Improving Parallel System Performance with a NUMA-aware Load Balancer

Laércio L. Pilla[1,2], Christiane Pousa Ribeiro[2], Daniel Cordeiro[2], Abhinav Bhatele[3],
Philippe O. A. Navaux[1], 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 – INRIA – Grenoble University – Grenoble, France
{christiane.pousa, daniel.cordeiro, jean-francois.mehaut}@imag.fr

[3]Department of Computer Science – University of Illinois at Urbana-Champaign – Urbana, IL, USA
{bhatele, kale}@illinois.edu

***Abstract*—Multi-core nodes with Non-Uniform Memory Access (NUMA) are now a common architecture for high performance computing. On such NUMA nodes, the shared memory is physically distributed into memory banks connected by a network. Owing to this, memory access costs may vary depending on the distance between the processing unit and the memory bank. Therefore, a key element in improving the performance on these machines is dealing with memory affinity. We propose a NUMA-aware load balancer that combines the information about the NUMA topology with the statistics captured by the Charm++ runtime system. We present speedups of up to 1.8 for synthetic benchmarks running on different NUMA platforms. We also show improvements over existing load balancing strategies both in benchmark performance and in the time for load balancing. In addition, by avoiding unnecessary migrations, our algorithm incurs up to seven times smaller overheads in migration, than the other strategies.**

***Keywords*-load balancing, non-uniform memory access, memory contention, performance, object migration**

I. Introduction

The importance of Non-Uniform Memory Access (NUMA) architectures has been increasing as a scalable solution to alleviate the memory wall problem and to provide better scalability for multi-core machines. Clusters based on AMD Opteron processors and Intel Nehalem ones are examples of multi-core machines with NUMA design. A NUMA platform is a multi-processor system where the processing elements share a single global memory that is physically distributed into several memory banks. These memory banks are interconnected by a specialized network. Due to this interconnection, memory access costs may vary depending on the distance (latency) between processing elements and memory banks, and based on the number of processing elements accessing the same memory bank (bandwidth). Since these platforms are becoming ubiquitous in high performance computing (HPC), it is important to reduce the access latency and to increase the available bandwidth for data access on them. Therefore, enhancing the memory affinity becomes a key element to improve performance on these machines.

Memory affinity is enhanced when the thread and data placement is done in a such way that the access latency and memory contention perceived by threads to get data is reduced [1]. This improvement may happen through different approaches, such as the use of efficient memory allocation mechanisms or by balancing the load appropriately among the different processing elements. The first approach focuses on distributing data and bringing it closer to its users, so as to reduce latency and memory contention. The second approach deals with doing a better distribution of the work among processing elements in order to avoid hot spots and improve communication among threads. The implementation of these approaches is usually linked to the characteristics of the target parallel programming environment.

Several popular options for programming multi-core and NUMA architectures are available, and their performance can be improved in different ways. In OpenMP, an interface may be employed in the standard to allow memory affinity control [1], or a hardware-aware runtime system can be used to control thread scheduling [2]. When using MPI on shared memory, the efficiency problem is usually addressed by improving the process mapping [3], [4]. Another environment that may benefit from the improvement of memory affinity is Charm++ [5].

Charm++ is a C++-based parallel programming model and runtime system (RTS) designed to enhance programmer productivity by providing a high-level abstraction of the parallel computation while delivering good performance. Charm++ programs are decomposed into communicating objects called *chares*, which exchange data through remote method invocations. One of the main advantages of Charm++ is that the RTS captures statistics for the chares during the execution [6], which can be used to improve the load balance [7] and to enhance memory affinity on multi-core machines with NUMA

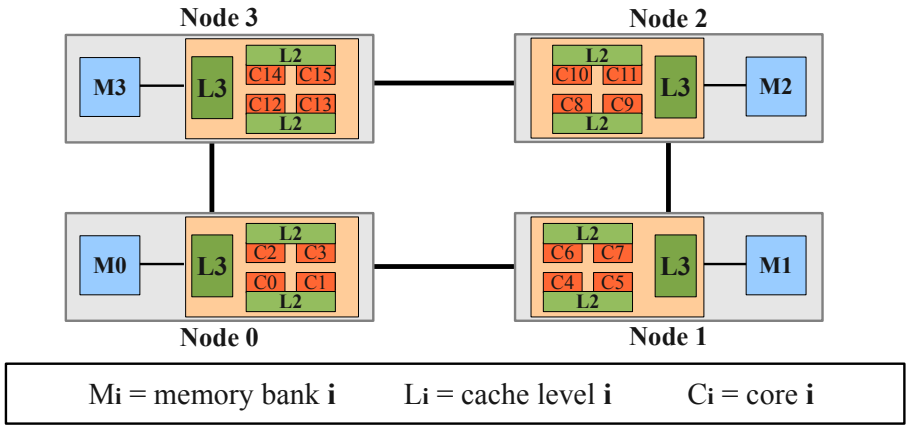

Fig. 1. Schema of a multi-core NUMA machine with 16 cores and 4 NUMA nodes.

design. However, it still lacks information about memory access costs, which represents an important aspect of the NUMA platform.

Using CHARM++ as a test bed, we try to address the following key questions: 1) How can we obtain information about the NUMA architecture? 2) How can we use this information to improve the parallel system performance? 3) How does this improvement compare to other strategies on different multi-core machines with different workloads?

In this context, this paper presents a NUMA-aware load balancer — named **NUMALB**, which combines the information about the machine topology with the statistics captured by the CHARM++ RTS. It aims to improve the load balance while avoiding unnecessary migrations and reducing across-core communication.

The rest of this paper is organized as follows: in Section II we briefly describe multi-core platforms with NUMA design and the CHARM++ runtime system. Section III introduces and describes the proposed load balancer for the CHARM++ runtime. In Section IV, we present the platforms and benchmarks used in our experiments. We evaluate the performance of the proposed load balancer in Section V. In Section VI, we discuss some related work and present concluding remarks and future work in Section VII.

II. BACKGROUND

In this section, we describe the main characteristics of multi-core platforms with NUMA design. Furthermore, we also present the CHARM++ parallel system and some of its design characteristics.

A. Multi-core Platforms with NUMA Design

Multi-core platforms are a growing trend in computer science, especially in HPC. A multi-core machine consists of multiple cores grouped into sockets that share different levels of cache hierarchies and the main memory. This aims to alleviate some important issues such as the instruction level parallelism within a chip and the power wall problem [8]. The multi-core design allows computer architecture engineers to build powerful shared memory machines with tens or even hundreds of cores. However, the increasing number of cores demands an efficient memory hierarchy solution, since several cores might use the same network interconnect to access the shared memory generating the memory wall problem [9].

In order to support this high number of cores and to reduce the memory wall problem, multi-core platforms with Non-Uniform Memory Access design are being deployed. In these multi-core NUMA machines, several cores access the same global shared memory. Furthermore, their shared memory is physically distributed into several memory banks which are interconnected by a network. The memory wall problem is thus reduced, since cores can use different paths and memory banks to access data. However, this design generates an asymmetry on access latency to get the data [1], [10], leading to the concept of local and remote accesses. A local access is performed when a core accesses a memory bank that resides in its node. Contrary to this, a remote access occurs when a core requests data that is allocated on some other node.

Figure 1 shows the schema of a multi-core NUMA machine with sixteen cores and four NUMA nodes. The global shared memory is distributed over the machine in four memory banks. In this architecture, four cores each have their local memory bank (local access) and other memory banks are accessed using the interconnection network (remote access). Additionally, this machine has multiple levels of shared cache to reduce latency costs. In this case, each pair of cores share a L2 cache and four cores each share a L3 cache.

In multi-core machines with NUMA design, it is particularly important to ensure an efficient usage of memory banks to reduce NUMA costs in the application. In order to do so, mechanisms such as thread scheduling, memory allocation and load balancing can be used, depending on the application and runtime system characteristics [1], [2], [10].

B. CHARM++ Parallel System

CHARM++ is a parallel runtime system that provides an object oriented parallel programming language with a 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 its objects [5], [11].

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 takes care of all messages generated from these interactions. Chares communicate through remote method invocation (a message-driven model). Further, the CHARM++ RTS is responsible for physical resource management on the target machine.

In the current version of CHARM++, all communication on shared memory machines is done in memory. In the shared memory (SMP) build of CHARM++, communication proceeds through the exchange of pointers between CHARM++ threads. Due to this, the CHARM++ runtime is able to avoid high overheads due to messages and reduce communication time. However, in the case of NUMA machines, this mechanism can be affected by asymmetric memory latencies and bandwidth. CHARM++ relies on the operating system memory affinity and does not explicitly control the placement of shared data in the memory.

Particularly on some operating systems such as Linux and Windows, the default policy to manage memory affinity on NUMA machines is *first-touch*. This policy places data on the NUMA node that first accesses it [12]. In the case of CHARM++ communication mechanism, once the data (e.g. a message) is touched, this memory policy will not perform any data migration to enhance memory affinity. This might result in sub-optimal data placement in CHARM++ applications running on NUMA platforms. For instance, we can imagine a situation where some messages have been generated and originally allocated on core 0 of NUMA node 0. After that, these messages are sent to core 1 of NUMA node 1 and after several hops they end up on core N of NUMA node N. All message sends are pointer exchanges of data that were originally allocated and touched in the memory of core 0. In such a scenario, several remote accesses will be generated for every communication.

Owing to the design of the CHARM++ communication mechanism and the ubiquity of multi-core platforms with NUMA design, it is important to provide NUMA support in the CHARM++ parallel system to manage the machine resources efficiently and reduce the memory access costs to get data.

III. NUMA-aware Load Balancer

The new generation of NUMA multi-core platforms, combined with the availability of easy-to-use parallel runtime systems like CHARM++, are enabling the development of very large parallel programs composed of several tasks. In order to ensure good performance, it is crucial to fully utilize the platform, ensuring that no processor will be underutilized due to imbalance of the tasks being executed.

The problem of load balancing is known to be NP-complete [13]. In NUMA machines, the problem becomes more challenging due to its memory hierarchy. In these systems, an action taken by the load balancer to equalize the load of the available processors may actually decrease the overall performance because of the latency in remote memory accesses.

In order to cope with the complexity introduced by NUMA machines, we have developed a new heuristic that considers the specifics of NUMA multi-core machines to perform load balancing. In this section, we describe the information that can be obtained from the underlying system which can be used by a NUMA-aware load balancer. In light of this information, we present a new heuristic and its implementation using the CHARM++ runtime system.

A. Obtaining Runtime Information

A NUMA-aware load balancer can benefit from two different classes of information obtained dynamically (at runtime) from the underlying system: *application data* and *NUMA topology*.

Application data comprises all information about the parallel application that can be probed at runtime: task execution times, communication information, and the assignment chosen by the scheduler at a given time. In CHARM++ RTS, this information can be dynamically obtained during the execution of the application.

CHARM++ provides a mature load balancing framework to balance computational and communication load on the processors [7]. Load balancing in CHARM++ is measurement-based and depends on instrumented data from previous time steps to balance load for future time steps. The RTS provides information about the total work assigned to each processing element (load) and execution time of each chare. The execution time of each chare includes its computational and communication load. The load on each processing element (core) is the sum of loads of all its chares and other runtime overheads.

The CHARM++ RTS also provides detailed information about the communication graph of the application. It is possible to obtain details about the number of messages and the amount of bytes exchanged among chares. A NUMA-aware load balancer can take advantage of this information to reduce communication overhead by bringing communicating chares closer to each other.

The *NUMA topology* comprises all information that can be gathered at runtime about the machine hardware that is executing the application. A NUMA machine can be characterized in terms of the number of NUMA nodes, cache memory sizes, sharing of cache hierarchies among cores and grouping of NUMA nodes.

Using this information, a NUMA-aware load balancer can create a model that represents the machine topology and use it to infer its memory access penalties. Since there is no tool that can automatically discover the physical topology among NUMA nodes, we define a **NUMA factor** to synthesize both the topology and the memory penalties. The NUMA factor

represents the overhead to access remote data and is defined as:

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

where i and j represent different NUMA nodes. This factor is computed for all NUMA nodes of the target machine, resulting in a square matrix of NUMA factors. Thus, the main advantages of using the NUMA factor as a topology indicator is that it is generic (can be easily computed for different NUMA machines) and aggregates the differentiating features of NUMA machines. In addition, the NUMA factor can be precomputed, which reduces the overhead of using it.

B. Load Balancing Heuristic

It is not possible to compute an assignment of tasks on to available processors that optimally equalizes the load in polynomial time (unless P = NP). Moreover, in the general case, a CHARM++ load balancer cannot make any assumptions about the application that will be executed, so it is also impossible to use precomputed assignments instead of online scheduling. Thus, in practice, in order to compute a good (approximated) assignment in a reasonable amount of time, a heuristic must be employed.

We have developed a load balancing heuristic that uses *application data* and *NUMA topology* information to reduce the load imbalance of parallel applications. The heuristic works like a classical List Scheduling algorithm [13], where tasks (chares) are rescheduled from a priority list and assigned to less loaded processors in a greedy manner. List schedule algorithms usually are fast to compute and provide good results in practice.

The main idea of the heuristic is to improve application performance by mapping chares to cores while reducing the costs of unbalanced computation and remote communications. The heuristic is based on the following cost function for mapping of a chare c on to core p:

$$\begin{aligned} cost(c,p) = \; & load(p) + \\ & \alpha \times (r_{comm}(c,p) \\ & \quad \times NUMA\ factor(comm(c), node(p)) \\ & \quad - l_{comm}(c,p)) \end{aligned}$$

In the equation, $load(p)$ represents the total load of core p, l_{comm} represents the number of messages sent from chare c to chares on cores of the same NUMA node (with the same local memory bank) as core p, and r_{comm} expresses the number of messages sent from chare c to chares on other NUMA nodes and is multiplied by the NUMA factor between the NUMA node of core p ($node(p)$) and the NUMA nodes where these communicating chares are mapped ($comm(c)$). Finally, α controls the weight that the communication costs have over the execution time. The heuristic uses the number of exchanged messages because it represents the amount of accesses to the shared memory. Since messaging time is related to the access latency, the cost is multiplied by the NUMA factor when considering remote accesses. In addition, local communications are subtracted from the overall cost to favor their occurrence.

C. NUMALB's Algorithm

By combining the information described in III-A and the heuristic presented in III-B, we have implemented a new load balancer for CHARM++, named NUMALB, which is better adapted for NUMA environments. It is a List Scheduling, greedy algorithm, that picks the heaviest (largest execution time) unassigned chare and assigns it to the core that presents the smaller cost. The choice for 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 NUMALB is presented in Algorithm 1.

Algorithm 1: NUMALB.

Input: C set of chares, P set of cores, M mapping of chares to cores
Output: M' mapping of chares to cores

1 $M' \leftarrow M$
2 **while** $C \neq \emptyset$ **do**
3 $\quad c \leftarrow v \mid v \in \arg\max_{u \in C} load(u)$
4 $\quad C \leftarrow C \setminus \{c\}$
5 $\quad p \leftarrow q, q \in P \wedge \{c,q\} \in M$
6 $\quad load(p) \leftarrow load(p) - load(c)$
7 $\quad M' \leftarrow M' \setminus \{(c,p)\}$
8 $\quad p' \leftarrow q \mid q \in \arg\min_{r \in P} cost(c,r)$
9 $\quad load(p') \leftarrow load(p') + load(c)$
10 $\quad M' \leftarrow M' \cup \{(c,p')\}$

Considering n chares and m cores, this algorithm presents a complexity of $\mathcal{O}(n^2 m)$ in the worst-case scenario — when chares present all-to-all communications. However, since this kind of behavior is usually avoided in CHARM++ applications, NUMALB shows a complexity of $\mathcal{O}(nm)$ for a constant vertex degree of the communication graph.

Another important fact about this algorithm is that it avoids unnecessary migrations. Since there is no information available about the size of the chares, there is no way to estimate the overhead brought by migrations. They are avoided by considering the initial scheduling and by subtracting the chare's load from its current core, as depicted on line 6 of Algorithm 1.

D. Implementation Details

NUMALB was implemented on top of the load balancing framework in CHARM++. This framework provides all necessary information about the application and only requires the new mappings of the chares from the load balancing strategy, to execute the migrations. It also enables the allocation of dynamic structures and gathering of information during CHARM++'s startup.

To extract the node hierarchy (which cores dwell on which node) and the machine's NUMA penalties, we implemented a library that retrieves the machine characteristics. The node hierarchy is retrieved from the operating system kernel. After

that, information such as number of nodes, number of cores and the mapping between cores and NUMA nodes is stored in files for later use. For the NUMA penalties, we use the LMbench benchmark [14] to get the access latency and to compute the NUMA factor. LMbench is a set of synthetic benchmarks that measures scalability of multi-processor platforms and the characteristics of the processor micro-architecture. During the installation of CHARM++, our library runs a script that executes LMbench between each pair of nodes. These latencies between nodes are saved in temporary files. We then use these latencies to compute the NUMA factor for each pair of nodes. The NUMA factor is also stored in files for later use by our library. During the initialization of NUMALB, these files are loaded into dynamic structures that are then used by its algorithm.

IV. EXPERIMENTAL SETUP

In this section we present our experimental setup to evaluate the NUMA-aware load balancer. We have selected two representative multi-core platforms with NUMA characteristics:

- **NUMA16**: based on eight dual-core AMD Opteron 875 processors. The cores have private L1 (64 KB) and L2 (1 MB) caches and no caches are shared.
- **NUMA32**: four eight-core Intel Xeon X7560 processors. Each core has a private L1 (32 KB) and L2 (256 KB) caches and all cores on the same socket share a L3 cache (24 MB).

Both machines run Linux (kernel 2.6.32) with GNU Compiler Collection.

Table I summarizes the hardware characteristics of these machines. Memory bandwidth (obtained from Stream - Triad operation [15]) and NUMA factor are also reported in this table. NUMA factors are shown in intervals, meaning the minimum and maximum penalties to access remote memory in comparison to local memory.

TABLE I
OVERVIEW OF THE NUMA MULTI-CORE PLATFORMS.

Characteristic	NUMA16	NUMA32
Number of cores	16	32
Number of sockets	8	4
NUMA nodes	8	4
Clock (GHz)	2.22	2.27
Highest level cache (MB)	1 (L2)	24 (L3)
DRAM capacity (GB)	32	64
Memory bandwidth (GB/s)	9.77	35.54
NUMA factor (Min;Max)	[1.1; 1.5]	[1.36; 3.6]

We used the CHARM++ release 6.2.1 with the optimized multi-core build [16]. To evaluate the load balancer, we selected three benchmarks from CHARM++ programs: (i) *kNeighbor*, a synthetic iterative benchmark where a chare communicates with k other chares at each step; (ii) *lb_test*, a synthetic unbalanced benchmark that can choose from different communication patterns; and (iii) *jacobi2D*, an unbalanced two-dimensional five-point stencil computation.

For comparison, the performance of other load balancers was also evaluated. They are: GREEDYLB, RECBIPARTLB, METISLB and SCOTCHLB. These four load balancers do not consider the original mapping of the chares and they are oblivious to the machine topology.

GREEDYLB reassigns the chares in a greedy fashion. The algorithm iteratively maps the heaviest chare to the least loaded core. Hence, it does not consider the communications among chares. Despite that, this strategy performs well due to its simplicity and speed. RECBIPARTLB does a recursive bipartition of the communication graph based on their loads. This is done by a breadth-first traversal until the required load (execution time) is gathered in one group.

METISLB is based on the graph partition algorithms implemented in METIS [17]. This strategy considers both the execution time and communication graph to improve the load balance. Similarly, SCOTCHLB follows the same principles, but it is based based on the algorithms in SCOTCH [18].

The results shown in the next section are the averages obtained over a minimum of 25 executions. They present a statistical confidence of 95% by Student's t-distribution and a 5% relative error.

V. RESULTS

The performance improvements obtained by rebalancing load in CHARM++ programs may depend 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. In this section, in order to exemplify the impact of load balancing, we first show the performance improvements obtained by the load balancers. Afterwards, we provide details about the overheads induced by the execution of the load balancer and migrations of chares.

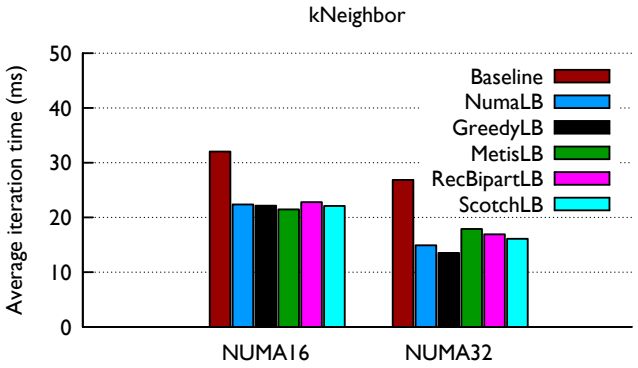

Fig. 2. Average iteration time for load balancers with the *kNeighbor* benchmark.

A. Performance Improvements

The results presented in this section represent the average iteration time before (as *Baseline*) and after applying a load balancing algorithm. Fig. 2 shows the performance obtained for the *kNeighbor* benchmark on both NUMA platforms

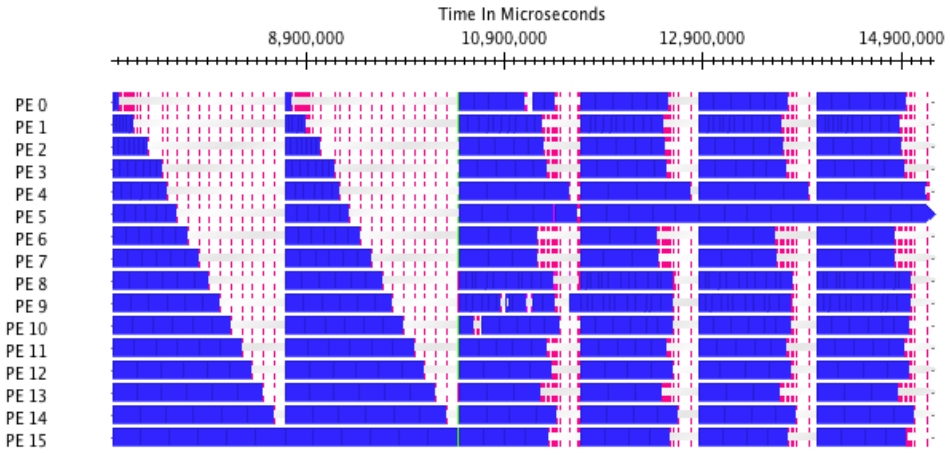

Fig. 3. Timeline view of *jacobi2D* using Projections — two time steps before and four after load balancing (using METISLB) are shown.

using 200 chares, for number of neighbors k equal to 8 and messages of 16 KB. On NUMA16, all load balancers present a speedup of 1.45 over the baseline time. On the other hand, on NUMA32, the best performance is obtained using the GREEDYLB, which reduces the iteration time to 50% and presents a speedup of 1.1 over NUMALB. While GREEDYLB distributes the load (that also considers the communication time but not the communication graph) more homogeneously over the cores, METISLB, SCOTCHLB and RECBIPARTLB tend to group chares and migrate them together to cores, and NUMALB tends to only migrate the heavier chares, which happens usually to nearby (in the same NUMA node) cores. The greater differences in performance on NUMA32 happen because this machine has a larger number of cores to distribute the chare's communication overhead. Furthermore, it has cores that share a cache, which results in faster communication among cores in the same NUMA node (which is exploited by NUMALB).

It is important to emphasize that this benchmark represents an extreme case, where there is only communication and no computation. In addition, its iteration time is small (tens of milliseconds), which makes it more vulnerable to minor load imbalances.

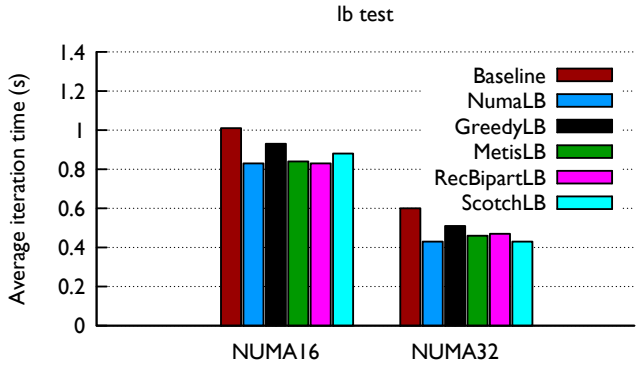

Fig. 5. Average iteration time for load balancers with the *lb_test* benchmark.

Fig. 5 depicts the performance obtained for the *lb_test* benchmark using 200 chares and a random communication graph. Each chare is randomly assigned a load between 50 and 200 ms. The best performance is obtained by the communication-aware load balancers. NUMALB shows speedups of 1.21 and 1.39 when compared to the baseline on NUMA16 and NUMA32, respectively. These results are similar to the ones of METISLB and RECBIPARTLB on NUMA16, and SCOTCHLB on NUMA32. Especially, NUMALB presents the best average performance improvement over these two machines. These results highlight the importance of considering the communication in addition to the execution times when rescheduling.

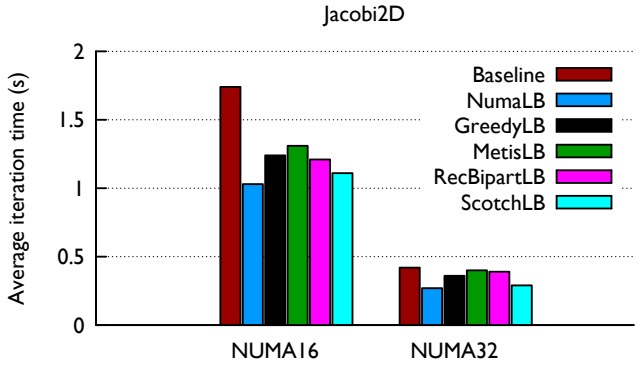

Fig. 6. Average iteration time for load balancers with the *jacobi2D* benchmark.

The best results for NUMALB are obtained with the *jacobi2D* benchmark, as shown in Fig. 6. These iteration times are for 100 chares and a 32^2 data array. NUMALB reduces the iteration time of *jacobi2D* over 40% (speedup of 1.69) on NUMA16 and over 35% (speedup of 1.55) on NUMA32. NUMALB balances the load among cores while keeping part of the original proximity among chares, both in core and NUMA node levels. This happens because NUMALB considers the

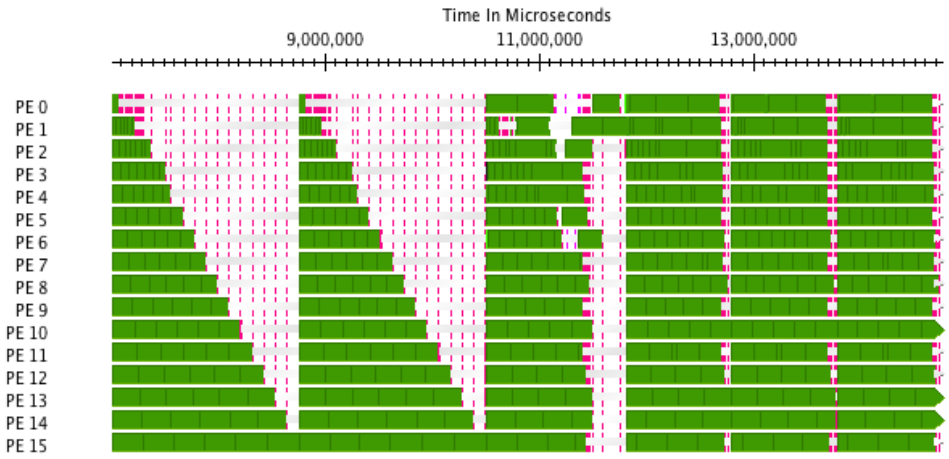

Fig. 4. Timeline view of *jacobi2D* using Projections — two time steps before and four after load balancing (using NUMALB) are shown.

TABLE II
TOTAL EXECUTION TIMES (IN SECONDS).

Benchmark	Machine	Load Balancer					
		Baseline	NUMALB	GREEDYLB	METISLB	RECBIPARTLB	SCOTCHLB
kNeighbor	NUMA16	0.609	0.500	0.510	0.511	0.509	0.512
	NUMA32	0.510	0.364	0.377	0.390	0.392	0.399
lb_test	NUMA16	19.213	17.401	18.441	17.476	17.356	17.899
	NUMA32	11.320	9.754	10.462	9.942	10.039	9.708
jacobi2D	NUMA16	17.323	13.868	14.896	15.189	14.743	14.208
	NUMA32	4.182	3.457	3.906	4.073	4.047	3.560

NUMA topology, while the other load balancers are oblivious to this information.

To get a better idea of the improvements obtained by load balancing, *jacobi2D* execution traces were captured and analyzed using the Projections performance analysis tool [19]. Fig. 3 shows a time line view of the application – the load distribution across the 16 cores for two time steps before and four after load balancing with METISLB on NUMA16. The benchmark presents an extreme case of load imbalance, where the heavier chares share the same core. Since this is an iterative application, the time of each step is defined by the slowest core. As Fig. 3 illustrates, starting from the second step after load balancing, the iteration behavior stabilizes with an efficiency of only 75%. On the other hand, we achieve an efficiency of 93.5% when using NUMALB, as shown in Fig. 4.

For all benchmarks, NUMALB gives the best performance improvements, with an average speedup of 1.51 over the baseline iteration time. This represents a 10% improvement over the other load balancers with the exception of SCOTCHLB. The latter obtains an average speedup of 1.44 over the baseline. Still, the improvements on application execution time depend on the load balancing frequency and number of iterations. For instance, Table II presents the average total execution time for all benchmarks. *jacobi2D* had a total of 10 iterations and one load balancing call after the fifth iteration, while *kNeighbor* and *lb_test* had 19 iterations and one load balancing call after the ninth iteration. For this configuration, an average speedup of 1.22 is obtained over the baseline with NUMALB.

Additionally, these results do not show the complete picture, because they consider the improvements on average iteration time but none of the rescheduling overheads. These overheads are reported in the next section.

B. Load Balancing Overhead

The two main overheads brought by load balancing are the execution time of the load balancing algorithm and the time spent on migration of chares. The average load balancing times for the different machines and benchmarks are presented in Table III. The faster load balancers are GREEDYLB and RECBIPARTLB, which do not use any external libraries. Still, even the slowest load balancer, SCOTCHLB, does not take more than 7 ms. This overhead is easily hidden by the improvements brought by load balancing. In addition to the execution time of the load balancer, NUMALB also incurs an initialization overhead to read the NUMA topology from files, as discussed in Section III-D. However, this process takes at most 3 ms and only has to be done once per execution.

Table IV presents the average number of migrations for each load balancer. *kNeighbor* and *lb_test* were executed with 200 chares, while *jacobi2D* had only 100. All load balancers present more migrations on NUMA32 than NUMA16 because the former has more cores. NUMALB's migration avoidance

TABLE III
AVERAGE LOAD BALANCING TIMES (IN MILLISECONDS).

Benchmark	Machine	Load Balancer				
		NUMALB	GREEDYLB	METISLB	RECBIPARTLB	SCOTCHLB
kNeighbor	NUMA16	3.804	2.648	4.392	1.571	5.930
	NUMA32	3.418	2.468	3.772	2.066	6.387
lb_test	NUMA16	1.876	1.629	2.027	0.981	2.552
	NUMA32	5.507	3.547	4.340	3.242	4.725
jacobi2D	NUMA16	1.029	0.859	1.124	0.722	1.671
	NUMA32	1.177	0.978	1.540	1.061	2.074

TABLE IV
AVERAGE NUMBER OF CHARES MIGRATED PER LOAD BALANCING INVOCATION.

Benchmark	Machine	Load Balancer				
		NUMALB	GREEDYLB	METISLB	RECBIPARTLB	SCOTCHLB
kNeighbor	NUMA16	25	189	188	176	185
	NUMA32	57	194	195	185	194
lb_test	NUMA16	40	188	187	184	184
	NUMA32	48	194	194	192	192
jacobi2D	NUMA16	26	94	94	91	93
	NUMA32	33	97	96	93	98

is clear, as it migrates at most 33% of the chares, while all other load balancers usually migrate 90% or more.

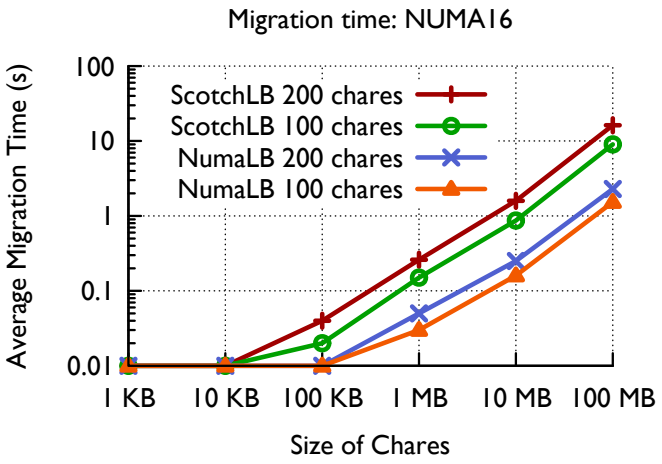

Fig. 7. Average migration time with the *lb_test* benchmark on NUMA16.

We also did several tests which vary the size of the chares with the *lb_test* benchmark to show the effect of the number of migrations on the load balancing overhead. For this, NUMALB and SCOTCHLB were used with 100 and 200 chares. SCOTCHLB was chosen because the number of migrations for it was similar to the other load balancers, but it had better overall performance, as presented in Section V-A.

The results for NUMA16 are shown in Fig. 7. The vertical axis represents the average time for migration in seconds for the different load balancers. The horizontal axis represents the size of the chares. Both axes are on a logarithmic scale.

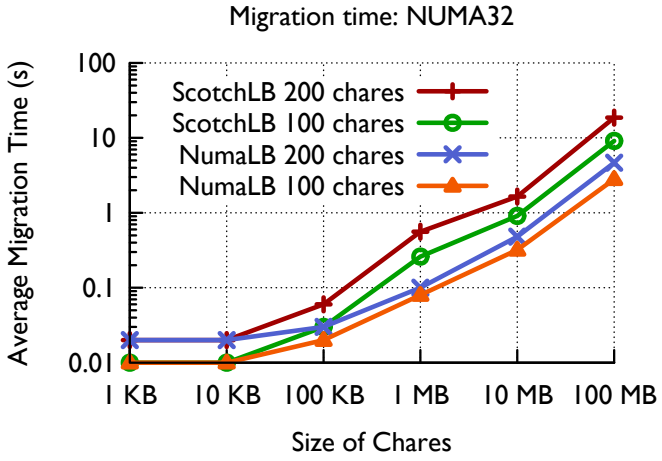

Fig. 8. Average migration time with the *lb_test* benchmark on NUMA32.

As the figure shows, both load balancers present the same migration time for small chares (up to 10 KB). After that, the migration costs for SCOTCHLB increase rapidly, taking double the time than NUMALB for 100 KB chares. As the chares grow in size, the difference of migrating only a few chares becomes even more noticeable. This culminates in a difference of 7 times when using 200 chares with a size of 100 MB, when SCOTCHLB's decisions incur 16.26 seconds in migrating chares, while NUMALB takes only 2.29 seconds. In addition, the migration costs increase by 60% when increasing the number of chares from 100 to 200 for NUMALB, and by 80% for SCOTCHLB.

Similar results for NUMA32 are presented in Fig. 8. When

using chares of 1 MB or more, SCOTCHLB takes more than 3 times to migrate all chosen chares when compared to NUMALB. For the largest size of chares considered, this difference goes to almost 4 times for 200 chares and 3.2 times for 100 chares. These results illustrate the important of avoiding unnecessary migrations to sustain a small load balancing overhead.

VI. RELATED WORK

The complexity of current parallel machines and applications has demanded efficient techniques to place tasks on processors. In this context, significant research has been done proposing schedulers and load balancers that improve the overall system performance [3], [20], [21], [22], [23], [24].

Agarwal, Sharma and Kale [20] propose topology-aware task mapping algorithms for CHARM++. They present load balancing strategies that reduce communication contention. The proposed algorithms exploit information about the application communication graph and the network topology of large parallel machines. They combine this information into a heuristic that reduces the hops-bytes for the application. The *hop-bytes* is a performance metric defined in the paper, which is based on the total number of bytes exchanged between processors weighted by the distance between them. Results show that the algorithms lead to performance improvements when compared to a random placement and a greedy strategy. Although this work considers the machine topology, it focuses on inter-node topology.

Bhatele, Kale and Kumar [22] studied the impact of load balancing algorithms in a molecular dynamics application over large parallel machines. The study focuses on static and dynamic topology-aware mapping techniques on 3D mesh and torus architectures. Results show that these techniques can improve the performance of NAMD [25] up to 10%. Similar to the work presented by Agarwal [20], the performance metric used to evaluate the load balancing algorithms is hop-bytes. However, the techniques do not consider the NUMA and multi-core design of large parallel platforms.

Rodrigues et al. [24] discuss a strategy to reduce load imbalance on 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. With this strategy, they obtained a small performance improvement over METISLB. However, this strategy can only be mapped to applications with regular communication patterns such as structured grids.

Work-stealing [23] is another well-known technique used to distribute computational tasks among a set of processes ("workers"). The main idea of work-stealing is that if a worker becomes idle (i.e., finishes the execution of its own tasks) then it will "steal" tasks from other workers. XKAAPI is a parallel system that relies on such technique to distribute the workload among the processors of the machine [21]. XKAAPI is a C++ library that provides support for asynchronous parallel and interactive programming. This parallel system also supports both shared and distributed memory parallel platforms. Work-stealing in XKAAPI is performed using a data flow representation of the application, which is built at execution time by the XKAAPI RTS. Differently from CHARM++, XKAAPI is well-suited to parallelize recursive algorithms specifically.

Tchiboukdjian et al. [23] propose an adaptive work-stealing algorithm for applications based on parallel loops. 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 made 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, [3] tries 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 [18] to compute the mapping of processes to cores. Similarly, in [4] a hierarchical algorithm is presented that uses information about the NUMA machine gathered by HWLOC [26]. This approach focuses only on improving communication latencies among processes, ignoring application load imbalance.

VII. CONCLUSION

The complexity of the memory subsystem of multi-core with NUMA design introduces new challenges to the problem of load balancing. In this context, an efficient load balancer algorithm must take into account the existing asymmetries in memory latencies and bandwidth.

To deal with load imbalance in this context, we designed NUMALB, a NUMA-aware load balancer that combines application statistics provided by CHARM++ and information about the NUMA machine topology. The machine's topology and memory penalties were synthesized as the NUMA factor. It represents the machine topology in a generic fashion while aggregating the different features 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 an average speedup of 1.51 on the iteration time with NUMALB (with a minimum of 1.22) when compared to not balancing the load at all. This represents a 10% improvement over most of the considered load balancers. In addition, NUMALB obtained this performance while migrating only up to 33% of the chares, which results in a migration overhead up to 7 times smaller than the other load balancers. These results are obtained by distributing the load over the cores while maintaining proximity of the communicating chares with regard to the NUMA topology.

Future work includes the extension of the load balancing algorithm to include the cache hierarchy in its decisions. This would require the measurement of the different communication latencies among cores. As a base, we plan to use the representation of the cache hierarchy provided by HWLOC [26]. By gathering and organizing this information, we can also provide

it to other libraries and algorithms, such as SCOTCH [18], to improve the quality of their scheduling decisions.

REFERENCES

[1] 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

[2] 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.

[3] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_17

[4] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15291-7_20

[5] 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.

[6] R. K. Brunner and L. V. Kale, "Handling application-induced load imbalance using parallel objects," in *Parallel and Distributed Computing for Symbolic and Irregular Applications*. World Scientific Publishing, 2000, pp. 167–181.

[7] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[8] M. Liu, W. Ji, Z. Wang, and X. Pu, "A memory access scheduling method for multi-core processor," *International Workshop on Computer Science and Engineering (WCSE 2009)*, vol. 1, pp. 367–371, 2009.

[9] W. A. Wulf and S. A. Mckee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, vol. 23, pp. 20–24, 1995.

[10] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT 2010)*. New York, NY, USA: ACM, 2010, pp. 319–330. [Online]. Available: http://dx.doi.org/10.1145/1854273.1854314

[11] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming Petascale Applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.

[12] A. Joseph, J. Pete, and R. Alistair, "Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport," in *International Conference on High Performance Computing (HiPC 2006)*, 2006, pp. 338–352.

[13] 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.

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

[15] 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/

[16] 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. [Online]. Available: http://doi.acm.org/10.1145/1838574.1838586

[17] G. Karypis and V. Kumar, "METIS: Unstructured graph partitioning and sparse matrix ordering system," *The University of Minnesota*, vol. 2, 1995.

[18] 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.

[19] S. Biersdorff, A. D. Malony, C. W. Lee, and L. V. Kale, "Integrated Performance Views in Charm++: Projections Meets TAU," in *International Conference on Parallel Processing (ICPP 2009)*, 2009, pp. 140–147. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2009.49

[20] T. Agarwal, A. Sharma, and L. V. Kale, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)*, 2006. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2006.1639379

[21] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, ser. PASCO '07. New York, NY, USA: ACM, 2007, pp. 15–23. [Online]. Available: http://doi.acm.org/10.1145/1278177.1278182

[22] 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)*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 110–116. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542295

[23] M. Tchiboukdjian, V. Danjean, T. Gautier, F. L. Mentec, and B. Raffin, "A work stealing algorithm for parallel loops on shared cache multicores," in *Proceedings of the 4th Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2010, pp. 1–10.

[24] 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," *22th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2010)*, vol. 0, pp. 71–78, 2010. [Online]. Available: http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/SBAC-PAD.2010.18

[25] 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.

[26] 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.