LLNL-CONF-536171

A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems

L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. A. Navaux, F. Broquedis, J. F. Mehaut, L. V. Kale

March 8, 2012

The 41st International Conference on Parallel Processing
Pittsburgh, PA, United States
September 10, 2012 through September 13, 2012

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems

Laércio L. Pilla[1,2], Christiane Pousa Ribeiro[2], Daniel Cordeiro[2], Chao Mei[3], Abhinav Bhatele[4],
Philippe O. A. Navaux[1], François Broquedis[2], Jean-François Méhaut[2], Laxmikant V. Kale[3]

[1]Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
{laercio.pilla, navaux}@inf.ufrgs.br
[2]LIG Laboratory – CEA / INRIA – Grenoble University – Grenoble, France
{christiane.pousa, daniel.cordeiro, francois.broquedis, jean-francois.mehaut}@imag.fr
[3]Department of Computer Science – University of Illinois at Urbana-Champaign – Urbana, IL, USA
{chaomei2, kale}@illinois.edu
[4]Center for Applied Scientific Computing – Lawrence Livermore National Laboratory – Livermore, CA, USA
bhatele@llnl.gov

***Abstract*—Multi-core compute nodes with non-uniform memory access (NUMA) are now a common architecture in the assembly of large-scale parallel machines. On these machines, in addition to the network communication costs, the memory access costs within a compute node are also asymmetric. Ignoring this can lead to an increase in the data movement costs. Therefore, to fully exploit the potential of these nodes and reduce data access costs, it becomes crucial to have a complete view of the machine topology (i.e. the compute node topology and the interconnection network among the nodes). Furthermore, the parallel application behavior has an important role in determining how to utilize the machine efficiently. In this paper, we propose a hierarchical load balancing approach to improve the performance of applications on parallel multi-core systems. We introduce NUCOLB, a topology-aware load balancer that focuses on redistributing work while reducing communication costs among and within compute nodes. NUCOLB takes the asymmetric memory access costs present on NUMA multi-core compute nodes, the interconnection network overheads, and the application communication patterns into account in its balancing decisions. We have implemented NUCOLB using the CHARM++ parallel runtime system and evaluated its performance. Results show that our load balancer improves performance up to 20% when compared to state-of-the-art load balancers on three different NUMA parallel machines.**

***Keywords*-load balancing, non-uniform memory access, memory affinity, multi-core, topology, cluster**

I. INTRODUCTION

The importance of non-uniform memory access (NUMA) architectures in the assembly of large-scale parallel machines has been increasing. The NUMA architecture is a scalable solution to alleviate the memory wall problem, and to provide better scalability for multi-core compute nodes. A NUMA architecture features distributed shared memory with asymmetric memory access costs. This characteristic, together with communication overheads of interconnection networks, can lead to significant communication costs on parallel applications. Therefore, to achieve high performance on such hierarchical machines, it becomes necessary to take the machine topology into account to avoid high communication costs.

In order to reduce communication costs among and within NUMA compute nodes, different approaches such as efficient memory allocation mechanisms, scheduling policies, and load balancing strategies can be used. The first approach focuses on distributing data, using the memory hierarchy of the NUMA multi-core node efficiently, and bringing data closer to the cores that will process it. This approach only reduces the communication costs within a compute node. The second approach reduces communication costs by dynamically mapping tasks of an application to the processing elements of the parallel machine. In this approach, some application characteristics are generally used to improve the affinity between tasks and data. The load balancing approach deals with this problem by doing a better distribution of the work among cores in order to avoid hot-spots and improve communication. The implementation of these approaches is usually linked to the characteristics of the target parallel programming environment [1], [2], [3], [4].

There are several options for programming applications on multi-core machines where some kind of optimization has been done, such as OPENMP [3], MPI [5], CHARM++ [6], XKAAPI [7] and UPC [8]. Usually, these runtime systems are responsible for some assignments like task orchestration, communication management, and work distribution among cores. However, most of these runtime systems provide a flat view of the machine, lacking information about the compute node topology and the interconnection network. They are not aware of the physical distance between the hardware components of the compute node or the interconnection network between the compute nodes. As a consequence, they can lead to sub-optimal application performance, since they may lack necessary information to efficiently match application communication patterns to the target machine topology.

In this paper, we propose a hierarchical load balancing approach to perform efficient distribution of the application load on parallel NUMA machines. We introduce the **NUCOLB** load balancer, which combines information about the NUMA multi-core topology, the interconnection network latencies and statistics of the application captured during execution. NUCOLB aims to improve load balance while avoiding unnecessary migrations and reducing intra- and inter-node communication.

In order to evaluate our load balancer, we have implemented it using CHARM++ as a test bed. CHARM++ is a C++-based parallel programming model and runtime system (RTS) that includes a load balancing framework. In our experiments, we use three benchmarks and a molecular dynamics application. Results show that the proposed load balancer can achieve performance improvements of up to 20% over existing load balancers on three different multi-core NUMA machines.

This paper makes the following contributions:

- We propose a novel model to represent and compose: (i) the way the memory system of a NUMA multi-core machine is connected together and (ii) the interconnection network overheads of parallel machines.
- We combine application and parallel machine characteristics into a hierarchical approach for load balancing.
- We present an efficient load balancing algorithm for parallel applications on NUMA multi-core machines.

This paper is organized as follows. In Section II, we characterize the problem handled by this research. Section III describes our approach and the proposed load balancer. Implementation details of the load balancing algorithm are presented in Section IV. In Section V, we present our evaluation methodology. The evaluated performance is reported in Section VI. In Section VII, we discuss some related work. Finally, we present concluding remarks and future work in Section VIII.

II. PROBLEM CHARACTERIZATION

We focus on the problem of enabling high efficiency and performance for dynamic parallel applications, such as molecular dynamics simulations, on hierarchical machines. When working with dynamic applications, an initial task mapping may provide poor performance due to load imbalance. This imbalance may come from different sources, such as input data, making it difficult to predict an optimal work distribution. An approach to solve this kind of problem involves the use of an online load balancing algorithm, as it is able to dynamically review the way tasks are distributed on the processing units. However, we believe that to consider only the load of the parallel application may result in performance losses due to communication overheads of the hierarchical machine. Thus, our objective for load balancing is to both maximize the use of the cores (minimize idleness) and also minimize the communication costs experienced by the application (maximize locality and affinity). We will elaborate on this problem in this section.

To develop a load balancing algorithm, one needs to gather and keep statistics about the execution behavior of the application, such as the execution time of tasks and their communication patterns. This can be done directly in the application code. This information may be sufficient to optimize our first objective. Examples of this would be timing the parallel sections of an OPENMP application, and keeping a list of communicating processes on MPI. However, developing a load balancing algorithm directly in the application may be a time consuming task, and the solution may work only for that particular application. A generic approach is to develop load balancing algorithms at the runtime level. This, however, requires the runtime to provide application information and the ability to move tasks on the parallel machine.

The runtime system may provide and work with a flat view of the parallel machine. This eases the development of applications for different systems. Nonetheless, this may also lead to poor performance, as current machines are built with complex compute nodes. These compute nodes may have asymmetric memory latencies and network communication costs. Figure 1 illustrates two scenarios where the communication impacts the application performance on NUMA multi-core machines. An arrow is used to represent the communication between two tasks. Although each scenario may have an equitable work distribution, the disregard of communication costs may hurt performance. For instance, in Figure 1(a), the communicating tasks are not able to benefit from a nearby shared memory bank and have to access data on a remote memory bank. This memory access may take 1.53 times longer than a local access, as measured in one of our test beds. This can be even worse, as depicted in Figure 1(b). Here, the communication has to go through a slower network connection, which presents a 3.4 times longer latency. These communications may play a bigger role in the application's performance with an increase in the number of communicating tasks, and asymmetries of the parallel machine.

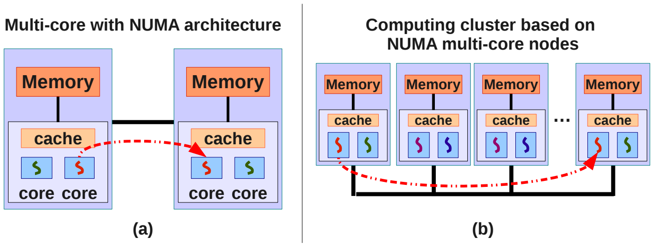


Fig. 1. Communication on hierarchical parallel machines.

To avoid the aforementioned problems, one approach involves providing the machine topology and communication costs to the load balancing algorithm. This information combined with the communication behavior of the application may be used to minimize its communication costs. This data can be measured and gathered by the programmer, who can add it to the application or runtime. This leads to the same shortcoming as before, since this may be time consuming and will only be effective for a particular machine. A better approach would be to provide information about the communication costs and machine topology in an automatic way that works on any machine and could be used by the load balancing algorithm.

To implement what we discussed above, we need to manage the gathering and combining of information about the dynamic application and the complex parallel machine in order to improve application performance. This improvement is obtained by minimizing the communication costs and maximizing the work distribution. In addition, we have to deal with doing this independently of the application and architecture. In the next section, we describe our proposal to solve this problem.

III. Hierarchical Load Balancing Approach

Scheduling theory shows that the problem of finding the optimal load balance for a parallel program is NP-hard [9]. On clusters of NUMA machines, the problem becomes even more challenging due to intra-node asymmetric memory access costs and inter-node communication costs. On these systems, an action taken by the load balancer to equalize the load on the available processors may actually make the overall performance worse by increasing the communication time.

In order to cope with the complexity introduced by these new platforms, we have developed a new load balancing algorithm that considers the specifics of NUMA parallel machines. In this section, we present NucoLB (Non-Uniform COmmunication costs Load Balancer), a load balancing algorithm developed for platforms composed of NUMA compute nodes.

A. Load Balancing Heuristic

Our load balancer relies on a heuristic that combines information about the application and machine topology to reduce the load imbalance of parallel applications. The heuristic works like a classical list scheduling algorithm [9], where tasks or jobs are rescheduled from a priority list and assigned to less loaded processors in a greedy manner. The priority list is dynamically computed at each load balancing step and is based on the load of the tasks. List scheduling algorithms are efficient, as they typically run in polynomial time, and provide good results in practice.

The goal of the heuristic is to improve the application performance by balancing the load on the nodes while reducing the remote memory access costs perceived by the application. The algorithm exploits information about the *application data* and the *machine topology*. Application data comprises of all information about the parallel application that can be probed at runtime: task execution times, communication information, and the current task mapping. The machine topology is all information that can be gathered at runtime about the machine hardware that is executing the application. A parallel machine can be characterized in terms of the number of NUMA nodes, cache memory sizes, sharing of cache among cores, and grouping of nodes.

The main idea of this load balancing algorithm is to iteratively map tasks — from the most to least CPU-intensive ones — assigning each of them to the core that offers the least overhead to its execution. In order to decide if a task t should be scheduled on core c to rebalance the load, the algorithm considers information about the execution of the application, such as the current load on core c, the amount of communication between task t and other tasks currently assigned to the NUMA node of core c (and, likewise, the communication between task t and tasks currently outside this NUMA node), and information about the communication costs related to the topology of the parallel machine. More formally, the cost of placing a task t on a core c is estimated using the following equation:

$$\begin{aligned}\text{cost}(c,t) = \text{coreLoad}(c)\, +& \\ \alpha \times [r_{\text{comm}}(t,c)& \\ \times \text{NUCO FACTOR}(\text{comm}(t), \text{node}(c))& \\ - l_{\text{comm}}(t,c)]&\end{aligned}$$

The equation presents a weighted linear sum of the costs involved in the execution of task t on core c. In this equation, coreLoad(c) represents the total load of all tasks currently assigned to core c (in other words, it is the sum of execution times, in seconds, of the tasks currently assigned to core c).

All communication costs are normalized by a factor α that controls the weight that they have over the execution time. l_{comm} represents the local communication i.e. the number of messages received by task t from tasks on other cores of the same NUMA node as core c. In other words, it represents the gains obtained by scheduling this task near other communicating tasks already mapped to this NUMA node. r_{comm} expresses the remote communications i.e. the number of messages received by task t from tasks on other NUMA nodes.

The topology of the platform is considered to infer the costs associated with non-uniform inter-node and intra-node communication latencies. To model this topology, we combine the hardware characteristics of the machine with its communication costs represented by the NUCO FACTOR. The communication cost of receiving a message on a NUMA node i from a NUMA node j is defined as:

$$\text{NUCO FACTOR }(i,j) = \frac{\text{Read latency from } i \text{ to } j}{\text{Read latency on } i}$$

If i and j are on the same multi-core compute node, this cost refers to the well-known NUMA FACTOR that measures the cost of accessing a different memory bank inside the node. If i and j are on different nodes, this cost relates to the cost of using the network to communicate. Section IV refers to the intra-node communication cost as NUMA FACTOR and the inter-node communication cost as NETWORK FACTOR, and presents details on how these costs can be measured. For the purpose of the algorithm, it is enough to note that this factor measures the latency caused by the communication between nodes i and j and that it can be easily calculated at runtime. In addition, it is important to notice that the value of the NUCO FACTOR that multiplies remote communications depends on the involved NUMA nodes, and that multiple NUMA nodes may be the source of communication. The NUCO FACTOR is calculated for the NUMA node of core c (represented by node(c)) and all the NUMA nodes of remote tasks communicating with task t (comm(t)).

The discussed heuristic considers the number of exchanged messages because it represents the number of accesses to the distributed shared memory. Since messaging time is related to the memory access costs, it is multiplied by the NUCO FACTOR when considering remote accesses. In addition, local communications are subtracted from the overall cost to favor their occurrence.

B. NucoLB*'s Algorithm*

Combining application and topology information in the heuristic presented in Section III-A, we have implemented a new load balancer, named NucoLB, which is better adapted for platforms with NUMA parallel machines. It is a greedy list scheduling algorithm that assigns the task with the largest execution time to the core that presents the smallest cost. The choice of a greedy algorithm is based on the idea of fast convergence to a balanced situation by mapping the greatest sources of imbalance first. The pseudocode for NucoLB is presented in Algorithm 1. Table I summarizes the parameters considered by NucoLB.

Algorithm 1: NucoLB

```
Input: $\mathcal{T}$ set of tasks, $\mathcal{C}$ set of cores, $\mathcal{M}$ mapping of
       tasks to cores
Output: $\mathcal{M}'$ mapping of tasks to cores
$\mathcal{M}' \leftarrow \mathcal{M}$
while $\mathcal{T} \neq \emptyset$ do
    $t \leftarrow k \mid k \in \arg\max_{l \in \mathcal{T}} \text{taskLoad}(l)$
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
    $c \leftarrow p, p \in \mathcal{C} \wedge \{(t,p)\} \in \mathcal{M}$
    $\text{coreLoad}(c) \leftarrow \text{coreLoad}(c) - \text{taskLoad}(t)$
    $\mathcal{M}' \leftarrow \mathcal{M}' \setminus \{(t,c)\}$
    $c' \leftarrow p \mid p \in \arg\min_{q \in \mathcal{C}} \text{cost}(q,t)$
    $\text{coreLoad}(c') \leftarrow \text{coreLoad}(c') + \text{taskLoad}(t)$
    $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{(t,c')\}$
```

The algorithm iteratively tries to map the task t with the largest execution time that has not been evaluated yet. t is initially mapped to core c. This information is obtained from the initial mapping $\mathcal{M}$. The load of this task is subtracted from the load of c before evaluating the cost function (cost(c,t) is explained in Section III-A). This subtraction is an important characteristic of this algorithm, as it avoids unnecessary migrations by reducing the load of c. This is important because the overhead resulting from migrations cannot be estimated without information about the size of the task in memory. After this subtraction, the algorithm evaluates all possible mappings for t and chooses the core c' that presents the smallest cost. Once this choice is made, the load of task t is added to the load of c'. The algorithm continues by searching for a new mapping for the next task.

TABLE I
Summary of the main parameters of NucoLB.

Parameter	**Definition**
cost(c,t)	Cost to map task t to core c.
taskLoad(t)	Execution time (load) of task t.
coreLoad(c)	Sum of the loads of tasks mapped to core c.
$l_{\text{comm}}(t,c)$	Number of local messages received by task t.
$r_{\text{comm}}(t,c)$	Number of remote messages received by task t.
comm(t)	NUMA nodes of remote tasks communicating with task t.
node(c)	NUMA node of core c.
NUCO FACTOR(i,j)	Latency factor between NUMA nodes i and j.

Considering n tasks and m cores, this algorithm presents a complexity of $\mathcal{O}(n^2m)$ in the worst-case scenario involving all-to-all communications. However, since this kind of communication is usually avoided in parallel applications for scalability, assuming a small, constant vertex degree of the communication graph, NucoLB has a complexity of $\mathcal{O}(nm)$.

IV. Implementation Details

Designing an efficient load balancing policy for a target hierarchical parallel machine can be a difficult and time-consuming effort. Detailed information from both the parallel application and the machine topology is required. This includes knowledge about how tasks are generated and executed, how data is allocated and shared among tasks, how processing units are organized, and the multiple levels of distributed memory over the machine. In this section, we present the implementation details of NucoLB, as well as information about obtaining the inputs for this load balancer.

To implement NucoLB, we use the Charm++ parallel runtime system (RTS) as the underlying framework. It provides an object oriented parallel programming language with the goal of improving programmer productivity. It abstracts architectural characteristics from the developer and provides portability over platforms based on shared and distributed memory. Parallel Charm++ applications are written in C++ using an interface description language to describe their objects [10].

Computation in Charm++ applications is decomposed into objects called *chares*. The programmer describes the computation and communication in terms of how these chares interact and the Charm++ RTS manages all messages generated from these interactions. Chares communicate through asynchronous messages. Furthermore, the Charm++ RTS is responsible for physical resource management on the target machine. Chares represent the tasks considered by NucoLB.

Charm++ also has a mature load balancing framework. It provides detailed information about the application, as required by NucoLB. Examples are the load of each chare, the load on each core, the number of messages and the amount of bytes exchanged among chares, and the current mapping of chares to cores. The load of each chare includes the time spent on computation and communication. The load on each core is the sum of the loads of all its chares and other runtime overheads. All this information comes from previous time steps, as load balancing in Charm++ is measurement-based. This implies that this is useful for applications with persistent load patterns or those with abrupt but infrequent changes.

NucoLB has been implemented on top of the Charm++ load balancing framework, as it provides an interface to plug-in new algorithms. It also enables the allocation of dynamic structures and the gathering of information during Charm++'s startup. We believe that NucoLB could also be used with other environments. However, it would require that they provide the necessary information about the tasks and the ability to choose a new scheduling.

Although Charm++ provides the necessary information about the parallel application, it does not provide any information about the intra-node topology. In order to create our topology model, we need the parallel machine characteristics, such as the number of NUMA nodes, cache memory sizes, sharing of caches among cores, grouping of NUMA nodes, and the interconnection network topology. Several libraries can provide this information about the machine topology. For instance, the hwloc [11] library generates a generic view of any compute node, in which processing units and memory caches are organized in a tree-based fashion. However, hwloc does not provide information about the way NUMA nodes are connected together and how compute nodes are interconnected. This information is critical in the context of hierarchical load balancing. The load balancer needs to know how *far* from each other the tasks are mapped, so that it can reduce the communication costs. To get this information, we extended the hwloc architecture model with our topology model of NUCO factor. This is done by adding **NUMA factors** and **NETWORK factors** to represent the way NUMA nodes and compute nodes are connected together.

The NUMA factor is obtained for all NUMA nodes of the target multi-core compute node through benchmarking, resulting in a square matrix of NUMA factors. Each NUMA factor is computed using the LMbench benchmark [12] that obtains the access latency on a memory bank. LMbench is a set of synthetic benchmarks that measures the scalability of multiprocessor platforms and the characteristics of the processor micro-architecture [12].

We use a ping-pong benchmark from the OpenMPI distribution to obtain the NETWORK factor. This benchmark allows us to get the latency to access data on remote compute nodes. The LMbench benchmark is used to get the access time to a memory bank on the local compute node. Then, these latencies are used to compute the NETWORK factor, resulting in a square matrix of factors that represents how compute nodes are connected together.

Both factors needs to be computed and stored only once for a target parallel machine. During the installation of Charm++, we run a script that executes LMbench and ping-pong benchmarks on the machine to obtain the factors. During the initialization of NucoLB, they are loaded into dynamic structures used later by its algorithm.

V. Experimental Setup

In this section, we describe the NUMA platforms, benchmarks and load balancers used for conducting the experiments.

A. Description of NUMA Machines

We have conducted tests on three representative multi-core platforms with different NUMA characteristics:

- **NUMA32**: A single node with four eight-core Intel Xeon X7560 processors. Each core has private L1 (32 KB) and L2 (256 KB) caches and all cores on the same socket share a L3 cache (24 MB).
- **NUMA48**: A single node with four twelve-core AMD Opteron processors. The cores have private L1 (64 KB) and L2 (512 KB) caches, and an L3 cache (10 MB) shared among six cores.
- **20xNUMA4**: This cluster has 20 compute nodes, each composed of two dual-core AMD Opteron 2218 processors. The cores have private L1 (64 KB) and L2 (2 MB) caches. The compute nodes are interconnected by Gigabit Ethernet using a bus topology.

All machines run the Linux kernel 2.6.32 and have the GNU Compiler Collection. 20xNUMA4 is part of Grid'5000 [13]. We used the Charm++ release 6.3.0 with the optimized multi-core build [6]. The MPI library used on the parallel machine 20xNUMA4 is OpenMPI.

In Table II, we summarize the hardware characteristics of these machines. Memory bandwidth obtained from the Stream-Triad operation [14] is also reported. For each machine, NUMA factor is also reported in intervals, meaning the minimum and maximum costs to access remote memory in comparison to local memory.

TABLE II
Overview of the NUMA multi-core platforms.

Characteristic	**NUMA32**	**NUMA48**	**20xNUMA4**
Number of nodes	1	1	20
Number of cores	32	48	4
Number of sockets	4	4	2
NUMA nodes	4	8	2
Clock (GHz)	2.27	2.2	2.6
Highest level cache (MB)	24 (L3)	10 (L3)	2 (L2)
DRAM capacity (GB)	64	256	4
Memory bandwidth (GB/s)	35.54	75.19	6.6
NUMA factor (Min;Max)	[1.36; 3.6]	[1.10; 2.8]	[1.52;1.53]

B. Charm++ Benchmarks and Mini-apps

To investigate the performance of our load balancer, we selected three benchmarks from the Charm++ distribution and one mini-app from the molecular dynamics field. These were selected due to their varied range of communication and computation characteristics (see Table III).

TABLE III
Benchmarks and mini-app characteristics.

	Number of Chares	Number of Iterations	Communication Topology	Type	LB Frequency
lb_test	200	50	random	compute-bound	10
kNeighbor	400	50	ring	comm-bound	10
stencil4D	256	50	4D mesh	compute-bound	10
LeanMD	1875	300	irregular	compute-bound	60

The selected benchmarks are: (i) *lb_test*, a synthetic imbalanced benchmark that can choose from different communication patterns; (ii) *kNeighbor*, a synthetic iterative benchmark where a task communicates with k other tasks at each step ($k = 7$ for this paper); and (iii) *stencil4D* an imbalanced four-dimensional stencil computation. Each benchmark has a different communication topology. For instance, *kNeighbor* uses a ring whereas the *lb_test* benchmark has a random communication graph. *stencil4D* communicates with neighbors on a four dimensional grid.

LeanMD is a molecular dynamics (MD) mini-app written in Charm++ and based on the popular MD application NAMD [15]. It simulates the behavior of atoms based on the Lennard-Jones potential, which is an effective potential that describes the interaction between two uncharged molecules or atoms. In *LeanMD*, the computation is parallelized using a hybrid scheme of spatial and force decomposition. In each iteration, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is assigned to a different set of objects called computes. Based on the forces sent by the computes, the cells perform the force integration and update various properties of their atoms – acceleration, velocity and positions. The communication pattern of *LeanMD* is more complex than the other benchmarks. In this application, there are two arrays of tasks that communicate with each other. In the first step of communication, tasks of the three dimensional (3D) array communicate using a multicast operation with tasks in the six dimensional (6D) array. Then, in the second step, tasks of the 6D array communicate with one or two tasks of the 3D array.

C. Load Balancers

To evaluate the performance of the proposed load balancer, we compare the results obtained using it to the baseline (without any load balancing algorithm) and to three load balancers available in Charm++. These load balancers are: GreedyCommLB, ScotchLB and TreeMatchLB.

GreedyCommLB reassigns tasks in a greedy fashion. The algorithm iteratively maps the heaviest task to the least loaded core while considering the communications among tasks. ScotchLB is based on the graph partitioning algorithms of Scotch [16]. This strategy considers both the execution time and communication graph to improve load balance. TreeMatchLB is based on a match of the communication graph of the application with the machine topology [5]. In order to reduce communication overhead, the load balancer assigns tasks to cores of the machine depending on the amount of data they exchange.

Similar to NucoLB, these load balancers consider the execution time and communication graph of the application. However, these algorithms (with the exception of ScotchLB) do not consider the original mapping of objects nor measure the communication costs of the machine.

As mentioned before, a Charm++ application calls the load balancing framework at pre-determined steps. For *LeanMD*, a load balancing call is made every 60 iterations. For the other benchmarks, this is done at every 10 iterations. The results in the next section show an average of a minimum of 20 runs for each benchmark. They present a statistical confidence of 95% by Students t-distribution and a 5% relative error. The value of α (see Section III-A) used in the experiments is 10^{-5}.

VI. Results

The performance impact of balancing the load in Charm++ programs depends on several different parameters, such as the iteration time of the application, the number of chares, the load balancing frequency, the load balancing algorithm's execution time, etc. Our main metric to evaluate the impact of load balancing is the total execution time of the application, or makespan. We also consider the average iteration time after load balancing and the average load balancing algorithm execution time. The iteration time does not consider the iterations before load balancing, nor overheads related to load balancing. The load balancing time is comprised of the time spent executing the algorithm and migrating the objects. In the following subsections, we first present results on NUMA48 and NUMA32, and then we explain the results obtained with *LeanMD* on the 20xNUMA4 cluster.

A. Performance Improvements on NUMA Compute Nodes

The *lb_test* benchmark presents a large variation in object loads, from 50 ms to 200 ms, and a small number of objects per core. It starts in an imbalanced state. Figure 2 illustrates the initial imbalanced iterations of *lb_test* and the balanced state after applying NucoLB's algorithm, as presented by the Projections performance analysis tool [17].

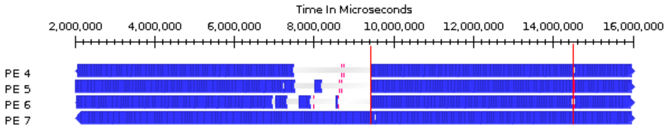


Fig. 2. Execution of *lb_test* on cores 4 to 7 of NUMA32, as captured by Projections. Each bar represents a core. The blue areas represent the execution of objects, and the white areas represent idleness. The period between 2 and 9.4 seconds illustrates the initial imbalanced state of the benchmark. PE 7 is the most loaded core. The period between 9.4 and 14.5 seconds represents the 10 iterations after calling NucoLB and before calling it again. The period after 14.5 seconds shows the beginning of remaining iterations.

Table IV summarizes the execution times obtained for *lb_test*. Most load balancers obtain speedups between 1.25 and 1.39 when compared to the baseline. The baseline represents the execution time of the application without using a load balancer. The performance improvements come from similar speedups in iteration time. The best results are obtained by using NucoLB. It achieves speedups of 1.39 over the baseline on NUMA48 and of 1.30 on NUMA32. This represents keeping the idleness of NUMA48 and NUMA32 cores as low as 5% and 4%, respectively. NucoLB migrates an average of 14 objects per load balancing call, while other load balancers migrate 195 objects on the average. This, however, has a small impact on the total execution time, as these objects do not use much memory. For instance, NucoLB's load balancing time is 9 ms on NUMA32, while that of ScotchLB is 12 ms. The small amount of communication, which adds to a total of 1 MB per iteration, reduces the impact of considering the communication costs of the machines. Nonetheless, the positive results with this computation-bound benchmark emphasize the advantages of a list scheduling approach.

The execution times measured for the *kNeighbor* benchmark are presented in Figure 3. *kNeighbor* is a communication-bound benchmark. In this experiment, objects communicate

TABLE IV
TOTAL EXECUTION TIME IN SECONDS AND SPEEDUP FOR THE LB_TEST BENCHMARK FOR DIFFERENT LOAD BALANCERS.

Load balancers	**NUMA32**		**NUMA48**	
	Time	Speedup	Time	Speedup
Baseline	36.19	—	27.93	—
NucoLB	27.95	1.30	20.09	1.39
GreedyCommLB	28.57	1.27	20.27	1.39
ScotchLB	28.91	1.25	21.53	1.30
TreeMatchLB	38.40	0.94	29.56	0.94

approximately 9 GB per iteration. Unlike *lb_test*, this benchmark starts in a balanced state. Charm++ distributes objects in a round-robin fashion through the cores. *kNeighbor* has twice as many objects as *lb_test*, and these objects have a regular and similar behavior. This explains the original balanced state of the application. NucoLB is able to reduce the average iteration time of *kNeighbor* on NUMA32 by 30 ms. This reduction is less noticeable in the total execution time due to the load balancing overhead of 150 ms per load balancing call. This cost comes from migrating an average of 116 objects. On the same machine, other load balancers migrate an average of 388 objects, which results in a migration time of more than 700 ms per load balancing call. This helps to explain why other load balancers end up increasing the total execution time. In particular, GreedyCommLB tries too hard to keep communicating objects on the same core. This leads to overloaded cores, which increases the iteration time by 50%. On NUMA48, NucoLB is the only load balancer that improves performance over the baseline (a speedup of 1.14). This was possible because NucoLB has a small load balancing overhead, and benefits from the knowledge of the machine topology. By using it, NucoLB is able to keep communicating objects on the same NUMA node and reduce communication costs. This results in a 6% reduction in the average load of each core, or 80 CPU-seconds.

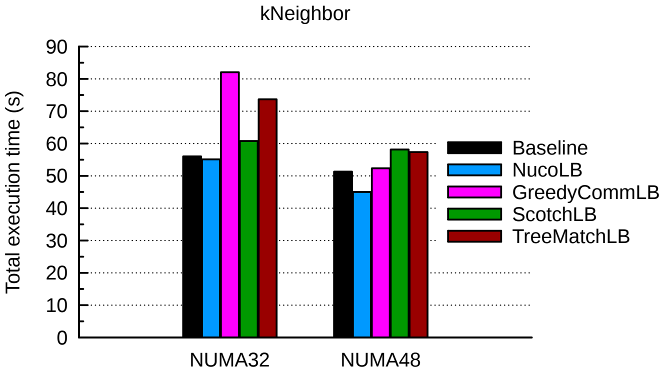


Fig. 3. Comparison of different load balancers for the *kNeighbor* benchmark.

Figure 4 illustrates the total execution time for the *stencil4D* benchmark. *stencil4D* has a similar number of objects as *lb_test*, and it also starts in an imbalanced state. Imbalance comes from an original mapping that underloads some cores, and from objects with different computational loads. Its objects send boundary information using messages of approximately 260 KB each, which adds to a total communication of 460 MB per iteration. *stencil4D* occupies 5.5 GB in memory, which means that it has large objects. For both machines, NucoLB shows the best performance. It obtains a speedup of 1.18 over the second best load balancer, GreedyCommLB, on NUMA32. When considering only the iteration time after load balancing, the speedup is 1.21. The difference in these speedups comes from the initial iterations of the benchmark, which happen before any load balancer can fix the imbalance.

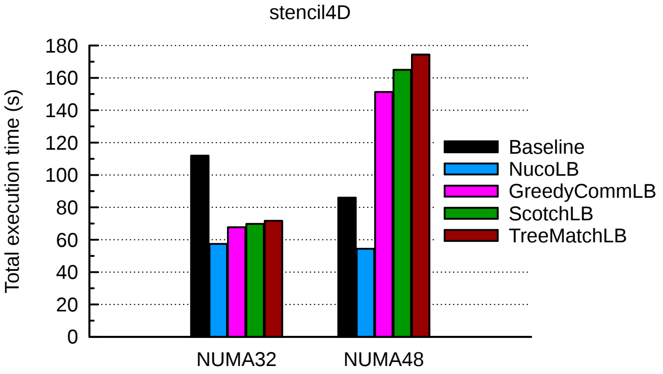
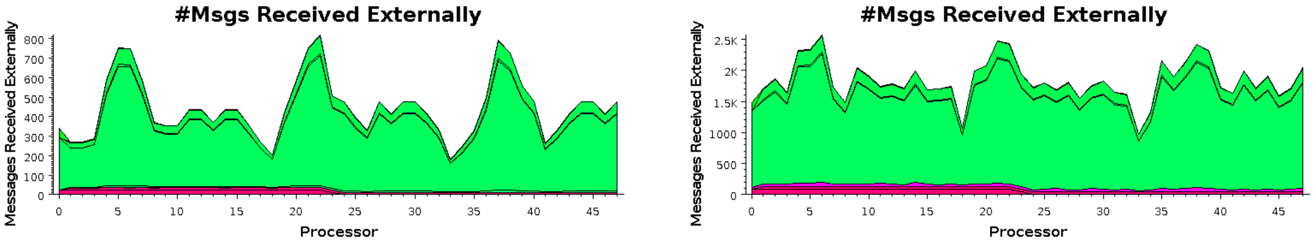


Fig. 4. Comparison of different load balancers for the *stencil4D* benchmark.

NucoLB migrates an average of 47 objects per load balancing call on NUMA32, which translates to 100 ms spent migrating objects. NucoLB is able to avoid unnecessary migrations because it considers the original mapping of the application and knows the machine topology. Exploiting the machine topology, it minimizes migrations that can worsen performance by increasing communication costs. On the other hand, other load balancers migrate 5.3 times more objects. This results in an overhead 11 times greater than that of NucoLB. This is related to the cost of migrating *stencil4D*'s large objects. Table V summarizes the load balancing costs of *stencil4D*. The large number of migrations leads to an increase in the memory footprint of the benchmark, which ends up affecting the performance of some objects. For instance, other load balancers increase last level cache misses by 32% and page faults by 19% on NUMA48 when compared to the baseline, while NucoLB shows an average reduction of 1% on both parameters. Meanwhile, NucoLB achieves a speedup of 1.58 over the baseline by reducing the average iteration time from 1.84 s to 0.78 s.

NucoLB also balances the communication between cores, as can be seen in Figure 5. Before load balancing, core 22 receives the largest amount of messages, 4.5 times more than the smallest number of messages received by any core. This difference is reduced to 2.6 times after load balancing with NucoLB.

The performance of the *LeanMD* application can be seen in Figure 6. This application is comprised of many small objects. Each occupies a small amount of memory and communicates



(a) Messages received per core before load balancing (first 10 iterations).

(b) Messages received per core after load balancing with NucoLB (remaining 40 iterations).

Fig. 5. Distribution of messages received from other cores for *stencil4D* on NUMA48, as captured by Projections. The horizontal axis identifies each core. The vertical axis represents the number of messages received from other cores. These figures show how NucoLB balances communications, as the difference in the most and the least number of messages received by any core is reduced.

TABLE V
Load balancing times in seconds for the stencil4D benchmark for different NUMA machines.

Load balancers	**NUMA32**	**NUMA48**
Baseline	—	—
NucoLB	0.11	0.27
GreedyCommLB	1.17	0.84
ScotchLB	1.13	1.02
TreeMatchLB	1.14	1.04

through small messages of approximately 100 bytes. This application starts in a balanced state. For instance, its original object mapping presents an efficiency of 95% on NUMA32, meaning an idleness of only 5%. This helps to explain why no load balancer was able to significantly improve performance on this machine. This is related to the large amount of objects per core in this experiment. We have an average of 58 objects per core, and the difference between the cores with the greatest and the smallest number of objects is only 3. Although this brings an initial balanced state to the application, this incurs a higher processor overhead, as many objects have to be scheduled on each core. We have a different situation on NUMA48, with its larger number of cores. There, NucoLB is able to reduce the iteration time by 12%, resulting in a speedup of 1.12 over the baseline total execution time. Other load balancers show similar performances as, in this case, migration costs are negligible and communication plays a minor part in the overall performance. However, this does not hold true in a cluster, as we will see in the next section.

B. Performance Improvements on a NUMA Cluster

In this section, we focus on the performance of *LeanMD* on 20xNUMA4, a cluster composed of 20 computes nodes and a total of 80 cores. The total execution times for *LeanMD* running on 80 cores can be seen in Figure 6 with the label 20xNUMA4. The number of objects and the size of their messages is the same as mentioned before. As the number of available cores increases, the number of objects per core decreases. In the case of 20xNUMA4, we have an average of 23 objects per core.

NucoLB was able to obtain speedups of 1.19 over the baseline and 1.18 over GreedyCommLB, the load balancer with the second best performance. NucoLB's hierarchical algorithm is able to balance the load over the available cores while reducing the communication costs. The iteration time reduces by 21% — from 115 ms to 91 ms, as can be seen in Table VI. This table presents the average iteration time and load balancing time for the different load balancers. The load balancing time includes the time spent executing the load balancing algorithm and migrating the objects.

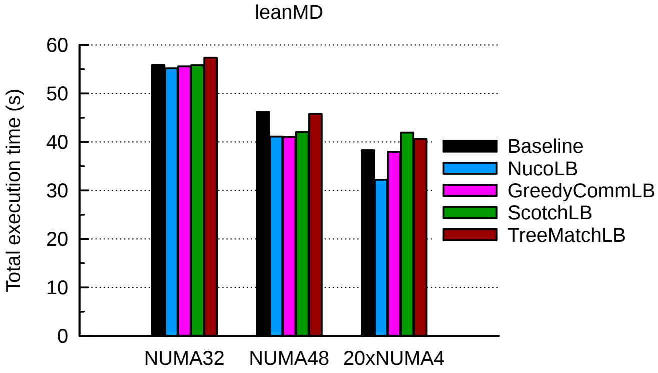


Fig. 6. Comparison of different load balancers for the *LeanMD* benchmark.

TABLE VI
Average iteration time and load balancing time (ms) for *LeanMD* with different load balancers on 20xNUMA4.

Load balancers	**Average iteration time**	**Load balancing time**
Baseline	114.94	0.00
NucoLB	90.64	104.45
GreedyCommLB	111.94	96.05
ScotchLB	123.76	144.08
TreeMatchLB	116.50	355.64

The load balancing decisions of NucoLB are able to achieve an efficiency of 93% in this configuration, implying an idleness of 7%. The algorithm chooses to keep up to 30% more objects than the average on some cores. This leads to a better performance than spreading these objects, as spreading incurs increased communication among NUMA nodes and compute nodes. However, this is only helpful when considering the whole machine hierarchy. For instance, other load balancing algorithms that do not consider the communication costs (as measured by NUCOLB) end up increasing the processor overhead by 50%. This overhead is related to the time spent by the CHARM++ RTS in managing network communications.

NUCOLB migrates approximately 300 objects when it is first called by the application. These migrations quickly converge to a more balanced state. Subsequent calls result in the migration of 100 objects. On this machine, the application's performance does not improve much after the second or third load balancing call. Nevertheless, NUCOLB's total load balancing time is equivalent to one iteration of *LeanMD*, as can be seen in Table VI. With a load balancing call after every 60 iterations, NUCOLB's overhead is easily compensated by the performance improvements that it brings.

VII. RELATED WORK

The complexity of current parallel machines and applications demands efficient techniques to place tasks on processors. Significant research has been done in developing schedulers and load balancers that improve the overall system performance [1], [4], [7], [8], [18], [19].

Bhatele et al. [18] study the impact of topology-aware load balancing algorithms on a molecular dynamics application running on large parallel machines. The study focuses on static and dynamic topology-aware mappings on 3D mesh and torus architectures. Results show that these techniques can improve the performance of NAMD by up to 10%. The metric used to evaluate the load balancing algorithms is *hop-bytes*, which is based on the total number of bytes exchanged between the processors weighted by the distance between them. However, the techniques used in this paper do not consider the NUMA and multi-core design of large parallel platforms.

Rodrigues et al. [19] discuss a strategy to reduce load imbalance in weather forecast models. They try to preserve the spatial proximity between neighbor tasks (and, by consequence, reduce communication overheads) by traversing them with a Hilbert curve and recursively bisecting it according to the load of the threads. Using this strategy, they obtain a small performance improvement over METISLB. However, this strategy can only be mapped to applications with regular communication patterns such as structured grids.

Tchiboukdjian et al. [7] propose an adaptive work-stealing algorithm for applications based on parallel loops on XKAAPI. The objective of their algorithm is to ensure that multi-core machines sharing the same cache, work on data that are close in memory. This is done to reduce the total number of cache misses. The proposed work stealing algorithm presents performance improvements of up to 30%, although its utilization is restricted to applications based on parallel loops.

On NUMA platforms, Mercier and Clet-Ortega [1] try to improve the placement of MPI processes by combining hardware's hierarchy information from the PM^2 runtime system, application's communication information from traces and the SCOTCH library [16]. Similarly, Jeannot and Mercier [5] presented a hierarchical algorithm that uses information about the NUMA machine gathered by HWLOC [11]. Chen et al. [20] also try to reduce communication costs of MPI applications on SMP clusters and multi-clusters. Their proposal maps the communication graph of the application to the topology graph of the machine. The communication is profiled during a previous execution, and the machine topology is obtained by running a parallel ping-pong benchmark. This approach focuses only on improving communication latencies among processes, ignoring application load imbalance.

Hofmeyr et al. [8] propose a pro-active load balancer for multi-core machines, named JUGGLE. It works as a user-level scheduler on Linux and tries to reduce the execution time of SPMD applications written using UPC or OPENMP. Their approach considers some of the memory costs of NUMA machines by forbidding migrations between NUMA nodes. They show performance improvements of up to 80% over static scheduling. However, JUGGLE works only on imbalances coming from a number of cores m that is a not a factor of the number of threads n. In other words, this algorithm does not focus on load balancing problems coming from the application's dynamism nor communication behavior.

Majo and Gross [4] have developed scheduling algorithms for NUMA architectures that try to improve data locality and cache contention. They obtain performance improvements of up to 32% over the default Linux scheduler. Their algorithms consider the NUMA penalty of the application, which is a factor between the CPI (cycles per instruction) of a program executing on the same NUMA node of its data and its CPI executing on a different NUMA node. This approach requires previous executions of the application with different data and process mappings. Also, they do not consider parallel applications and their communication schemes in their work.

VIII. CONCLUSION AND FUTURE WORK

The complexity of the memory subsystem of multi-core compute nodes with NUMA design introduces new challenges to the problem of load balancing. In order to efficiently utilize a parallel machine, a load balancing algorithm must consider not only the computational load of the application, but also the existing asymmetries in memory latencies and bandwidth, and network communication costs.

We designed NUCOLB, a topology-aware load balancer that combines application statistics and information about the NUMA compute nodes topology. The memory access costs and network communication costs were synthesized as the NUMA and NETWORK factors, respectively. The combination of these two factors results in our view of the machine topology as the NUCO FACTOR. This approach models the machine topology in a generic fashion, as it works on both NUMA compute nodes and clusters of NUMA machines. The chosen approach does not make any assumptions about the application nor requires prior executions.

Our experimental results showed that the proposed load balancer enhances the performance of CHARM++ applications. We obtained speedups of up to 1.19 over state-of-the-art load balancers on different NUMA platforms. In addition, NUCOLB obtained this performance while migrating a maximum of 30% of the tasks on average, which results in a migration overhead up to 11 times smaller than other load balancers. NUCOLB fulfilled its objectives, as it is able to reduce idleness down to 4%, and to reduce the communication costs experienced by the application (a 6% reduction in the average load of each core for *kNeighbor*). These results were obtained by distributing the load over the cores while maintaining proximity of the communicating chares with regard to the machine topology.

Future work includes the extension of the load balancing algorithm to include the cache hierarchy in its decisions. This requires the measurement of the different communication latencies among cores. As a baseline, we plan to use the representation of the cache hierarchy provided by HWLOC [11]. We also plan to use the machine representation of HWLOC to store communication latencies. By gathering and organizing this information, we can also provide it to other libraries and algorithms to improve the quality of their scheduling decisions.

ACKNOWLEDGMENT

This research has been partially supported by CAPES-BRAZIL under grants 5854/11-3 and 4874/06-4. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-536171). Some of the experiments presented in this paper were carried out on the Grid'5000 experimental test bed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This research was accomplished in the context of the International Joint Laboratory LICIA. This work was also supported by INRIA-Illinois Joint Laboratory for Petascale Computing and the ANR RESCUE and G8 ECS (Toward Exascale Climate Simulations) projects. We thank Harshitha Menon of the Parallel Programming Laboratory (Illinois) for helping with CHARM++ benchmarks and applications.

REFERENCES

[1] G. Mercier and J. Clet-Ortega, "Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2009, vol. 5759, pp. 104–115.

[2] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009)*, 2009, pp. 59–66. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2009.16

[3] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P. A. Wacrenier, and R. Namyst, "Structuring the execution of OpenMP applications for multicore architectures," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010)*. IEEE Computer Society, 2010, pp. 1–10.

[4] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead," in *Proceedings of the International Symposium on Memory Management*, ser. ISMM '11. New York, NY, USA: ACM, 2011, pp. 11–20.

[5] E. Jeannot and G. Mercier, "Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Springer Berlin / Heidelberg, 2010, vol. 6272, pp. 199–210.

[6] C. Mei, G. Zheng, F. Gioachin, and L. V. Kale, "Optimizing a parallel runtime system for multicore clusters: a case study," in *Proceedings of the 2010 TeraGrid Conference (TG 2010)*. New York, NY, USA: ACM, 2010.

[7] M. Tchiboukdjian, V. Danjean, T. Gautier, F. Le Mentec, and B. Raffin, "A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores," in *4th Workshop on Highly Parallel Processing on a Chip (HPPC)*, aug 2010.

[8] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz, "Juggle: proactive load balancing on multicore computers," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 3–14.

[9] J. Y.-T. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*, ser. Chapman & Hall/CRC computer and information science series. Chapman & Hall/CRC, 2004.

[10] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*. ACM, 1993, pp. 91–108.

[11] 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," *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2010)*, vol. 0, pp. 180–186, 2010.

[12] LMbench, "LMbench benchmark," 2010. [Online]. Available: http://www.gelato.unsw.edu.au/IA64wiki/lmbench3

[13] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, and E. Al, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[14] J. D. Mccalpin, "STREAM: Sustainable memory bandwidth in high performance computers," University of Virginia, Tech. Rep., 1995. [Online]. Available: http://www.cs.virginia.edu/stream/

[15] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008)*, April 2008, pp. 1–12.

[16] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *International Conference on High-Performance Computing and Networking (HPCN 1996)*. Springer, 1996, pp. 493–498.

[17] L. Kalé and A. Sinha, "Projections: A preliminary performance tool for charm," in *Parallel Systems Fair, International Parallel Processing Symposium*, Newport Beach, CA, April 1993, pp. 108–114.

[18] A. Bhatele, L. V. Kale, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *Proceedings of the 23rd international Conference on Supercomputing (ICS 2009)*. New York, NY, USA: ACM, 2009, pp. 110–116.

[19] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kale, "A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model," *Computer Architecture and High Performance Computing, Symposium on*, vol. 0, pp. 71–78, 2010.

[20] 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 *Proceedings of the 20th annual international conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 353–360.