Communication and Topology-aware Load Balancing in Charm++ with TreeMatch

Emmanuel Jeannot*, Esteban Meneses†, Guillaume Mercier‡*, François Tessier§* and Gengbin Zheng†
*Inria, France
†UIUC, USA
‡Bordeaux Polytechnic Institute, France
§University of Bordeaux, France

***Abstract*—Programming multicore or manycore architectures is a hard challenge particularly if one wants to fully take advantage of their computing power. Moreover, a hierarchical topology implies that communication performance is heterogeneous and this characteristic should also be exploited. We developed two load balancers for Charm++ that take into account both aspects, depending on the fact that the application is compute-bound or communication-bound. This work is based on our TreeMatch library that computes process placement in order to reduce an application communication costs based on the hardware topology. We show that the proposed load-balancing schemes manage to improve the execution times for the two aforementioned classes of parallel applications.**

I. Introduction

Simulation is now wide-spread in the fields of science and technology, because actual experimentation, even when possible at all, is becoming increasingly costly both in terms of time and resources. This simulation step itself is often conducted as a scientific application executed on a dedicated computer. Moreover, since these simulations are expected to be more and more refined and precise and to deal sometimes with very large time scales, the required computing power is therefore huge. As a consequence, to provide this computing power to the scientific applications is a major challenge that needs to be addressed.

From a hardware perspective, only the parallel computers are able to deliver this sought-after computing power. However, programming a parallel architecture is not a trivial undertaking because of the intrinsic concurrency of these machines and of the multiplicity of the computing nodes and cores. As a consequence, dedicated paradigms, tools and environments have to be used to ease the task of developing parallel applications.

Structuring the application and exposing its parallelism as much as possible is an efficient solution advocated by several programming environments such as Cilk [1] or Charm++ [2], [3] for instance. Such environments rely on fine grain parallelism and create software computing entities that usually outnumber the physical computing units available in the parallel architecture. This induces execution issues that have to be handled by the environment's runtime system. For instance, the workload may vary from one core to the other, resulting in a global imbalance that can harm the application's performance. As a consequence, load balancing schemes are usually proposed in these environments and improve the performance by optimizing the use of the available resources.

978–1–4799–0898–1/13/$31.00

However, load balancing schemes fail to grasp an important aspect of today's parallel computers: their hierarchical and heterogeneous nature, communication wise. Indeed, since the advent of multicore/manycore CPUs, the organization of the memory banks along with the presence of several cache levels lead to non-uniform memory access times. That is, the time to access or move data depends on the physical location of the process inside of the computing node. Moreover, a good load balancing scheme does not always guarantee that the communication in the application will be optimized as well. As a consequence, load balancing schemes can be enhanced in order to take into consideration not only the load of the various cores, but also the amount of communication between the processes executing the various tasks of the parallel application. This can be achieved by matching the communication pattern of the application to the underlying hardware.

Therefore, the multiple application processes have to be placed carefully on the various cores/CPUs of the machine. This technique can improve application performance and ultimately scalability since communication costs are decreased. So far, a compelling idea is to try and regroup on the same computing node the processes sharing the most data so as to reduce the application communication costs [4], [5]. However, it now becomes necessary to push further this idea and apply it to the subsets of application processes *within* a computing node to take advantage of the complex structure of its memory.

In a previous work [6], we proposed a library called TreeMatch that computes a relevant process placement and targets clusters of multicore NUMA nodes. Since TreeMatch yields promising results for message-passing based applications on such targets [7], we believe that it could benefit to other programming models, and in particular the fine grain ones, by enhancing their load balancing schemes so as to better take advantage of clusters of NUMA multicore nodes. In this paper, we extend this library to deal with constraint placement and we study the results achieved by two new topology-aware load-balancers for applications developed with the Charm++ framework. The choice of the load-balancer depends on the class of the application: compute-bound or communication bound. By introducing TreeMatch in the load balancing schemes of Charm++, we managed to outperform some other communication-aware schemes available.

This paper is organized as follows: Section II exposes the motivation and context of the work while Section III lists various related works. The TreeMatch algorithm is described in Section IV and its integration in the load-balancing scheme of

Charm++ is described in Section V. The results achieved are detailed in Section VI while Section VII concludes this paper.

II. Motivations and Context

As discussed in the introduction, scientific applications that are in need of a massive computing power rely on parallel computers since they are the only architectures able to deliver the requested resources. However, a gap remains between the performance achieved at the application level and the performance of the underlying hardware. This issue can be addressed in several ways: first, using an appropriate programming model can help to improve the overall performance of the target application. Indeed, according to the application structure and its degree of parallelism, different programming models or frameworks might be more efficient. For instance, coarse grain applications with explicit communications are likely to use an interface such as the Message Passing Interface (MPI), while finer grain ones without explicit communications are likely to employ a standard such as OpenMP for instance. The target architecture is also a determinant factor in the choice of the programming tools. For instance, for cluster-based architectures featuring an interconnection network, it is necessary to exchange messages between the nodes, thus a message-passing based paradigm is likely to be the best fit, while in the case of shared-memory machines, the range of choices is wider since models with implicit communications are also possible (e.g. PGAS, multithreading, etc.) [1].

Second, it is necessary to use an optimized implementation of the chosen programming standard. This encompasses two distinct aspects: on one hand the implementation of the functions defined by the standard should obviously be the most efficient possible, on the other hand, the interactions of this implementation with the underlying target hardware should also be carefuly defined and enforced. This is usually the role of the underlying runtime system (e.g. a process manager for an MPI implementation). This runtime system will also trigger mechanisms that can improve the overall performance of the application. A mechanism such as load balancing falls into this category. By dispatching the workload dynamically (as the application executes itself) on the various processors of the architecture, one can expect a decrease in execution times. However, most of load balancing schemes fail to fully take into account other factors that impact application performance. Such factors include objects migration (when balancing the load, processing entities are likely to migrate from one core to another one) and communication costs for instance.

However, these communication costs are increasingly difficult to understand, because the current parallel architectures have undergone tremendous changes over the past few years. Indeed, the amount of computing cores available inside a node has increased dramatically. It is not uncommon to find machines featuring a dozen of cores per processor, and this number is expected to grow steadily in the forthcoming years. Moreover, this trend also impacts the memory organization and layout: memory banks are scattered throughout the node and the cache levels now form a complex, multi-level hierarchy. As a consequence, the communication costs within a single node are not homogeneous anymore. Practically speaking, these costs for exchanging data or messages between processes sharing the same node depend on their physical location (in the node). This is known as Non-Uniform Memory Access (NUMA) effects. An intuitive idea is therefore to place processes that communicate the most to processors/cores that are physically the closest to each other, because they share more cache levels and the NUMA effects are less prominent. Moreover, the increase of the memory resources does not follow the same trend as the number of processors. Indeed, the amount of memory available *per core* is expected to decrease. As a consequence, the issue of reducing the communication costs in a parallel application is going to become more and more crucial, even for compute-bound applications as the amount of communications in such applications is likely to increase due to the scarcity of the memory resources.

The underlying physical architecture has thus to be modeled in a convenient but realistic and usable fashion. One way is to assess the performance of the hardware with several benchmarks and to make use of these results to place the processes accordingly. This quantitative approach lacks dynamicity, requires to gather information prior to any application run and is prone to measurement errors. Another way to deal with this issue is to use a qualitative approach where hardware organization and structure shall guide the placement. The advantages of this approach are its flexibility since it does not rely on a prior collection of information, its genericity and dynamicity, provided that a relevant tool is used to perform this step. Currently, no such standard tool exists, but one can nevertheless rely on recent efforts such as HWLOC [8] that is available for a wide range of hardware.

To sum up, our approach is to consider additional factors when balancing the load for applications using a fine grain programming model. These factors include the migration costs and the communication costs between computing entities (e.g. tasks, processes, objects, threads). As for the communication costs, we decrease them thanks to a qualitative model of the underlying hardware, which ensures dynamicity, flexibility and genericity.

III. State of the Art

The issue of topology-aware mapping has been studied previously [5], [9]. In particular, MPI implementations make it possible to gather communication information such as the number of messages or the amount of data exchanged. Besides, some of these implementations feature means to bind processes on processing units in their runtime systems (e.g. process managers).

MPI 2.2 is a relevant example [10]. Beyond a static placement, some works focus on online placement by relying on a technique called *rank reordering* ([7], [4]). Finally, Dummler [11] explored the issue of hybrid, MPI + OpenMP application multithreaded process mapping. TreeMatch [6] is an algorithm which takes an application's communication pattern as input and the target machines's architecture to compute a relevant process placement.

Charm++ [2], [12] is a message passing-based programming environment based on the C++ language. However, while MPI considers processes in its programming model (with a

[1] Our course, PGAS and OpenMP implementations for distributed memory do exist, but their runtime system rely on the message passing paradigm for their internode communications.

granularity that is most of the time coarse), Charm++ model is based on a finer granularity by splitting computation in smaller tasks called *chares*. These chares are characterized by their CPU load, their input and output communication and some other useful fields. This makes it easier to introduce load balancing mechanisms. A strong advantage of Charm++ is therefore the possibility to design, plug and test load-balancers transparently without changing the application code. Moreover, several applications have been developed using Charm++ in different scientific topics. For instance, it is the case of NAMD [13] and LeanMD [14] (molecular dynamic applications), or ChaNGa [15] (cosmological computations).

Common load balancing schemes, which take into account the CPU load on each processing unit, have been extended in some works to take into account the topology of the underlying architectures. In a previous work, Bhatelé & Kalé [16] present the benefits of topology aware mapping on a torus topology. NucoLB and HwTopoLB [17] [18] apply load balancing based on a quantitative approach of the topology links (latency and bandwidth figures are necessary). Besides, this kind of strategy requires to assess the target architecture communication performance with appropriate tools before running any application. Our solution based on TREEMATCH is fully dynamic because we use only a qualitative approach for our representation of the hardware topology.

IV. EXTENSION OF TREEMATCH TO ACCOUNT FOR PLACEMENT CONSTRAINTS

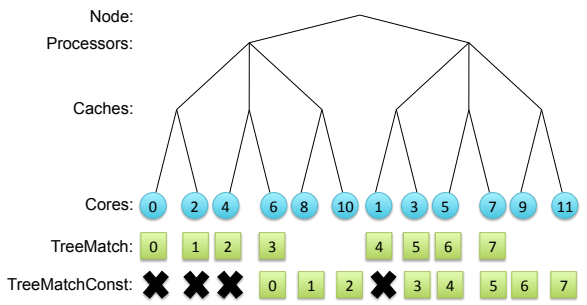

Fig. 1. A $2:3:2$ topology tree modeling a node with two processors featuring 6 PUs/cores each, where two PUs/cores share a common cache. Note that the core numbering follows the physical one and not the logical one. Two results are displayed. The basic TREEMATCH one and the TREEMATCH-CONST one with constraints on leaves/cores number 0, 2, 4 and 1 as in the example.

In this section, we present the process placement method based on our TREEMATCH solution. The aim of our work is to assign each process to its dedicated processing unit (usually a core) in order to reduce communication costs between processes. The core algorithm takes as input a matrix modeling the amount of communication between processes and a representation of the underlying hierarchical architecture modeled as a tree, where the leaves represent the processing units (PUs) on which the processes should be mapped (see Fig. 1). TREEMATCH considers balanced trees, that is, trees in which the arity of the node at each level is the same. This enables the following compact notation of trees: $a_1 : a_2 : \ldots : a_n$ where n is the number of levels and a_i is the arity of all the nodes at level i. Therefore, the tree depicted in Fig. 1 is modeled with this notation as: $2 : 3 : 2$. TREEMATCH relies on several algorithms and a first version was proposed in [6] and [7]. In this earlier version, when the number of PUs is larger than the number of processes, the user has no means to specify which PUs shall *not* be used and the algorithm decides where to allocate the processes using the communication cost as the sole decision criterion. However, such approach is suitable only in the case of shared-memory machines and when no load balancing is applied. In our case, we target clusters of multicore nodes addressing load-balancing issues. Moreover, if we consider eight processes to be mapped on the architecture modeled in Fig. 1, there is no guarantee that the earlier version of TREEMATCH would map the processes evenly. Indeed, as shown in Fig 1, leaves number 8, 10 , 9 and 11 are not used. It is not possible to mark a given subset of leaves as unused. To ensure an allocation that comply to this type of use, we have designed an enhanced version of the algorithm that is now able to take into account *constraints* explicitly by listing the PUs/leaves that cannot be used for the mapping. This new version is called TREEMATCHCONST to account for the constraints given by the user.

For instance, imagine that the user wants to prevent PUs/leaves number 0, 2, 4, and 1 to be used for some reason. We give this information as new input to the algorithm along with the topology description and the communication pattern as shown in Algorithm 1. For the sake of simplicity, we assume that the number of constraints plus the dimension of the communication matrix is equal to the number of leaves of the tree. If it is not the case, a workaround consists of padding the communication matrix with null values until we reach the required dimension and then, once the result is output, we simply ignore the mapping of the virtual processes corresponding to the padded values.

TREEMATCHCONST is a recursive algorithm. In our example, for the first call we have T, the tree depicted if Fig 1, m the communication matrix displayed in Fig 2 and $C = \{0, 4, 2, 1\}$. Let k be the arity of tree T for the corresponding recursive step (k=2 for the first recursive step). At line 2 of the algorithm, we k-partition the communication matrix but we take the constraints into account. As the constraints state that three leaves on the left subtree and one on the right subtree cannot be used, we need to partition the communication matrix so that exactly three processes are allocated on the left subtree and exactly five on the right one, with the goal of minimzing the communication cost between each partition. Unfortunately, no graph partitioner is able to provide an unbalanced and exact partitioning at the same time. For instance, Scotch [19] provides only balanced partitioning while Metis [20] never guarantees that a given partition has the exact specified size. Therefore, we have implemented a simple, greedy, randomized k-partitioner for performing this task. Being random, greedy k-partitioning does not necessarily provide a very good solution, so we perform this step a thousand times and keep the best solution. In our example, the first three processes are mapped on the left subtree and the five last processes on the right one. We can then call recursively TREEMATCHCONST with new inputs on each subtree. For instance in the left subtree we have only the communication matrix corresponding to the irst three processes (as shown in Fig. 3), the tree T is the one starting at the *processors* level and is of arity $k = 3$ and the constraints are $C = 0, 4, 2$. When the algorithm reaches the bottom of the tree, the result can be aggregated. In our example, it is

Proc	0	1	2	3	4	5	6	7
0	0	1000	10	1	100	1	1	1
1	1000	0	1000	1	1	100	1	1
2	10	1000	0	1000	1	1	100	1
3	1	1	1000	0	1	1	1	100
4	100	1	1	1	0	1000	10	1
5	1	100	1	1	1000	0	1000	1
6	1	1	100	1	10	1000	0	1000
7	1	1	1	100	1	1	1000	0

Fig. 2. Communication matrix example

Proc	0	1	2
0	0	1000	10
1	1000	0	1000
2	10	1000	0

Fig. 3. Communication matrix of the left subtree after step 1

displayed by Fig. 1.

It is worth to note that regardless of the version of the algorithm used, TREEMATCH uses structural information (a topology tree) and never needs quantitative information about the underlying hardware (e.g. bus speed, network bandwidth, latencies, etc.), as opposed to other approaches such as NucoLB [18] or Scotch [21]. We believe that this advocates for our approach, as dealing with qualitative and structural information does not require to assess the hardware performance and is therefore insensitive to incorrect or partial measures.

Algorithm 1: The TREEMATCHCONST Algorithm

Input: T// The topology tree
Input: m // The communication matrix
Input: C // The constraints array
1 $k \leftarrow$ arity at the top of the tree T.;
2 $p \leftarrow$constraint_k_partition(k, m, C); // finds partitions of size k taking the constraints into account
3 tab_m $\leftarrow$ split_com_mat(C, k, p); // Splits the communication matrix in k parts according to the partition just found above
4 tab_C $\leftarrow$ split_constraints (C, k, T) ; // Constructs a tab of constraints of size k: one for each partitions
5 **if** T *is not a leaf* **then**
// recursively calls TREEMATCHCONST on the k subtrees of the root of T;
6 **foreach** i *in* $0..k-1$ **do**
7 executes TREEMATCHCONST on the i^{th} subtree of T, tab_m[i], tab_C[i].
8 $r \leftarrow$ aggregates results of each subtrees;
9 **else**
10 $r \leftarrow$ assigns the process/constraint to T;
11 returns r as result for T;

V. LOAD-BALANCING IN CHARM++ WITH TREEMATCH

Charm++ is a runtime system and a programming language implementing the message passing paradigm. Unlike MPI, Charm++ does not manipulate processes but independant computing objects called *chares*. These fine grain objects can be in higher numbers than the hardware processing units (e.g. CPUs, cores). Moreover, if this model is used jointly with a dynamic load balancing system, it can easily level the CPU consumption and consequently improve the execution time of applications.

But while some applications are limited because of a huge imbalance (LeanMD [14] for example), some others feature chares which exchange lots of data (e.g. kNeighbor or Stencil3D). These differents behaviors led us to create two distinct load balancers based on TREEMATCH. The first one, called TMLB_MIN_WEIGHT, applies a communication-aware load balancing scheme by favouring the CPU load balance. To do so, it solves a maximum weight matching problem in order to minimize the chares migration. The second algorithm, designed for communication-bound applications and called TMLB_TREEBASED, computes a placement of groups of chares on each processing unit. Then, it considers each node and enforces some load balancing while keeping as much balanced as possible the chare placement in order to minimize communication costs. We detail these two algorithms in the following two sections.

A. Load balancing with communication and migration minimization

As explained previously, some applications are very unbalanced in terms of CPU load. However, such applications also exchange data. Our goal is thus to address both issues, that is, improve the load balancing whilst decreasing the communication costs. That is why we designed an algorithm that carries out some load balancing while keeping as close as possible communicating chares, taking into account the topology. However, if we only consider both problems, results show marginal improvements. This stems from the chares migration. When we consider all chares and reorder them according to their communication exchanges or their load, the probablity that a chare will stay on its original processing unit is low. The goal is therefore threefold : to balance the load, to minimize communication costs and to minimize chares migration. We detail how we reduce these migrations in our algorithm by solving a maximum weight matching problem.

This first load balancing algorithm can be found on Algorithm 2. To explain it, we can consider an application which creates a hundred chares and that shall be executed on four cores. After a few iterations, Charm++ calls the load balancing algorithm. First, it extracts the chares communication pattern from the application monitoring (provided by the Charm++ runtime). This results in a 100×100 matrix. Then, we create a fake topology tree by decomposing the order of this matrix in prime factors. In our example, the topology tree will be: $2 : 2 : 5 : 5$. Then, we run TREEMATCH to find an appropriate chares permutation in order to have the most communicating chares as close as possible in the tree. Once we obtain this permutation, we split it in a number of parts equal to the number of cores of the underlying architecture (four in our example). In this case, each part corresponds to the group of chares of each of the four subtrees in the fake topology's second level.

Then, we apply the `AssignChareOnCore` function as follows: we sort each part by decreasing load and thus each part corresponds to a core. The main loop first assigns each chare to its corresponding core by considering at each iteration the less loaded core and its most loaded chare. When a core has received all the chares of its part but remains the less loaded, we select chares from an other part. This algorithm keeps the most communicating chares together and applies

load balancing using the less loaded ones. Therefore, we can make a fine CPU load balancing. Moreover, at each new chare assignement, we update a migration matrix such as we increment $m\_mig[old\_core][new\_core]$ when the chare has to move from old_core to new_core.

At the end of this phase, we have as many groups of chares as we have cores. The remaining question is how to map these groups of chares onto the physical cores. We solve this problem with the goal of minimizing the chares migration. Minimizing the migrations corresponds to solve a minimum weight matching problem on the migration matrix. Indeed, each entry $m\_mig[i][j]$ gives the migration cost when allocating group i on core j. A minimum weight matching of this matrix is therefore an assignment of the groups to the cores such that the sum of migration costs is minimized. Finding such matching can be performed in polynomial time by the Hungarian algorithm [22].

Algorithm 2: The TMLB_MIN_WEIGHT Algorithm

```
   Input: m_chares The communication matrix between chares
   Input: n Order of m_chares
   Input: fake_T A fake topology tree with n leaves
   Input: m_mig The migration matrix between cores
1  ;
2  p ←Permutation(m_chares, fake_T);
3  SortEachPartDesc(p, nb_cores)
4  foreach i in 0..n do
5      c ←LessLoadedCore();
6      chare ←ChooseChare(c);
7      AssignChareOnCore(chare, c);
8      UpdateMigrationMatrix(m_mig, chare)
9
10 h ←HungarianAlgorithm(m_mig);
11 foreach chare do
12     SetNewPe(chare, h)
```

B. Tree-based chares placement and load balancing

The algorithm of the previous section is designed for compute-bound applications. In this section, we tackle the issue of balancing the load for communication-bound applications. Here, the load balancer based on TREEMATCH first reduces communication costs while balancing the load on each processing unit. Our algorithm is presented on Algorithm 3.

For example, consider a 110 chares applications to be executed on two nodes, eight cores architecture (four cores per node). At the beginning of the execution, Charm++ groups these chares to the cores. At each load-balancing step, the first part of the algorithm consists in building the communication matrix of these groups of chares, gathering the topology and applying TREEMATCH to permute these groups to the cores so that communication costs between cores are minimized. The second part of our algorithm goes up in the topology tree and considers the nodes, that is, the set of cores that share the same memory banks. For each node in our architecture, we apply locally and in parallel a load-balancing algorithm. For example, assume that the first step assigns 55 chares to each node. This means that these 55 chares are grouped in four parts (one per core). In order to balance the load of these chares on each of the four cores, we have to create a fake topology on which we will be able to bind the chares. Because of the grouping algorithm of TREEMATCH, we could fail to find a good placement if the prime factorization contains a large prime number. To avoid this problem, we use a new feature in TREEMATCHCONST: the possibility to put constraints on leaves in our topology. Thus, to create the needed fake topology, we take the number of chares to be bound and we increment this value to obtain a number for which the prime factorization will be only decomposed in 2 and 3. This increment will improve the ability of TREEMATCH to create groups and to find a good process placement. In our case, the increment leads to consider a topology of 64 leaves and yields the following binary tree: $2:2:2:2:2:2$. To map the 55 real chares to the 64 leaves of this tree and keep a good chares balance, we apply constraints regularly on each smallest subtree. These constraints will be given to the TREEMATCHCONST algorithm that is based on an unbalanced k-partitioning. Once we have a satisfactory chares permutation, we assign them on their new core. This is done using the same `AssignChareOnCore` method as in the previous algorithm.

Algorithm 3: The TMLB_TREEBASED Algorithm

```
   Input: m_pu The communication matrix between Pe
   Input: T The topology tree
2  p ←Permutation(m_pu, T);
3  foreach chare do
4      SetNewPe(chare, p)
5
6  foreach node in parallel do
       Input: m_chares The communication matrix between
              chares in the current node
       Input: n Order of m_chares
7      fake_T ←CreateFakeTopology();
8      p ←PermutationWithConstraints(m_chare, fake_T);
9
10     foreach i in 0..n do
11         c ←LessLoadedCore();
12         chare ←ChooseChare(c);
13         AssignChareOnCore(chare, c);
```

VI. EXPERIMENTAL RESULTS

In this section, we present the results obtained with our two load balancers. We selected three applications: the first one, called LeanMD [14], is a molecular dynamic application known to have a huge load imbalance. We did our tests for this application with TMLB_MIN_WEIGHT. The last two applications, kNeighbor (an iterative application where each chare communicates with k others) and Stencil3D (a three dimensional stencil application)[2] are known to exchange a lot of data. That is why we conducted the tests using the TMLB_TREEBASED scheme. All experiments were carried out on 16 nodes of the PlaFRIM platform. The nodes are linked with an InfiniBand interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR). Each node contains two Quad-core-INTEL XEON NEHALEM X5550 (2.66 GHz) processors. 8 Mbytes of L3 cache are shared between the four cores of a CPU. There are also 24 GB of 1.33GHz DDR3 RAM on each node. The operating system is a SUSE Linux (2.6.27 kernel). We used the repository version of Charm++ which was a 6.4.0 development version. Finally, for all these experiments, the metric accounting for chares/process affinity is the number of messages exchanged.

We compare our solutions with an execution without any load balancing (DummyLB or Baseline) on one hand and on

[2]kNeighbor and Stencil3D are part of the Charm++ benchmarks suite.

the other hand with the standard load-balancers that are available by default in Charm++. These load balancers implement greedy strategies or are the suggested ones for the application. In particular, GreedyLB (resp. GreddyCommLB) uses load (resp. load and communication) to assign chares on cores with the following strategy: the highest loaded chare is mapped on the less loaded core. RefineCommLB take objects from overloaded cores and assign them in order to reach an average load. RefineCommLB is one of the suggested strategies for KNeighbor. We did not compare our approach to NucoLB as it was designed for shared-memory machines.

A. LeanMD

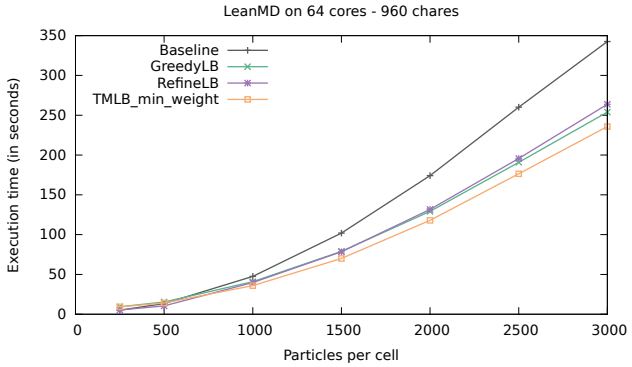

Fig. 4. Execution time (including load-balancing) vs. number of particles per cell of the LeanMD application on 64 cores for various load-balancing strategies

LeanMD is a molecular dynamics application which is very unbalanced. Among others parameters, the number of particles per cell (per chare in a software manner) can be modified in order to generate more computation and communication. We made this parameter vary in our experiments. This application provides some communication flow but it is insignificant compared to the load imbalance. Our TMLB_MIN_WEIGHT algorithm, which favours the load balancing, is a perfect case for this test. The results we obtained with this load balancer are presented on Figure 4. On this figure, we plotted the whole execution time (including the load balancing time) according to the number of particles per cell. We can see that except for the small cases, we outperform the other algorithms. On problems equal or larger than 2000 particles per cell, we reduce by 30% the execution time without load balancing (Baseline).

When we compare the load balancing time for each strategy we can notice that TMLB_MIN_WEIGHT takes on average 233 ms where RefineLB takes 1 ms. As for GreedyLB, the load balancing is calculated in 14.5 ms. However, even if TMLB_MIN_WEIGHT is slower than the two other load balancers, this drawback is counterbalanced by the benefits obtained regarding the total execution time of the application.

In Fig. 5 we present the number of chares that migrate from one core to a new one for the same execution than the one depicted in Fig 4. We see that GreedyLB does not take into account this aspect as almost all the chares do migrate. RefineLB is mainly an incremental strategy that balances the load by moving as few chares as possible, therefore, the number of migration is very small. The TMLB_MIN_WEIGHT

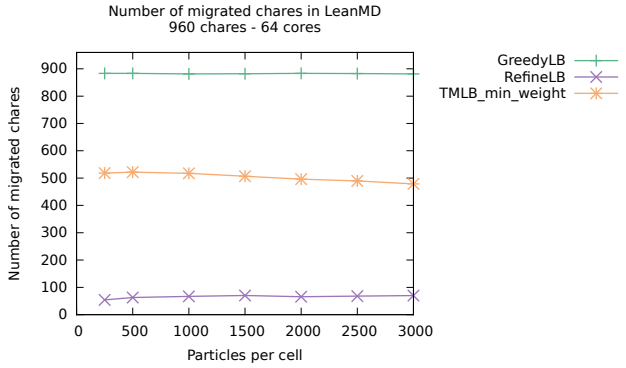

Fig. 5. Average number of migrated chares (among a total of 960 chares) for each load balancer and for each set of runs of LeanMD.

strategy finds a good compromize between migration cost and other factors impacting the execution time (e.g. load, communication, topology). Without migration minimization, the amount of migrations is much larger (around 680 migrations) and so is the execution time (approximately a 5% increase). Therefore this feature is necessary to achieve good performance.

B. kNeighbor

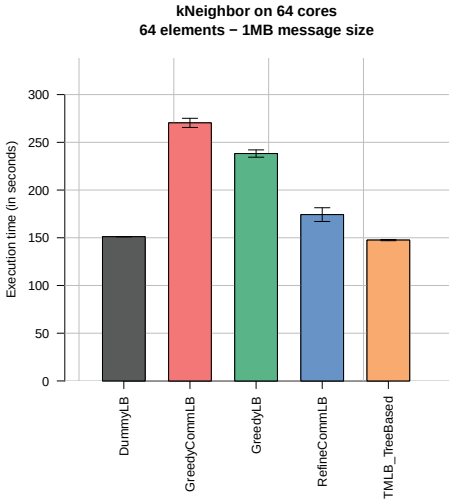

Fig. 6. Execution time (including load-balancing) of the KNeighbor application on 64 cores for several load balancing strategies with 64 chares and 1MB messages.

kNeighbor is a application designed to simulate intensive communication between a fixed number of chares (seven by default). Our results are presented in Figure 6, 7 and 8.

We carried out these experiments with respectively 64, 128 and 256 elements (chares). We can see that in all experiments, our solution is faster than all the other ones. When there are only a few elements, no load balancer yields good performance as compared to the native charm++ load balancers. When we reach 256 elements, the native load balancers can achieve interesting improvements, especially RefineCommLB. However, TMLB_TREEBASED manages to outperform it, by

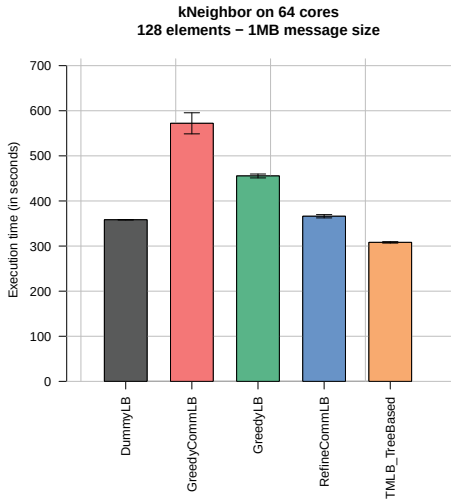

Fig. 7. Execution time (including load-balancing) of the KNeighbor application on 64 cores for several load balancing strategies with 128 chares and 1MB messages.

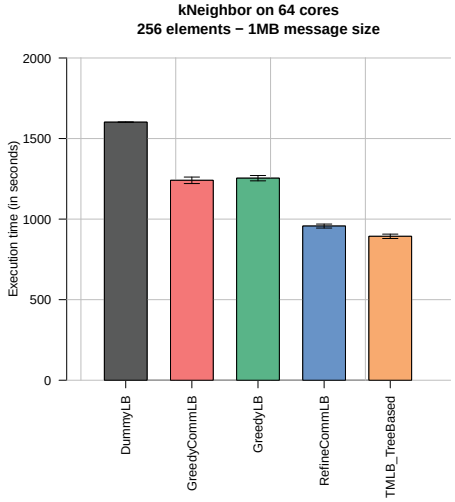

Fig. 8. Execution time (including load-balancing) of the KNeighbor application on 64 cores for several load balancing strategies with 256 chares and 1MB messages.

roughly 10%. Moreover, we see that the relative performance of TREEMATCH improves with the input size.

The Figure 9 represents the execution time of each load balancing strategy used in our kNeighbor experiments. Each load balancer follows a linear trajectory while the number of chares is doubled at each step on the x-axis. TMLB_TREEBASED is clearly slower than the other strategies but yields a good chare placement that improves the total execution time as shown in Figures 6, 7 and 8.

C. Stencil3D

Stencil3D is a 3-dimensional stencil with regular communication with some fixed neighbors. Because of this, it is not beneficial to apply a load balancing scheme every

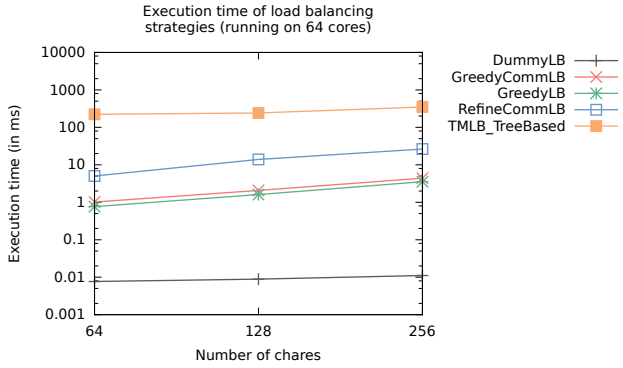

Fig. 9. Load balancing time of the different strategies vs. number of chares for the KNeighbor application.

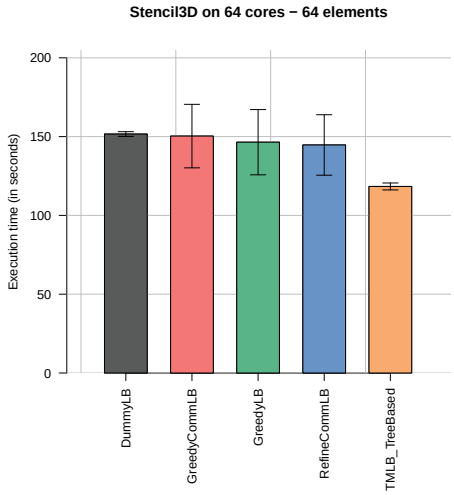

Fig. 10. Execution time (including load-balancing) of the Stencil3D application on 64 cores and 64 chares

ten iterations. The best results we achieved, for each load balancer, was when we balanced the load *only once*, after ten iterations. The experiments presented in Figure 10 follow this principle. Like kNeighbor, Stencil3D is a communication-bound application, so we applied TMLB_TREEBASED. We notice two important results. First, we obtain a gain of roughly 18% compared to the other strategies. Then, our algorithm offers a better execution time stability as shown by the error bars representing the standard deviation.

The load balancing time is very short for greedy strategies in Stencil3d experiments. Except for RefineCommLB which takes approximately 7.5 ms, the others greedy load balancers take less than 1 ms to determine the new chares assignment. Conversely, TMLB_TREEBASED takes on average 214.8 ms. As for the kNeighbor case, this load balancing time has to be compared to the total execution time of the application. The time needed by our strategy to find a good chare placement is completely hidden by the gain in execution time.

VII. CONCLUSION

Efficiently executing high-performance applications on parallel computers is a hard task. One efficient solution consists

in structuring the application and exposing its parallelism as much as possible in order to express all its parallelism. However, fine-grain parallelism raises the question of efficient load-balancing schemes.

In the literature, there exists many such load-balancing strategies. However, architectural advances have led to highly hierarchical computing platforms and therefore it is necessary to propose new solutions and strategies taking into account the load, the communication, the topology or the migration cost.

In this paper, we have studied two new load-balancing strategies and we have implemented them in the Charm++ computing environment. Both strategies address the issue of topology-aware load-balancing but one is targeted towards compute-bound applications while the other targets communication-bound applications. These solutions are based on the TREEMATCH library that is designed for mapping computation to tree-structured topologies and which have the advantage of using only qualitative information.

We have tested them on a distributed memory platform and compared them against standard Charm++ load balancing strategies. We have chosen real applications (LeanMD, Stencil3D and Kneighbor). Our results show that the proposed strategies lead to better execution times even if computing the load-balancing is higher when using TREEMATCH.

Future works are directed towards extending the scalability of the strategies by improving their parallelism and the gathering of the information required by them. For instance, we want to improve the adaptation of TMLB_TREEBASED concerning the choice of the hierarchy level where the algorithm is distributed in order to improve its scalability.

ACKNOWLEDGMENTS

This work was supported by INRIA-Illinois Joint Laboratory on Petascale Computing. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, 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/).

REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/209936.209958

[2] L. 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*. ACM Press, September 1993, pp. 91–108.

[3] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[4] B. Brandfass, T. Alrutz, and T. Gerhold, "Rank Reordering for MPI Communication Optimization," *Computer & Fluids*, Jan. 2012.

[5] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: an Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters," in *ICS*, G. K. Egan and Y. Muraoka, Eds. ACM, 2006, pp. 353–360.

[6] E. Jeannot and G. Mercier, "Near-optimal placement of mpi processes on hierarchical numa architectures," *Euro-Par 2010-Parallel Processing*, pp. 199–210, 2010.

[7] G. Mercier and E. Jeannot, "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering," in *EuroMPI*, ser. Lecture Notes in Computer Science, vol. 6960. Santorini, Greece: Springer, Sep. 2011, pp. 39–49.

[8] 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: IEEE Computer Society Press, Feb. 2010. [Online]. Available: http://hal.inria.fr/inria-00429889

[9] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-Scale Parallel Architectures," in *ICS*, D. K. Lowenthal, B. R. de Supinski, and S. A. McKee, Eds. ACM, 2011, pp. 75–84.

[10] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Traeff, "The Scalable Process Topology Interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, Aug. 2010.

[11] J. Dümmler and T. Rauber and G. Rünger, "Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008, pp. 141–148.

[12] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[13] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kale, R. D. Skeel, and K. Schulten, "NAMD—a parallel, object-oriented molecular dynamics program," *Intl. J. Supercomput. Applics. High Performance Computing*, vol. 10, no. 4, pp. 251–268, Winter 1996.

[14] V. Mehta, "LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines," Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[15] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[16] A. Bhatelé and L. V. Kalé, "Benefits of Topology Aware Mapping for Mesh Interconnects," *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, vol. 18, no. 4, pp. 549–566, 2008.

[17] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. Navaux, F. Broquedis, J.-F. Méhaut, and L. V. Kale, "A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems," in *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE, 2012, pp. 118–127.

[18] L. L. Pilla, P. O. Navaux, C. P. Ribeiro, P. Coucheney, F. Broquedis, B. Gaujal, and J.-F. Mehaut, "Asymptotically optimal load balancing for hierarchical multi-core systems," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 236–243.

[19] François Pellegrini, "Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," in *Proceedings of SHPCC'94, Knoxville*. IEEE, may 1994, pp. 486–493.

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

[21] F. Pellegrini, SCOTCH *and* LIBSCOTCH 5.1 *User's Guide*, ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, August 2008, http://www.labri.fr/perso/pelegrin/scotch/.

[22] S. Micali and V. V. Vazirani, "An $o(\sqrt{(v)}e)$ algorithm for finding a maximum matching in general graphs." in *Proc. 21st Ann IEEE Symp. Foundations of Computer Science*, 1980, pp. 17–27.