the physical core number 1, and so on. On the contrary, *No LB Packed* assigns groups of chares by following the logical identity: group 0 of n chares on logical core number 0, group 1 of n chares on logical core number 1, and so on. On the PlaFRIM platform where the logical numbering differs from the physical one, this initial placement is decisive[6]. Therefore, the user has to know the characteristics of the underlying architecture to assign groups of chares optimally: we have a difference of up to 30% between the two cases. Thanks to TreeMatchLB, we are able to be oblivious to this initial placement. We computed an average of TreeMatchLB runtime starting with the two initial placements described earlier. This result is depicted on Figure 8 with the orange line. This graph shows that regardless of the initial placement, TreeMatchLB is able to move the chares to converge to an optimal placement.

6 Conclusion

Being able to dynamically balance the load of applications is necessary in the context of high-performance computation due to the hierarchical nature of modern architectures and the cost of communication and the irregular nature of many applications. In this paper, we propose a distributed and hierarchical load-balancer. Our approach is a two-level one. At the network level, we rely on LibTopoMap heuristics for dealing with arbitrary network topologies. For the fine

[6]Indeed, the cores of the Intel Xeon Nehalem included in a PlaFRIM node present this physical numbering scheme: 0,2,4,6,1,3,5,7

chare-to-core mapping, we use TreeMatch. Nonetheless, we showed that in general, a global load-balancing step is mandatory and we provide such step with a refinement algorithm. At the end, our algorithm optimizes both the communication and the computation costs.

We carried out many experiments on different architectures (PlaFRIM with an Infiniband fat-tree network and Blue Waters with a Gemeni 3D torus) and various applications and benchmarks. Results show that our topology-aware approach is in general the best strategy and, in the worst case, on par with the best solution. Statistically, our solution is the best on average. Moreover, thanks to its decentralized approach it is able to scale up to one million of chares while standard load balancers do not work for the largest setting we tested. An other interesting feature of our approach is that it relieves the user from the burden of taking into account the core numbering when launching its application.

In future work, we could improve the network awareness. First, some links are not considered. Concerning a 3d torus, two ways are possible in the same axis to link two nodes while we only consider one. Secondly, it could be interesting to evaluate the routing rules for each routers. This would require to manage the way the routing is performed. Another prospect could be to work on load-balancing time and more particularly to let TreeMatchLB chose the two levels of hierarchy to work on. For now, our solution first work at the network level then on the cores level. Should the number of cores per node be very substantial, an interesting feature could be to automatically chose to compute a placement of chares on groups of cores sharing the same cache. This improvement should increase the scalability of this hierarchical approach.

Acknowledgment

This research is partially supported by the NCSA-Inria-ANL-BSC-JSC-Riken Joint-Laboratory on Extreme Scale Computing (JLESC) in which we had allocated hours to the Blue Waters machine. We would like to thank Torsten Hoefler for providing us with the LibTopoMap code. The PlaFRIM experimental testbed, is being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS (see `https://plafrim.bordeaux.inria.fr`). Part of this work has also been funded by the ANR MOEBUS project (ANR-13-INFR-0001). We would like to thanks Fabrice Dupros et Rafael Keller Tesser for providing us with the Ondes3D code.

References

[1] Hideo Aochi, Ariane Ducellier, Fabrice Dupros, Mickael Delatre, Thomas Ulrich, Florent de Martin, and Masayuki Yoshimi. Finite difference simulations of seismic wave propagation for the 2007 mw 6.6 niigata-ken chuetsu-oki earthquake: Validity of models and reliable input ground motion in the near-field. *Pure and Applied Geophysics*, 170(1-2):43–64, 2013.

[2] Abhinav Bhatele. Topology Aware Task Mapping. In D. Padua, editor, *Encyclopedia of Parallel Computing (to appear)*. Springer Verlag, 2011.

[3] Abhinav Bhatelé and Laxmikant V. Kalé. Benefits of Topology Aware Mapping for Mesh Interconnects. *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, 18(4):549–566, 2008.

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. Hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia, February 2010. IEEE Computer Society Press.

[5] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[6] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *ICS*, pages 75–84, 2011.

[7] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[8] Raj K Jain. The art of computer systems performance analysis. 1991.

[9] E. Jeannot and G. Mercier. Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference*, volume 6272 of *Lecture Notes on Computer Science*, pages 199–210, Ischia Italie, SEPT 2010. Springer.

[10] Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier, and Gengbin Zheng. Communication and Topology-aware Load Balancing in Charm++ with TreeMatch. In *IEEE Cluster 2013*, Indianapolis, États-Unis, September 2013. IEEE.

[11] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Trans. Parallel Distrib. Syst.*, 25(4):993–1002, 2014.

[12] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[13] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[14] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[15] Laxmikant Kale, Anshu Arya, Abhinav Bhatele, Abhishek Gupta, Nikhil Jain, Pritish Jetley, Jonathan Lifflander, Phil Miller, Yanhua Sun, Ramprasad Venkataraman, Lukasz Wesolowski, and Gengbin Zheng. Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Technical Report 11-49, Parallel Programming Laboratory, November 2011.

[16] Laxmikant Kale, Anshu Arya, Nikhil Jain, Akhil Langer, Jonathan Lifflander, Harshitha Menon, Xiang Ni, Yanhua Sun, Ehsan Totoni, Ramprasad Venkataraman, and Lukasz Wesolowski. Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge. Technical Report 12-47, Parallel Programming Laboratory, November 2012.

[17] Laxmikant V. Kale. Programming Models at Exascale: Adaptive Runtime Systems, Incomplete Simple Languages, and Interoperability. *The International Journal of High Performance Computing Applications*, 23(4):344–346, October 2009.

[18] Laxmikant V. Kale and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[19] L.V Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) 93*, pages 91–108. ACM Press, September 1993.

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

[21] Rafael Keller Tesser, Laércio Pilla, Fabrice Dupros, Philippe Navaux, Jean-François Mehaut, and Celso Mendes. Improving the performance of seismic wave simulations with dynamic load balancing. In *Proceedings of 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2014)*, Torino, Italy, 2014. Accepted.

[22] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato. Adaptive techniques for clustered N-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2:1, March 2015.

[23] Harshitha Menon and Laxmikant Kalé. A distributed dynamic load balancer for iterative applications. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 15:1–15:11, New York, NY, USA, 2013. ACM.

[24] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2(1):1–16, 2015.

[25] G. Mercier and E. Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 39–49, Santorini, Greece, September 2011. Springer.

[26] Laercio L Pilla, Philippe OA Navaux, Christiane P Ribeiro, Pierre Coucheney, Francois Broquedis, Bruno Gaujal, and Jean-Francois Mehaut. Asymptotically optimal load balancing for hierarchical multi-core systems. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 236–243. IEEE, 2012.

[27] Laércio L Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Chao Mei, Abhinav Bhatele, Philippe OA Navaux, François Broquedis, Jean-François Méhaut, and Laxmikant V Kale. A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 118–127. IEEE, 2012.

[28] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, Alvaro Fazenda, Celso L. Mendes, and Laxmikant V. Kale. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil, 2010.

[29] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[30] Gengbin Zheng, Abhinav Bhatele, Esteban Meneses, and Laxmikant V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

[31] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, San Diego, California, USA, September 2010.