HAL
open science

Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems

Laércio L. Pilla

To cite this version:

Laércio L. Pilla. Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Grenoble; UFRGS, 2014. English. NNT : . tel-00981136

HAL Id: tel-00981136

https://theses.hal.science/tel-00981136

Submitted on 21 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

préparée dans le cadre d'une cotutelle entre l'*Université de Grenoble* et *Universidade Federal do Rio Grande do Sul*

Spécialité : **Mathématiques-informatique**

Arrêté ministériel : le 6 janvier 2005 -7 août 2006

Présentée par

Laércio LIMA PILLA

Thèse dirigée par **Jean-François MÉHAUT** et **Philippe O. A. NAVAUX**

préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique** et **Programa de Pós-Graduação em Computação**

Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems

Thèse soutenue publiquement le **XX avril 2014**,
devant le jury composé de :

M. Raymond NAMYST
Professeur, Université de Bordeaux 1, Rapporteur
M. Wagner MEIRA JUNIOR
Professeur, Universidade Federal de Minas Gerais (UFMG), Rapporteur
M. Alfredo GOLDMAN VEL LEJBMAN
Professeur, Universidade de São Paulo (USP), Examinateur
M. Nicolas MAILLARD
Professeur, Universidade Federal do Rio Grande do Sul (UFRGS), Examinateur
M. Jean-François MÉHAUT
Professeur, Université de Grenoble - CEA, Directeur de thèse
M. Philippe O. A. NAVAUX
Professeur, Universidade Federal do Rio Grande do Sul (UFRGS), Directeur de thèse

Université Joseph Fourier / Université Pierre Mendès France / Université Stendhal / Université de Savoie / Grenoble INP

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LAÉRCIO LIMA PILLA

Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Philippe Olivier Alexandre Navaux
Advisor

Prof. Jean-François Méhaut
Advisor

Porto Alegre, February 2014

CIP – CATALOGING-IN-PUBLICATION

Laércio Lima Pilla,

Topology-Aware Load Balancing for
Performance Portability over
Parallel High Performance Systems /

Laércio Lima Pilla. – Porto Alegre: Programa de Pós-Graduação em Computação da UFRGS, 2014.

115 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Philippe Olivier Alexandre Navaux; Advisor: Jean-François Méhaut.

1. Computer architecture, Parallel programming, Profiling, Scheduling. I. Navaux, Philippe Olivier Alexandre. II. Méhaut, Jean-François. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Alexandre Netto
Pró-Reitor de Coordenação Acadêmica: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 5

LIST OF FIGURES 7

LIST OF TABLES 9

ABSTRACT 11

1 INTRODUCTION 13
1.1 Problem statement 14
1.2 Objectives and thesis contributions 14
1.3 Research context 15
1.4 Document organization 15

2 BACKGROUND 17
2.1 Components of high performance computing platforms 17
2.1.1 Composition 18
2.1.2 Asymmetry and nonuniformity at topology levels 21
2.2 Characterization of scientific applications 22
2.2.1 Tasks: load and communication 22
2.2.2 Application irregularity and dynamicity 23
2.2.3 Examples of scientific applications 25
2.3 Task mapping 26
2.3.1 Load imbalance and costly communications 26
2.3.2 Performance portability 27
2.3.3 Information required for mapping tasks 28
2.4 Machine topology identification 29
2.4.1 Hierarchy perception 29
2.4.2 Communication cost scanning 30
2.5 Discussion 31

3 MODELING THE TOPOLOGIES OF HPC PLATFORMS 33
3.1 Topology representation 33
3.1.1 Topology tree 33
3.1.2 Communication costs 34
3.2 Topology modeling 35
3.2.1 Memory hierarchy of multicore platforms 35
3.2.2 HPC network topology 37
3.3 Topology model evaluation 38

3.3.1 Latency and bandwidth measurements . . . 39
3.3.2 Use of midpoints . . . 41
3.3.3 Nonuniformity measurement at memory level . . . 42
3.4 Machine topology library . . . 45
3.4.1 Tools . . . 45
3.4.2 Data storage . . . 45
3.4.3 Library interface . . . 46

4 PROPOSED LOAD BALANCING ALGORITHMS . . . 47
4.1 Centralized algorithms . . . 47
4.1.1 NUCOLB: Nonuniform communication costs load balancer . . . 48
4.1.2 HWTOPOLB: Hardware topology load balancer . . . 51
4.1.3 Comparison between proposed centralized algorithms . . . 56
4.2 Hierarchical algorithms . . . 56
4.2.1 HIERARCHICALLB: Hierarchical load balancer composition . . . 56
4.3 Implementation details . . . 59
4.4 Conclusion . . . 62

5 RELATED WORK ON TASK MAPPING . . . 63
5.1 Load balancing algorithms . . . 63
5.1.1 Graph partitioning . . . 64
5.1.2 Runtime level load balancers . . . 65
5.1.3 Application-focused load balancers . . . 68
5.2 Work stealing . . . 69
5.3 Process mapping . . . 72
5.4 Discussion . . . 73

6 PERFORMANCE EVALUATION . . . 79
6.1 Evaluation methodology . . . 79
6.1.1 Experimental platforms . . . 79
6.1.2 Benchmarks and applications . . . 80
6.1.3 Load balancers . . . 82
6.1.4 Experimental setup . . . 82
6.2 Load balancing on multicore machines . . . 83
6.2.1 NUCOLB . . . 84
6.2.2 HWTOPOLB . . . 89
6.2.3 NUCOLB and HWTOPOLB . . . 92
6.3 Load balancing on clusters . . . 94
6.3.1 NUCOLB . . . 95
6.3.2 HWTOPOLB . . . 96
6.3.3 HIERARCHICALLB . . . 99
6.4 Conclusion . . . 102

7 CONCLUSION AND PERSPECTIVES . . . 105
7.1 Contributions . . . 105
7.2 Perspectives . . . 107

REFERENCES . . . 109

LIST OF ABBREVIATIONS AND ACRONYMS

AMPI	Adaptive MPI
ASIC	Application-Specific Integrated Circuit
BC	Betweenness Centrality
BRAMS	Brazilian developments on the Regional Atmospheric Modeling System
BSP	Bulk Synchronous Parallel
CN	Compute Node
coNCePTuaL	Network Correctness and Performance Testing Language
C-PML	Convolutional Perfectly Matched Layer
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPPD	Parallel and Distributed Processing Group
HPC	High Performance Computing
HWLOC	Portable Hardware Locality
JLPC	Joint Laboratory for Petascale Computing
LB	Load Balancing
LICIA	International Laboratory in High Performance and Ambient Informatics
LIG	Grenoble Informatics Laboratory
LLC	Last Level Cache
MPI	Message Passing Interface
MPIPP	MPI Process Placement
MSTII	Mathematics, Information Sciences and Technologies, and Computer Science
MTS	Multithreaded Shepherds
NAMD	NAnoscale Molecular Dynamics
Nanosim	Nanosimulations and Embedded Applications for Hybrid Multi-core Architectures
NL 6	NUMAlink 6

NUCA	Nonuniform Cache Access
NUMA	Nonuniform Memory Access
PPL	Parallel Programming Laboratory
PM	Process Mapping
PU	Processing Unit
QPI	QuickPath Interconnect
RAM	Random Access Memory
RAMS	Regional Atmospheric Modeling System
RTS	Runtime System
RTT	Round-Trip Time
SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading
SPMD	Single Program, Multiple Data
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol
UFRGS	Federal University of Rio Grande do Sul
UMA	Uniform Memory Access
XML	Extensible Markup Language
WS	Work Stealing

LIST OF FIGURES

2.1 Example of machine topology of a UMA machine with 24 PUs distributed in 4 sockets. 18
2.2 Example of communication through memory on a UMA machine. . . 19
2.3 Example of a machine topology of a NUMA machine with 24 PUs distributed in 4 NUMA nodes. 19
2.4 Example of communication through memory on a NUMA machine. . 20
2.5 Example of a machine topology with multiple compute nodes. 4 UMA compute nodes are illustrated. 20
2.6 Example of communication through network organized as a fat tree. . 21
2.7 Different levels of load irregularity and dynamicity. 24
2.8 Different levels of communication irregularity and dynamicity. 24
2.9 Challenges for performance portability. 28

3.1 Machine topology representation of one socket of machine Xeon24. Different levels of the cache hierarchy present different latencies and bandwidths. 35
3.2 Midpoints for a memory hierarchy including the main memory and three cache levels. Midpoints represent the middle point in logarithm scale between the sizes of two levels. 36
3.3 Memory latency measured with `lat_mem_rd` for different data sizes on Opt48. 39
3.4 Memory bandwidth measured with `bw_mem` for different data sizes on Opt48. 40
3.5 Product between memory latency and bandwidth for machine Opt48. 40
3.6 Interconnections of the first eight NUMA nodes of machine Xeon192 based on topology file. 43
3.7 Interconnections of the first eight NUMA nodes of machine Xeon192 based on the machine specification of processor nodes. 43

4.1 Load balancers simplified class diagram. 61

6.1 Execution of *lb_test* on PUs 4 to 7 of Xeon32, as captured by Projections. 84
6.2 Total execution time of *kNeighbor* with NUCOLB and other load balancers. 86
6.3 Total execution time of *stencil4D* with NUCOLB and other load balancers. 86
6.4 Messages received per PU before load balancing (first 10 timesteps). . 87

6.5 Messages received per PU after load balancing with NUCOLB (remaining 40 timesteps). 88
6.6 Total execution time of *LeanMD* with NUCOLB and other load balancers. 88
6.7 Total execution time of *kNeighbor* with HWTOPOLB and other load balancers. 89
6.8 Total execution time of *stencil4D* with HWTOPOLB and other load balancers. 90
6.9 Total execution time of *LeanMD* with different load balancers. 92
6.10 Average timestep and average PU load for *Ondes3D* on Xeon32. Values averaged at each 20 timesteps. 93
6.11 Average timestep duration for *Ondes3D* on Xeon32 with HWTOPOLB and NUCOLB. Values averaged at each 20 timesteps. . 94
6.12 Total execution time of *LeanMD* on 20xOpt4. 95
6.13 Total execution time of *LeanMD* with HWTOPOLB and other load balancers on a varying number of Opt32 compute nodes. The problem size is increased with the number of compute nodes. 97
6.14 Total execution time of *LeanMD* with different load balancers on up to 16 Opt32 CNs. The problem size is increased with the number of compute nodes. 99
6.15 Total execution time of *LeanMD* with different load balancers on up to 16 Opt32 CNs. The problem size is the same for all numbers of compute nodes. 101
6.16 Speedup for different load balancers over the execution time of the baseline on 2 compute nodes. 101

LIST OF TABLES

2.1 Characterization of the example platforms in regards to machine topology symmetry and uniformity. 22
2.2 Characterization of the example applications in regards to application dynamicity and irregularity. 26
2.3 Characterization of the example applications in regards to sources of load imbalance and costly communications. 27

3.1 Memory latency (ns) as measured by `lat_mem_rd` on different machines. 41
3.2 Comparison of the different representations of communication distance from NUMA node 0 on Xeon192. 44

5.1 Task mapping algorithms comparison in terms of application modeling. 74
5.2 Task mapping algorithms comparison in terms of machine topology modeling. 76
5.3 Task mapping algorithms comparison in terms of techniques employed and objectives. 78

6.1 Overview of the platforms' hardware characteristics. 80
6.2 Overview of the platforms' software components. 80
6.3 Benchmarks and application characteristics. 81
6.4 Load balancers comparison in terms of machine topology modeling. . 82
6.5 Load balancers comparison in terms of application modeling and techniques employed. 83
6.6 Total execution time in seconds of *lb_test* and speedups for NUCOLB and other load balancers. 85
6.7 Load balancing times in seconds for *stencil4D* for different NUMA machines. 87
6.8 Average number of LLC misses (in millions) for *stencil4D* with HWTOPOLB and other load balancers on Opt32. 91
6.9 Total execution time in seconds of *Ondes3D* and speedups for NUCOLB, HWTOPOLB and other load balancers. 93
6.10 Average timestep duration and load balancing time (ms) for *LeanMD* with NUCOLB and other load balancers on 20xOpt4. 96
6.11 Average timestep duration of *LeanMD* with HWTOPOLB and other load balancing algorithms on a varying number of Opt32 compute nodes. 98

6.12 Average load balancing time of HwTopoLB and other load balancing algorithms for *LeanMD* on a varying number of Opt32 compute nodes. 98

ABSTRACT

This thesis presents our research to provide performance portability and scalability to complex scientific applications running over hierarchical multicore parallel platforms. Performance portability is said to be attained when a low core idleness is achieved while mapping a given application to different platforms, and can be affected by performance problems such as load imbalance and costly communications, and overheads coming from the task mapping algorithm. Load imbalance is a result of irregular and dynamic load behaviors, where the amount of work to be processed varies depending on the task and the step of the simulation. Meanwhile, costly communications are caused by a task distribution that does not take into account the different communication times present in a hierarchical platform. This includes nonuniform and asymmetric communication costs at memory and network levels. Lastly, task mapping overheads come from the execution time of the task mapping algorithm trying to mitigate load imbalance and costly communications, and from the migration of tasks.

Our approach to achieve the goal of performance portability is based on the hypothesis that precise machine topology information can help task mapping algorithms in their decisions. In this context, we proposed a generic machine topology model of parallel platforms composed of one or more multicore compute nodes. It includes profiled latencies and bandwidths at memory and network levels, and highlights asymmetries and nonuniformity at both levels. This information is employed by our three proposed topology-aware load balancing algorithms, named NUCOLB, HWTOPOLB, and HIERARCHICALLB. Besides topology information, these algorithms also employ application information gathered during runtime. NUCOLB focuses on the nonuniform aspects of parallel platforms, while HWTOPOLB considers the whole hierarchy in its decisions, and HIERARCHICALLB combines these algorithms hierarchically to reduce its task mapping overhead. These algorithms seek to mitigate load imbalance and costly communications while averting task migration overheads.

Experimental results with the proposed load balancers over different platform composed of one or more multicore compute nodes showed performance improvements over state of the art load balancing algorithms: NUCOLB presented improvements of up to 19% on one compute node; HWTOPOLB experienced performance improvements of 19% on average; and HIERARCHICALLB outperformed HWTOPOLB by 22% on average on parallel platforms with ten or more compute nodes. These results were achieved by equalizing work among the available resources, reducing the communication costs experienced by applications, and by keeping load balancing overheads low. In this sense, our load balancing algorithms provide performance portability to scientific applications while being independent from application and system architecture.

Keywords: Computer architecture, Parallel programming, Profiling, Scheduling.

1 INTRODUCTION

For years now, science has advanced thanks to the insights brought by numerical simulations. These scientific applications are used to reproduce and predict complex phenomena, with scales varying from the way a protein behaves to the effects caused by an earthquake. These applications have ever increasing demands in performance and resources. For instance, the higher the resolution of a weather forecasting model, the more precise and accurate would be its predictions, as local phenomena would be better represented in the system. In order to profit from the multiple resources available in high performance computing (HPC) platforms, scientific applications are developed using parallel languages and runtimes.

Parallel applications are decomposed into smaller parts, which can be implemented as processes, threads, objects, etc. We refer to them as tasks. Each task participates in the simulation by processing a part of the total workload. Before starting an application, it may not be possible to predict the amount of work, or load, each task will receive or produce. Load irregularity may come from tasks playing different roles in the application, or simulating different aspects or materials in the system. The load of a task can also change dynamically through time, as it achieves a different state in the simulation, or receives feedback from other tasks. When tasks are distributed over a parallel platform without taking such behaviors into account, load imbalance occurs.

In addition to the effects that tasks' loads have over the execution of a scientific application, there is also the affinity of tasks that affect it. A task's affinity group refers to the other tasks that it shares data or communicates with. It is in the best of interests to have communicating tasks mapped in the parallel platform in a way that reduces their communication times. However, communication times are not only affected by the communication behavior of a task, which can be irregular and dynamic as can happen to its load, but also by the organization of the HPC platform where the application is executing.

Typical HPC platforms are composed of multiple processing units, varying in scale from a multicore machine to a cluster composed of many multicore compute nodes. Their topologies include different cache, memory, and network levels organized hierarchically. As a consequence, the communication time between a pair of processing units or compute nodes will also depend on their distance in the whole topology.

Besides the effects caused by a hierarchical organization, communication times may not be uniform in some of the topology levels. At an interconnection network level, this happens because having a direct and dedicated interconnection between every pair of compute nodes is unfeasible in scale. Meanwhile, in order to mitigate the effects of the "memory wall problem" (WULF; MCKEE, 1995), not only is memory organized in a hierarchical fashion, but the main memory is also being assembled from different memory banks in a nonuniform memory access (NUMA) design. Besides nonuniformity, symme-

try may not be held, as the communication time between two components can depend on which direction it is happening, and the routing algorithms involved. If not taken into account when mapping tasks to the available resources, these characteristics of HPC platforms (namely nonuniform and asymmetric communication times, and hierarchical organization), together with an application's irregular and dynamic communication behaviors, can result in costly communications to said application.

Mitigating costly communications and balancing load are two different ways to improve application performance. Nevertheless, they can be orthogonal: an optimal task distribution for balancing load can negatively affect performance by increasing communication costs, as can the opposite happen. In this context, an equilibrium has to be found when focusing in both objectives for performance.

1.1 Problem statement

A challenge lies on managing the execution of a scientific application in order to explore the resources of a parallel platform to their fullest and, consequently, achieve high performance and scalability. This requires identifying characteristics of the application and platform, and managing which processing and communication resources get to be shared among tasks based on their loads and affinity. This problem is also aggravated by the fact that platforms and applications evolve independently from one another.

Finding the best task distribution to balance load and/or communication costs is a NP-Hard problem (LEUNG, 2004), thus requiring impracticable amounts of time for any application or platform of interest. To mitigate this problem, heuristics are employed to schedule tasks in feasible time. Although different algorithms exist, they may not provide performance portability.

A task mapping algorithm is said to attain performance portability when it is able to map a given application to different platforms and still achieve high efficiencies, which comes from a low core idleness. Although an algorithm can be tuned to a specific application and platform, this sacrifices portability in the process, as extensive work has to be done to provide scalable performance when running the same application in another platform, or another application in the same platform.

Performance portability also requires the task mapping algorithm to be performant, as its execution time will influence the application's performance. In this sense, a challenge lies in displaying performance portability with task mapping algorithms, as one needs to equilibrate balancing load, reducing communication costs, and keeping the algorithm's execution time small.

1.2 Objectives and thesis contributions

The main objective of our research is to **provide performance portability and scalability to complex scientific applications running over hierarchical multicore parallel platforms**. We follow the hypothesis that **precise machine topology information can help load balancing algorithms in their decisions**. Considering this objective and hypothesis, our contributions are the following:

- We develop a generic machine topology model of parallel platforms composed of one or more multicore compute nodes. It includes profiled information at memory and network levels. This model is used to provide detailed information to task

mapping algorithms. Our model is able to automatically measure, store, and make available the machine topology at a user level. It is also kept generic by using tools and benchmarks independent of application and system architecture.

- We propose load balancing algorithms that work at runtime level, which keeps them independent of a specific platform or application. They combine application information gathered during execution to our proposed machine topology model that considers real communication costs over the platform. They improve the distribution of tasks over a parallel platform in order to mitigate load imbalance and costly communications. They are able to handle applications with irregular and dynamic behaviors, and machine topologies with nonuniform and asymmetric communications costs (PILLA et al., 2012, 2014).

- We experimentally evaluate our approach with real benchmarks and applications over multiple parallel platforms. We implement our load balancing algorithms with a parallel language and compare their performance to other state of the art algorithms.

1.3 Research context

This research is conducted under the context of a joint doctorate between the *Institute of Informatics* of the Federal University of Rio Grande do Sul (UFRGS); and the *Mathematics, Information Sciences and Technologies, and Computer Science* (MSTII) Doctoral School, part of the University of Grenoble (UdG). This cooperation is held within the *International Laboratory in High Performance and Ambient Informatics* (LICIA).

At UFRGS, research is developed in the *Parallel and Distributed Processing Group* (GPPD). This research group possesses a vast experience with task mapping algorithms and scientific applications, which include load balancing algorithms for Bulk Synchronous Parallel applications (RIGHI et al., 2010) and weather forecasting models (RODRIGUES et al., 2010).

At UdG, this research is conducted in the *Nanosimulations and Embedded Applications for Hybrid Multi-core Architectures* (Nanosim) team, which is part of the *Grenoble Informatics Laboratory* (LIG). This team counts with an extensive background in multicore architecture modeling and memory management (RIBEIRO, 2011; CASTRO, 2012).

This research also profits from a collaboration with the *Parallel Programming Laboratory* (PPL) of the University of Illinois at Urbana-Champaign through the *Joint Laboratory for Petascale Computing* (JLPC). They are the developers of a parallel language and runtime system named CHARM++, which was used as a test bed for the implementation of our proposed load balancing algorithms.

1.4 Document organization

The remaining chapters of this thesis are organized as follows:

- A review of the main characteristics of parallel platforms and applications is presented in Chapter 2. It also discusses the basics of task mapping algorithms and machine topology probing.

- Our machine topology model is presented in Chapter 3. Its rationals, advantages,

and limitations are explored in this chapter. An evaluation of the parameters measured in the topology is also detailed.

- Two centralized load balancing algorithms are proposed in Chapter 4. They are: NUCOLB, a load balancer that focuses on the nonuniform aspects of parallel platforms; and HWTOPOLB, a load balancer that considers the whole machine topology hierarchy in its decisions. This chapter also presents HIERARCHICALLB, a hierarchical topology-aware load balancer based on the composition of centralized algorithms.

- The experiments conducted to evaluate our load balancers are shown in Chapter 6. This chapter includes information about the parallel platforms, benchmarks, applications, and other load balancers used to measure the performance and scalability of our algorithms.

- Concluding remarks and research perspectives are discussed in Chapter 7.

2 BACKGROUND

The performance of scientific applications can be affected by many different variables related to attributes intrinsic to the application itself and to the platform. For instance, at platform level, the communication time between two tasks will depend on the time that it takes for data to travel from one core to another through the machine topology. Meanwhile, when focusing in the application, the execution time of a task at a certain timestep will depend on the phenomena being simulated and the input data. These problems are accentuated as applications become more complex, and parallel platforms are designed with more hierarchical characteristics. In this context, task mapping algorithms are employed to reduce the impact of such attributes. They ease the process of distributing work over a parallel machine.

A task mapping algorithm provides performance portability by achieving a low core idleness when mapping an application to different platforms. This not only demands the algorithm to handle load imbalance and communication performance problems, but also to be performant by keeping its time overhead to a minimum. For an algorithm to decide which of these problems is the most relevant in a given situation, knowledge about the parallel platform and application of interest is required.

In this chapter, we discuss the core concepts of parallel platforms and applications, as well as their characteristics that affect performance. First, we introduce the features of parallel system architectures, how they are assembled, and how they can influence the communication time of applications. After that, we describe how parallel applications are composed, and exemplify how they behave using three different scientific applications. The performance challenges faced by task mapping algorithms and their basic concepts are explained next. We follow with a presentation of tools to control and probe the machine characteristics, which serve to provide detailed information to task mapping algorithms. We conclude this chapter with a discussion on the aforementioned topics.

2.1 Components of high performance computing platforms

High performance computing (HPC) platforms are parallel machines designed to run applications with high demands of computing power and other resources. The main approach to increase their computing power is to increase their parallelism. The amount of processing units per chip has been growing in the last decade, as has been the number of compute nodes used to compose a parallel system. For instance, the most performing parallel platform in the world as evaluated by the TOP500 list in November 2013 (DONGARRA; MEUER; STROHMAIER, 2013), known as Tianhe-2, is composed of $16,000$ compute nodes, and includes a total of $3,120,000$ processing units. Such platform design leads to highly hierarchical architectures, with complex memory subsys-

tems and network topologies. This makes achieving scalable performance with different applications a challenge.

2.1.1 Composition

When considering a bottom-up approach, the basic component of this system architecture is the **processing unit** (**PU**). A processing unit is the place where a task is executed. In this scenario, one core in a simple multicore processor would represent one PU, while a simultaneous multithreading (SMT) core would have two or more PUs. This is the same definition used by HWLOC (BROQUEDIS et al., 2010; HWLOC, 2013), as will be discussed later in Section 2.4.1.

When one or more PUs are grouped in a single shared memory space, we have a **compute node** (**CN**). Besides PUs and main memory, a compute node includes many levels of cache memory to accelerate memory access. These components are hierarchically organized, as illustrated in Figure 2.1. It shows the machine topology of one compute node composed of 24 PUs (6 PUs per socket). In this example, the memory hierarchy is organized as follows: each PU has its own L1 cache; pairs of PUs share a L2 cache; L3 cache is shared among all PUs inside the same socket; and the main memory is shared among all PUs.

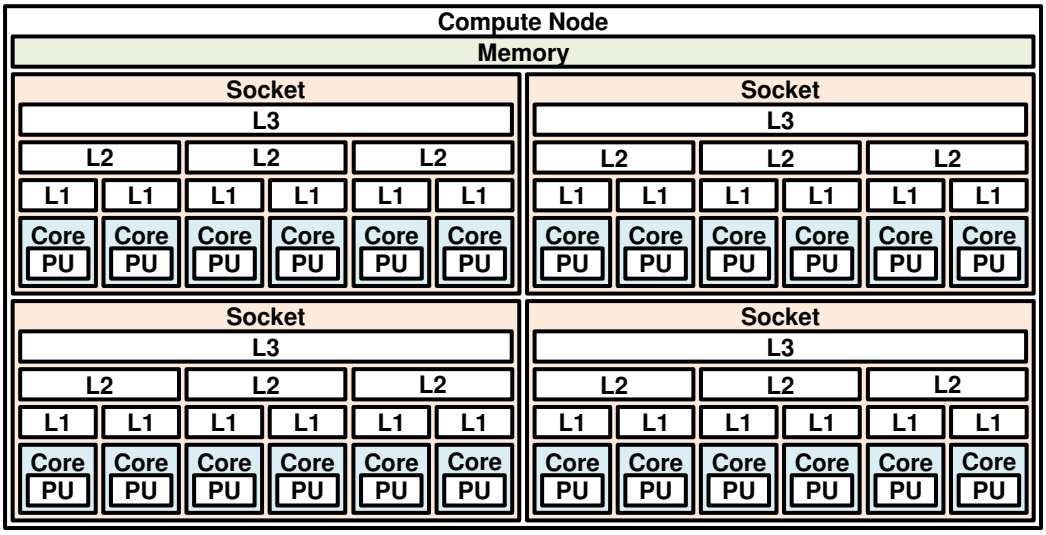

Figure 2.1: Example of machine topology of a UMA machine with 24 PUs distributed in 4 sockets.

Inside a compute node, tasks are considered to communicate through shared memory. It is also considered that they are able to benefit from the memory hierarchy when communicating. Data sent from one task to another is regarded as stored in the first level of the topology that is shared among the involved PUs. If the involved PUs do not share any cache level, then data can be found in the main memory. In the case of a uniform memory access (UMA) machine, all tasks accessing the main memory will take similar times. This happens because all use the same interface (for instance, a bus) to read data, as illustrated in Figure 2.2. However, the same does not happen with nonuniform memory access (NUMA) architectures.

NUMA architectures are a current trend in the design of parallel compute nodes. As the number of PUs inside a CN increases, so does increase the stress to the shared memory controller hub in a UMA CN. Meanwhile, on NUMA architectures, the main memory

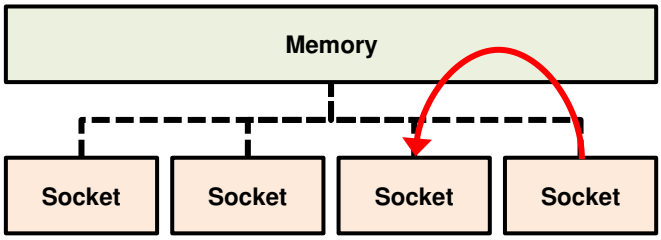

Figure 2.2: Example of communication through memory on a UMA machine.

is partitioned in multiple memory banks that are physically distributed, but the memory space is still shared among all PUs in the same compute node. This design has the advantage of spreading the memory accesses over these different banks. Each group of PUs sharing one memory bank is called a **NUMA node**. Figure 2.3 shows a compute node similar to the one seen in Figure 2.1, but partitioned into 4 NUMA nodes.

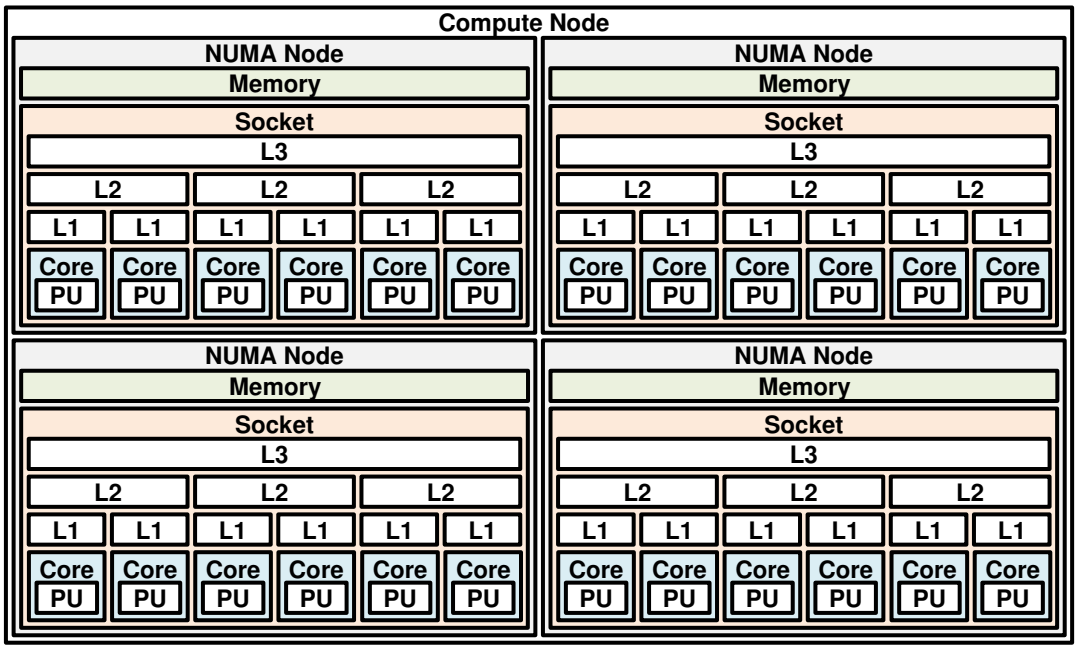

Figure 2.3: Example of a machine topology of a NUMA machine with 24 PUs distributed in 4 NUMA nodes.

When tasks residing in PUs from different NUMA nodes communicate, data is usually considered to be stored in the sender's memory bank due to a *first-touch* memory policy, as it stores data in the NUMA node of the first thread to access it (LOF; HOLMGREN, 2005). In this scenario, the receiver will have to make a remote memory access to read data. This is depicted in Figure 2.4. The arrow represents the path that data travels to get from sender to receiver. One of the main characteristics of NUMA machines is that accessing data in a remote memory bank takes longer than accessing it in local memory.

Besides the memory hierarchy of a compute node, a network hierarchy is present when clustering more than one CN. When using the same representation illustrated in Figures 2.1 and 2.3, an additional level is added to the machine topology to include the network interconnection. This is depicted in Figure 2.5. Tasks communicating in this platform use the same mechanisms discussed before if both tasks happen to be in the

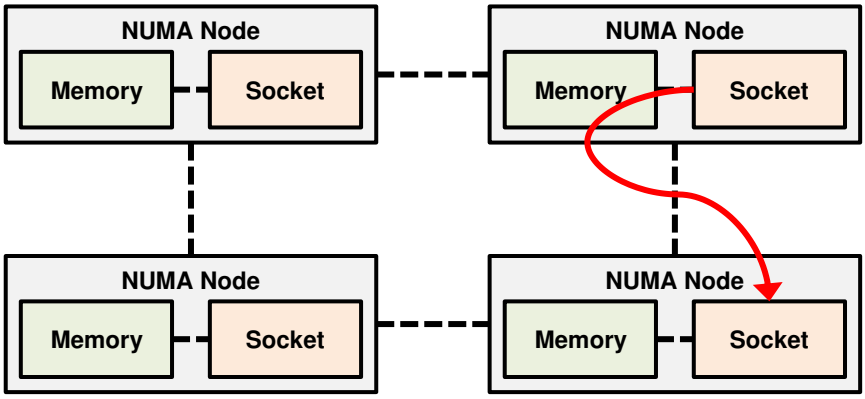

Figure 2.4: Example of communication through memory on a NUMA machine.

same CN. If that is not the case, then data will be sent to the receiver's CN. This communication organization is displayed in Figure 2.6. The time that it takes for two tasks to communicate through network is usually considered to be greater than the time it takes for them to communicate through memory.

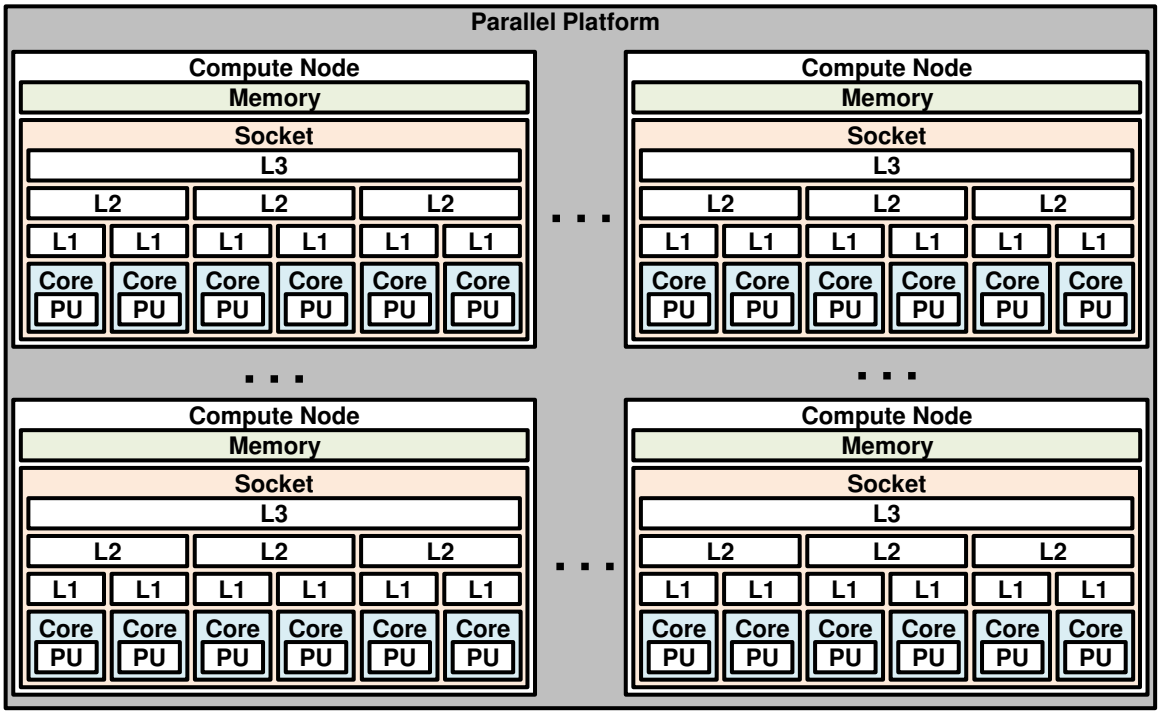

Figure 2.5: Example of a machine topology with multiple compute nodes. 4 UMA compute nodes are illustrated.

The communication time between two PUs is strongly influenced by the machine topology level where it happens. The closer the topology level is to the PUs, the smaller this time is. In this sense, communication at a cache level is faster than at a memory or network level. Still, other factors can influence communication performance. For instance, contention happens when there is conflict for a shared resource (e.g., a network link), which decreases the performance of tasks using it. Besides that, differences inside a topology level can affect performance, as discussed next.

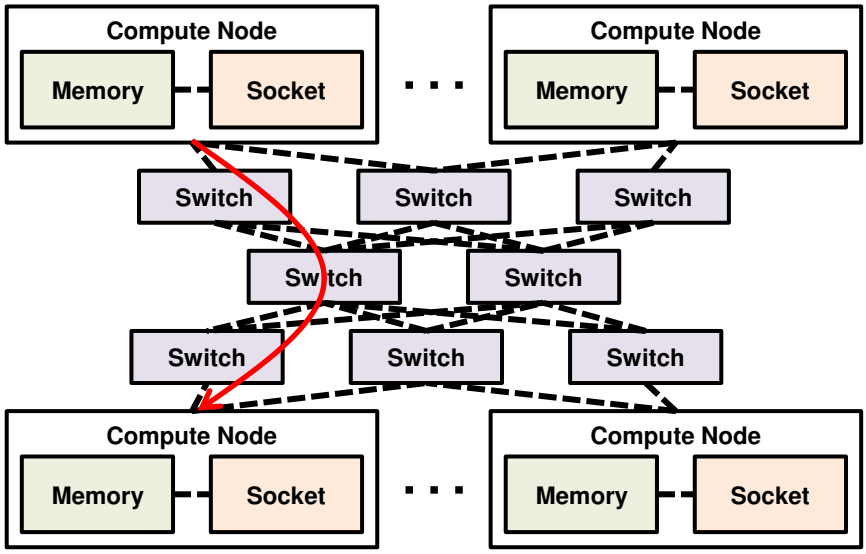

Figure 2.6: Example of communication through network organized as a fat tree.

2.1.2 Asymmetry and nonuniformity at topology levels

When studying the communication time among PUs in the machine topology, two important properties emerge: **symmetry** and **uniformity**. They are explained below:

- **Symmetry**: a level in the machine topology is said to be **symmetric** if the communication time of a first task sending data to a second one is the same than the communication time of the second task sending data to the first one. When this does not happen, a topology level is said to be **asymmetric**.
- **Uniformity**: a level in the machine topology is said to be **uniform** if all tasks communicating through that level have the same communication time. When this does not happen, a topology level is said to be **nonuniform**.

For a more formal definition, a machine topology can be denoted as a quadruple $\mathcal{O} = (\mathcal{P}, \mathcal{L}, S, C)$, with $\mathcal{P}$ the set of PUs, $\mathcal{L}$ the set of levels of the topology, the first topology level shared by two PUs as a function $S : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{L}$, and the communication time of one PU to another as a function $C : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{R}_{>0}$. S is symmetric and transitive.

A topology level $l \in \mathcal{L}$ is considered to be symmetric if

$$\forall a, b \in \mathcal{P} \wedge S(a, b) = l \Rightarrow C(a, b) = C(b, a). \tag{2.1}$$

Additionally, a topology is said to be symmetric if Equation 2.1 holds for all $l \in \mathcal{L}$.

A topology level $l \in \mathcal{L}$ is considered to be uniform if

$$\forall a, b, c \in \mathcal{P} \wedge S(a, b) = S(a, c) \Rightarrow C(a, b) = C(a, c). \tag{2.2}$$

Likewise, a topology is said to be uniform if Equation 2.2 holds for all $l \in \mathcal{L}$. It is important to notice that a topology level cannot be asymmetric and uniform at the same time due to $\mathcal{S}$ being a symmetric function. All other combinations are valid.

Table 2.1 relates symmetry and uniformity to the machine topologies illustrated in Figures 2.1, 2.3, and 2.5. Other combinations can happen in real platforms, such as a cluster of NUMA CNs. Cache levels are usually, but not exclusively, symmetric and uniform.

Table 2.1: Characterization of the example platforms in regards to machine topology symmetry and uniformity.

Platform	Cache levels	Memory level	Network level
UMA CN	Symmetric	Symmetric	—
	Uniform	Uniform	—
NUMA CN	Symmetric	Asymmetric	—
	Uniform	Nonuniform	—
Clustered UMA CNs	Symmetric	Symmetric	Symmetric
	Uniform	Uniform	Nonuniform

Nonuniformity can be seen in non-uniform cache access (NUCA) architectures (KIM; BURGER; KECKLER, 2002). The **memory level** is uniform for UMA compute nodes, and nonuniform for NUMA CNs. NUMA compute nodes may also present asymmetry (RIBEIRO, 2011). The **network level** strongly depends on the network topology. In Figure 2.6, all compute nodes are interconnected through a tree of switchs. This results in a symmetric but nonuniform network level. Asymmetry may come in the network level as a result of routing (HOEFLER; SCHNEIDER, 2012).

The hierarchical design of a system architecture, combined with asymmetric and nonuniform topology levels, impacts the communication time of an application. If the different communication times between pairs of processing units are not taken into account, then communication can hinder application performance. Still, the machine topology is not the only factor affecting the total execution time of a parallel application, as characteristics of the application itself play a role too. These characteristics are discussed in the next section.

2.2 Characterization of scientific applications

Scientific applications are used to simulate phenomena through time at different scales (e.g., from the way molecules interact in nanoseconds, to the climate in years). Simulations involve large datasets and/or much processing. These parallel applications have their work and data split into **tasks** which populate the resources available in parallel platforms. The actual implementation of these tasks depends on the programming language. For instance, tasks may be implemented as threads in OPENMP (DAGUM; MENON, 2002), processes in MPI (GROPP; LUSK; SKJELLUM, 1999), and active objects in CHARM++ (KALE; KRISHNAN, 1993). The number of tasks in an application can be much larger than the number of PUs available in the parallel platform.

2.2.1 Tasks: load and communication

Each task has an amount of processing to do. This is going to be referred as **load** in this thesis. A task's load is measured as the time it takes running on a processing unit in a machine. The bigger the load, the longer the execution time. Although load could be decomposed into two parameters, amount of work of a task and PU performance, we keep it as single entity for simplicity. All platforms considered in this research are homogeneous, which means that their processing units have the same performance. This is the same approach seen in related works (CHEN et al., 2006; HOFMEYR et al., 2011; JEANNOT;

MERCIER, 2010; LIFFLANDER; KRISHNAMOORTHY; KALE, 2012; TCHIBOUKDJIAN et al., 2010; OLIVIER et al., 2011). This is discussed in more details in Chapter 5.

As tasks compute, the simulation iteratively evolves, and tasks start to communicate. Even though simulations commonly evolve in timesteps, they are not required to follow a Bulk Synchronous Parallel (BSP) (VALIANT, 1990) approach.

Task communication may happen through memory or network, depending on where the involved tasks are mapped, as previously discussed in Section 2.1. For the sake of simplicity, we are going to refer to data exchanges as **messages**. All messages sent and received during the execution of an application (or a part of it) can be seen as its **communication graph**, where vertices represent tasks and edges represent communication. Communication is measured by the number of messages sent from a task to another and the amount of bytes communicated. The time that it takes for a task to receive a message depends on both factors, and is also strongly affected by the distance between the processing units where tasks are mapped.

Considering these characteristics, an instance of an application execution (or part of it) can be defined as a quintuple $\mathcal{A} = (\mathcal{T}, Load, Size, Msgs, Bytes)$, with $\mathcal{T}$ the set of tasks, their loads as a function $Load : \mathcal{T} \rightarrow \mathbb{R}_{>0}$, their sizes in bytes as a function $Size : \mathcal{T} \rightarrow \mathbb{R}_{>0}$, the number of messages sent from one task to another as a function $Msgs : \mathcal{T} \times \mathcal{T} \rightarrow \mathbb{N}$, and the number of bytes sent from one task to another as a function $Bytes : \mathcal{T} \times \mathcal{T} \rightarrow \mathbb{N}$. This organization is used to explain how application behavior interacts with its performance next.

2.2.2 Application irregularity and dynamicity

The way an application and its tasks behave can affect performance in different ways, e.g., tasks may have different loads; their loads may change through time; and communication may follow different patterns in different phases of the application. In the context of this thesis, we focus on two task characteristics: (i) how their load behaves; and (ii) how their communication behaves. This does not take into account other characteristics that may influence the final performance of an application, such as the way tasks interact with the file system. Communication and load vary from one application to another in two axes: **regularity** and **dynamicity**. They are explained in more details below:

- **Regularity** refers to how the load or communication of a task differs from another task. An application with **regular load** has tasks that compute for approximately the same time. When this is not the case, an application is said to have **irregular load**. **Regular communication** is present in applications with well-defined communication graphs, where the number of messages and amount of bytes exchanged is the same among different tasks. Meanwhile, an application with **irregular communication** has a complex communication graph, or the amount of bytes or messages communicated varies between pairs of tasks.

- **Dynamicity** refers to how the load or communication of a task varies through time. An application with **dynamic load** has tasks that compute for different amounts of time at different timesteps. If loads are constant through time, an application is said to have **static load**. An application with **static communication** has a communication graph that does not change. If that is not the case for an application, it is said to exhibit **dynamic communication**.

Figures 2.7 and 2.8 illustrate how regularity and dynamicity affect the load and communication graph of an application, respectively. The vertical axis represents changes

in regularity, while the horizontal axis represents a variation in dynamicity through timesteps. The horizontal bars in Figure 2.7 represent the load of four different tasks during three timesteps. The circles in Figure 2.8 represent four tasks in two different timesteps, while the arrows represent messages, and their thickness imply the volume of data exchanged. As these figures illustrate, regularity and dynamicity are not absolute characteristics, as they vary in levels.

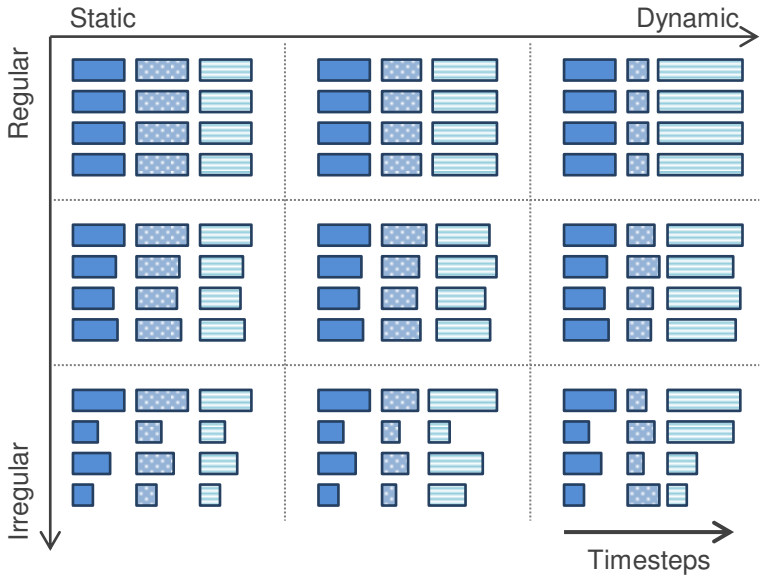

Figure 2.7: Different levels of load irregularity and dynamicity.

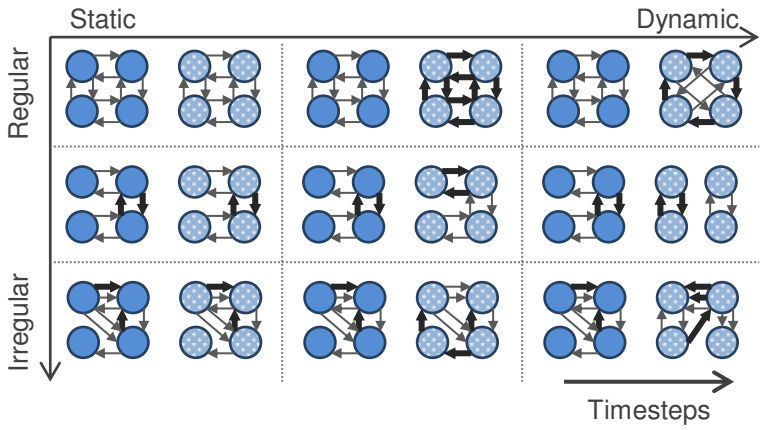

Figure 2.8: Different levels of communication irregularity and dynamicity.

Irregularity and dynamicity affect application performance at both load and communication sides. If the total load of tasks mapped to one PU is greater than in others, then PUs will be idle while waiting for the slowest (most loaded) PU. This scenario can easily happen in an application with a large load irregularity. Additionally, load dynamicity can generate this scenario during execution time, making it hard to predict and avoid. Meanwhile, communication time will reduce if tasks that communicate more than others are mapped closer in the machine topology, and it will suffer if the opposite happens. Communication dynamicity can change which tasks have their performance affected by the communication time.

All these characteristics can be present in different levels in a scientific application. We exemplify this next.

2.2.3 Examples of scientific applications

To better understand how dynamicity and irregularity present themselves in real scenarios, we analyze three scientific applications from different areas: seismology, weather forecasting, and molecular dynamics.

Seismic wave propagation models are mainly employed to estimate the damage in future earthquake scenarios. Simulations are applied in a regional scale, where the domain is divided into a three-dimensional grid. Each sub-domain can be seen as a task. In addition to the simulated region, the domain also includes artificial borders used to process and absorb the outgoing energy of the seismic waves. In these models, task communication is regular and static, as tasks only communicate with their neighbors. Their load is mostly static, but wave propagation can spread dynamism in nonlinear simulations (DUPROS et al., 2010). These applications can also present load irregularity. This happens because the physical domain simulation has a different load than the artificial absorbing borders; and because different geological layers may have different constitutive laws, which results in different computational costs (DUPROS et al., 2008, 2010; TESSER et al., 2014).

Weather forecasting models are used to predict the state of the atmosphere at given place and time. This prediction can vary from tomorrow's weather to how the climate is going to be in the following years. Models such as the Regional Atmospheric Modeling System (RAMS) (WALKO et al., 2000), and its Brazilian variant, BRAMS (BRAMS, 2013), split part of the globe and the atmosphere in a three-dimensional mesh. Each of these parts can be seen as a task. Their communication follows a regular and static behavior as seismology models do. Weather forecasting models can present load irregularity, as tasks may have different workloads depending on input data. Load dynamicity can also happen due to phenomena moving through the simulated area (e.g., thunderstorms) (RODRIGUES et al., 2010)(XUE; DROEGEMEIER; WEBER, 2007). This dynamism happens in a scale smaller than the one seen in seismic wave propagation.

Molecular dynamics simulations are employed to study the dynamics and properties of biomolecular systems. Typical experiments with applications such as Nanoscale Molecular Dynamics (NAMD) (NELSON et al., 1996) simulate the behavior of the molecular system for hundreds of nanoseconds. This takes millions of femtoseconds steps to simulate (BHATELE et al., 2009). NAMD uses a hybrid of spatial and force decomposition methods, where the simulation space is divided into cubical regions called *cells*, and the forces between two cells are the responsibility of *computes*. These two kinds of tasks, cells and computes, bring load irregularity to the application. Load dynamicity happens as simulated atoms can move from one cell to another. Communication is also irregular, as it involves different communication patterns at the same time, such as pair to pair and multicasts.

Table 2.2 summarizes the irregular and dynamic behaviors of the three applications. Although these applications have similarities, as being iterative and involving a three-dimensional space, the differences in the simulated phenomena results in various combinations of performance challenges to be handled when executing them in parallel platforms. In the next section, we discuss the potential performance problems that should be considered when mapping tasks to physical resources.

Table 2.2: Characterization of the example applications in regards to application dynamicity and irregularity.

Application	Irregularity	Dynamicity
Seismic wave propagation	Load (two kinds of tasks)	Load
Weather forecasting	Load (physical regions)	Load
Molecular dynamics	Load (two kinds of tasks) and communication	Load

2.3 Task mapping

Task mapping algorithms serve to, as their name says, guide the way tasks are mapped to physical resources (mainly the processing units). More formally, we define a task mapping as a function $\mathcal{M} : \mathcal{T} \rightarrow \mathcal{P}$, where $\mathcal{T}$ represents the set of tasks of an application, and $\mathcal{P}$ is the set of PUs of a platform.

Some variations of task mapping algorithms can be called load balancers, schedulers, process mapping algorithms, and others. They play a central role in achieving performance portability with scientific applications running on parallel HPC platforms, as properties of both can affect application performance. For instance, a naïve task mapping involving an equal number of tasks per PU can perform well with a regular, static application over a symmetric and uniform platform. However, any irregular or dynamic behaviors can result in an increase of core idleness and loss of scalability. We discuss the main performance problems considered in this thesis below.

2.3.1 Load imbalance and costly communications

The aforementioned behaviors can negatively affect performance in two ways: (i) through **load imbalance**; and (ii) through **costly communication**. An application is said to be load unbalanced when its current task mapping presents processing units with significant load differences. The load of a processing unit is considered to be the sum of the load of the tasks mapped to it. These load differences result in processing units being idle while waiting its tasks to synchronize with others. Such idleness affects the parallel efficiency and scalability of an application. Regular applications are the easiest to reduce load imbalance, as an even task distribution results at most in an *off-by-one* imbalance, where the number of tasks on each processing unit is within one of each other. Meanwhile, dynamic applications are harder to keep balanced, as changes in behavior during execution time are more difficult to predict and mitigate.

Load imbalance is mostly independent of the machine topology. However, the opposite happens for costly communications. The time that it takes for two tasks to communicate depends on how many messages are exchanged, the data volume, and where these tasks are mapped. For instance, communication through network is usually considered to take longer than through shared memory. If we consider that the communication time of an application is the sum of the time that all its messages take, then an application is considered to have costly communications if its current communication time is much greater than its optimal communication time. In other words, an application is said to have costly communications if its current task mapping does not benefit from the machine topology. Applications with regular communication are easier to map to the machine topology in a way that reduces communication costs.

Table 2.3: Characterization of the example applications in regards to sources of load imbalance and costly communications.

Application	Load imbalance	Costly communication
Seismic wave propagation	Load irregularity and dynamicity	Bad task mapping
Weather forecasting	Load irregularity mainly	Bad task mapping
Molecular dynamics	Load irregularity and dynamicity	Irregular communication

The impact of load imbalance and costly communications will depend on the application and platform of interest. For instance, considering the three applications discussed in Section 2.2.3, Table 2.3 summarizes possible sources of load imbalance and costly communications. Seismic and weather simulations are less likely to suffer from costly communications, as their communication behavior is strongly static and regular. Still, a task mapping that leaves communicating tasks far from each other could generate a performance problem. Meanwhile, load irregularity and dynamicity pose challenges to a load balanced task distribution for the three applications. All these factors and some more have to be taken into consideration to provide performance portability to parallel applications, as is discussed next.

2.3.2 Performance portability

Performance portability is achieved when an application can be mapped to different platforms and still achieve low core idleness. This requires a task mapping that mitigates the effects of load imbalance and costly communications. Besides these two challenges, the task mapping algorithm itself must not be a liability to performance.

To better illustrate these concepts, Figure 2.9 depicts a scenario at its top where an application running on four PUs is unbalanced and suffering from costly communications. The vertical bars represent the load on each PU, and the arrows represent the communication time among PUs. Communication is illustrated after the computation phase of the application only to evidence it, as communication can happen at the same time tasks are computing.

Below the initial task mapping in Figure 2.9, five different task mappings are presented. Mapping (a) improves communication, but does not solve the load imbalance problem. Mapping (b) fixes the load imbalance, but increases the communication costs of the application. This could be a result of the task mapping algorithm not taking into account the communication behavior of the application or the machine topology. Mapping (c) solves both problems, but application performance is affected by the execution time of the task mapping algorithm. While this algorithm is running, the application is stalled. A similar problem happens with mapping (d), where the task migration overhead increases the total execution time of the application. The ideal case in Figure 2.9 is illustrated by mapping (e), where both performance problems are handled by a task mapping algorithm with a low overhead.

For performance portability to be achieved, a task mapping algorithm must take informed decisions regarding where to map tasks. In the next section, we discuss the main sources of application and platform knowledge used by task mapping algorithms.

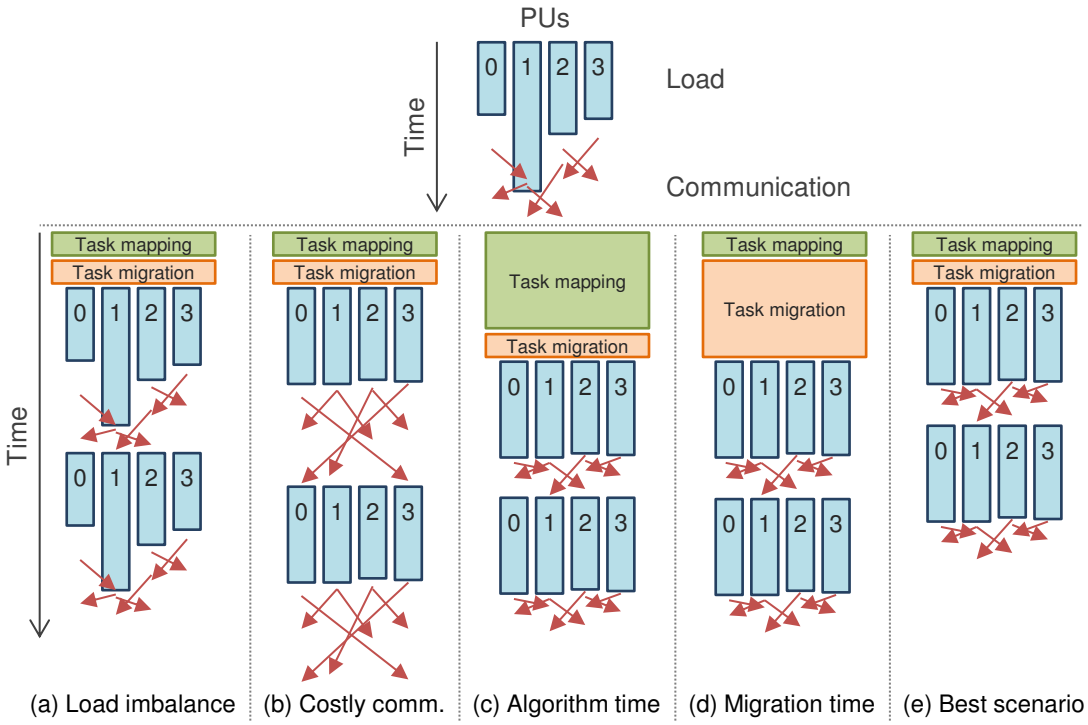

Figure 2.9: Challenges for performance portability.

2.3.3 Information required for mapping tasks

As previously discussed in Sections 2.2.1 and 2.1.2, we can see an application execution as the quintuple $\mathcal{A} = (\mathcal{T}, Load, Size, Msgs, Bytes)$, and the machine topology of the parallel platform where it is executing as a quadruple $\mathcal{O} = (\mathcal{P}, \mathcal{L}, S, C)$. Nonetheless, different task mapping algorithms may only consider part of this representation for its decisions, as they focus on different objectives or scenarios where part of this information is not available.

Task mapping algorithms that focus on mitigating load imbalance usually require some knowledge of the execution time of the tasks or the current utilization of the processing units. Tasks' loads, or $Load$, can be obtained by timing the execution of each task (for instance, at a runtime (KALE; KRISHNAN, 1993) or operating system (HOFMEYR et al., 2011) level), or by predicting their processing time based on some previous knowledge of the application. However, this second approach can only be applied to applications with static loads mostly. By using $Load$, an algorithm will seek to distribute tasks in a way that the sum of their loads in each PU is similar.

The total time spent of migrating tasks, or the migration overhead, can be estimated using the amount of bytes private to each task, or their $Size$. The current task mapping M is also important in this situation, as it enables the task mapping algorithm to keep some tasks where they currently are, avoiding unnecessary task migrations.

The communication graph of an application plays an important role when trying to reduce costly communications. Task mapping algorithms can use the number of messages ($Msgs$) or the data volume exchanged between tasks ($Bytes$) to evaluate which tasks should be mapped close to each other. Such information can be captured at operating system level (DIENER; CRUZ; NAVAUX, 2013), runtime level (FRANCESQUINI; GOLDMAN; MEHAUT, 2013), or even by tracing an execution of the application (MERCIER; CLET-ORTEGA, 2009). Still, with no knowledge about the machine topology, one can

only map communicating tasks to the same PU.

The machine topology can be used by task mapping algorithms in two different detail levels. The first one involves using the machine topology hierarchy, as the set of levels in the topology $\mathcal{L}$ and the first topology level shared by two PUs S, to designate which PUs are closer to each other. However, this approach does not differentiate nonuniform or asymmetric levels. The second step includes knowledge about the communication time between PUs C. By combining this information with the communication graph of the application, one can estimate the communication costs of different mappings. The mechanisms used to discover the machine topology hierarchy and communication times are the subject of the next section.

2.4 Machine topology identification

Existing approaches to identify the characteristics of a system architecture can be divided into two different groups. The first group focuses on controlling and discovering the different topology levels in a parallel platform, while the second group involves benchmarks and applications to obtain the communication times of the platform and other parameters. We present examples of both groups in the next sections and discuss which characteristics of the machine topology they could be able to provide to task mapping algorithms.

2.4.1 Hierarchy perception

A first technique to organize and model the hierarchical topology of a machine would be to read the vendors' architecture specifications and to describe it manually. An advantage of such approach lies in the ability to obtain architecture details that may not be available for tools (e.g., the physical distance between two components). However, this faces several issues, such as: (i) the problems of being manual work (portability, propensity to errors, scalability, etc.); (ii) limited information made available by vendors; and (iii) at a communication costs level, the differences between what is specified and what the system provides under different workloads. Due to such limitations, this approach is usually not chosen.

A tool used in different works to provide information about the system's architecture is named Portable Hardware Locality, or **HWLOC** (BROQUEDIS et al., 2010)(HWLOC, 2013). HWLOC provides a portable abstraction of the underlying machine hardware, describing the system architecture hierarchy. It automatically gathers the machine topology of a compute node, and it provides the ability to extend it to include multiple compute nodes. HWLOC contains supplementary information about machine components, such as cache sizes, line sizes, and associativity. Its interface supports the manipulation of the hardware abstractions, and the binding of tasks and memory onto PUs and NUMA nodes. HWLOC is able to represent nonuniformity. For instance, it uses a distance matrix made available by the BIOS to represent the distance among NUMA nodes. However, it does not express asymmetries at a topology level.

Another tool like HWLOC is **LIKWID** (TREIBIG; HAGER; WELLEIN, 2010)(LIKWID, 2013). The main difference between the two is that LIKWID supports hardware performance counters for a target application and architecture. Nonetheless, this feature depends on the support provided by the hardware. It represents nonuniformity for NUMA nodes the same way that HWLOC does, and it also does not report asymmetries. Moreover, LIKWID is limited to one compute node, and it does not offer an API to manipulate the

hardware abstraction as data structures.

Instead of using general tools like HWLOC and LIKWID, some companies prefer to provide machine topology information through special interfaces to developers. For instance, the UV 2000 series server from SGI includes a topology file with its operating system that contains all NUMAlink 6 interconnections between pairs of NUMA nodes inside the parallel platform (SGI UV 2000 System User Guide, 2012). With this information, one can measure the distance between two NUMA nodes in the machine as the number of hops.

A last approach to probe the machine topology involves benchmarking the hardware, as done by **Servet** (GONZALEZ-DOMINGUEZ et al., 2010)(SERVET, 2013). Servet is a benchmark suite to characterize the communication costs of platforms composed of multiple UMA compute nodes. It assesses the cache hierarchy, memory access costs and bottlenecks, and the communication latency among PUs. This communication latency is used to group PUs into levels, which form the machine topology. However, Servet does not explicitly contemplate nonuniformity in the memory level.

Besides Servet, there are different sets of benchmarks to evaluate the memory and network levels of a machine topology. A brief list is presented next.

2.4.2 Communication cost scanning

Benchmarks that work inside a single compute node usually focus in more than just scanning communication costs. For instance, **BlackjackBench** (DANALIS et al., 2012)(BLACKJACK, 2013) is a suite composed of several micro-benchmarks to probe the hardware characteristics of a compute node, such as caches' line size, size, latency, and associativity; TLBs number, page size, and working set; instruction latencies and throughputs; and others. BlackjackBench's focus is to provide hardware information for algorithm-guided tuning, and architecture-aware compiler environments. It also contains a collection of scripts to ease and automate the statistical analysis of benchmarked data. Analysis techniques include enforcing monotonicity to reduce noise when it makes sense, e.g., considering that cache access latency only increases as we move farther from the processing unit; and finding the steps in curves that represent hardware changes, such as the size of a cache level in the hierarchy, by finding the biggest gradients (relative values increase).

A very traditional benchmark suite that considers characteristics of a compute node and its network interconnections can be found in **LMBENCH** (STAELIN, 1996)(LMBENCH, 2013). It includes benchmarks to evaluate PUs, memory, network, file system, and disk. Although current implementations are able to run benchmark instances in parallel, the original suite was focused on evaluating CNs with one PU only. For this reason, they cannot uncover asymmetric and nonuniform levels in the topology by themselves. Memory benchmarks assess read latency and read bandwidth of the different cache and memory levels, and include an implementation of the STREAM benchmark (STAELIN, 1996, ap. (MCCALPIN, 1995)). Meanwhile, the network benchmark measures the round-trip time (RTT) between two CNs using two different transport protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

When focusing at an interconnection network level only, different frameworks are available for the development of benchmarks. An example of this is **Netgauge** (HOEFLER et al., 2007)(NETGAUGE, 2013). Netgauge splits communication patterns, such as one-to-one or one-to-many, from the communication protocols and interfaces used, such as InfiniBand or TCP. It also provides the ability to implement new modules and

insert them in the framework. Similarly to BlackjackBench, Netgauge automates the statistical analysis of benchmarked data.

Meanwhile, at a higher level than Netgauge, **coNCePTuaL**, or Network Correctness and Performance Testing Language (PAKIN, 2007)(CONCEPTUAL, 2013), is a domain-specific language to write network benchmarks. It eases the expression of complex communication patterns with its high level language. coNCePTuaL's focus is to provide a portable, readable, and reproducible way to evaluate networks. Benchmarks described with coNCePTuaL are used to generate C code using different communication interfaces, such as TCP and MPI, that are then executed. When run, they automatically document the state of the system. coNCePTuaL is also able to process output data to ease its statistical analysis.

It can be noticed that no tool or benchmark alone is able to capture information from both memory and interconnection levels while also detecting the communication costs of the platform. This motivates the development of new approaches to model the machine topology.

2.5 Discussion

In this chapter, we presented the main characteristics of parallel HPC platforms and scientific applications that influence performance. We have seen that a parallel platform is composed of multiple processing units distributed over one or more compute nodes. Its machine topology is organized hierarchically, including multiple cache, memory, and network levels. When tasks mapped to processing units exchange messages, their communication time can be influenced by asymmetry and nonuniformity at different topology levels. When communication time is unoptimized, we say that the task mapping suffers from costly communications.

Besides the influence of the machine topology on the communication time, the parallel application also plays an important role. Communication performance is affected by its communication graph, which includes the number of messages exchanged between its tasks and their data volume, and may present irregular and dynamic behaviors.

Irregularity and dynamism can also be present at a task load level, as exemplified with scientific applications from three different areas: seismic wave propagation, weather forecasting, and molecular dynamics. If a mapping results in a large task load difference among PUs, the faster PUs will have to idly wait for the slowest one to continue their computations. In this scenario, we say that the task mapping suffers from load imbalance.

We defined that a task mapping achieves performance portability when it is able to map an application to different parallel platforms while maintaining a low core idleness. This involves mitigating load imbalance, costly communications, and also keeping its own overhead to a minimum. Task mapping overhead is related to the time spent by an algorithm computing a new task mapping, and the time spent migrating tasks. To achieve all that, a task mapping algorithm requires detailed information about the application and the machine topology.

Different approaches to obtain information about the machine topology were discussed. Most tools and applications focus on identifying the different topology levels in a parallel platform or evaluating the communication costs at different levels, but not both. To fill this gap, we propose our own approach to model the machine topology of HPC platforms in the next chapter, and employ it in novel topology-aware load balancing algorithms in Chapter 4.

3 MODELING THE TOPOLOGIES OF HPC PLATFORMS

As discussed in the previous chapter, a parallel platform can include complex memory and network hierarchies in its topology. Such topology may present asymmetries and nonuniformity in different levels, which influence the performance of scientific applications. This is specially true when considering their communication performance. To mitigate this problem, task mapping algorithms can take into account properties of the parallel platform and organize them into their own machine topology model. However, current tools are usually focused on modeling the memory hierarchy and/or the network hierarchy, or capturing the communication costs of platform, but not both.

In this chapter, we present the rationale of our machine topology model, as well as its features, limitations, techniques, and tools involved. Our main objective is to provide a unified machine topology model that includes both memory and network levels, and that exposes asymmetry and nonuniformity in different levels, so they can be used by task mapping algorithms and others. Additional objectives include:

- Presenting a general vision of the topology, allowing the model to be applied over different platforms in a portable way;
- Providing an interface that allows the machine topology model to be used by different algorithms, specially task mapping ones;
- Modeling communication times over the topology with real, precise, and accurate costs;
- Profiling the machine topology without incurring in large overheads. This includes limiting the scope of information collected; and
- Building our solution over well established tools.

3.1 Topology representation

We start this chapter by introducing how different topology levels are represented in our model, followed by a discussion about the parameters used to represent communication costs.

3.1.1 Topology tree

We base our approach to machine topology modeling over the abstraction provided by HWLOC (BROQUEDIS et al., 2010). HWLOC exposes information about a compute node system's architecture to applications and runtimes. Examples of such information are the

number of processing units, cache sizes and sharing, and the presence of different memory nodes. HWLOC also enables process, thread, and memory binding, which are important features required to profile the communication costs of a platform. This is discussed in more details in Section 3.2.1.

HWLOC represents the machine topology as a tree, where processing units (PUs) are leaves and the compute node (CN) is the root. All other components in the machine, such as cache memories, sockets, and NUMA nodes serve as intermediary nodes. This tree can be traversed, for instance, to find the first topology level shared by two PUs, or to which NUMA node a PU belongs.

Although HWLOC provides an automatic machine topology detection mechanism, it is limited to a CN only. However, to overcome this limitation, HWLOC provides to the developer the ability to extend its representation to include multiple compute nodes. For a machine topology $\mathcal{O}$ as described in Section 2.1.2, HWLOC provides the set of PUs $\mathcal{P}$, the set of topology levels $\mathcal{L}$, and how levels are shared as a function S. In other words, it only lacks the communication costs C.

HWLOC considers nonuniformity at a memory level by exposing the distance matrix made available by the BIOS to represent the distance among NUMA nodes. These values can be seen as synthetic communication costs, as they do not represent actual parameters of the topology. We extend HWLOC's machine topology model by benchmarking the cache, memory, and network hierarchies to compute the communication costs C. More details are presented in Sections 3.2.1 and 3.2.2.

3.1.2 Communication costs

Two metrics were chosen to portray the communication costs achieved through the machine topology. These are latency and bandwidth. They are able to represent the different factors that impact communication performance in a simplified way.

Latency (C_{lat}) expresses the cost to first access a message and bring it closer to a task. Latency affects the waiting time for data directly. The longer it takes to access data, the farther the communicating tasks are considered to mapped. Latency is also called as *delay* in the LogP model for networks (CULLER et al., 1993).

Bandwidth (C_{band}) reflects the limit in amount of data that can be accessed per unit of time. It can be used to observe bottlenecks in the memory and network topologies. Bandwidth is seen as the inverse of the *overhead* parameter in LogP. When used in combination with latency, it provides an estimation of the time that it takes to access a certain data quantity. It is important to clarify that latency and bandwidth are not directly related. This is discussed in more details with experimental results in Section 3.3.1.

The communication costs between two PUs are related to the first topology level that is shared by them, as we consider that this is where their communication happens. In this context, latency and bandwidth have to be provided for all cache, memory, and network levels. Figure 3.1 illustrates how communication latency and bandwidth are represented for different topology levels on one socket of a machine named Xeon24, where Xeon represents its processor model and 24 is the number of PUs in a compute node. More information about this machine can be found in Section 6.1.1. The rectangles represent PUs and the three levels of cache present in a socket, while the arrows indicate at which topology level communication happens.

We obtain the communication costs of a parallel platform by profiling all levels of the machine topology. The mechanisms involved in this process are explained next.

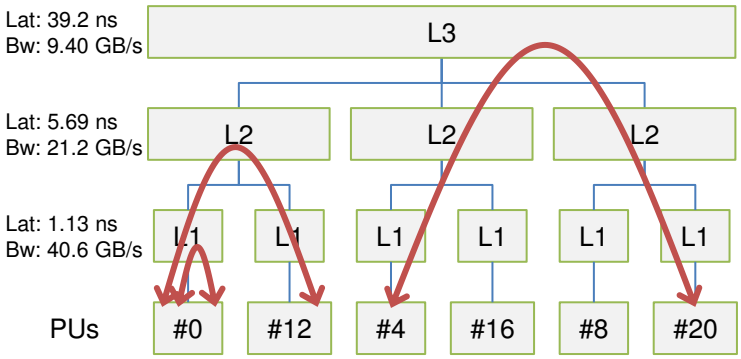

Figure 3.1: Machine topology representation of one socket of machine Xeon24. Different levels of the cache hierarchy present different latencies and bandwidths.

3.2 Topology modeling

To model the topology and provide this knowledge concisely and in feasible time, some assumptions regarding the machine have to be made. We list the main assumptions of our model below. These assumptions are in pair with current and popular technologies.

- All PUs have the same cache hierarchy and cache sizes. In other words, the cache hierarchy is homogeneous, as is the usual case in current platforms. This assumption helps to reduce the benchmarking time by evaluating the cache hierarchy for one PU instead of all PUs. This can easily be changed, but would incur in a longer profiling time due to the identification of different cache hierarchies and their respective benchmarking.

- There is no combination of different kinds of memory in the system, such as volatile and non-volatile random-access memories, or caches and scratchpads. The present model could be extended to include these characteristics if tools are made available to identify and control memory allocation.

- The benchmarks do not explicitly consider contention. We assume that the platform is dedicated to the application, which removes contention from consolidation, and contention from the parallel application alone can be modeled in the algorithms that use our model. Both are currently out of the current scope of our research.

- Communication costs are optimistic, as they represent the best latency and bandwidth achievable for a topology level by a single PU. Measurements are made sequentially and require the platform to be dedicated for them.

We opted to use well-known benchmarks and tools as a mean to maintain our model generic. Still, there is no common mechanism to measure latency and bandwidth at both memory and network levels. For this reason, we explain how we model the communication costs for each of these levels in the next sections.

3.2.1 Memory hierarchy of multicore platforms

The memory characteristics of a parallel platform are profiled using two portable microbenchmarks from the LMBENCH suite (STAELIN, 1996; LMBENCH, 2013). We use

`lat_mem_rd` to measure memory read latency and `bw_mem` for memory read bandwidth. These benchmarks run multiples times to gather statistical measurements.

`lat_mem_rd` measures memory read latency using a technique known as pointer chasing, where each memory access reads the address of the next position to access. A stride of the size of the cache line is used to force each memory read to mitigate prefetching effects.

`bw_mem` tests memory read bandwidth by accessing a contiguous vector of integers with a stride of the size of four integers. While STREAM (STAELIN, 1996, ap. (MCCALPIN, 1995)) is a well-known bandwidth benchmark, its focus resides in the main memory of the machine. It did not show much difference between cache levels and partially avoided caching in some experiences.

To measure these metrics for the different levels of the memory hierarchy, data used by the benchmarks is required to: (i) fit inside the level of interest; and (ii) be big enough to avoid being stored in a cache level closer to the PU. Our approach is based on measuring latency and bandwidth for one data size only for each level of the memory hierarchy. Such data size represents the middle point in logarithm scale between the sizes of two levels in the memory hierarchy. In the context of this thesis, these points are called **midpoints**. Figure 3.2 illustrates the midpoints for a machine with a memory hierarchy containing three cache levels. The horizontal axis represents in logarithmic scale the size in bytes of the different memories: 32 KB of L1 cache, 2 MB of L2 cache, 32 MB of L3 cache, and 2 GB of main memory. The size of a cache line is considered to be 512 bytes.

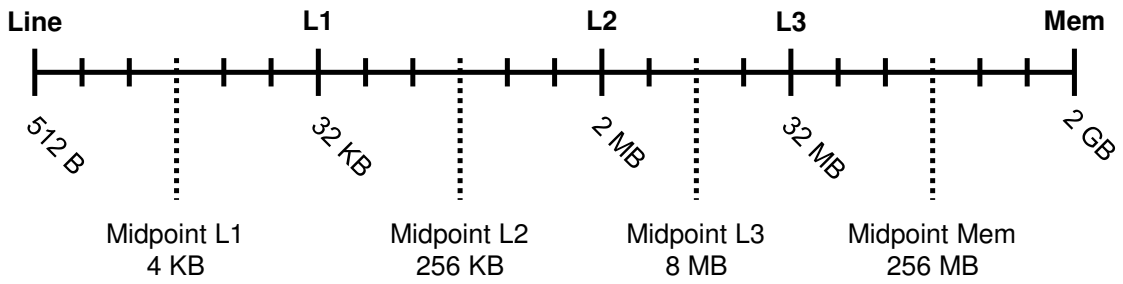

Figure 3.2: Midpoints for a memory hierarchy including the main memory and three cache levels. Midpoints represent the middle point in logarithm scale between the sizes of two levels.

By measuring these metrics for the midpoints only, the overhead of benchmarking the machine topology is significantly reduced when compared to measuring several data sizes to compute statistics for each topology level. We compare our approach to using the mean and median latencies in Section 3.3.2.

In the case of nonuniform and asymmetric topology levels, benchmarking is done for each pair involved. We call this approach **permutation-based**. For instance, when considering a NUMA compute node, the measurements involve keeping the benchmark's thread in the first PU of a NUMA node while its memory resides in another NUMA node. This process is facilitated by HWLOC, as it provides the ability to bind threads and memory in the machine topology. As usually no information is available regarding the way NUMA nodes are physically interconnected, all combinations of NUMA nodes are benchmarked. This is done sequentially to avoid having measurements interfering with each other.

Testing all permutations of NUMA nodes results in $O(m^2)$ latency and bandwidth measurements for m NUMA nodes. As this number is usually small for current compute

nodes, benchmarking time tends not to be a problem. Nevertheless, we provide a different alternative for the scenario where m is large. Instead of measuring latency and bandwidth for all combinations of NUMA nodes, we use the distance matrix available in HWLOC to guide the benchmarking by measuring the communication costs only once for each distance value. We call this approach **distance-based**. This involves a constant number of measurements instead of a quadratic one. For instance, in a machine where distances are represented as 1 for the local memory and 2 for the other NUMA nodes, communication costs would be measured only twice, and the obtained latencies and bandwidths would be replicated for the different NUMA node combinations. We provide a precision comparison of the two approaches in Section 3.3.3.

The procedure used to benchmark the network topology is similar to the one used to measure nonuniformity inside a compute node, but instead of considering NUMA nodes, pairs of compute nodes are evaluated. This is presented next.

3.2.2 HPC network topology

The benchmarks used for measuring parameters in the network level are different from the ones used in a memory level. This happens due to differences in the basic mechanism used for communication. Passing a message in shared memory requires allocating space to store the message's content and sending a pointer containing this memory position, while communication in distributed memory involves sending the message through the network to be read by the receiver.

We gather network statistics using a ping-pong benchmark written with coNCePTuaL (PAKIN, 2007; CONCEPTUAL, 2013). As discussed in Section 2.4.2, coNCePTuaL is a domain-specific language to write network benchmarks that focuses on portability, readability, and reproducibility. The abstract code used to describe our ping-pong benchmark is illustrated in Code 3.1. The benchmark returns the round-trip times (RTT) between all pair of compute nodes (line 26) for different sizes of messages. This process is sequential, as is the measurement done for nonuniform memory levels. An advantage of such approach is that it is able to profile the network without previous knowledge of its topology.

The use of the round-trip time is related to the absence of a global clock to measure the time of a single message. Different message sizes are benchmarked so that a simple linear regression (Equation 3.1) can be applied to decompose the time in latency and bandwidth. Equation 3.2 represents this process, where $time$ represents the RTT for a size of message $bytes$. Latency is considered two times because two messages are involved in the measurement. However, bandwidth is considered only once because the return message has a null size, which results in a smaller influence in the overall measurement. This enables the capture of asymmetric bandwidths.

$$y = \alpha + \beta x \tag{3.1}$$

$$time = 2 \times latency + \frac{bytes}{bandwidth} \tag{3.2}$$

For the linear regressions, message sizes are split into two groups: small messages and big messages. Small messages are more strongly influenced by communication latency, while big messages depend on the bandwidth. By using small messages to compute latency and big messages for bandwidth, we are able to obtain more precise results. By combining this information with the measurements done at cache and memory levels, we

are able to provide a complete view of the machine topology with unified communication costs.

Code 3.1: Ping-pong benchmark description with coNCePTuaL.

```
1  #ping-pong latency test written in coNCePTuaL
2  Require language version "1.4".
3  # Parse the command line.
4  reps is "Number of repetitions of each message size" and
      comes from
5   "--reps" or "-r" with default 1000.
6  maxbytes is "Maximum number of bytes to transmit" and comes
      from
7   "--maxbytes" or "-m" with default 1M.
8  warm is "Number of repetitions for warmup" and comes from
9   "--warmup" or "-w" with default 50
10 # Ensure that we have a peer with whom to communicate.
11 Assert that "the ping-pong test requires at least two tasks
      " with num_tasks>=2.
12
13 # Perform the benchmark.
14 For each sender in {0, ..., 1000} {
15   for each receiver in {neighbor for each neighbor in {0,
        ..., 1000} where sender <> neighbor} {
16     for each msgsize in {512, 1024, 2048, ..., maxbytes} {
17       task sender resets its counters then
18       for reps repetitions plus warm warmup repetitions {
19         task sender sends a msgsize bytes message to task
              receiver then
20         task receiver sends a 0 byte message to task sender
21       } then
22         if receiver < num_tasks then
23         task sender logs
24                     receiver as "Neighbor" and
25                     msgsize as "Bytes" and
26                     elapsed_usecs as "Total time (usecs)"
27     }
28   }
29 }
```

3.3 Topology model evaluation

We evaluate different aspects and decisions of our machine topology model in this section. Firstly, we show how the achievable communication latency and bandwidth depend on the topology level being used, and justify the use of both metrics to represent the communication costs of the machine topology. Secondly, we discuss the impact of using midpoints to measure communication costs. Lastly, we compare different approaches to identify nonuniformity and asymmetry at a memory level. All machines used in this evaluation are detailed in Section 6.1.1.

3.3.1 Latency and bandwidth measurements

To better understand how the communication costs of a platform vary among topology levels, different parallel platforms have been evaluated. Nonetheless, they all presented the same behavior. For this reason, we will discuss the results obtained with only one parallel platform.

Latencies measured with `lat_mem_rd` on the parallel platform named Opt48 are illustrated in Figure 3.3. The vertical and horizontal axes represent latency and the size of data used in the benchmark, respectively. The three labels in the graph indicate the size of each cache level. For each level of the memory hierarchy, 50 data sizes were used for profiling. These follow an arithmetic progression inside each level. For this experiment, the benchmark runs 50 repetitions for each data size. Different numbers of data sizes and repetitions were also evaluated, but showed similar results. As the horizontal axis is represented in a logarithmic scale, there are more data points near the end of each memory level than at the beginning.

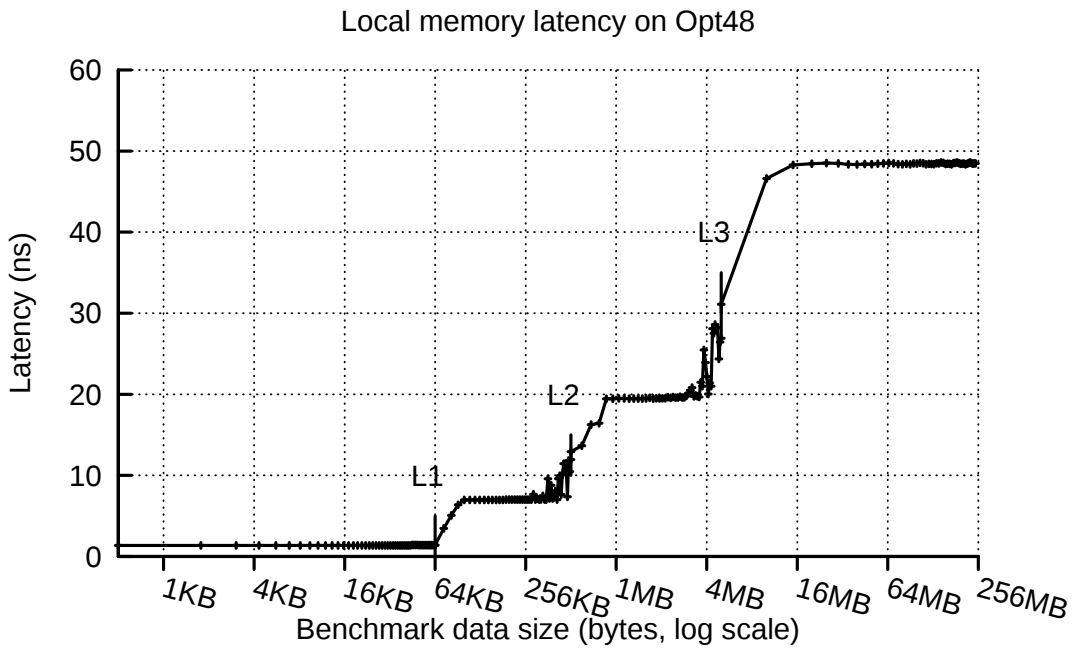

Figure 3.3: Memory latency measured with `lat_mem_rd` for different data sizes on Opt48.

An area of stability can be seen for each level of the memory hierarchy in Figure 3.3. However, the edges of each memory level tend to show some disturbance. Latencies are smaller when data is slightly bigger than the preceding cache level because that cache is still able to accelerate part of the memory reads. Similarly, latencies are bigger when data approaches the limit of a cache level, as conflicts and other processes from the operating system can reduce the effectiveness of a cache level. These extremities could be prejudicial to our topology model, as they add noise to the measurement process.

The same stability behavior seen in Figure 3.3 can be seen in the bandwidth profile obtained with `bw_mem` and shown in Figure 3.4. These results follow the same method discussed before. In this case, the vertical axis represents the memory bandwidth achieved for the different data sizes.

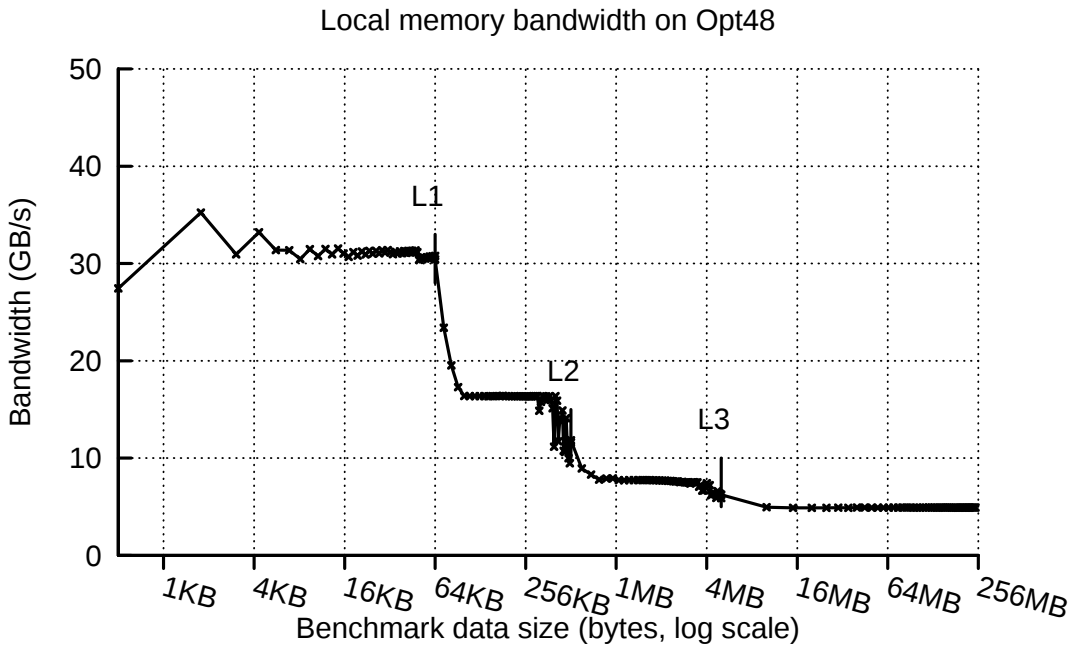

Figure 3.4: Memory bandwidth measured with `bw_mem` for different data sizes on Opt48.

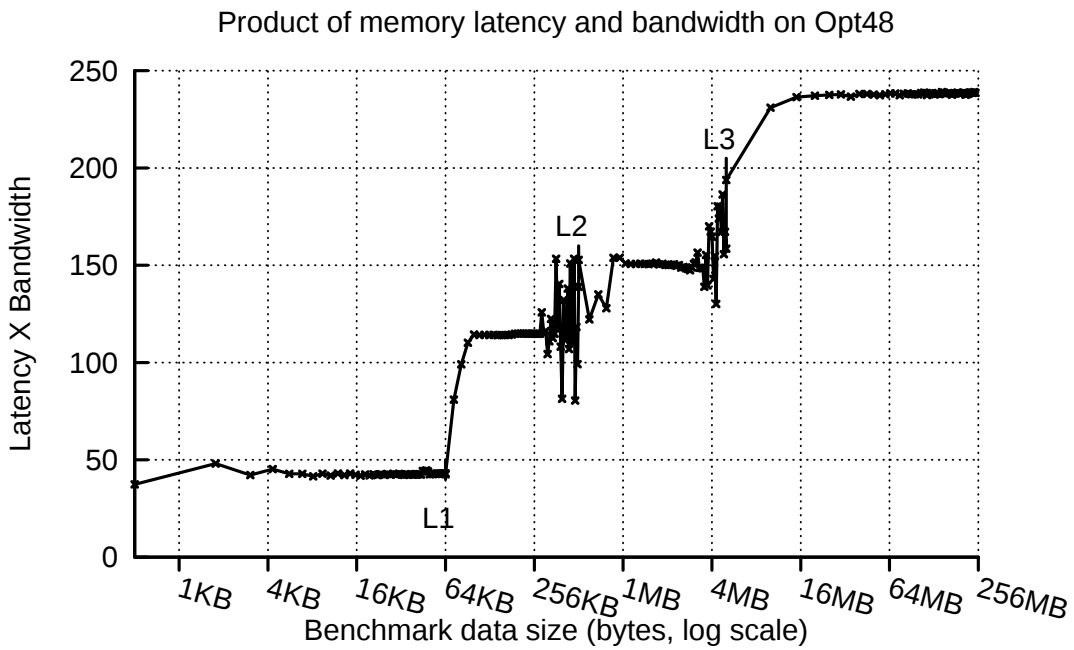

Figure 3.5: Product between memory latency and bandwidth for machine Opt48.

While latency increases as we move farther from the processing unit, bandwidth decreases. However, these changes do not happen at the same order of magnitude. We see an increase in latency from L1 to the memory node at the scale of hundreds, while bandwidth decreases in order of tens. This indicates that these two metrics are in part disconnected. To better visualize this, Figure 3.5 shows the product of latencies and bandwidths obtained for different data sizes. The unity of the vertical axis is of no meaning in this case. The product value increases with data size, which indicates that latency is not inversely

proportional to bandwidth, as it increases faster than bandwidth decreases. The strong variance seen around the L2 label at 512 KB comes from same reasons explained before. They affect slightly different data sizes for each benchmark, which results in those spikes.

These results show the significance and difference of latency and bandwidth to model the communication costs. They also stress how noisy measurements can be when they are close to the limits of different memory hierarchy levels. We discuss how we overcome these noises next.

3.3.2 Use of midpoints

Our approach to measure latency and bandwidth at different memory topology levels is based on the use of midpoints, which represent the middle point in logarithm scale between the sizes of two levels in the memory hierarchy. To better understand how this affects the measured values, we compare the use of midpoints to other statistical tools, namely the arithmetical mean and median.

As Figures 3.3 and 3.4 illustrate, latency and bandwidth measurements show variations near the limits of each memory level. This affects statistics as the difference between each level is not symmetric. To better display this, Table 3.1 represents three different approaches to define the latencies of the memory hierarchy on three different machines named Xeon24, Xeon32, and Opt48. All approaches execute the benchmark 50 times for each data size.

Table 3.1: Memory latency (ns) as measured by `lat_mem_rd` on different machines.

	Xeon24			
	L1	L2	L3	Memory node
Midpoint	1.133	5.694	39.20	185.8
Mean	1.132	9.416	57.24	183.9
(% change vs. midpoint)	(0.1%)	65%	46%	(1.0%)
Median	1.133	5.917	46.84	185.3
(% change vs. midpoint)	0.0%	3.9%	19%	(0.3%)
	Xeon32			
	L1	L2	L3	Memory node
Midpoint	1.791	4.480	20.90	118.2
Mean	1.786	7.197	20.70	116.0
(% change vs. midpoint)	(0.3%)	60%	(1.0%)	(1.8%)
Median	1.790	4.563	20.93	119.0
(% change vs. midpoint)	0.0%	1.8%	0.1%	0.7%
	Opt48			
	L1	L2	L3	Memory node
Midpoint	1.364	6.973	19.46	48.58
Mean	1.377	7.821	21.21	48.41
(% change vs. midpoint)	1.0%	12%	9.0%	(0.4%)
Median	1.364	7.022	19.66	48.44
(% change vs. midpoint)	0.0%	0.7%	1.1%	(0.3%)

The difference between latencies computed with the midpoint approach and the arithmetic mean goes up to 65%, as seen on the L2 cache latency of Xeon24. Meanwhile, the difference between use of midpoints and the median stays under 4%, with the only exception being the L3 cache latency on Xeon24. These results indicate that midpoints provide stable means to measure the communication costs of different topology levels. Its main advantage when compared to the median is that it requires only one run for each level of the memory hierarchy, while the median requires running benchmarks with several data sizes, which results in a larger benchmarking time.

3.3.3 Nonuniformity measurement at memory level

As discussed in Section 3.2.1, we provide two different ways to capture nonuniform communication costs at a memory level: a permutation-based approach, which benchmarks all possible pairs of NUMA nodes; and a distance-based approach, which uses the distance matrix made available by the BIOS and HWLOC to guide which pairs should be benchmarked. The first approach provides more detailed information, while the second has a benchmarking time much smaller than the first one. In this section, we compare both techniques to other information available in parallel compute nodes.

We focus our experiments on Xeon192, an SGI UV2000 machine composed of 24 NUMA nodes, where each NUMA node contains 8 PUs. This parallel platform provides an interesting environment for our experiments due to its substantial number of NUMA nodes inside a single compute node, and to their interconnection, as NUMA nodes in an SGI UV2000 machine are interconnected through a proprietary fabric named NUMAlink 6, or NL 6 (SGI UV 2000 System User Guide, 2012).

Figure 3.6 illustrates how the first eight NUMA nodes of Xeon192 are interconnected among themselves and with other NUMA nodes. Each circle represents a NUMA node, while each arrow represents a NL 6 interconnection. Dashed arrows represent interconnection using Intel QuickPath Interconnect (QPI) (An Introduction to the Intel QuickPath Interconnect, 2009). This information is collected from a topology file made available with the operating system on this platform.

As it can be seen in Figure 3.6, Xeon192's nodes form pairs interconnected through QPI, such as nodes 0 and 1, or 4 and 5. Additionally, each NUMA node has two NL 6 connections with one other node (e.g., nodes 4 and 6). Based only on this view of the platform, one could expect to see different communication performances between node 0 and nodes 8, 2, or 3, as the first is connected to node 0 by only one NL 6, the second is connected by two NL 6, and the third would require data to go through QPI and a double NL 6 interconnect.

Another vision of the same eight NUMA nodes of Xeon192 is shown in Figure 3.7. This representation is based on information made available by SGI on manuals (SGI UV 2000 System User Guide, 2012). In this figure, each square represents a processor node. A processor node consists of two NUMA nodes interconnected by QPI to one application-specific integrated circuit (ASIC) named HARP. This ASIC is connected to the NUMAlink interconnect fabric through different NL 6 ports. Each pair of NL 6 interconnections is represented by an arrow in Figure 3.7.

These two representations of Xeon192 provide different expectations of the communication performance at memory level. To better demonstrate that, Table 3.2 presents estimations of the communication costs achievable by node 0 accessing data on various NUMA nodes. The first line of the table depicts the minimum number of hops data would have to traverse to be accessed by a PU on node 0 using the machine representation of

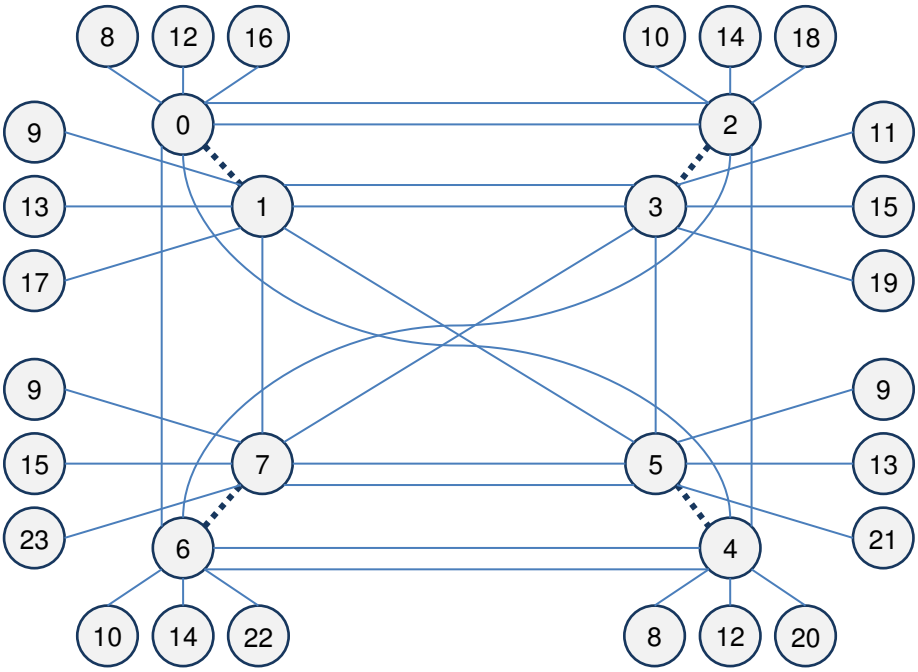

Figure 3.6: Interconnections of the first eight NUMA nodes of machine Xeon192 based on topology file.

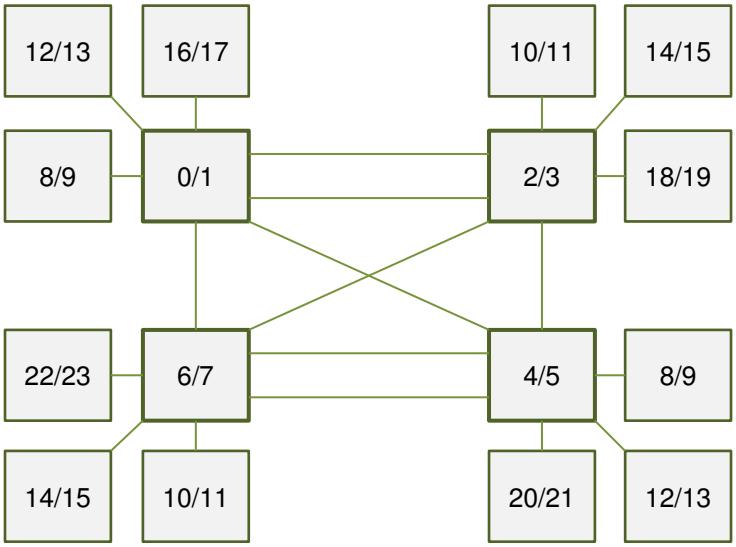

Figure 3.7: Interconnections of the first eight NUMA nodes of machine Xeon192 based on the machine specification of processor nodes.

Figure 3.6, while the second line is based in Figure 3.7. Although these approaches provide the same estimation for several combinations of NUMA nodes, they disagree on 28% of the cases (160 of 576) for this machine.

As previously discussed, a different approach to characterize nonuniformity at memory level is to use the distance matrix made available by the BIOS and accessible using HWLOC. The distances provided by this matrix for node 0 on Xeon192 are presented on the third line of Table 3.2. Four different distance levels can be seen: a distance 1 for the node itself; a distance 5 for the other NUMA node inside the same processor node;

Table 3.2: Comparison of the different representations of communication distance from NUMA node 0 on Xeon192.

Method	NUMA nodes								
	0	1	2	3	8	9	10	16	18
# of hops (NL 6 and QPI)	0	1	1	2	1	2	2	1	2
# of hops (Processor nodes)	0	0	1	1	1	1	2	1	2
Distances (BIOS/HWLOC)	1	5	6.5	6.5	6.5	6.5	7.9	6.5	7.9
Latency (ns)	71	457	573	572	572	576	682	575	696
(remote/local)	1.0	6.4	8.1	8.1	8.1	8.1	9.6	8.1	9.8
Bandwidth (GB/s)	10.5	2.1	1.7	1.7	1.7	1.7	1.4	1.7	1.4
(local/remote)	1.0	5.0	6.2	6.2	6.2	6.2	7.5	6.2	7.5

a distance 6.5 for NUMA nodes in neighboring processor nodes; and a distance 7.9 for NUMA nodes at a distance of two hops. It is important to notice that this does not differentiate between NUMA nodes (or processor nodes) connected by more NL 6 ports. Still, none of these techniques can detect asymmetries.

The last four lines of Table 3.2 present the latencies and bandwidths measured using our permutation-based approach. For each permutation, each benchmark is executed 5 times to warmup the memory, and 30 times to gather statistical data. For each of the two metrics, a factor comparing local and remote communication costs is also provided. These measured communication costs behave similarly to the distances matrix, as four different levels can be seen. Results indicate that the distances matrix is based on bandwidth factor on Xeon192, as they differ from one another by less than 5%. However, the differences in latency between local and remote communications are 25% larger than the aforementioned distances. In this context, using the distance matrix would result in a underestimation of the real communication costs among NUMA nodes. Overestimation can also happen. For instance, on a machine such as Xeon32 (see Chapter 6.1.1), distances are 40% to 50% greater than the actual communication cost differences.

Using the permutation-based approach to measure latencies and bandwidths, we are also able to reveal asymmetries at a memory level. The main limitation of this approach lies in its execution time. For Xeon192, which is composed of 24 NUMA nodes, the memory hierarchy profiling takes approximately 500 minutes, which can be prohibitive. In this same machine, our distance-based approach takes around 2 minutes only. Even though some accuracy and details are lost, this technique showed an average difference of 1.1% to its permutation-based counterpart, with a maximum difference of 2.8%. In addition, all communication costs measurements involved in our machine topology model showed differences between runs of 1% at most.

All these results involving the use of both latency and bandwidth to represent communication costs, the use of midpoints to decrease profiling time, and the two techniques to measure nonuniformity at memory levels justify some of decisions taken in our machine topology model. We discuss next how we make our machine topology available to task mapping algorithms, so they can benefit from this precise information.

3.4 Machine topology library

Our machine topology model is made available to task mapping algorithms as a library named HIESCHELLA (HIESCHELLA, 2013). HIESCHELLA represents a machine topology as $\mathcal{O}_{\text{HieSchella}} = (\mathcal{P}, \mathcal{L}, S, \{C_{\text{lat}}, C_{\text{band}}\})$. It uses the topology tree abstraction provided by HWLOC to organize PUs and topology levels, and augments it with the communication costs of the platform as latencies and bandwidths.

This section is organized as follows: Firstly, we present HIESCHELLA's tools used to profile the machine topology and organize its communication costs. Secondly, we explain how the topology model is stored for previous use by task mapping algorithms. Lastly, we show how HIESCHELLA interfaces its machine topology model with other algorithms.

3.4.1 Tools

HIESCHELLA includes three tools that use its library. They are `memory_profile`, `network_extension`, and `read_topology`.

The memory hierarchy of a multicore platform is modeled by **`memory_profile`**. It starts by gathering the machine topology tree and computing the midpoints for the different memory levels. It continues by measuring the memory read latencies and bandwidths with benchmarks `lat_mem_rd` and `bw_mem`, respectively. The machine topology model including the communication costs is later stored on an Extensible Markup Language (XML) file. `memory_profile` can profile nonuniform memories using both permutation-based and distance-based techniques. It is important to emphasize that `memory_profile` has to be run only once to model the machine topology of a compute node.

When a parallel platform is composed of multiple CNs, the interconnection network topology is modeled by **`network_extension`**. It uses as input the machine topology model of a single CN and the parsed results of our network benchmark (see Section 3.2.2), and provides as output a complete model including both memory and network topology levels. This model is also stored on an XML file for posterior use by task mapping algorithms.

Our last tool, **`read_topology`**, is used to provide a simplified, human-readable view of the machine topology model. It includes the communication costs among all PUs. This information is read from the topology files generated by `memory_profile` or `network_extension`. The way this information is stored is explained next.

3.4.2 Data storage

HIESCHELLA extends HWLOC's XML format to represent and store the communication costs benchmarked in the machine topology. As our model intends to represent the communication costs among PUs as a function $C : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$, a simple approach to store this information would be to use matrices of size n^2 for a number of PUs n, where each cell represents the cost between two PUs. This would provide a low cost to build, and fast access to data. Nonetheless, we chose to store data in a tree structure and profit from HWLOC's topology representation. This approach has two main advantages over using matrices: (i) it is closer to the hierarchical structure of the machine; and (ii) it has better scalability, as the memory footprint of a tree is $O(n)$, while a matrix would take $O(n^2)$.

In the case of nonuniform and asymmetric costs, like the ones between two different NUMA nodes, a matrix is stored in their common ancestor in the machine topology (the

first level of the topology that is shared among those components). This lets us express the different communication costs when necessary while avoiding the use of a whole n^2 matrix.

3.4.3 Library interface

HIESCHELLA implements a dynamic library to be linked to task mapping algorithms and our tools. It provides an interface to a set of functions written using the programming language C.

A first function is used to load the machine topology model from an XML file. It includes a custom callback used to read the communication costs from said file, as this information is not standard to HWLOC. The result of this function call is a `hwloc_topology_t` object that is used by all other HIESCHELLA functions. Functions are also provided to write the communication costs of a topology level in the model. They are exclusively used by `memory_profile` and `network_extension`.

The most important functions provided by HIESCHELLA are the ones used to obtain the communication costs between two PUs. Several functions are made available for that use, such as `get_latency`, `get_bandwidth`, and `get_comm`. They take as input the previously loaded topology object and the identifiers of the two PUs of interest. With this simplified interface, our library is able to provide a detailed topology model for task mapping algorithms. We present how our topology-aware load balancing algorithms are able to profit from this model in the next chapter.

4 PROPOSED LOAD BALANCING ALGORITHMS

The main hypothesis of our research revolves around the idea that precise machine topology information should be considered by task mapping algorithms as it can influence application performance. For instance, the communication time between application tasks can vary according to the processing units that they are mapped due to the nonuniform and asymmetric communication costs present on different topology levels. To help task mapping algorithms overcome these issues, we provide them with a detailed machine topology model, as discussed in the previous chapter. Still, the availability of this information does not make task mapping trivial.

Scheduling theory shows that the problem of finding an optimal task distribution for a parallel program is NP-hard (LEUNG, 2004). Since an optimal solution cannot be found in feasible time, the development of task mapping algorithms is strongly based on the use of heuristics. To improve the performance of an application, techniques involve mitigating load imbalance, and reducing communication costs, as discussed in Section 2.3.1. Still, to achieve performance portability, a scheduling algorithm needs to map tasks to different platforms and achieve a low core idleness, which also depends of the task mapping algorithm itself not being a performance problem.

In this context, we propose novel topology-aware load balancing algorithms. To keep our approach generic, we decouple it from a specific application or platform by doing load balancing at a middleware level. Application information is captured during runtime, while the machine topology is independently provided by our library. The load balancing technique was chosen as it has been attested to be appropriate for both load imbalance and costly communication problems of applications with dynamic and irregular behaviors, which is further discussed in Section 5.1.

Our load balancing algorithms are introduced in the following order: First, we present our centralized algorithms, which are suitable for parallel platforms composed of few CNs. They are named **NucoLB** (PILLA et al., 2012) and **HwTopoLB** (PILLA et al., 2012, 2014). After that, we present HierarchicalLB, which involves the hierarchical composition of centralized algorithms, and is more suitable for larger parallel platforms. We end this chapter with a discussion on the implementation of these algorithms, and a comparison of their main characteristics and contributions.

4.1 Centralized algorithms

A load balancing algorithm is considered to be centralized when it decides where each and every task of an application will be mapped. The main advantage of this approach lies in its complete view of the application and parallel platform. This involves control of the load distribution over all processing units, and over the communication through the

whole parallel machine, which allows algorithms to make better decisions.

This complete knowledge over the mapping of tasks comes with a price: increases in the size of the parallel platform or application can significantly impact the execution time of centralized load balancing algorithms, which will affect performance portability. In other words, their scalability may be limited. In this sense, the algorithm's run time has to be traded with its ability to quickly mitigate load imbalance and costly communications.

We present two different centralized load balancing algorithms in this section. The first algorithm is named NUCOLB and focuses on the nonuniform aspects of parallel platforms. Meanwhile, our second algorithm is named HWTOPOLB and considers the whole topology in its decisions. Their characteristics and differences are detailed next.

4.1.1 NUCOLB: Nonuniform communication costs load balancer

NUCOLB (PILLA et al., 2012) is a load balancing algorithm developed for parallel platforms involving nonuniform levels in their topologies. This includes topologies with one or more NUMA compute nodes. While it aims at mitigating load imbalance, it also tries to reduce costly communications by keeping communicating tasks in the same NUMA node. In this section, we discuss the rationale of NUCOLB, its models, and its algorithm.

4.1.1.1 Rationale

NUCOLB focuses on the NUMA nodes of the machine topology model. This is done for two main reasons: Firstly, this helps to emphasize the nonuniform communication costs present in the platform, which have a large impact in the communication performance of the application. Secondly, this is used to reduce the execution time of the algorithm. This happens because NUCOLB assesses the communication time of a task at a NUMA node level instead of a PU level, which results in the communication time having to be computed only once for all PUs in the same NUMA node.

Its heuristic works like a classical list scheduling algorithm (LEUNG, 2004), where tasks or jobs are rescheduled from a priority list and assigned to less loaded processing units in a greedy manner. The priority list is dynamically computed and is based on the load of the tasks. This design involves evaluating a new mapping for all tasks at each load balancing call, which enables a quick mitigation of load imbalance and costly communications by NUCOLB. This is specially useful when dealing with application dynamism. Additionally, list scheduling algorithms are considered to be efficient, as they typically run in polynomial time, and provide good results in practice. Finally, 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.

Our algorithm is also refinement-based, which means that it considers the current task mapping on its decisions. Refinement-based algorithms migrate less tasks than algorithms that do not consider the current task mapping, which helps them to decrease their task migration overhead.

4.1.1.2 Application and platform modeling

NUCOLB models the application as $\mathcal{A}_{\text{NucoLB}} = (\mathcal{T}, Load, Size_0, Msgs, Bytes_0)$. In other words, it considers the current existing tasks, their loads, and their communication graph based only on the number of messages exchanged. The communication graph is used to compute the communication cost of mapping a task to a certain NUMA node, as

will be discussed later in this section.

The machine topology is seen as $\mathcal{O}_{\text{NucoLB}} = (\mathcal{P}, \mathcal{L}, S, C_{\text{lat}})$, with $\mathcal{P}$ the set of PUs, $\mathcal{L}$ the set of levels of the topology, and the first topology level shared by two PUs as a function $S : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{L}$. The function C_{lat} represents the communication latency in the machine topology at memory and network levels. This latency is used to compute the NUCO FACTOR, which acts as the difference between remote and local communication latencies. The NUCO FACTOR is represented by a function $F : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{R}_{\geq 1}$. The NUCO FACTOR F for two PUs p and q is presented in Equation 4.1. In this scenario, $C_{\text{lat}}(p, p)$ represents the latency that p has to access its local memory.

$$F(p, q) = \frac{C_{\text{lat}}(p, q)}{C_{\text{lat}}(p, p)} \tag{4.1}$$

The algorithm regards communication between NUMA nodes as a source of more costly communications than communication inside a NUMA node. To illustrate that, consider the set of NUMA nodes $\mathcal{N}$ and a function that maps PUs to NUMA nodes as $node : \mathcal{P} \rightarrow \mathcal{N}$. Two PUS p and q are said to reside in the same NUMA node if their NUCO FACTOR is equal to 1, as depicted in Equation 4.2.

$$node(p) = node(q) \iff F(p, q) = 1 \tag{4.2}$$

4.1.1.3 Algorithm

The main idea of this load balancing algorithm is to iteratively map tasks in decreasing order of $Load$, assigning each of them to the PU that offers the least overhead to its execution. In order to decide if a task t should be scheduled on PU p to balance load, the algorithm considers information about the execution of the application, such as the current load on PU p as $LoadPU(M, p)$, the amount of communication among task t and other tasks currently assigned to PUs in the same NUMA node than p (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 PU p is estimated as a function $cost : M \times \mathcal{P} \times \mathcal{T} \rightarrow \mathbb{R}$ using Equation 4.3.

$$cost(M, p, t) = LoadPU(M, p) + \alpha \times (remoteCost(M, p, t) - localCost(M, p, t)) \tag{4.3}$$

Equation 4.3 presents a weighted linear sum of the costs involved in the execution of task t on PU p. In this equation, all communication costs are normalized by a factor α that controls the weight that they have over the execution time. $remoteCost$ represents the communication costs of remote messages, as presented in Equation 4.4. Remote communications are multiplied by the NUCO FACTOR between the involved nodes. This way, the greater the difference between local and remote latencies, the greater the remote communication cost. Meanwhile, $localCost$ represents local communication, as given in Equation 4.5. Local communications are subtracted from the total cost. This leads to smaller costs for NUMA nodes that have many local message exchanges.

$$remoteCost(M, p, t) = \sum_{u \in \mathcal{T} \wedge node(p) \neq node(M(u))} (Msgs(u, t) \times F(p, node(M(u)))) \tag{4.4}$$

$$localCost(M, p, t) = \sum_{u \in \mathcal{T} \wedge node(p) = node(M(u))} Msgs(u, t) \tag{4.5}$$

NUCOLB's algorithm bases its mapping decisions in the aforementioned functions and equations. NUCOLB involves a refinement-based greedy list scheduling algorithm that assigns the task with the largest load to the PU that presents the smallest cost. The pseudocode for NUCOLB is presented on Algorithm 1.

Algorithm 1: NUCOLB's algorithm.

Input: Topology $\mathcal{O}$, Application $\mathcal{A}$, Mapping M
Output: Mapping M'

1 $M' \leftarrow M$
2 $\mathcal{T}' \leftarrow \mathcal{T}$
3 **while** $\mathcal{T}' \neq \emptyset$ **do**
4 $\quad t \leftarrow \underset{u \in \mathcal{T}'}{\text{argmax}}\ Load(u)$
5 $\quad \mathcal{T}' \leftarrow \mathcal{T}' \setminus \{t\}$
6 $\quad p \leftarrow M'(t)$
7 $\quad LoadPU(M', p) \leftarrow LoadPU(M', p) - Load(t)$
8 $\quad q \leftarrow \underset{r \in \mathcal{P}}{\text{argmin}}\ cost(M', r, t)$
9 $\quad LoadPU(M', q) \leftarrow LoadPU(M', q) + Load(t)$
10 $\quad M'(t) \leftarrow q$

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 PU p. This information is obtained from the initial mapping M. The load of this task is subtracted from the load of p before evaluating the cost function. This subtraction is an important characteristic of this algorithm, as it avoids unnecessary migrations by reducing the load of p. This is important because the overhead resulting from migrations can only be estimated with information about the size of the task in memory, which NUCOLB does not consider ($Bytes_0$). After this subtraction, the algorithm evaluates all possible mappings for t and chooses the core q that presents the smallest cost. Once this choice is made, the load of task t is added to the load of q. The algorithm continues by searching for a new mapping for the next task.

4.1.1.4 Algorithm properties

Considering $n = \text{card}(\mathcal{T})$ and $m = \text{card}(\mathcal{P})$, NUCOLB presents a complexity of $O(n^2 m \log m)$ 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 $O(nm \log m)$ for $\log n < m \log m$, or $O(n \log n)$ otherwise (due to sorting). The $\log m$ factor is related to the description of the machine topology as a tree, as provided by our library.

A characteristic of NUCOLB is that it will evaluate all possible mappings for a task even in a situation where load is balanced and communications are optimized, which results in a constant evaluation overhead. This issue is approached in a different way by HWTOPOLB.

4.1.2 HwTopoLB: Hardware topology load balancer

HwTopoLB (PILLA et al., 2012, 2014) aims at improving performance by decreasing the costs of communication through the whole machine topology, and by ensuring that no PU will be underutilized due to load imbalance. In addition, this algorithm is proved to asymptotically converge to an optimal solution, as we demonstrate in Theorem 1.

4.1.2.1 Rationale

HwTopoLB considers all levels of the machine topology model in its decisions. Although this approach requires more time to compute the communication costs of a task mapping, it provides a more accurate estimation.

In order to maintain a small load balancing time, specially when the current task mapping shows low core idleness, we designed HwTopoLB in a way to usually avoid computing a new mapping for most tasks. Instead of ordering all tasks and evaluating their migrations like NucoLB, HwTopoLB prioritizes the most loaded PUs and stops after deciding that a task should not be migrated from its current PU (more details are provided later in this section). This prioritization also helps to handle the greatest sources of PU idleness first, which hastens the convergence to a better task mapping.

Some design decisions are linked to requirements for the algorithm to asymptotically converge to an optimal solution. The first one is that the algorithm is refinement-based, which also has the advantage of decreasing the task migration overhead. The second one is that decisions related to choosing a task for evaluation and choosing a PU to map it are probabilistic. This is required to escape local optimum mappings.

4.1.2.2 Application and platform modeling

HwTopoLB models the machine topology as $\mathcal{O}_{\text{HwTopoLB}} = (\mathcal{P}, \mathcal{L}, S, \{C_{\text{lat}}, C_{\text{band}}\})$. The functions C_{lat} and C_{band} represent the communication latency and bandwidth, respectively, in the machine topology. This includes cache levels, the memory level, and the network level when considering more than one CN. These metrics are captured by benchmarking the machine topology, and made available by our library, as discussed in Chapter 3.

The application is modeled as $\mathcal{A}_{\text{HwTopoLB}} = (\mathcal{T}, Load, Size_0, Msgs, Bytes)$. Although it would be of interest to consider the size of the tasks to better predict their migration overhead, this is not done due to an implementation limitation. This is discussed in Section 4.3. The communication graph considers both the amount of messages exchanged and the volume of data communicated. They are used in conjunction with the latencies and bandwidths available in the machine topology model to measure communication costs.

4.1.2.3 Algorithm

HwTopoLB tries to find the best trade-off between mapping a task to a more underutilized PU and mapping a task closer to the other tasks it communicates with. In its context, the function to minimize is the total completion time of the tasks, also called the makespan, that we denote by $globalCost : (\mathcal{T} \rightarrow \mathcal{P}) \rightarrow \mathbb{R}$.

The total completion time can be expressed as the maximum completion time on each PU. Equation 4.6 expresses the total completion time based on $PUCost(M, p)$ as the completion time of PU p for mapping M. Similarly, the completion time on a PU is the sum of the time, denoted by $taskCost(M, t)$, needed to execute each task scheduled on

said PU. This is depicted in Equation 4.7.

$$globalCost(M) \stackrel{\text{def}}{=} \max_{p \in \mathcal{P}} PUCost(M, p) \tag{4.6}$$

$$PUCost(M, p) = \sum_{t \in \mathcal{T} \wedge M(t) = p} taskCost(M, t) \tag{4.7}$$

The time to execute a task depends on its load and its communication costs as represented in Equation 4.8. The computational load $Load(t)$ of task t can also be seen as equal to the size of the task over the speed of the PU. This is shown in a more simplified way as homogeneous PUs are being considered.

$$taskCost(M, t) = Load(t) + commCost(M, t) \tag{4.8}$$

The communication cost of a task, denoted by $commCost(M, t)$, is defined as the sum of (i) the number of messages received from different tasks, multiplied by the access latency to the nearest shared level in the machine topology; and (ii) the number of bytes received from different tasks, divided by the bandwidth of said topology level. This is depicted in Equation 4.9, where $Msgs(u, t)$ and $Bytes(u, t)$ represent the number of messages and bytes received by task t from task u, respectively.

$$\begin{aligned} commCost(M, t) = & \sum_{u \in \mathcal{T}} Msgs(u, t) \times C_{\text{lat}}(M(t), M(u)) \\ & + \sum_{u \in \mathcal{T}} \frac{Bytes(u, t)}{C_{\text{band}}(M(t), M(u))} \end{aligned} \tag{4.9}$$

The goal here is to design an algorithm that finds an optimal mapping, *i.e.* a mapping M such that $globalCost(M)$ is minimized. The proposed algorithm works as follows: at each of its iterations, the current mapping is modified in one component according to the three following steps.

- Firstly, we choose a PU. The PU with the highest load is selected with probability α (in practice the parameter α is close to one). Otherwise (with probability $1 - \alpha$), we select another PU uniformly.
- Secondly, we choose one task currently mapped to this PU. The task with the biggest load is chosen with probability β(again, $\beta$is large). Otherwise, the choice is uniform over all tasks on said PU.
- Finally, we move this task to a PU that minimizes the overall cost with a high probability.

For the last step, we chose to use a Gibbs distribution with temperature E. The Gibbs distribution with temperature $E > 0$ over the set of k real values $v_1 \ldots v_k$ is the probability vector on $\{1 \ldots k\}$:

$$\left(\frac{\exp(-v_i/E)}{\sum_{j=1}^{k} \exp(-v_j/E)} \right)_{i=1 \ldots k} \tag{4.10}$$

In our case, the values v are the costs associated to each PU. This distribution was chosen because it has two properties of interest: (i) the probability of choosing a PU with

minimal cost goes to one as the temperature goes to zero; and (ii) it is memoryless, which means, in our context, that the probability of choosing a new mapping depends only on the current mapping. This property, together with the property of the space of possible task mappings being discrete, allows our problem to be modeled as a Markov chain. This is important for the convergence proof provided in Section 4.1.2.4. Although other exponential distributions present the same properties, we chose the Gibbs distribution as it can be easily computed by our algorithm.

HWTOPOLB is designed as described in Algorithm 2. The parameters used are the temperature $E > 0$, the horizon h, $\alpha \in (0;1)$, and $\beta \in (0;1)$. We also use a function `RAND()` that represents a uniform pseudo-random generator of a real number in $(0;1)$. The horizon h defines the number of iterations of the algorithm, which plays an important role on Theorem 1. It is important to notice that, as discussed in Section 4.1.2.1, Algorithm 2 is implemented to stop after deciding that a task should not be migrated from its current PU, which means its value cannot be defined *a priori*.

Algorithm 2: HWTOPOLB's algorithm.

Input: Topology $\mathcal{O}$, Application $\mathcal{A}$, Mapping M_{init}
Output: Mapping M

1 **initialization**: $M \leftarrow M_{\text{init}}$, $i \leftarrow 0$
2 **while** $i < h$ **do**
  `/* Choice of a PU from which a task is selected */`
3 **if** `RAND()` $< \alpha$ **then**
4 Choose PU $\bar{p}$ uniformly in $\mathcal{P}^* \stackrel{\text{def}}{=} \underset{p \in \mathcal{P}}{\text{argmax}}\ PUCost(M, p)$
5 **else**
6 Choose PU $\bar{p}$ uniformly in $\mathcal{P} \backslash \mathcal{P}^*$
  `/* Choice of a task in` $\bar{p}$ `*/`
7 **if** `RAND()` $< \beta$ **then**
8 Choose task $\bar{t}$ uniformly in $\mathcal{T}^*_{\bar{p}} \stackrel{\text{def}}{=} \underset{t \in \mathcal{T} \wedge M(t) = \bar{p}}{\text{argmax}}\ taskCost(M, t)$
9 **else**
10 Choose task $\bar{t}$ uniformly in $\{t \in \mathcal{T} \wedge M(t) = \bar{p}\} \backslash \mathcal{T}^*_{\bar{p}}$
  `/* Choice of a PU to which the task` $\bar{t}$ `is moved */`
11 Choose a PU $\hat{p}$ according to a Gibbs distribution with temperature T over the set of values $(globalCost(M'_p))_{p \in \mathcal{P}}$, where $M'_p(t) = \begin{cases} p & \text{if } t = \bar{t} \\ M(t) & \text{otherwise} \end{cases}$
  `/* Mapping update */`
12 $M(\bar{t}) \leftarrow \hat{p}$
13 $i \leftarrow i + 1$
14 **return** M

4.1.2.4 *Algorithm properties*

The complexity of Algorithm 2 cannot be defined in terms of the number of tasks or PUs because it also depends on the horizon h. However, considering $n = \text{card}(\mathcal{T})$ and $m = \text{card}(\mathcal{P})$, an iteration of the algorithm (lines 2-13) presents a complexity

of $O(nm \log m)$ in the worst-case scenario involving all-to-all communications. However, for a sparse communication graph with a constant vertex degree, as it is usually the case, the complexity of an iteration is reduced to $O(m \log m)$. In this same scenario, the algorithm's initialization has a complexity of $O(n \log m)$.

By choosing a new mapping using the Gibbs distribution, the sequence of mappings computed in the successive iterations of the algorithm forms a random sequence that is an irreducible and aperiodic Markov chain over all possible mappings as long as $E > 0$, $0 < \alpha < 1$ and $0 < \beta < 1$. Therefore, as the number of iterates grows, its distribution converges to the unique stationary distribution of the chain. Although it is not possible to compute this limit distribution exactly, one can characterize its behavior as a function of E, as shown in the following theorem.

Theorem 1. *For all $\alpha, \beta \in (0; 1)$, if the temperature E is close to zero, then the mapping computed by* HWTOPOLB *is optimal with high probability when the horizon h grows to infinity, i.e. the probability can be rendered arbitrarily close to one by lowering the temperature.*

Before a proof of the optimality of HWTOPOLB is given, some remarks and comments about the algorithm are presented below.

- Another way to state the theorem is the following: let us denote by $\mathbb{P}^*(h, E, \alpha, \beta)$ the probability that the mapping returned by the algorithm is optimal. Then for all $0 < \alpha, \beta < 1$, $\lim_{E \to 0} \lim_{n \to \infty} \mathbb{P}^*(h, E, \alpha, \beta) = 1$.

- Introducing non deterministic decisions allows the algorithm to escape from locally optimal configurations where moving one task from one PU does not help but still better configurations exist. However, in the algorithm, greedy decisions (*i.e.* choosing the most loaded processor, or the biggest task) are taken with a high probability. This allows us to speed up convergence to good mappings. For example, picking the biggest task with high probability mimics the classical Largest Processing Time (LPT) algorithm that can be proved to provide a $\frac{4}{3}$-approximation of the optimal mapping (GONZALEZ; IBARRA; SAHNI, 1977).

- The algorithm resembles the Gibbs sampling method (BREMAUD, 1999). In the classical Gibbs sampling method, the moved task is chosen according to a uniform distribution over all tasks or using a fixed deterministic order among tasks. Here, the proof of optimality of the algorithm does not follow from standard arguments, because the mapping process is not a reversible Markov chain. In this context, a new proof is needed.

- Using the classical Gibbs sampling algorithm is not practical here because it is not easy to pick a task uniformly over all tasks. It is easier to first pick a PU and then pick a task on that PU, as done in the proposed algorithm. Also, an adequate tuning of the parameters α and βcan help to improve the speed of convergence.

- Some restrictions imposed on Gibbs sampling also apply here. Moving two tasks simultaneously is not allowed: the optimality result does not hold in this case.

- Finally, note that the convergence in the theorem is asymptotic ($h \to \infty$) and then, the optimal mapping may be reached after a prohibitive time. In order to have a good trade-off between optimality and convergence time, we set the parameters of

the algorithm to $E = 0.1$, $\alpha = 0.8$, $\beta = 0.8$ in the experimental evaluation present in Chapter 6, and the algorithm stops as soon as no performance improvement is achieved in the iteration of the main loop (lines 2-13).

Proof. Let us denote by $(M_E(i))_{i \leq h}$ the sequence of mappings produced by HWTOPOLB under temperature E (line 11). This is a Markov chain over the set of all possible mappings. It should be clear that, if the parameters α and β are strictly between 0 and 1, then all mappings are reachable by $(M_E(i))$ with a positive probability. Hence the unique stationary distribution of the Markov chain denoted by π_E gives the asymptotic distribution of $(M_E(i))$.

The proof proceeds in two steps. We first show that, if the task were picked uniformly in the algorithm, the Markov chain would be reversible. This makes the computation of the asymptotic distribution easy and the result follows in that case. However, note that, in Algorithm 2, the probability of selecting a given task cannot be made uniform even by tuning β and α. In the second stage, the limit distribution in the real nonuniform case is characterized by using spanning trees. This allows us to show that the set of mappings M such that $\lim_{E \to 0} \pi_E(M) > 0$ are exactly the same as in the uniform case. This will complete the proof.

4.1.2.4.1 Uniform Case

This case amounts to changing lines 3 to 10 in the algorithm by "choose task $\bar{t}$ uniformly in $\mathcal{T}$".

Let us denote by $\mathcal{N}_t(M)$ the neighborhood of the mapping M, which is the set of mappings that differ from M only for task t. The transition probability from M to M' ($M' \neq M$), denoted by $P_E(M, M')$, is given by Equation 4.11.

$$\begin{cases} \frac{1}{T} \frac{\exp(globalCost(M')/E)}{\sum_{M'' \in \mathcal{N}_t(M)} \exp(globalCost(M'')/E)} & \text{if } M' \in \mathcal{N}_t(M) \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

For all $E > 0$, $(M_E(i))$ is a reversible Markov chain, *i.e.* $\pi_E(M)P_E(M, M') = \pi_E(M')P_E(M', M)$. It is easy to check that $\pi_E(M) \propto \exp(globalCost(M)/E)$, so that $\pi_E(M)$ is positive only for mappings that minimize the global cost, when E is arbitrarily close to 0.

4.1.2.4.2 General Case

In the nonuniform case, which corresponds to the proposed algorithm, the Markov chain $(M_E(i))$ is no longer reversible. Theorem 1 by Álos-Ferrer and Netzer (2010) states that the mappings M^* ultimately reached by the algorithm (*i.e.* such that $\lim_{E \to 0} \pi_E(M^*) > 0$ for all E) are those minimizing the function $W(M) \stackrel{\text{def}}{=} \min_{S \in \mathcal{S}(M)} w(S)$, where $\mathcal{S}(M)$ is the set of all spanning trees with root M over the graph induced by the neighborhood relation $\mathcal{N}(.) \stackrel{\text{def}}{=} \cup_{t \in \mathcal{T}} \mathcal{N}_t(.)$, and $w(S)$ as defined in Equation 4.12.

$$w(S) \stackrel{\text{def}}{=} \sum_{(M,M') \in S} \left(globalCost(M') - \min_{M'' \in \mathcal{N}(M)} globalCost(M'') \right) \tag{4.12}$$

A close look at the definition of the function $w(S)$ should convince one that the spanning trees minimizing $w(S)$ do not depend on the probability that a given task is chosen.

Consequently, mappings such that $\lim_{E \to 0} \pi_E(M) > 0$ are the same as in the uniform case.

□

4.1.3 Comparison between proposed centralized algorithms

Although our proposed algorithms have the same objective of providing performance portability to scientific applications running on parallel platforms, they show many different characteristics. For instance, each considers different platforms attributes, even though they have access to the same machine topology model. NUCOLB works only with parallel platforms with CNs composed of NUMA nodes, and uses the NUCO FACTOR to weight communication distance between NUMA nodes, while HWTOPOLB accepts platforms without nonuniform memory levels, and uses both latency and bandwidth at different topology levels to estimate communication costs.

Their processes to decide where to map one task are also different. NUCOLB maps a task to the PU that presents the smallest cost deterministically, whereas HWTOPOLB maps it to the PU that would result in the smallest makespan with the highest probability.

NUCOLB evaluates a new mapping for all tasks at each load balancing call, which results in a more aggressive strategy to fix unbalance. This is specially useful with dynamic applications, but can result in a more significant load balancing overhead. Meanwhile, HWTOPOLB tries to remap less tasks per load balancing call. This strategy provides a smaller load balancing overhead, and benefits applications that have been well mapped and show small PU idleness.

Besides the aforementioned characteristics, both algorithms are centralized. They benefit from a complete control over the mapping of tasks, but their execution times can harm performance portability when the size of the application or parallel platform increases. To overcome this problem, we discuss a different approach in the next section.

4.2 Hierarchical algorithms

The process of mapping tasks to PUs is organized as a tree in hierarchical algorithms. At root level, centralized decisions are taken to map all tasks to different domains. At each level of the tree, a domain distributes the tasks that it has received over its own subdomains. These decisions are taken independently and in parallel at each level. This process is repeated until tasks are finally mapped to PUs. In this sense, a centralized algorithm can be seen as a hierarchical algorithm of one level only.

By splitting part of the scheduling decisions, hierarchical algorithms are able to reduce their execution time when compared to centralized algorithms in large scale parallel platforms. This comes at a cost of a reduced view and control of the load and communication distribution over all processing units, which can lead to a slower convergence to a task mapping with a low PU idleness.

We based our topology-aware hierarchical load balancing algorithm, named HIERARCHICALLB, on our centralized algorithms presented in Section 4.1. We provide detailed explanations next.

4.2.1 HIERARCHICALLB: Hierarchical load balancer composition

HIERARCHICALLB is a load balancing algorithm for parallel platforms constituted of multiple compute nodes. It hierarchically organizes our centralized load balancing algorithms in two levels. At root level, HIERARCHICALLB employs HWTOPOLB to

distribute tasks over CNs. Meanwhile, it can use NUCOLB or HWTOPOLB at leaf level to map tasks to PUs inside a compute node.

4.2.1.1 Rationale

We hierarchically compose our centralized algorithms instead of creating a hierarchical algorithm from scratch to benefit from their well understood properties and behaviors. This approach also provides more flexibility, as new centralized algorithms can be added to HIERARCHICALLB later.

HIERARCHICALLB employs task mapping in two levels: first, all tasks available are mapped to CNs; and later, tasks from each CN are mapped to its PUs. This second phase is done in parallel for each CN. This organization matches the machine topology, as different mechanisms are used for communication between CNs (interconnection network) and inside a CN (memory hierarchy). Additionally, Zheng et al. (2011) show that a tree with two load balancing levels can have the smallest execution time.

NUCOLB was not chosen as a load balancing algorithm to be employed at root level due to the following reasons: (i) it works only with platforms composed of NUMA nodes, which are not seen by the root load balancer; (ii) it would result in a longer load balancing time than HWTOPOLB, as it evaluates new mappings for all tasks at each load balancing call; and (iii) it would also result in more task migrations among CNs.

4.2.1.2 Application and platform modeling

Similarly to HWTOPOLB, HIERARCHICALLB models the application as $\mathcal{A}_{\text{HierarchicalLB}} = (\mathcal{T}, Load, Size_0, Msgs, Bytes)$, and the whole machine topology as $\mathcal{O}_{\text{HierarchicalLB}} = (\mathcal{P}, \mathcal{L}, S, \{C_{\text{lat}}, C_{\text{band}}\})$. However, these models are split into two different views, one for each of the task mapping levels.

At the root level (or centralized level), HIERARCHICALLB keeps the same application model but has a different view of the machine topology, as it focuses on mapping tasks to CNs instead of PUs. In this context, the machine topology is modeled as $\mathcal{O}_{\text{net}} = (\mathcal{N}, \mathcal{L}_{\text{net}}, S_{\text{net}}, \{C_{\text{net lat}}, C_{\text{net band}}\})$, with $\mathcal{N}$ the set of compute nodes that serve as virtual PUs for the centralized load balancing algorithm, $\mathcal{L}_{\text{net}}$ the set of network levels of the topology, the first topology level shared by two CNs as a function $S_{\text{net}} : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{L}_{\text{net}}$, and the communication latency and bandwidth of one CN to another as functions $C_{\text{net lat}} : \mathcal{N} \times \mathcal{N} \rightarrow \mathbb{R}_{>0}$ and $C_{\text{net band}} : \mathcal{N} \times \mathcal{N} \rightarrow \mathbb{R}_{>0}$. Additionally, the task mapping at this level is seen as $M_{\text{net}} : \mathcal{T} \rightarrow \mathcal{N}$.

At the leaves level (or distributed level), the load balancing algorithm at each compute node sees a different part of the application and machine topology. The leaf load balancer responsible for CN $i \in N$ models the parallel application as $\mathcal{A}_i = (\mathcal{T}_i, Load_i, Size_0, Msgs_i, Bytes_i)$, where $\mathcal{T}_i$ represents the set of tasks mapped to i, their loads as a function $Load_i : \mathcal{T}_i \rightarrow \mathbb{R}_{>0}$, the number of messages sent from one task to another as a function $Msgs_i : \mathcal{T}_i \times \mathcal{T}_i \rightarrow \mathbb{N}$, and the number of bytes sent from one task to another as a function $Bytes_i : \mathcal{T}_i \times \mathcal{T}_i \rightarrow \mathbb{N}$. $\mathcal{T}_i$ is a subset of $\mathcal{T}$ where Equation 4.13 holds. It is important to notice that all communication among tasks in different CNs is lost at this level.

$$\forall t \in \mathcal{T}_i \Rightarrow t \in \mathcal{T} \wedge M_{\text{net}}(t) = i \tag{4.13}$$

The load balancer responsible for a compute node sees only the PUs and topology levels that are part of it. Considering a function $cn : \mathcal{P} \rightarrow \mathcal{N}$ that relates PUs to CNs, the

machine topology of CN $i \in \mathcal{N}$ is seen as $\mathcal{O}_i = (\mathcal{P}_i, \mathcal{L}_{\text{mem}}, S_{\text{mem}}, \{C_{\text{mem lat}}, C_{\text{mem band}}\})$, with $\mathcal{P}_i$ the set of PUs in where $\forall p \in \mathcal{P}_i \Rightarrow cn(p) = i$, $\mathcal{L}_{\text{mem}}$ the set of memory levels of the topology in i, the first topology level shared by two PUs as a function $S_{\text{mem}} : \mathcal{P}_i \times \mathcal{P}_i \rightarrow \mathcal{L}_{\text{mem}}$, and the communication latency and bandwidth of one PU to another as functions $C_{\text{mem lat}} : \mathcal{P}_i \times \mathcal{P}_i \rightarrow \mathbb{R}_{>0}$ and $C_{\text{mem band}} : \mathcal{P}_i \times \mathcal{P}_i \rightarrow \mathbb{R}_{>0}$.

These models are provided by HIERARCHICALLB to the centralized algorithms that compose it. Nevertheless, these algorithms may disregard parts of the models. For instance, NUCOLB does not take into account $Bytes_i$ and $C_{\text{mem band}}$ in its decisions.

4.2.1.3 Algorithm

HIERARCHICALLB does not decide a task mapping by itself. Instead, it employs centralized load balancing algorithms in two levels. For this reason, we split HIERARCHICALLB's algorithm into two parts: a root algorithm and a leaf algorithm. The root algorithm is executed in a centralized fashion once at each load balancing call. Meanwhile, an instance of the leaf algorithm is run in parallel in each compute node of the platform.

HIERARCHICALLB can be said to operate in four steps: (i) the leaves send data to the root; (ii) the root computes a mapping of tasks to CNs; (iii) the leaves receive data from the root; and (iv) the leaves compute a mapping of tasks to PUs.

The operation of HIERARCHICALLB at root level is presented in Algorithm 3. It starts by receiving application information from all leaf load balancers. This includes their application models $\mathcal{A}_i$, and functions $Ext_Msgs_i : \mathcal{T}_i \times \mathcal{T} \rightarrow \mathbb{N}$ and $Ext_Bytes_i \mathcal{T}_i \times \mathcal{T} \rightarrow N$, which refer to the number of messages and bytes sent from tasks in CN i to tasks in other CNs, respectively. They are defined in Equations 4.14 and 4.15. The algorithm continues by organizing the application model and mapping at network level. On line 5, HIERARCHICALLB calls a centralized load balancing algorithm (currently HWTOPOLB). It then divides the application model according to its mapping decisions using function $split$, and sends this information to the leaf load balancers.

Algorithm 3: HIERARCHICALLB's root algorithm.

Input: Topology $\mathcal{O}_{\text{net}}$
1 **for** $i \in \mathcal{N}$ **do**
2 Receive $\mathcal{A}_i, Ext_Msgs_i, Ext_Bytes_i$ from leaf$_i$
3 $M_{net}(t) \leftarrow i, \forall t \in \mathcal{T}_i \wedge i \in N$
4 $\mathcal{A}_{\text{HierarchicalLB}} \leftarrow join(\mathcal{A}, Ext_Msgs, Ext_Bytes)$
5 $M'_{net} \leftarrow LB_algorithm(\mathcal{O}_{\text{net}}, \mathcal{A}_{\text{HierarchicalLB}}, M_{net})$
6 $\mathcal{A}' \leftarrow split(\mathcal{A}_{\text{HierarchicalLB}}, M'_{net})$
7 **for** $i \in \mathcal{N}$ **do**
8 Send $\mathcal{A}'_i$ to leaf$_i$

$$Ext_Msgs_i(t,q) = \begin{cases} Msgs(t,q) & \forall t,q \in \mathcal{T} \wedge t \in \mathcal{T}_i \wedge q \notin T_i \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

$$Ext_Bytes_i(t,q) = \begin{cases} Bytes(t,q) & \forall t,q \in \mathcal{T} \wedge t \in \mathcal{T}_i \wedge q \notin T_i \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

As mentioned before, HIERARCHICALLB leaf load balancers compute a new task mapping in parallel for each compute node. This is done as portrayed in Algorithm 4. Each leaf load balancer starts by sending its local information to the root load balancer and waiting for new application data. After receiving an answer (line 2), the leaf maps all new tasks to a PU chosen uniformly in $\mathcal{P}'$ using function $random_PU$ This process is used to give a valid task mapping to the centralized load balancing algorithm, which is called on line 4 to provide a new mapping of tasks to PUs. This algorithm could be HWTOPOLB or NUCOLB.

Algorithm 4: HIERARCHICALLB's leaf algorithm.

Input: Topology $\mathcal{O}_{\text{mem}}$, Application $\mathcal{A}_i$, Mapping M_i, Messages to remote tasks Ext_Msgs_i, Bytes to remote tasks Ext_Bytes_i

Output: Mapping M'_i

1 Send $\mathcal{A}_i, Ext_Msgs_i, Ext_Bytes_i$ to root
2 Receive $\mathcal{A}'_i$ from root
3 $M_i(t) \leftarrow random_PU(\mathcal{P}_i), \forall t \in \mathcal{T}'_i \wedge t \notin \mathcal{T}_i$
4 $M'_i \leftarrow LB_algorithm(\mathcal{O}_{\text{mem}}, \mathcal{A}'_i, M'_i)$

4.2.1.4 Algorithm properties

The pessimistic complexity of HIERARCHICALLB cannot be easily defined, as it depends on the centralized algorithms used. For instance, as explained in Section 4.1.2.4, HWTOPOLB's complexity cannot be defined in terms of number of tasks or PUs as its iterations depended on the horizon h.

HIERARCHICALLB computes a new task mapping in sequential steps that involve two communication phases and two load balancing algorithms phases. Meanwhile, a centralized load balancing algorithm executes only one step. In this context, HIERARCHICALLB requires a large number of tasks and PUs to compensate this overhead when compared to a centralized algorithm.

An advantage of HIERARCHICALLB over our centralized algorithms is that it reduces the memory required in a compute node to run a load balancer. This happens because no part of the load balancing tree requires a complete view of the mapping of tasks to processing units. This information is spread over the different leaf load balancers, and the root load balancer sees a compressed view of the task mapping (to CNs instead of PUs). In this sense, our hierarchical load balancing algorithm can handle larger problems than our centralized ones.

A last aspect that has to be noticed is that, although HIERARCHICALLB can employ HWTOPOLB in both of its levels, it does not have the same convergence guarantee of the centralized algorithm. This guarantee is lost because HIERARCHICALLB does not respect the restriction of moving only one task at a time since decisions are taken in parallel at leaf level.

4.3 Implementation details

NUCOLB, HWTOPOLB, and HIERARCHICALLB are implemented using CHARM++ (KALE; KRISHNAN, 1993)(CHARM++, 2013). Their source-code is available online free of charge (HIESCHELLA, 2013). CHARM++ is a parallel programming language and runtime system (RTS) based on C++ with the goal of improving

programmer productivity. It abstracts architectural characteristics from the developer and provides portability over platforms based on one or more compute nodes.

CHARM++ contains a load balancing framework (ZHENG et al., 2011), which provides an interface for developing load balancing plugins to CHARM++ applications. A load balancer is provided with information regarding the application ($\mathcal{A}$) and its current mapping (M_{init}), and the runtime system expects in return a new task distribution to migrate tasks. We discuss CHARM++ in more details in Section 5.1.

All load balancing information comes from previous timesteps of the application. Profiled information is reset after each load balancing call. CHARM++ models the application as $\mathcal{A}_{\text{Charm++}} = (\mathcal{T}, Load, Size_0, Msgs, Bytes)$. Although this representation provides plenty information for load balancing algorithms, its limitations also have an important impact over the scheduling decisions of NUCOLB and HWTOPOLB. Examples of trade-offs are given below.

- Information is aggregated from all timesteps between two load balancing calls. However, a load balancer does not know how many timesteps have passed in this period, nor can it split this profile to analyze the behavior of tasks during past timesteps. This precludes the dynamicity evaluation done by algorithms such as MIGBSP (RIGHI et al., 2010), which is discussed in Section 5.1.

  Storing only aggregated information is understandable, as keeping a profile for each timestep would increase the memory footprint of the load balancing framework, which could make centralized algorithms impractical.

  To circumvent this limitation, a load balancer call could be done at each application timestep. Still, this incurs in larger algorithm and task migration overheads affecting the application's performance, and a loss of performance portability.

- The moment in time when messages are sent is not captured. By knowing when tasks are communicating, a load balancer could evaluate if contention is happening at certain periods, and try to mitigate its effects. Still, storing this information could results in the same problem discussed above.

- The size of tasks in bytes is not provided ($Size_0$). Without this information, migration costs cannot be realistically predicted. In this situation, migrations have to be avoided to guarantee scalable performances.

  A way to discover task sizes would be to serialize tasks when calling a load balancer. This serialization is already done when a task is chosen for migration. Nonetheless, the overhead involved in allocating, deallocating, and copying memory could turn load balancing unfeasible. Serializing only one task would result in a smaller overhead, but could also lead to wrong decisions if tasks are irregular in size.

CHARM++ provides a flat view of the machine topology, which does not include any information about the topology levels, their sharing, and communication costs. To overcome this, we use our machine topology model described in Chapter 3. Our load balancing algorithms initialize the data structures responsible for the machine topology model at the start of the application, and then load this information from an XML file using our HIESCHELLA library.

To better explain how our load balancers are implemented, we show a simplified class diagram in Figure 4.1. It contains both the original CHARM++ load balancer classes and

the ones that we developed. The basic structures used by CHARM++ load balancers are implemented by the `BaseLB` class. The class `CentralLB` extends it to provide the basic mechanisms used by all centralized load balancers. Our centralized load balancing algorithms are based a new class named `TopologyAwareLB` that adds the machine topology model obtained from our library. Meanwhile, HIERARCHICALLB is implemented by extending the `HybridBaseLB` class, which serves as a base for hierarchical load balancers in CHARM++.

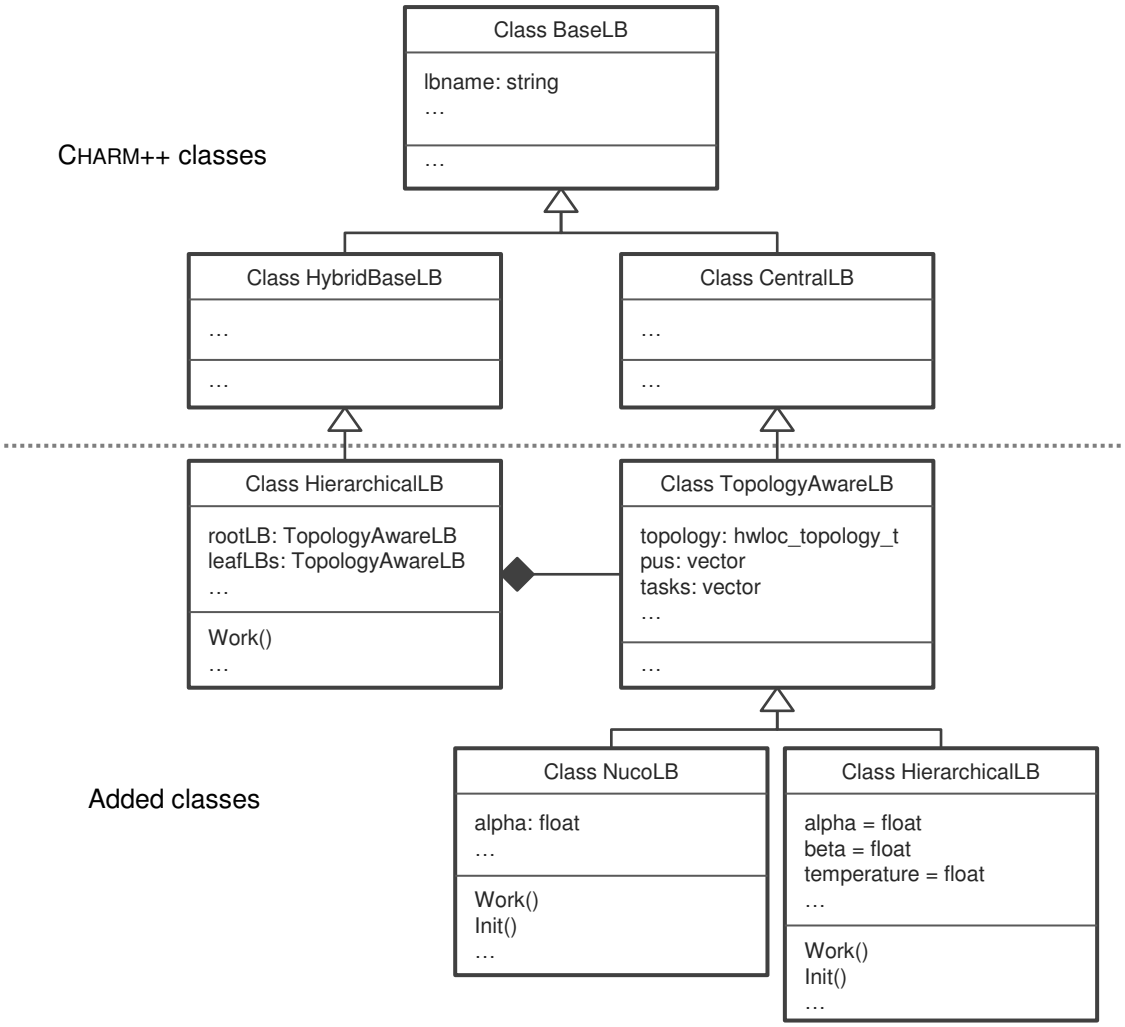

Figure 4.1: Load balancers simplified class diagram.

CHARM++ uses a token-based scheme for hierarchical load balancers in order to avoid unnecessary task migration overheads (ZHENG et al., 2011). Instead of migrating tasks between domains at each level of the load balancing tree, lightweight tokens containing only the tasks' workload information are used. At the end of the load balancing phase, all tokens are substituted by their original tasks to enforce the new task mapping.

Although we have implemented our load balancers using the CHARM++ parallel system, their characteristics and model allow them to be ported to other parallel libraries (e.g., MPI and OpenMP). Firstly, the machine topology model is provided by our library, being generic and independent of parallel language. The remaining information required, namely the application communication pattern and load, could be obtained through the

use of profile-guided tools and by extending parallel libraries. For instance, Lifflander, Krishnamoorthy and Kale (2012) developed a threaded active message library on MPI, which is discussed in Section 5.1. This library is able to collect the load of each task. The only information missing for our load balancing algorithms would be the communication graph, which should be trivial to obtain from the MPI runtime.

4.4 Conclusion

In this chapter, we presented our topology-aware load balancing algorithms, named NUCOLB, HWTOPOLB, and HIERARCHICALLB. They aim to provide performance portability to scientific applications running on parallel HPC platforms.

These algorithms involve different trade-offs regarding their control over load imbalance, costly communications, and load balancing execution time. NUCOLB is able to quickly fix performance problems, specially of dynamic applications, at the cost of a longer load balancing time and a greater number of task migrations. Meanwhile, HWTOPOLB provides a smaller load balancing overhead, and a guarantee of convergence to an optimal mapping, at the cost of a slower convergence to a better task mapping. Lastly, HIERARCHICALLB organizes centralized load balancing algorithms in a hierarchical fashion in order to scale to larger parallel platforms, at the cost of a reduced control over the load and communication distribution over all PUs when compared to a centralized algorithm such as NUCOLB and HWTOPOLB. The techniques used in our algorithms are compared to the state of the art in task mapping in Chapter 5.

Our algorithms were implemented using the CHARM++ parallel programming language in order to benefit from its load balancing framework, which provides detailed application information at a runtime level. Meanwhile, their machine topology models are obtained from our HIESCHELLA library. By using data from these two sources, our load balancing algorithms are kept independent of a specific application or platform. In addition, we are able to benefit from other load balancing algorithms and real application developed using CHARM++ to evaluate our own algorithms. We present an experimental evaluation of the proposed load balancing algorithms in Chapter 6.

5 RELATED WORK ON TASK MAPPING

The complexity of parallel platforms and applications demands effective task mapping techniques to achieve efficiency and scalability. This scheduling problem has been extensively studied over the past few decades and is known to be an NP-complete problem (LEUNG, 2004). Since no optimal solution can be computed in feasible time, different global scheduling algorithms and heuristics have been researched and developed (CASAVANT; KUHL, 1988).

Task mapping techniques can be employed at different times and levels. In the case of static applications, offline techniques can be used based on the analysis of their codes at compiler or pre-compiler level, or based on traces of previous executions. Meanwhile, more dynamic techniques can be employed at different levels, such as: at application level, which requires extensive knowledge about the application, and additional implementation effort when porting the technique to another application; at operating system level, which has a simplified view of the application, and is restrained to one compute only; and at middleware level, which requires the use of external runtime systems or libraries.

Some algorithms represent the machine topology $\mathcal{O}$ and application $\mathcal{A}$ in ways different from the ones presented in Sections 2.1 and 2.2. An algorithm or technique that models a flat machine topology has the set of levels of the topology $\mathcal{L}$ as $\mathcal{L}_{\text{flat}}$ with only one level, which is also the only level shared by all PUs. The communication time of one PU to another, also described as C, can be decomposed into different parameters, as the exact time that it takes to send one message also depends on characteristics of said message (e.g., its size in bytes), which are application-dependent. For instance, C could be decomposed into the communication bandwidth, delay, and overhead parameters of LogP (CULLER et al., 1993). Additionally, some approaches use synthetic communication costs C_{syn} that only represent the topology level where communication happens. In the applications' side, some models do not take into consideration their tasks' load ($Load_0$), size in memory ($Size_0$), number of messages ($Msgs_0$), or bytes communicated ($Bytes_0$).

In this chapter, the state of the art in task mapping and task scheduling techniques is presented. We split it in different categories according to the main technique applied: load balancing, work stealing, and process mapping. A comparison between the algorithms discussed in this chapter and the ones proposed in this thesis is presented at the end of the chapter.

5.1 Load balancing algorithms

Load balancing (LB) is a technique used to distribute an application's load and communication evenly over a platform in order to avoid having overloaded PUs. Most ap-

proaches try to fix load imbalances and costly communications periodically by remapping tasks ever so often. These approaches are **measurement-based**, which means that they require information about the application gathered during runtime. A tasks' load and communication graph captured during the last timesteps are used as an approximation of its behavior in future timesteps. This means that load balancers are more effective with more static applications, as less load balancing calls would be required to keep the platform and application in a balanced state. This is seen as the *principle of persistence* (ZHENG et al., 2011).

We discuss different approaches for load balancing in this section. Firstly, we will touch the subject of graph mapping and partitioning algorithms. Secondly, we present load balancing algorithms developed at runtime level, and follow with load balancing algorithms focused on application characteristics that go beyond the application model discussed in Section 2.2.1.

5.1.1 Graph partitioning

A technique to load balance dynamic applications involves using hypergraph partitioning, as done by Catalyurek at al. (2007). They focus on mitigating the effect of costly communications by reducing the communication volume among PUs and avoiding task migrations. The application is represented as a hypergraph, where vertices are tasks, and nets (hyperedges) represent communication and migrations costs. Communication and migration costs are computed using the amount of bytes a task communicates ($Bytes$) and contains ($Size$), respectively. All costs are scaled by a factor of α that represents the load balancing periodicity in application timesteps. Their algorithm tries to split the hypergraph into $k = \text{card}(P)$ partitions while keeping the load on each PU smaller than the average plus an overhead.

Hypergraph partitioning is done in phases using the ZOLTAN tool (DEVINE et al., 2002)(BOMAN et al., 1999). However, similar tools could be used to do graph partitioning, such as METIS (KARYPIS; KUMAR, 1995) (more specifically, HMETIS(METIS, 2013)) and SCOTCH (PELLEGRINI; ROMAN, 1996)(SCOTCH, 2013). Initially, a coarsening phase is used to reduce the number of vertices in the graph. Coarsening is done by merging pairs of vertices that have costly nets between them. This is done in steps until the amount of vertices is smaller than a fixed number (for instance, 2048), or when the last coarsening step does not reduce the hypergraph enough (e.g., by 10%). Coarsening steps are computed in parallel and the best match is chosen at each step. After coarsening the graph, each PU computes in parallel a different randomized greedy hypergraph growing algorithm to split it into k partitions. This is done through recursive bisection in ZOLTAN. Additionally, a refinement phase tries to move vertices among partitions in order to refine the load distribution.

Migrations costs are only considered after the first load balancing call. They are added to the hypergraph by including one fixed vertex to each partition of the graph. These vertices represent PUs and cannot be merged through the coarsening phase of the algorithm, nor moved to other partitions. Migration costs are added as nets between vertices in one partition and their local fixed vertex. This is done in order to reduce the time spent migrating tasks. Load balancing calls involving migration costs work to refine the initial task distribution.

5.1.2 Runtime level load balancers

Load balancing mechanisms have been proposed on different runtime systems. For instance, CHARM++ (KALE; KRISHNAN, 1993)(CHARM++, 2013) is a parallel programming language and runtime system (RTS) based on C++ with the goal of improving programmer productivity, as previously discussed in Section 4.3. It abstracts architectural characteristics from the developer and provides portability over platforms based on one or more compute nodes. However, it does not consider any information about the machine topology $\mathcal{O}$ besides the set of PUs $\mathcal{P}$. This results in a flat view of the machine topology ($\mathcal{L}_{\text{flat}}$).

CHARM++ applications are overdecomposed into active objects called *chares*. Programmers describe computation and communication in terms of how these chares interact and the RTS manages all messages generated from these interactions. Chares communicate through asynchronous messages. Furthermore, its RTS is responsible for physical resource management on the target platform.

A load balancing framework is also included in CHARM++. It provides a full view of an instance of an application execution $\mathcal{A}$ and its current mapping of tasks to PUs as a function $M : \mathcal{T} \rightarrow \mathcal{P}$. The load of each task includes the time spent on computation and communication. For a given mapping M, the load on each PU, $LoadPU : M \times \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$, is equal to the sum of the loads of all tasks mapped to it, plus runtime overheads ($Overhead : \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$), as in Equation 5.1.

$$LoadPU(M, p) = Overhead(p) + \sum_{t \in \mathcal{T} \wedge M(t) = p} Load(t) \tag{5.1}$$

All load balancing information comes from previous timesteps of the application. CHARM++'s load balancing framework provides the current task mapping M_{init} to a load balancer, and expects in return a new task distribution. This mechanism was used to implement several load balancing strategies (including ours), some of which are discussed below.

Zheng et al. (2011) study the performance and scalability of different periodic load balancers on CHARM++. The authors present four load balancing strategies:

- GREEDYLB is a centralized greedy algorithm that only uses task loads ($Load$) for its decisions (ZHENG, 2005). It is used to quickly mitigate load imbalance. GREEDYLB starts by removing tasks from their current mapping to create an empty mapping M_0. This includes removing the tasks' loads from their current PUs. It then sorts tasks in decreasing load order, and iteratively maps the unassigned task with the highest load to the least loaded PU. The initial task mapping M_{init} is not considered in this process, which leads to a large amount of task migrations. For a number of tasks $n = \text{card}(\mathcal{T})$, GREEDYLB is an $O(n \log n)$ algorithm.

- GREEDYCOMMLB is similar to GREEDYLB, with the addition of communication loads, $commLoad : M \times \mathcal{T} \rightarrow \mathbb{R}_{\geq 0}$. They are computed using the application's $Msgs$ and $Bytes$, and two synthetic communication cost parameters C_α and C_β . The communication cost of a task is related to how many messages and bytes it sends to tasks mapped to other PUs, as presented in Equation 5.2. Similarly, the communication load of a PU, $commLoadPU : M \times \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$, is equal to the sum of the communication load of the tasks currently mapped to it.

$$commLoad(M,t) = C_\alpha \times \sum_{u \in \mathcal{T} \wedge M(t) \neq M(u)} Msgs(t,u) + C_\beta \times \sum_{u \in \mathcal{T} \wedge M(t) \neq M(u)} Bytes(t,u) \quad (5.2)$$

$$commLoadPU(M,p) = \sum_{t \in \mathcal{T} \wedge M(t) = p} commLoad(M,t) \quad (5.3)$$

The algorithm starts by sorting tasks in decreasing load order, and uses an empty mapping M_0. GREEDYCOMMLB creates a new task mapping M_{i+1} by iteratively choosing a PU p_{min} to assign the unassigned task with the highest load t. This PU is chosen among the least loaded PU and all PUs that have tasks that t communicates with. p_{min} is the PU that presents the smallest total load, $totalLoad : M \times \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$, which is the sum of the current PU load, communication load, and communication load of task t. This is represented in Equation 5.4.

$$totalLoad(M_i,p) = LoadPU(M_i,p) + commLoadPU(M_i,p) + commLoad(M_i,t) \quad (5.4)$$

The communication load of task t is smaller in PUs that have tasks that it communicates with. The idea behind this is to mitigate costly communications while balancing load by keeping communicating tasks mapped to the same PU. Although GREEDYCOMMLB considers more information about the application than GREEDYLB, they have the same complexity for a communication graph with a constant vertex degree and card($\mathcal{T}$) $\gg$ card($\mathcal{P}$) (ZHENG, 2005).

- REFINELB improves load balance by incrementally adjusting the current task mapping. Based on M_{init}, it splits PUs as heavy or light according to their load. A processing unit p is considered heavy (or overloaded) if its load is greater than a threshold. This threshold is computed by multiplying the average PU load by a margin $\epsilon \geq 1.0$, as shown in Equation 5.5.

$$threshold(M) = LoadPU_{\text{avg}}(M) \times \epsilon \quad (5.5)$$

  REFINELB checks all possible task migrations from the most loaded PU to all light PUs, and migrates the task that leaves its new PU the closest to the threshold. This process is repeat until no migrations seem to improve load balance, or no PU is considered heavy. REFINELB contains an additional heuristic: it starts with a high ϵ at each load balancing call, and uses binary search to reduce it to achieve a state closer to the average load. As GREEDYLB, REFINELB does not consider the application's communication graph.

- HYBRIDLB is a hierarchical load balancer. It groups PUs in domains, which can be based on the network topology of the platform. Domains can be organized in multiple levels as a tree, where PUs are leaves. Different load balancing strategies are used depending on the level of the tree. This hierarchical approach is used to

reduce the load balancing algorithm time seen in centralized load balancers running on large scale parallel platforms. HYBRIDLB starts by representing all PUs in one domain as one virtual PU. Data is then sent to the domain leader, which will repeat this process through all intermediary tree levels. When load balancing data gets to the root, a refinement-based algorithm, such as REFINELB is applied. Using such algorithm has the advantage of reducing inter-domain migrations. As the load balancer chooses from where load should be migrated, it informs the leaders of its sub-domains. At intermediary levels, tasks are chosen to be migrated among domains. At the level closer to the leaves, a greedy algorithm, such as GREEDYCOMMLB or GREEDYLB, is used to decide the mapping of tasks to PUs in the domain.

The results presented by Zheng et al. show that: (i) centralized load balancers tend to provide better task distributions than hierarchical ones; (ii) refinement-based and hierarchical algorithms provide better scalability by reducing load balancing overheads, such as task migrations; and (iii) the hierarchical approach significantly reduces the memory footprint required for load balancing.

Although on a parallel environment other than CHARM++, Lifflander, Krishnamoorthy and Kale (2012) propose a load balancing mechanism for MPI that have similarities with the algorithms presented by Zheng et al. To achieve tasks with a finer granularity than the usual MPI process, they overdecompose the parallel application into tasks similar to the ones in CHARM++. However, their load balancing mechanism does not profile the application's communication graph. They propose a hierarchical load balancing strategy that blends REFINELB and GREEDYLB. It organizes PUs in a load balancing tree, where all PUs are leaves, and some are also repeated in different levels of the tree. The authors also discuss the possibility of using other organizations that better fit the machine topology.

At the beginning of the load balancing phase, the average PU load is determined. Each overloaded PU starts removing tasks from its task pool in increasing order of load until it is not considered overloaded anymore. A PU is considered overloaded when its load is greater than a $threshold(M)$ previously defined in Equation 5.5. The selected tasks and the new PU load are sent to its parent in the tree. This parent stores the heaviest tasks available in a greedy fashion, so it can later map them to its underloaded PUs while keeping such PUs under $threshold(M)$. It then sends the total load surplus or deficit in its sub-tree to its own parent, together with the remaining tasks. This procedure is repeated until load balancing information gets to the root. The root PU will map the heaviest tasks to the least loaded sub-trees in a greedy fashion. Sub-trees will repeat this process until the leaves receive their tasks.

The greedy approach used in each level of the tree has the objective of reducing load balancing communication and storage by only sending tasks and information necessary for the algorithm. Additionally, a load balancing tree with only two levels corresponds to a centralized algorithm, which is also discussed by the authors.

In a different context, Francesquini, Goldman and Mehaut (2013) employ load balancing on runtime environments based on the actor model running on one NUMA compute node. The main difference between these environments and the ones discussed previously is that tasks are dynamically created and destroyed during execution. They split tasks into two groups: regular tasks, which are expected to have a short lifespan; and hubs, which execute for a longer time and spawn several tasks. Communication happens mainly between a hub and the tasks it created, which are called its affinity group. This

division influences the initial task mapping, as hubs are spread through different NUMA nodes, and regular tasks are mapped to the same NUMA node than their hubs to reduce communication costs. Their load balancing algorithm works in three steps. It first tries to migrate tasks back to their original NUMA node (called home node). If there still load imbalance, then the algorithm evaluates task migrations between PUs in the same NUMA node. It considers migrations between NUMA nodes only if these two steps are not enough balance load. Besides this load balancing mechanism, the authors also propose a work stealing algorithm, which is discussed in Section 5.2.

5.1.3 Application-focused load balancers

Application-focused load balancing algorithms consider information that go beyond what is represented by our application model. For instance, in a different work using CHARM++, Bhatele, Kale and Kumar (2009) study the impact of network topology-aware load balancing algorithms on NAMD (see Section 2.2). Their approach benefits from knowledge specific to this application. Their research focuses on static and dynamic topology-aware mappings on 3D mesh and torus networks. In this context, a static mapping means choosing where a *cell* task is going to execute at the beginning of the application, while dynamic mapping is related to the periodic migration of *compute* tasks. A refinement-based algorithm is used for balancing load. Their results show that these static and dynamic techniques can improve the communication performance of NAMD by up to 10%. The metric used to evaluate load balancing algorithms is *hop-bytes*, which is based on the total number of bytes exchanged between PUs weighted by the distance between them in hops. They benefit from knowing the application to load balance, as they are able to decide the more important tasks to keep in place or migrate.

Still on CHARM++, Rodrigues et al. (2010) discuss a refinement-based strategy to reduce load imbalance and costly communications in weather forecast models (more specifically, BRAMS). They try to preserve the spatial proximity between neighbor tasks (and, by consequence, avoid costly communications) by traversing them with a Hilbert curve and recursively bisecting it according to their loads. This is repeated until the number of segments becomes equal to the number of PUs. This strategy is named HILBERTLB. This load balancer benefits from knowing that BRAMS has a static and regular communication graph. However, it does not use information about the number of messages or bytes in its decisions. Their results show that only HILBERTLB and METISLB, a graph partitioning load balancing algorithm that uses METIS (KARYPIS; KUMAR, 1995), are able to improve the performance of BRAMS.

In this same study, Rodrigues et al. also evaluated how the load balancing frequency and imbalance threshold influence the final application performance. Load balancing frequency is related to how many timesteps the application should run before calling HILBERTLB, while the imbalance threshold defines the acceptable load difference between the most loaded PU and the average. When this difference is smaller than the threshold, no task migrations are done. Their results showed that, for BRAMS, the best performance is achieved with an interval of 10 timesteps and a threshold of 10%. Bigger thresholds resulted in too few migration to fix load imbalance, while too small thresholds resulted in more migrations than necessary (which incur migration overheads). Meanwhile, although a higher frequency enabled perceiving load imbalance sooner, it also affected performance due to the time spent by the load balancer. Smaller frequencies resulted in taking too long to improve balance. This indicates that the optimal load balancing frequency depends on the application of interest.

In a different approach, Righi et al. (2010) propose an algorithm named MIGBSP for load balancing Bulk Synchronous Parallel (BSP) (VALIANT, 1990) applications running on parallel systems composed of multiple *sites* (e.g., clusters). Each site is organized as a set of PUs and has its own Set Manager. A Set Manager is responsible for load balancing decisions, and communication with other Set Managers. A load balancing call is done at each α timesteps. The value of α is changed according to the behavior of the application in the last iterations. If tasks execute for amounts of time that are too different from the average, the load balancing frequency is increased. Otherwise, it is decreased. In the context of MIGBSP, tasks have a granularity of process.

During a load balancing call, each Set Manager decides how likely it is to migrate some of its tasks to other sites. They compute a *Potential of Migration* $PM(t, s)$ for each task t and site s. PM is based on three metrics: computation, communication and memory. The higher the value of $PM(t, s)$, the more likely task t is to migrate to site s. Computation and communication affect PM positively, while memory does the opposite. The more stable a tasks' computation or communication is, the more these metrics will influence its potential of migration.

The computation metric $comp(t, s)$ is influenced by the load of t, and the performance of the PUs in s. This load is computed applying the Aging concept (TANENBAUM, 2008) over the load of task t at each of the last α timesteps. It uses the idea that the predicted load for the next timesteps is more influenced by recent timesteps. The communication metric $comm(t, s)$ applies the Aging concept over the amount of bytes sent from task t to other tasks in site s. Finally, the memory metric $mem(t, s)$ computes the amount of time it would take to move t from its current site to another one.

After the candidates for migration are chosen, their migration feasibility is evaluated. The Set Managers estimate the amount of time a candidate task t would take to run in its current site and its destination site. If the first is smaller than the second plus the estimated migration time, then t is not migrated. If no tasks are migrated after several load balancing calls, then MIGBSP decreases the load balancing frequency.

At the memory level, Hofmeyr et al. (2011) propose a user-level scheduler, named JUGGLE, to mitigate load imbalance of single program, multiple data (SPMD) static and regular applications running on a compute node. It runs on Linux and works with parallel application written using PTHREADS, UPC, and OPENMP. The authors focus on *off-by-one* imbalance. JUGGLE keeps a thread in each PU to monitor the progress of each task. Periodically, all monitored information is sent to a centralized scheduler that classifies PUs as fast and slow, and migrates tasks from slow PUs to fast PUs. Additionally, JUGGLE considers nonuniformity at memory level by forbidding migrations between NUMA nodes. Techniques similar to this one are employed on work stealing algorithms.

5.2 Work stealing

Work stealing (WS) algorithms try to mitigate load imbalance by moving tasks from loaded PUs to idle PUs. When a PU becomes idle, it becomes a *thief*. Thieves try to *steal* tasks from PUs that still have tasks to execute. The former PUs are called *victims* (FRIGO; LEISERSON; RANDALL, 1998)(BLUMOFE; LEISERSON, 1999). Following this definition, work stealing algorithms are mostly distributed or hierarchical scheduling algorithms. They can also be seen as receiver initiated load balancing algorithms (KUMAR; GRAMA; VEMPATY, 1994). This technique is specially useful for applications where $Load$ is unknown, and algorithms based on recursive, branch-and-bound, and divide-and-

conquer approaches.

Work stealing techniques differ from each other mainly by the way *task pools* are handled. Task pools are the abstraction used to organized tasks (DINAN et al., 2009). The scheduling algorithm may use either a centralized or a distributed approach. A centralized task pool resides in one PU and stores all tasks. All other PUs will have to steal tasks from it. Meanwhile, distributed tasks pools store patches of the set of tasks and are linked to different PUs. In this situation, a task pool may be local to one PU only, or shared by a group of PUs. In the case of distributed task pools, different approaches are used to define from where a thief PU will try to steal tasks. Examples of ways to pick a victim are: randomly, in round-robin, and from the nearest neighbors in the topology (KUMAR; GRAMA; VEMPATY, 1994). Some of the different approaches for work stealing are discussed below.

Francesquini, Goldman and Mehaut (2013) propose a NUMA-aware work stealing algorithm that complements their load balancing algorithm discussed in Section 5.1. In their context, each PU has its own local task pool. A thief chooses a victim following a order based on the number of hops between its NUMA node and the node of the victim. It first tries to steal tasks from task pools in its NUMA node, followed by task pools at a one hop distance, as so on. This acts to reduce the communication costs and task migration overhead while mitigating load imbalance.

At a memory level, Tchiboukdjian et al. (2010) propose a work-stealing algorithm for applications based on parallel loops using the KAAPI library (GAUTIER; BESSERON; PIGEON, 2007) running on one compute node. Their focus lies on reducing load imbalance while also reducing the amount of last level cache (LLC) misses. This is done without any knowledge of machine topology levels or communication times.

In KAAPI, each PU has its own task pool. An application has n independent tasks that reside sequentially in memory in a task pool organized as a queue. These tasks are the loop iterations. A thief steals more than one task at each time due to their granularity. Tasks are stolen from the beginning to the end of the list. The author's proposal lies in defining a window that limits how many tasks can be stolen and executed at a certain time. This is done to reduce the working set and improve cache usage.

Quintin and Wagner (2010) implement two new work stealing algorithms in KAAPI for parallel platforms composed of one or more CNs: Probability Work Stealing (PWS), and Hierarchical Work Stealing (HWS). In PWS, instead of selecting a victim uniformly, the probability of selecting a victim is proportional to the inverse of the distance between the PUs of the thief and the victim. This requires some topology information to be computed. Meanwhile, HWS organizes PUs in groups, such as all PUs in the same socket or cluster. Each group is composed of one leader and several slaves. Slaves can only steal local tasks in their group, while leaders steal global tasks between groups. The division of global and local tasks has to be set by the user. Both algorithms try to reduce the task migration overhead by avoiding steals from distant PUs.

In a different work using KAAPI, Hermann et al. (2010) present a work stealing algorithm for a compute node composed of PUs (called Central Processing Units in their approach, or CPUs) and graphics processing units (GPUs). Their technique focuses on applications seen as task dependency graphs. Dependency represents which tasks generate data used by other tasks. The initial task mapping is achieved by partitioning the graph with tools like METIS and SCOTCH, and mapping multiple partitions to each processing unit (including GPUs). Partitions are used to keep locality among tasks that share data. Load balance is later achieved by stealing a partition from a task pool.

Their work stealing policy takes into account the different performances that a task can achieve running on CPUs and GPUs. The runtime system captures task loads, and uses them to compute a ratio of the time taken to execute a partition in a CPU and in a GPU. If that ratio is over a threshold, then the partition can only be stolen by other GPUs. To avoid keeping a PU idle for too long, this threshold is increased each time a CPU fails to steal, and decreased each time a GPU fails.

In the context of OPENMP applications, Olivier et al. (2011) discuss and evaluate several scheduling algorithms. They also present a hierarchical work-stealing algorithm named *multithreaded shepherds* (MTS) for a CN composed of more than one socket. They focus on improving performance by reducing load imbalance and improving cache and memory usage.

MTS uses a task pool per socket to benefit from cache memory shared among PUs. PUs access their task pools as Last In, First Out (LIFO) queues, or stacks. The use of stacks improves cache usage by using temporal locality, as recently created tasks are scheduled first. Meanwhile, task pools are seen as First In, First Out (FIFO) queues for stealing tasks. This also helps locality as recent tasks are less likely to be migrated. In addition, when a PU becomes idle, it randomly chooses a task pool to steal tasks, and migrates a number of tasks equal to the number of PUs in its socket. This reduces the overhead of stealing remote tasks.

In the same scenario than Olivier et al., Broquedis et al. (2010) present a RTS for executing and tuning OPENMP applications named FORESTGOMP. FORESTGOMP focuses on reducing communication costs while balancing load by keeping tasks that are created in the same parallel region close together in the machine topology. The machine topology is captured using HWLOC (see Section 2.4). Every time tasks are created, they are grouped in a *bubble*. Bubbles are bound to specific parts of the machine, such as a core, a socket, or a NUMA node, to maximize cache reuse. These bubbles represent hierarchical task pools. If a PU has no task to execute in its local pool, it is going to search through the hierarchy before trying to steal from remote task pools. Additionally, FORESTGOMP stores where a bubble was last scheduled to be able to place it again in its previous location and improve cache reuse.

Besides their work on load balancing algorithms discussed in Section 5.1, Lifflander, Krishnamoorthy and Kale (2012) also propose a work stealing mechanism for MPI. What sets aside their proposal from others is the consideration of iterative applications as target. Their retentive work stealing algorithm keeps stolen tasks in their new PU's task pool, so that the application will start in a more balanced state in the next timestep.

In their approach, each PU has its own task pool, and task pools are randomly chosen for stealing. A window mechanism, similar to the one proposed by Tchiboukdjian et al., is used to define which tasks cannot be stolen. Besides keeping locality with recently created tasks, this also reduces the overhead of reading local tasks for execution, as it does not require obtaining a lock.

In the context of a specific application, Frasca, Madduri and Raghavan (2012) investigate methods to improve performance and power consumption of the Betweenness Centrality (BC) algorithm using OPENMP on a NUMA compute node. Their approach uses a task pool per PU and a retentive work stealing algorithm similar to the one proposed by Lifflander, Krishnamoorthy and Kale. However, instead of randomly choosing a task pool to steal, the thief's search for tasks is guided by the NUMA distance, which uses a synthetic value to represent communication costs. Synthetic communication costs are also applied in process mapping algorithms, as discussed in the next section.

5.3 Process mapping

Processing mapping (PM) is a technique used to define the initial task mapping of an application mainly. It largely focuses on mitigating costly communications, as applications are considered to have regular and static tasks, and do not suffer from load imbalance. Additionally, the number of tasks is usually the same as the number of PUs. Process mapping requires some level of knowledge about the machine topology. Different approaches are discussed below.

Mercier and Clet-Ortega (2009) try to improve the placement of MPI processes on a symmetric compute node. They gather the machine topology using a component of the PM^2 runtime system (THIBAULT; NAMYST; WACRENIER, 2007) and synthetic communication costs C_{syn}. This is used to generate a machine topology matrix. Meanwhile, application information is gathered by tracing an initial execution. The amount of bytes exchanged among tasks ($Bytes$) is extracted from the trace to generate a symmetric communication matrix.

Both the machine topology and communication matrices can be seen as complete, non-oriented graphs with weighted edges. By using the abstraction of graphs, the authors are able to use the SCOTCH library (PELLEGRINI; ROMAN, 1996) to statically map the communication graph to the topology graph.

Similarly, Jeannot and Mercier (2010) propose a process mapping algorithm named TREEMATCH for a NUMA compute node. Communication is gathered using the same approach as Mercier and Clet-Ortega. However, the machine topology is gathered using HWLOC (see Section 2.4) and represented as a tree. Using this information, TREEMATCH works similarly to a graph partitioning algorithm. A coarsening phase is applied over the communication graph to reduce the number of vertices by a factor of the arity of a level of the tree (from leaves to the root). When the number of vertices in the graph does not match the number of vertices in a level of the tree, disconnected vertices are added to the graph. When only one vertex remains, TREEMATCH starts to map the groups of vertices to the topology tree, in a way that tasks end mapped to PUs, which represent the leaves of the topology tree.

Cruz, Diener and Navaux (2012) present a dynamic process mapping algorithm that considers the memory hierarchy on its decisions. Tasks are remapped during their execution based on their current communication behavior. The information of which tasks are sharing data is captured from the Translation Lookaside Buffers (TLBs), or from page misses at operating system level (DIENER; CRUZ; NAVAUX, 2013). They employ the Edmonds graph matching algorithm, which solves the maximum weight perfect matching for complete weighted graphs. The weights for the communication graph are related to how much data two tasks share, while the ones for the platform are related to the first level of the topology shared by two PUs.

At a network level, Chen et al. (2006) propose a tool named MPI Process Placement toolkit (MPIPP) to map MPI processes to parallel platforms composed of multiple CNs. The network topology is benchmarked to obtain its communication costs C_{gap} and C_{overhead} of the LogP model (CULLER et al., 1993). For $P = \text{card}(\mathcal{P})$ PUs, $P/2$ ping-pong tests are run in parallel, where each PU communicates with only one other PU. This benchmark runs $P-1$ times, so all possible interactions between two PUs are measured. Meanwhile, a trace is used to gather information about the communication behavior of the application. Messages sizes are computed by dividing the amount of bytes exchanged between two tasks ($Bytes$) by their number of messages ($Msgs$). Using this data, the median message size is computed and compared to a threshold. If the median message size is bigger than

the threshold, then $Bytes$ and C_{gap} are used to compute the task mapping. Otherwise, $Msgs$ and C_{overhead} are used.

MPIPP mapping decision starts by placing each task in a random PU. Then, at each iteration, the algorithm selects which pairs of communicating tasks are going to exchange PUs, so that the communication cost reduction is the maximum possible. The communication costs depends on the chosen parameters, as discussed above. This process is repeated until no gains are achieved.

In a similar context, Hoefler and Snir (2011) study the problem of mapping applications with irregular communication patterns to the network topology. The network topology includes CNs and switches, while the application's communication graph considers $Bytes$ only. The authors focus on optimizing two different metrics: *Worst Case Congestion*, or congestion; and *Average dilation*, or dilation.

- **Congestion** is the ratio between the amount of bytes that are communicated thought a link in the network, and the capacity of the link. The **worst case congestion** gives a lower bound on the time needed for communication.

- **Dilation** is the average length of the path taken by a message sent from a task to another one. The **average dilation** is computed by weighting each communication between tasks and its frequency. Thus, it gives how many hops messages will have to traverse in the network, or how much the network will be stressed.

The authors propose a topology mapping library that employs different algorithms to try to find the task mapping that minimizes congestion and dilation. Firstly, in the case where more than one task will be mapped to each CN, a graph partitioner is used to group tasks such that the number of groups equals the number of compute nodes. This generates a new communication graph representing the application. If the number of tasks is equal to the number of CNs, then a graph similarity algorithm is used to generate a task mapping. Other algorithms employed include a recursive bisection algorithm and a greedy algorithm that maps the tasks with the most communication to the CNs with the largest bandwidths between them. In addition to these algorithms, each of the generated mappings goes through an optimization phase where random pairs of tasks are swapped. The new generated mappings have their congestion and dilation computed and compared, and only the best mapping found is used.

5.4 Discussion

Although task mapping algorithms use techniques much different from each other, many common points can be seen among them. In this section, we summarize the techniques and algorithms presented in this chapter and compare them to the algorithms proposed in Chapter 4 in light of their application modeling, machine topology modeling, and task mapping techniques employed. These comparisons are depicted in Tables 5.1, 5.2, and 5.3, where task mapping algorithms are ordered by their appearance in the text.

Task mapping algorithms are compared in Table 5.1 according to the kinds of applications that they focus at. This is done in terms of the following characteristics:

- The parallel application as a 5-uple $\mathcal{A} = (\mathcal{T}, Load, Size, Msgs, Bytes)$. $\mathcal{T}$ is omitted as all algorithms consider the tasks to map. The other parameters are shown with a check mark (✓) when considered by the mapping algorithm.

- The application behavior in terms of irregularity as: *Tasks* when only load irregularity is considered; *Comm.* when only communication irregularity is considered; *Both* when both are considered; and *Reg.* when the algorithm does not model irregularity.

- The application behavior in terms of dynamicity as: *Tasks* when only load dynamicity is considered; *Comm.* when only communication dynamicity is considered; *Both* when load and communication dynamicity are considered; and *St.* when the algorithm models static loads and communication.

Table 5.1: Task mapping algorithms comparison in terms of application modeling.

Algorithm	$Load$	$Size$	$Msgs$	$Bytes$	**Irreg.**	**Dyn.**
NucoLB	✓		✓		Both	Both
HwTopoLB	✓		✓	✓	Both	Both
HierarchicalLB	✓		✓	✓	Both	Both
Catalyurek at al.	✓	✓		✓	Both	Both
GreedyLB	✓				Tasks	Tasks
GreedyCommLB	✓		✓	✓	Both	Both
RefineLB	✓				Tasks	Tasks
HybridLB	✓		✓	✓	Both	Both
Lifflander et al.: LB	✓				Tasks	Tasks
Francesquini et al.: LB					Tasks	Tasks
Bhatele et al.	✓			✓	Both	Both
HilbertLB	✓				Tasks	Tasks
Righi et al.	✓	✓		✓	Both	Both
Juggle	✓				Reg.	St.
Francesquini et al.: WS					Tasks	Tasks
Tchiboukdjian et al.					Tasks	Tasks
PWS					Tasks	Tasks
HWS					Tasks	Tasks
Hermann et al.	✓		✓		Tasks	St.
MTS					Tasks	Tasks
ForestGOMP					Tasks	Tasks
Lifflander et al.: WS					Tasks	Tasks
Frasca et al.					Tasks	Tasks
Mercier et al.				✓	Comm.	St.
TreeMatch				✓	Comm.	St.
Cruz et al.				✓	Comm.	Comm.
MPIPP			✓	✓	Comm.	St.
Hoefler et al.				✓	Comm.	St.

The proposed load balancing algorithms presented in Chapter 4, NucoLB, HwTopoLB, and HierarchicalLB, consider applications with irregularity and dynamicity at both task and communication levels. This is a common feature in load balancing algorithms that consider the communication graph at some level. However, load

balancing algorithms are more sensible to dynamic loads than work stealing algorithms, as the latter are more reactive when perceiving idle PUs. Additionally, while algorithms are able to handle irregular and dynamic loads, dynamic communications are left aside in more than half of the studied approaches.

Our load balancers use the number of messages exchanged among tasks to represent the communication graph at some level (HWTOPOLB also considers the data volume). This is less commonly seen in other algorithms. One of the only algorithms that differentiates small messages from big messages is MPIPP (CHEN et al., 2006). Although large volumes of data communicated can have an impact in performance due to bottlenecks in the parallel platform, applications with a large number of small messages can also be affected by non negligible communication latencies. By considering both, an algorithm becomes more adapted to handle different kinds of applications.

On a final note, tasks' sizes are rarely accounted by the algorithms, as the main approach to reduce migration costs usually involves avoiding migrations whenever possible.

Many differences can also be found when comparing task mapping algorithms according to their representation of the machine topology. This is done in Table 5.2 in terms of the following characteristics:

- The machine topology as a 4-uple $\mathcal{O} = (\mathcal{P}, \mathcal{L}, S, C)$. $\mathcal{P}$ is omitted as all algorithms consider the PUs available to map tasks. $\mathcal{L}$ and S are grouped in one column because the way topology levels are shared is directly linked to the number of levels in the machine topology model. C is listed as empty if no communication costs are considered.

- The parallel platform levels considered for task mapping as: *Mem.* when only one compute node is modeled; *Net.* when multiple CNs are taken into account, but their internal organization is not considered; and *Plat.* when a whole parallel platform is modeled.

- The topology levels where nonuniformity is treated by the algorithm as: *Mem.* for the memory level; *Net.* for the network level; *Plat.* for both levels; and *Unif.* if all levels are seen as uniform.

- The topology levels where asymmetry is treated by the algorithm as: *Mem.* for the memory level; *Net.* for the network level; *Plat.* for both levels; and *Sym.* if all levels are seen as symmetric.

A first thing that can be noticed in Table 5.2 is that our algorithms are the only ones to consider nonuniformity at platform level, which includes both memory and network levels. All other algorithms that consider the whole platform in their decisions see a flat, symmetric and uniform topology, which has the quality of being simple. Nevertheless, instead of modeling a flat machine topology like most load balancers implemented using CHARM++, NUCOLB, HWTOPOLB and HIERARCHICALLB have a hierarchical view of the topology. Their approach to model communication costs is more strongly related to process mapping algorithms than to load balancing ones.

While task mapping algorithms that work at network level consider real communication costs in their decisions, such as the latency, bandwidth, and number of hops among CNs, others use mostly synthetic values to represent distant communications. Although synthetic values reduce the amount of work required to model the machine topology while

Table 5.2: Task mapping algorithms comparison in terms of machine topology modeling.

Algorithm	$\mathcal{L}$ **and** S	C	**Level**	**Nonunif.**	**Asym.**
NUCOLB	✓	C_{lat}	Plat.	Plat.	Mem.
HWTOPOLB	✓	$\{C_{\text{lat}}, C_{\text{band}}\}$	Plat.	Plat.	Plat.
HIERARCHICALLB	✓	$\{C_{\text{lat}}, C_{\text{band}}\}$	Plat.	Plat.	Plat.
Catalyurek at al.	flat	$\emptyset$	Plat.	Unif.	Sym.
GREEDYLB	flat	$\emptyset$	Plat.	Unif.	Sym.
GREEDYCOMMLB	flat	$\{C_{\text{syn}}, C_{\text{syn}}\}$	Plat.	Unif.	Sym.
REFINELB	flat	$\emptyset$	Plat.	Unif.	Sym.
HYBRIDLB	flat	$\{C_{\text{syn}}, C_{\text{syn}}\}$	Plat.	Unif.	Sym.
Lifflander et al.: LB	flat	$\emptyset$	Plat.	Unif.	Sym.
Francesquini et al.: LB	✓	$\emptyset$	Mem.	Mem.	Sym.
Bhatele et al.	✓	C_{hops}	Net.	Net.	Sym.
HILBERTLB	flat	$\emptyset$	Plat.	Unif.	Sym.
Righi et al.	✓	C_{band}	Net.	Net.	Sym.
JUGGLE	✓	$\emptyset$	Mem.	Mem.	Sym.
Francesquini et al.: WS	✓	C_{hops}	Mem.	Unif.	Sym.
Tchiboukdjian et al.	flat	$\emptyset$	Mem.	Unif.	Sym.
PWS	✓	C_{syn}	Plat.	Unif.	Sym.
HWS	✓	$\emptyset$	Plat.	Unif.	Sym.
Hermann et al.	flat	$\emptyset$	Mem.	Unif.	Sym.
MTS	✓	$\emptyset$	Mem.	Mem.	Sym.
FORESTGOMP	✓	$\emptyset$	Mem.	Mem.	Sym.
Lifflander et al.: WS	flat	$\emptyset$	Plat.	Unif.	Sym.
Frasca et al.	✓	C_{syn}	Mem.	Mem.	Sym.
Mercier et al.	✓	C_{syn}	Mem.	Mem.	Sym.
TREEMATCH	✓	C_{syn}	Mem.	Mem.	Sym.
Cruz et al.	✓	C_{syn}	Mem.	Mem.	Sym.
MPIPP	✓	$\{C_{\text{lat}}, C_{\text{band}}\}$	Net.	Net.	Sym.
Hoefler et al.	✓	$\{C_{\text{hops}}, C_{\text{band}}\}$	Net.	Net.	Net.

still providing the possibility of some vision of nonuniformity, they can also provide completely wrong communication costs, depending on the machine.

Although our algorithms are based on the same topology model, they consider different aspects of parallel systems. NUCOLB uses latency to represent the different communication costs among NUMA nodes, while HWTOPOLB estimates real communication times using latencies and bandwidths. Both consider asymmetries at memory level, but only HWTOPOLB considers asymmetry at network level too. This happens due to the way latency and bandwidth are measured at network level, as discussed in Section 3.2.2. Benchmarking the network involves measuring the RTT between pairs of compute nodes. Although we are able to differentiate bandwidth in the two directions, latency ends up being averaged. Since NUCOLB considers only latency in its decisions, it ends with a symmetric view of the network. The only other machine topology model that considers asymmetries is the one presented by Hoefler and Snir (2011). Their network asymmetry comes from modeling the routing algorithm employed in the platform. Routing differen-

tiates the probability of messages being sent through different paths in the network.

Finally, task mapping algorithms are compared according to their objectives and techniques employed in Table 5.3. Comparison is done in terms of the following characteristics:

- The employed technique: load balancing (LB), work stealing (WS), and process mapping (PM).
- The consideration of mitigating load imbalance and costly communications when mapping tasks. A check mark (✓) is used when an algorithm has one of them as objective.
- Taking the current task mapping into account when computing a new task mapping.
- The way the algorithm is managed as: *Cent.* for centralized algorithms; *Hier.* for hierarchical algorithms; and *Dist.* for distributed algorithms.

By crossing the comparisons made in Tables 5.1, 5.2, and 5.3, some insights can be brought to light.

- Most hierarchical algorithms take into account the original task mapping, as they focus in reducing task mapping overheads, such as the time spent migrating tasks.
- One may try to reduce costly communications without considering the communication costs of the platform or the communication graph in the task mapping algorithm, as done by Rodrigues et al. in their load balancing algorithm HILBERTLB (RODRIGUES et al., 2010). However, this requires extensive knowledge about the application of interest, and reduces generality.
- Different algorithms employ greedy strategies for task mapping, such as NUCOLB, GREEDYCOMMLB, HYBRIDLB, and the hierarchical load balancing algorithm proposed by Lifflander, Krishnamoorthy and Kale. What sets NUCOLB apart from them the most is its view of the machine topology, which influences its cost function and view of locality at a NUMA node level.
- Most task mapping algorithms make their decisions in a deterministic way, which results in the possibility of finding and stopping at a local optimum solution. In this sense, probabilities such as the one used by HWTOPOLB are rarely seen. The only exception in this case is PWS.

Additionally, the study of the state of the art in task mapping algorithms helped us see how load balancing algorithms are the most used for mitigating both load imbalance and costly communications, how taking into account the initial task mapping is a relevant approach used to avoid migrations and their overhead, and how a detailed machine topology model is important when focusing on communication costs. We evaluate how our load balancing algorithms are able to provide performance portability to applications running on parallel platforms when compared to other algorithms in the next chapter.

Table 5.3: Task mapping algorithms comparison in terms of techniques employed and objectives.

Algorithm	**Tech.**	**Load Imb.**	**Costly Comm.**	M_{init}	**Mgt.**
NucoLB	LB	✓	✓	✓	Cent.
HwTopoLB	LB	✓	✓	✓	Cent.
HierarchicalLB	LB	✓	✓	✓	Hier.
Catalyurek at al.	LB	✓	✓	✓	Hier.
GreedyLB	LB	✓			Cent.
GreedyCommLB	LB	✓	✓		Cent.
RefineLB	LB	✓		✓	Cent.
HybridLB	LB	✓	✓	✓	Hier.
Lifflander et al.: LB	LB	✓		✓	Hier
Francesquini et al.: LB	LB	✓	✓	✓	Cent.
Bhatele et al.	LB	✓	✓	✓	Cent.
HilbertLB	LB	✓	✓		Cent.
Righi et al.	LB	✓	✓	✓	Hier.
Juggle	LB	✓		✓	Hier.
Francesquini et al.: WS	WS	✓	✓	✓	Dist.
Tchiboukdjian et al.	WS	✓			Dist.
PWS	WS	✓	✓		Dist.
HWS	WS	✓	✓		Hier.
Hermann et al.	WS	✓		✓	Dist.
MTS	WS	✓			Dist.
ForestGOMP	WS	✓		✓	Dist.
Lifflander et al.: WS	WS	✓			Dist.
Frasca et al.	WS	✓			Dist.
Mercier et al.	PM		✓		Cent.
TreeMatch	PM		✓		Cent.
Cruz et al.	PM		✓		Cent.
MPIPP	PM		✓		Cent.
Hoefler et al.	PM		✓		Cent.

6 PERFORMANCE EVALUATION

The process of scheduling tasks in a parallel platform is usually based on heuristics, as an optimal mapping cannot be computed in feasible time. Task mapping algorithms consider different aspects of the application and machine topology to mitigate load imbalance, costly communications, or both. As so, an algorithm's final performance will depend on the number of tasks of the application, their loads, their dynamicity and irregularity, the algorithm's overhead, the number of PUs in the parallel platform, and others.

In this chapter, we present a performance evaluation of the proposed load balancing algorithms NUCOLB, HWTOPOLB, and HIERARCHICALLB in comparison to other algorithms of the state of the art discussed in Chapter 5. We first present the evaluation methodology common to our experiments. Experiments with NUCOLB and HWTOPOLB on a multicore compute node are presented next. This is followed by the performance evaluations on clusters with all proposed algorithms. A discussion over the obtained results is presented at the end of this chapter.

6.1 Evaluation methodology

Experiments were set over a combination of parallel platforms, applications, and load balancing algorithms. To ease the comprehension of our methodology, we first characterize the parallel platforms that served as testbed. This is followed by a description of the CHARM++ benchmarks and applications used in the evaluations. Next, other state of the art load balancers are detailed. This section ends with a discussion on final details of the experiments.

6.1.1 Experimental platforms

We conducted our evaluation on five representative parallel platforms. Table 6.1 summarizes some of the hardware characteristics of these machines. All characteristics listed, with the exception of the number of compute nodes, refer to one compute node only.

In addition to the attributes listed in Table 6.1, Xeon24 has a UMA (Uniform Memory Access) design. The compute nodes of Opt4 and Opt32 are interconnected through Gigabit Ethernet and a Gemini 3D torus network, respectively. Communication is performed using MPI. Opt4 is part of Grid'5000 (BOLZE et al., 2006).

The version of MPI available on Opt4 and Opt32, and other software used in these platforms are listed in Table 6.2. All machines use CHARM++ release $6.4.0$ for the experimental evaluation, with the exception of the results with NUCOLB presented in Sections 6.2.1 and 6.3.1, which use CHARM++ release $6.3.0$. Experiments using one compute node only use the CHARM++ multicore-linux64 build. Experiments running on multiple

Table 6.1: Overview of the platforms' hardware characteristics.

Characteristic	Xeon24	Xeon32	Opt48	Opt4	Opt32
#PUs	24	32	48	4	32
#Sockets	4	4	4	2	2
#NUMA nodes	—	4	8	2	4
Processor	Xeon X7460	Xeon X7560	Opteron 6174	Opteron 2218	Opteron 6272
Clock (GHz)	2.66	2.27	2.20	2.6	2.10
LLC (MB)	16 (L3)	24 (L3)	10 (L3)	2 (L2)	12 (L3)
DRAM (GB)	64	64	256	4	32
#CNs	1	1	1	20	16
Total PUs	24	32	48	80	512
Network Interconnection	—	—	—	Gigabit Ethernet	Cray Gemini

Table 6.2: Overview of the platforms' software components.

Software	Xeon24	Xeon32	Opt48	Opt4	Opt32
Linux kernel version	3.2.0	3.5.2	3.2.0	2.6.32	2.6.32
GCC version	4.6.3	4.7.1	4.6.3	4.4.5	4.3.4
MPI library	-	-	-	OpenMPI	cray-mpich2

CNs on Opt4 and Opt32 use the net-linux-x86_64-smp and mpi-crayxt-smp builds, respectively.

6.1.2 Benchmarks and applications

To evaluate the performance of our proposed load balancers, we selected three benchmarks from CHARM++, *lb_test*, *kNeighbor*, and *stencil4D*, and two applications: one from the molecular dynamics field named *LeanMD*, and another from the seismic wave propagation field named *Ondes3D*. They were chosen due to their varied range of communication patterns and workload characteristics. They are described below.

lb_test is an imbalanced iterative benchmark which supports different levels of load irregularity and communication patterns. A static load variation from 50 ms to 200 ms was used in our experiments.

kNeighbor is a synthetic iterative benchmark where a task communicates with k other tasks at each step (in these experiments, $k = 7$). Its communication pattern is a ring. In our experiments, messages of 8 KB are exchanged among tasks.

stencil4D is an imbalanced stencil computation. It uses a four dimensional mesh to represent its communication pattern. In these experiments, an array of side 128 was decomposed into blocks of size 32.

LeanMD is a molecular dynamics (MD) application written in CHARM++ and based on the popular MD application, NAMD (NELSON et al., 1996)(BHATELE et al., 2009). It simulates the behavior of atoms based on the Lennard-Jones potential, which is an ef-

fective potential that describes the interaction between two uncharged molecules or atoms. The computation on *LeanMD* is parallelized using a hybrid scheme of spatial and force decomposition. In each timestep, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is done by a different set of tasks 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. Communication in *LeanMD* happens between two arrays of tasks. In the first step of communication, tasks of the three dimensional cell array communicate using a multicast operation with tasks in the six dimensional array. Then, in the second step, tasks of the six dimensional array communicate with one or two tasks of the three dimensional array.

Ondes3D is a seismic wave simulator employed to estimate the damage in future earthquake scenarios (DUPROS et al., 2010). It simulates the propagation of seismic waves over a three-dimensional space. *Ondes3D* implements the standard fourth-order in space numerical scheme, and uses the Convolutional Perfectly Matched Layer (C-PML) method for its absorbing boundary conditions (KOMATITSCH; MARTIN, 2007). It uses a standard MPI Cartesian two dimensional grid decomposition. Although originally implemented using MPI, it has been ported to Adaptive MPI (AMPI) (HUANG; LAWLOR; KALE, 2004) to benefit from CHARM++'s load balancing framework (TESSER et al., 2014). For that to happen, the application has to be overdecomposed into multiple virtual MPI processes per PU. *Ondes3D* presents load irregularity due to the boundary conditions producing additional work, and load dynamicity from the simulation of waves spreading through space. For our experiments, we ran a simulation based on the Mw 6.6 2007 Niigata Chuetsu-Oki, Japan earthquake (AOCHI et al., 2013) with a resolution of 72 million grid points. The complete simulation is comprised of 6000 timesteps, but only the initial 600 ones were used in our experiments due to its long execution time.

Table 6.3 summarizes the benchmarks characteristics and parameters used in our experiments. Additional details over *LeanMD* parameters are presented separately for NUCOLB, HWTOPOLB, and HIERARCHICALLB in their respective sections. Different load balancing frequencies have been chosen for different applications in order to strike a balance between the benefits of remapping tasks and the overheads of moving tasks and computing a new task mapping. Deciding the optimal moment to call a load balancer is a challenging problem (MENON et al., 2012) and is out of the scope of this thesis.

Table 6.3: Benchmarks and application characteristics.

	Number of Tasks	Number of Timesteps	Communication Graph	Type	LB Frequency	Number of LB Calls
lb_test	200	50	random	load-bound	10	4
kNeighbor	400	50	ring	comm-bound	10	4
stencil4D	256	50	4D mesh	load-bound	10	4
LeanMD	*	*	complex	load-bound	*	*
Ondes3D	512	600	2D mesh	load-bound	20	29

6.1.3 Load balancers

We compare the performance of our load balancers to the baseline and to other state of the art algorithms. The baseline represents the use of no load balancer during the execution of the application. The state of art load balancers are represented by GREEDYLB, GREEDYCOMMLB, HYBRIDLB, SCOTCHLB, SCOTCHREFINELB, REFINECOMMLB, and TREEMATCHLB. These load balancers present a variety of task mapping strategies, including greedy, refinement-based, and graph partitioning. Their algorithms are discussed in more details in Chapter 5.

SCOTCHLB and SCOTCHREFINELB are based on the graph partition algorithms implemented in the SCOTCH library (PELLEGRINI; ROMAN, 1996). The difference between them is that SCOTCHREFINELB considers the current task mapping on its decisions. TREEMATCHLB is based on the process mapping algorithm TREEMATCH (JEANNOT; MERCIER, 2010). To group tasks in a way to mimic the granularity of processes, TREEMATCHLB starts by applying a greedy algorithm based on the tasks' loads. REFINECOMMLB is similar to REFINELB, but it considers the communication graph in its decisions like GREEDYCOMMLB. The employed version of HYBRIDLB uses REFINELB as a root load balancer and GREEDYLB as a leaf load balancer. These load balancers are compared to our proposed algorithms according to their machine topology model, application model and techniques employed in Tables 6.4 and 6.5, respectively. The characteristics used for comparison are a subset of the ones presented in Section 5.4.

Table 6.4: Load balancers comparison in terms of machine topology modeling.

Algorithm	$\mathcal{L}$ **and** S	C	**Nonunif.**	**Asym.**	**Mgt.**
NUCOLB	✓	C_{lat}	Plat.	Mem.	Cent.
HWTOPOLB	✓	$\{C_{\text{lat}}, C_{\text{band}}\}$	Plat.	Plat.	Cent.
HIERARCHICALLB	✓	$\{C_{\text{lat}}, C_{\text{band}}\}$	Plat.	Plat.	Hier.
GREEDYLB	flat	$\emptyset$	Unif.	Sym.	Cent.
GREEDYCOMMLB	flat	$\{C_{\text{syn}}, C_{\text{syn}}\}$	Unif.	Sym.	Cent.
HYBRIDLB	flat	$\emptyset$	Unif.	Sym.	Hier.
SCOTCHLB	flat	$\emptyset$	Unif.	Sym.	Cent.
SCOTCHREFINELB	flat	$\emptyset$	Unif.	Sym.	Cent.
REFINECOMMLB	flat	$\{C_{\text{syn}}, C_{\text{syn}}\}$	Unif.	Sym.	Cent.
TREEMATCHLB	✓	C_{syn}	Plat.	Sym.	Cent.

These load balancing algorithms are all implemented and made available with CHARM++, which results in several similarities. They are all centralized or hierarchical load balancers that work at a platform level, including one or more compute nodes in their decisions. Most are able to handle task load and communication that are both irregular and dynamic. Besides HYBRIDLB, all considered load balancers focus on mitigating load imbalance and costly communications. Additionally, all lack the knowledge of the task sizes in bytes.

6.1.4 Experimental setup

The results shown in this chapter are the average of a minimum of 20 runs for each benchmark and application, and present a statistical confidence of 95% by Student's t-

Table 6.5: Load balancers comparison in terms of application modeling and techniques employed.

Algorithm	$Load$	$Msgs$	$Bytes$	$M_{\textbf{init}}$
NucoLB	✓	✓		✓
HwTopoLB	✓	✓	✓	✓
HierarchicalLB	✓	✓	✓	✓
GreedyLB	✓			
GreedyCommLB	✓	✓	✓	
HybridLB	✓			✓
ScotchLB	✓		✓	
ScotchRefineLB	✓		✓	✓
RefineCommLB	✓	✓	✓	✓
TreeMatchLB	✓		✓	

distribution and a 5% relative error, unless noted. The execution time of all tests, and hardware counters, such as LLC cache misses, were captured using `perf stat` (STAT, 2013). Additionally, Charm++ threads were bound to PUs at the start of the experiments. This avoids thread migration overheads and guarantees that threads are pinned in the machine topology.

HwTopoLB uses three different parameters, named α, β, and T, as discussed in Section 4.1.2. Results shown in Section 6.2.2 were obtained with $\alpha = 0.8$, $\beta = 0.8$ and $T = 0.1$. Different combinations of $\alpha \in (0.5; 1)$, $\beta \in (0.5; 1)$, and $T \in (0.1; 1)$ were evaluated, but their results were all inside the relative error margin and hidden for the sake of simplicity. NucoLB uses a parameter α to control the weight that communication has over the execution time, as presented in Section 4.1.1. Results shown in this chapter were obtained with $\alpha = 10^{-5}$.

The performance impact of load balancing algorithms depends on several different parameters, such as the duration of application timesteps, the number of tasks, 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 timestep duration after load balancing and the average load balancing algorithm execution time. The timestep duration does not consider the timesteps before load balancing, nor overheads related to load balancing. The load balancing time is comprised of the time spent executing the algorithm and migrating tasks. These metrics are employed to better understand the performance of load balancing algorithms in the next sections.

6.2 Load balancing on multicore machines

We start this evaluation by focusing on the performance of the proposed algorithms when mapping tasks over parallel platforms composed of one compute node only. The multicore machines used for these experiments present varied machine topologies, including different numbers of PUs and NUMA nodes, which highlight different features of our centralized load balancers. Additionally, the results obtained at this smaller scale serve to guide decisions when working at a cluster level in Section 6.3.

We organize this section in three parts. Firstly, we present the results obtained with NUCOLB. This is followed by the results with HWTOPOLB. Lastly, we compare the performance of both algorithms with the seismic wave simulator *Ondes3D*.

6.2.1 NUCOLB

This section provides a performance comparison between NUCOLB and GREEDYCOMMLB, SCOTCHLB, and TREEMATCHLB on parallel platforms Xeon32 and Opt48. We first discuss results with benchmark *lb_test*, followed by *kNeighbor*, *stencil4D*, and, lastly, *LeanMD*.

The *lb_test* benchmark presents a large variation in task loads, from 50 ms to 200 ms, and a small number of tasks per PU. It starts in an imbalanced state, as illustrated in Figure 6.1. This figure shows the initial imbalanced timesteps of *lb_test*, and its balanced state after calling NUCOLB as captured by the Projections performance analysis tool (KALE; SINHA, 1993). Each horizontal bar represents a PU (called PE on Projections). The blue areas represent the execution of tasks, and the white areas represent idleness. The period between 2 and 9.4 seconds (7.4 seconds total) illustrates the initial imbalanced state of the benchmark. PE 7 is the most loaded PU. Meanwhile, the period between 9.4 and 14.5 seconds (5.1 seconds total) represents the 10 timesteps after calling NUCOLB and before calling it again. This scenario illustrates how NUCOLB is able to reduce PU idleness to a minimum.

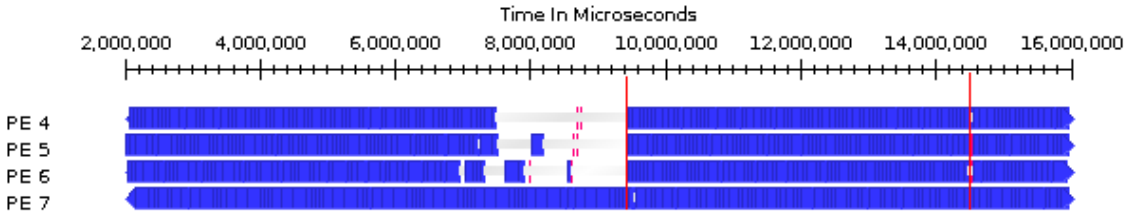

Figure 6.1: Execution of *lb_test* on PUs 4 to 7 of Xeon32, as captured by Projections.

Table 6.6 summarizes the execution times obtained for *lb_test* on machines Xeon32 and Opt48. 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. Performance improvements come from similar speedups in timestep duration. The best results are obtained with NUCOLB, as it achieves speedups of 1.39 over the baseline on Opt48 and of 1.30 on Xeon32. This represents keeping the idleness of Opt48 and Xeon32 PUs as low as 5% and 4%, respectively. NUCOLB migrates an average of 14 tasks per load balancing call, while other load balancers migrate 195 tasks on average. This, however, has a small impact on the total execution time, as these tasks have a small memory footprint. For instance, NUCOLB has a load balancing time of 9 ms on Xeon32, while SCOTCHLB takes 12 ms. The small amount of communication of the benchmark, which adds to a total of 1 MB per timestep, reduces the impact of considering the communication costs in these platforms. Nonetheless, the positive results with this compute-bound benchmark still emphasize the advantages of a list scheduling approach.

The execution times measured for the *kNeighbor* benchmark are presented in Figure 6.2. *kNeighbor* is a communication-bound benchmark. In this experiment, tasks communicate approximately 9 GB per timestep. Unlike *lb_test*, this benchmark starts in a balanced state. CHARM++ distributes tasks in a round-robin fashion through the PUs.

Table 6.6: Total execution time in seconds of *lb_test* and speedups for NUCOLB and other load balancers.

Load balancers	**Xeon32**		**Opt48**	
	Time	Speedup	Time	Speedup
Baseline	36.19 s	—	27.93 s	—
NUCOLB	27.95 s	1.30	20.09 s	1.39
GREEDYCOMMLB	28.57 s	1.27	20.27 s	1.39
SCOTCHLB	28.91 s	1.25	21.53 s	1.30
TREEMATCHLB	38.40 s	0.94	29.56 s	0.94

kNeighbor has twice as many tasks as *lb_test*, and these tasks have a regular and static behavior. This explains the original balanced state of the application. NUCOLB is able to reduce the average timestep duration of *kNeighbor* on Xeon32 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 tasks. On the same machine, other load balancers migrate an average of 388 tasks, which results in a migration time of more than 700 ms per load balancing call. This helps explain why other load balancers end up increasing the total execution time. In particular, GREEDYCOMMLB tries too hard to keep communicating tasks on the same PU. This leads to overloaded PUs, which increase the timestep duration by 50%. On Opt48, 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 tasks on the same NUMA node (instead of simply the same PU) and reduce communication costs. This results in a 6% reduction on the average load of each PU, or 80 CPU-seconds.

Figure 6.3 illustrates the total execution time of the *stencil4D* benchmark. *stencil4D* has a number of tasks similar to *lb_test*, and also starts in an imbalanced state. Imbalance comes from an original mapping that underloads some PUs, and from tasks with irregular computational loads. Its tasks send boundary information using messages of approximately 260 KB each, which adds to a total communication of 460 MB per timestep. *stencil4D* occupies 5.5 GB in memory, which means that it has large tasks. For both machines, NUCOLB shows the best performance. It obtains a speedup of 1.18 over the second best load balancer, GREEDYCOMMLB, on Xeon32. When considering only the timestep duration after load balancing, it obtains a speedup of 1.21. The difference in these speedups comes from the initial timesteps of the benchmark, which happen before any load balancer can fix the load unbalance.

NUCOLB migrates an average of 47 tasks per load balancing call on Xeon32, which translates to 100 ms spent migrating tasks. NUCOLB is able to avoid unnecessary migrations because it considers the original mapping of the application and knows the machine topology. By exploiting the machine topology, it is able to minimize migrations that could worsen performance by increasing communication costs. Meanwhile, other load balancers migrate 5.3 times more tasks. This results in a migration overhead 11 times greater than that of NUCOLB. This is related to the cost of migrating *stencil4D*'s large tasks. Table 6.7 summarizes the load balancing costs of *stencil4D*. The large number of

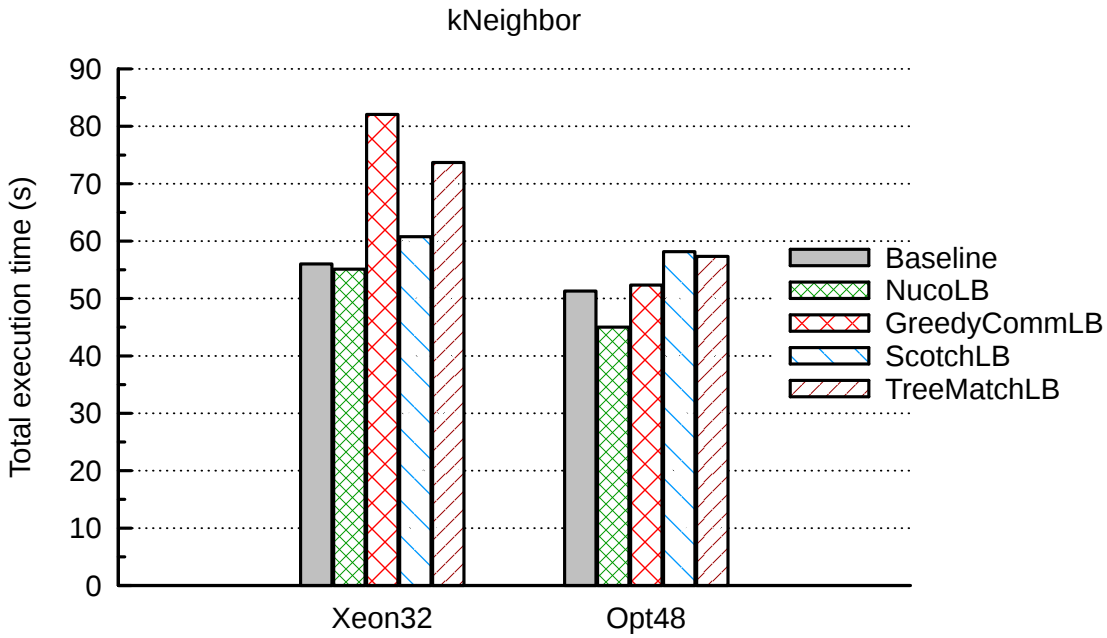

Figure 6.2: Total execution time of *kNeighbor* with NUCOLB and other load balancers.

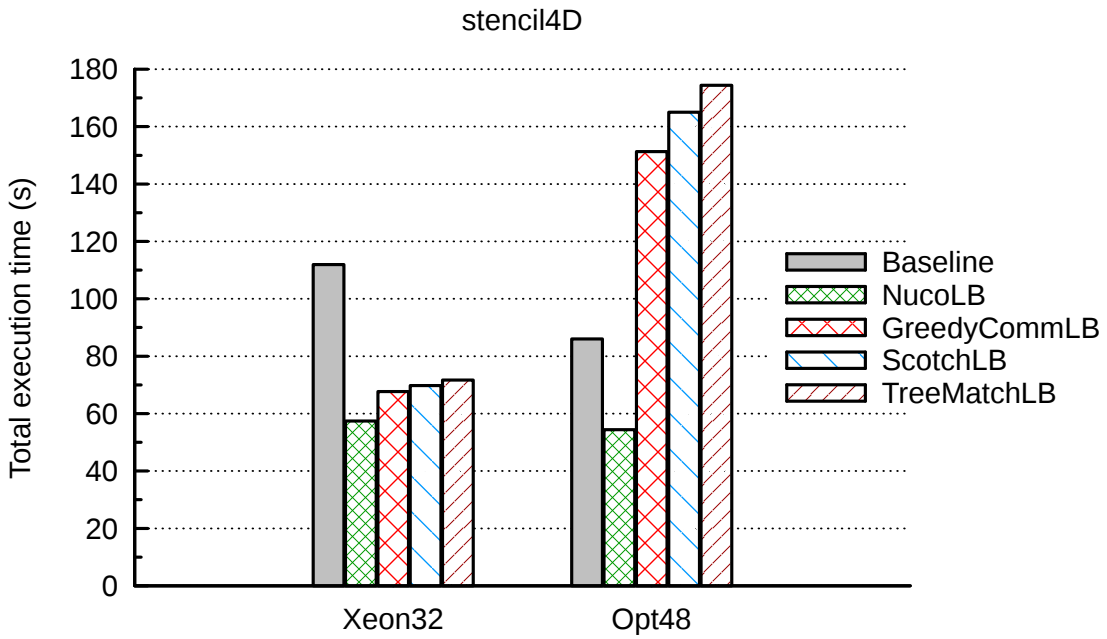

Figure 6.3: Total execution time of *stencil4D* with NUCOLB and other load balancers.

migrations leads to an increase in the memory footprint of the benchmark, which ends up affecting the performance of some tasks. For instance, other load balancers increase last level cache misses by 32% and page faults by 19% on Opt48 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 timestep duration from 1.84 s to 0.78 s.

NUCOLB also balances the communication between PUs, as can be seen in Fig-

Table 6.7: Load balancing times in seconds for *stencil4D* for different NUMA machines.

Load balancers	**Xeon32**	**Opt48**
Baseline	—	—
NucoLB	0.11 s	0.27 s
GreedyCommLB	1.17 s	0.84 s
ScotchLB	1.13 s	1.02 s
TreeMatchLB	1.14 s	1.04 s

ures 6.4 and 6.5. They show the distribution of messages received from other PUs for *stencil4D* on Opt48, as captured by Projections. The horizontal axis identifies each PU, while the vertical axis represents the number of messages received from other PUs. These figures show how NucoLB balances communications, as the difference in the most and the least number of messages received by any PU is reduced. Before load balancing, PU 22 receives the largest amount of messages, 4.5 times more than the smallest number of messages received by any PU. This difference is reduced to 2.6 times after load balancing.

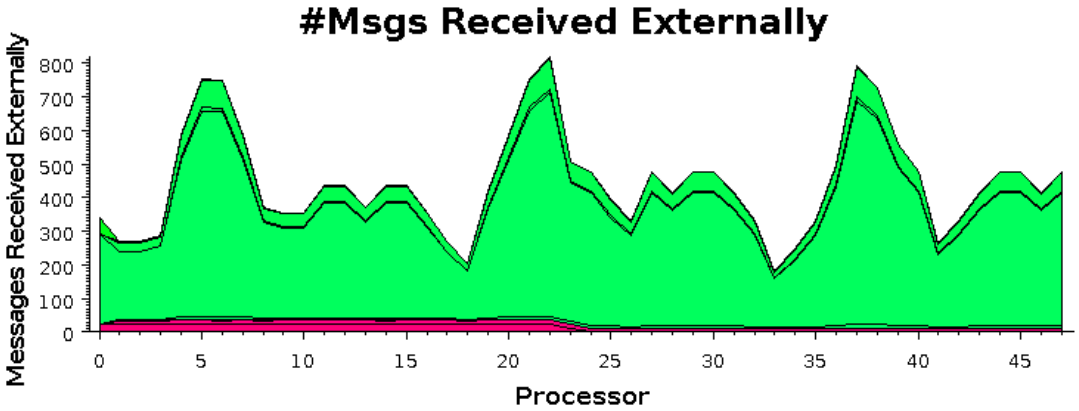

Figure 6.4: Messages received per PU before load balancing (first 10 timesteps).

The performance of the *LeanMD* application can be seen in Figure 6.6. This application is comprised of many small tasks. Each occupies a small amount of memory and communicates through small messages of approximately 100 bytes. In these experiments, *LeanMD* runs 1875 tasks for 300 timesteps, and a load balancing call is done at each 60 timesteps, for a total of four load balancing calls. This application starts in a balanced state in this experiment. For instance, its original task mapping presents a PU idleness of only 5% on Xeon32. 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 tasks per PU in this experiment. We have an average of 58 tasks per PU, and the difference between the PUs with the greatest and the smallest number of tasks is only 3. Although this brings an initial balanced state to the application, this incurs a higher processor overhead, as many tasks have to be scheduled on each PU. We have a different situation on Opt48, with its larger number of PUs. In this situation, NucoLB is able to reduce the timestep duration by 12%, resulting in a speedup of 1.12 over the baseline. Other load balancers show simi-

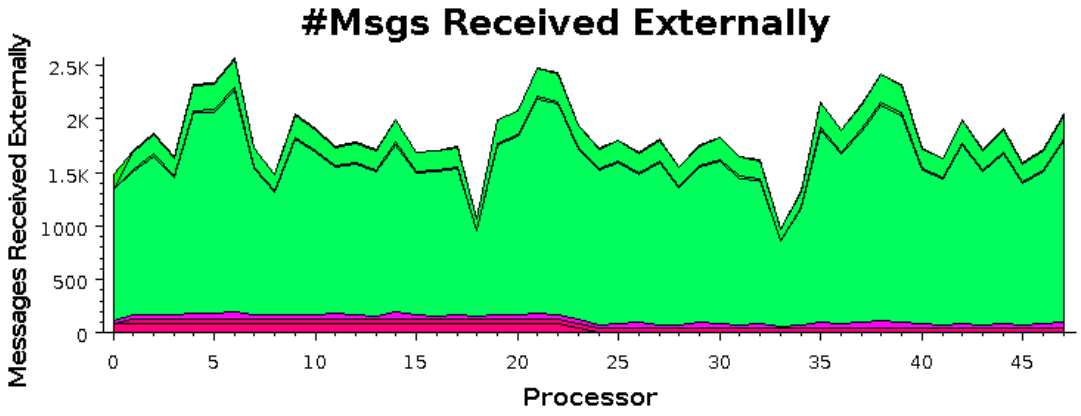

Figure 6.5: Messages received per PU after load balancing with NUCOLB (remaining 40 timesteps).

lar performances as, in this case, migration costs are negligible and communication plays a minor part in the overall performance.

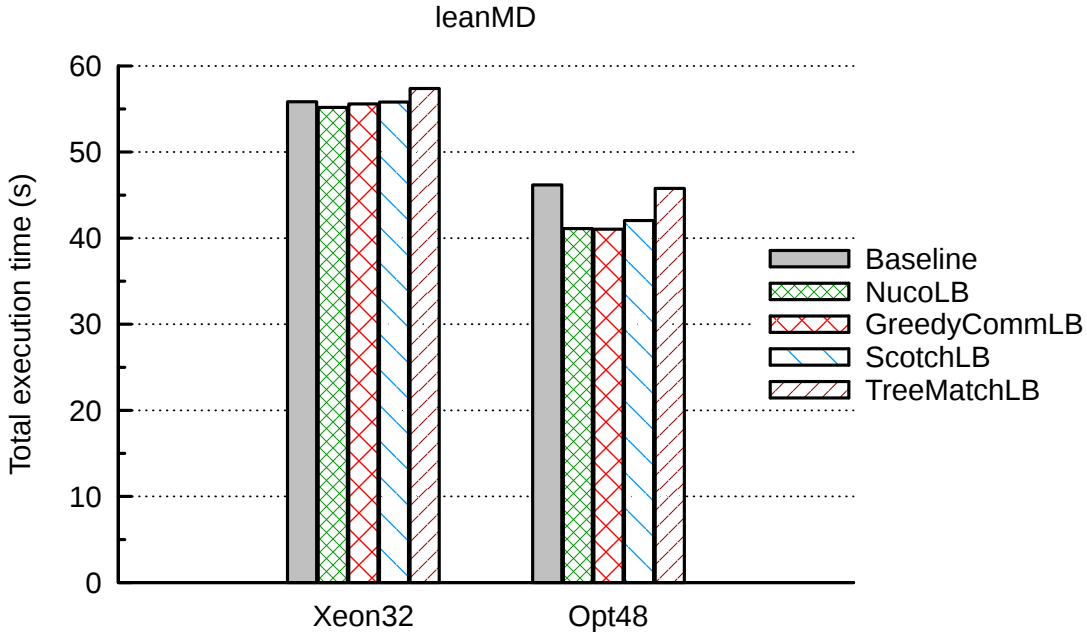

Figure 6.6: Total execution time of *LeanMD* with NUCOLB and other load balancers.

These experiments illustrated how NUCOLB is able to provide performance portability to applications running on different multicore machines, as it reduced (or maintained) the total execution time of applications in all tested scenarios. It was able to reduce PU idleness down to 4%, and to improve the communication costs experienced by the application with a 6% reduction on the average load of each core for *kNeighbor*. In addition, it kept a small load balancing overhead with migrations limited to a maximum of 30% of all application tasks on average. We evaluate how HWTOPOLB behaves in similar scenarios in the next section.

6.2.2 HwTopoLB

This section provides a performance comparison between HwTopoLB and GreedyCommLB, ScotchLB, and RefineCommLB. We present the results of experiments with *kNeighbor*, *stencil4D* and *LeanMD* on machines Xeon24, Xeon32, Opt48, and Opt32.

The total execution times of *kNeighbor* are presented in Figure 6.7. As previously discussed, *kNeighbor* starts in a balanced state, as its tasks have regular loads and communication. For instance, the difference in load between the slowest PU and the average is less than 9% on Xeon32, and 18% on Xeon24. Nonetheless, HwTopoLB was able to reduce the average timestep duration on Xeon24 by 220 ms, and by 160 ms on Opt48. Each time HwTopoLB is called during the execution of *kNeighbor*, it reduces the timestep duration of the benchmark some more. For instance, timesteps take approximately 1.26 s at the beginning of the execution of *kNeighbor* on Xeon24. After the first load balancing call, HwTopoLB reduces the timestep duration to 1.09 s on average. Subsequent calls reduce it to 1.03 s, and 1.01 s. This results in a reduction of 10 seconds in the total execution time, or a speedup of 1.18 over the baseline. In this same scenario, HwTopoLB obtained a speedup of 1.13 over the second best load balancer, RefineCommLB. While RefineCommLB keeps a similar number of tasks per PU, HwTopoLB is able to measure the impact of the communication costs and overloads some PUs with 15% more tasks than RefineCommLB. By using knowledge about the machine topology, HwTopoLB is able to reduce the communication costs felt by the application. This translates in an 11% reduction on the average load of each PU, or 2.7 CPU seconds per timestep.

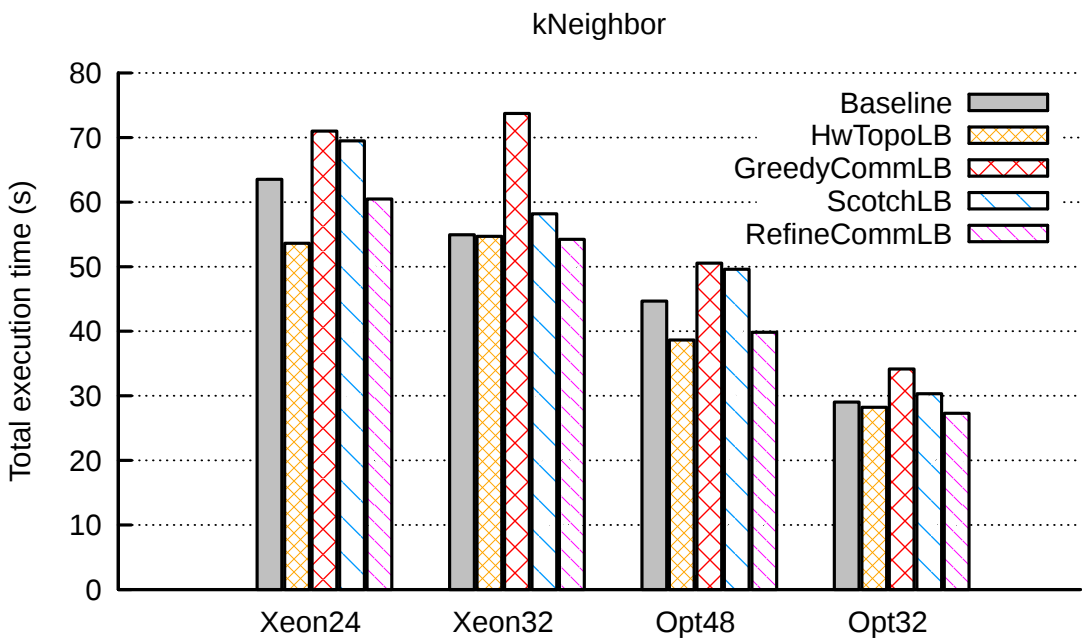

Figure 6.7: Total execution time of *kNeighbor* with HwTopoLB and other load balancers.

Figure 6.7 also shows that no significant performance improvements were achieved on machines Xeon32 and Opt32. In this situation, GreedyCommLB overloads some PUs while trying to improve communication. However, this results in load imbalance. For instance, GreedyCommLB increases the average timestep duration on Xeon32 by 33%.

In this same machine, SCOTCHLB is able to reduce the average timestep duration by 3%. Still, the overall performance of *kNeighbor* is decreased by the time spent remapping tasks at every load balancing call. SCOTCHLB migrates an average of 386 tasks each time it is called. This results in an overhead of 4 seconds in the total execution time of the benchmark. In these same situations, HWTOPOLB provides a low load balancing overhead of 45 ms per load balancing call, and is able to compensate them with a minor performance improvement of 7.4 ms in timestep duration.

The total execution times obtained on the experiments with *stencil4D* are shown in Figure 6.8. The benchmark starts in an imbalanced state due to its original mapping that underloads some PUs, and tasks with different computational loads. We were not able to achieve a 95% confidence interval for the results with GREEDYCOMMLB and SCOTCHLB on Opt48, as the measured execution times show a large variance. The total execution times for GREEDYCOMMLB varied between 74 and 233 seconds for GREEDYCOMMLB, and between 71 and 350 seconds for SCOTCHLB. Still, the maximum execution time measured for HWTOPOLB is of only 52 seconds.

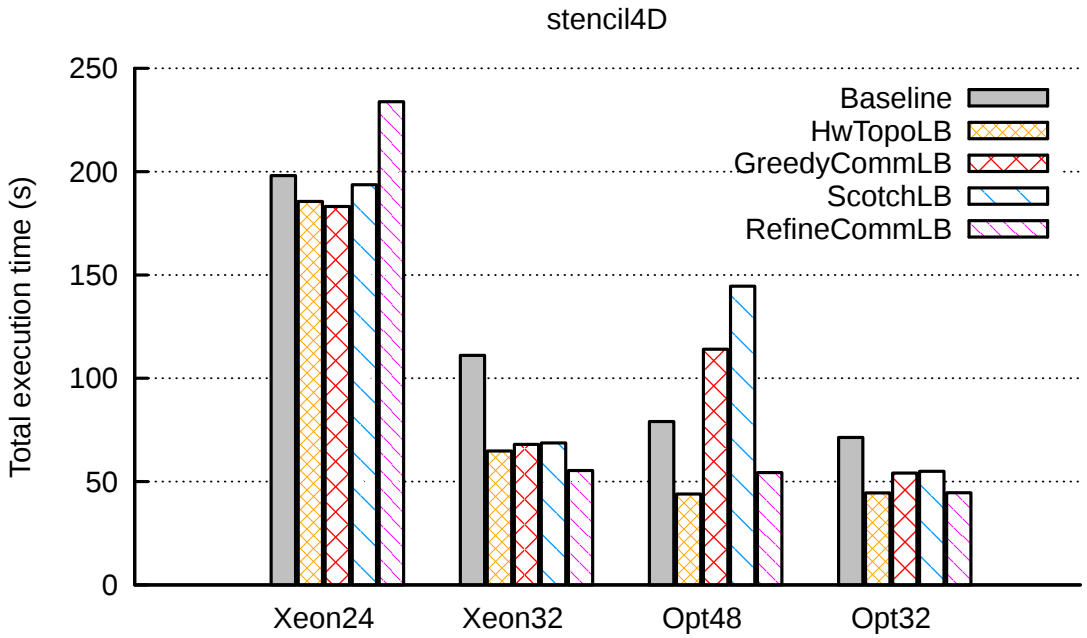

Figure 6.8: Total execution time of *stencil4D* with HWTOPOLB and other load balancers.

HWTOPOLB achieved speedups of approximately 1.8 and 1.71 over the baseline on Opt48 and Xeon32, respectively, and a speedup of 1.23 over the second best load balancer on Opt48. In this same machine, it was able to reduce the average timestep duration of *stencil4D* by 69%. On Xeon32, HWTOPOLB decreases the timestep duration by 57%. After the first load balancing call, the average timestep duration goes from 2.4 s to 1.28 s by correcting the initial load imbalance. An average of 50 tasks are migrated for that. Subsequent load balancing calls refine the performance of *stencil4D* by reducing the timestep duration to 1.03 s, and 0.94 s after the last call. The number of tasks migrated also decreases at each load balancing call, arriving to an average of 14 tasks being migrated in the last one. These results illustrate how HWTOPOLB is able to correct unbalance and refine the work distribution. By doing this, HWTOPOLB is able to provide average speedups of 1.54 over the baseline and 1.40 over the other load balancing algorithms on the four

parallel platforms considered in this experiment.

HWTOPOLB improves the memory locality of *stencil4D* by keeping communicating tasks close together. For instance, it reduced the LLC misses on Opt32 by 7% when compared to the baseline and REFINECOMMLB, and by 40% when compared to GREEDYCOMMLB and SCOTCHLB, as can be seen in Table 6.8. These cache misses are also related to the number of tasks migrated, as data movement incurs in a smaller cache reuse. These two load balancers do not consider the current task mapping, which results in severe migration overheads.

Table 6.8: Average number of LLC misses (in millions) for *stencil4D* with HWTOPOLB and other load balancers on Opt32.

Load balancers	**LLC misses**
Baseline	$7,201$
HWTOPOLB	$6,673$
GREEDYCOMMLB	$10,796$
SCOTCHLB	$10,769$
REFINECOMMLB	$8,374$

The results obtained with load balancing on the molecular dynamics application *LeanMD* are presented in Figure 6.9. For this experiment, the cell array dimensions X, Y, and Z were set to 6, 6, and 5, respectively, resulting in a total of 2700 tasks. It simulates 501 timesteps with load balancing calls after the 20th timestep, and at each 100 timesteps thereafter, for a total of five load balancing calls. Tasks present a small memory footprint, and the cost to migrate them in a shared memory machine is negligible. Differently from previous benchmarks, tasks communicate by multicast extensively. *LeanMD* starts in a balanced state, where the difference between the average and the maximum PU loads varies from 7.5% on Opt48 down to 5.5% on Xeon24. This helps explain why no load balancer was able to achieve performance improvements over 5% in this experiment.

GREEDYCOMMLB was able to achieve the best timestep duration for *LeanMD* by aggressively remapping tasks. The tasks' small memory footprint helps this behavior by providing a small migration overhead. However, the difference between the timestep duration's achieved by HWTOPOLB and GREEDYCOMMLB is less than 3% for all cases, and the difference between total executions times is less than 2%, which is considered to be inside the error margin.

HWTOPOLB's knowledge about the machine topology enables the improvement of *LeanMD*'s memory locality. It reduces LLC misses by 13% on average when compared to the baseline on Xeon24, Opt48 and Opt32, and by 6% when compared to the other load balancers in the same machines. HWTOPOLB also reduces the total PU overhead by 5% on Xeon32 when compared to the baseline. PU overhead comprises all time spent by CHARM++ on activities other than executing tasks, such as managing communication.

The results presented in this section show how HWTOPOLB is able to improve application performance on different scenarios. When considering all benchmarks and parallel platforms, HWTOPOLB showed performance improvements of 23% and 19% on average when compared to the absence of a load balancer and to the other load balancing algorithms, respectively. We compare HWTOPOLB's performance to the one of NUCOLB when load balancing *Ondes3D* in the next section.

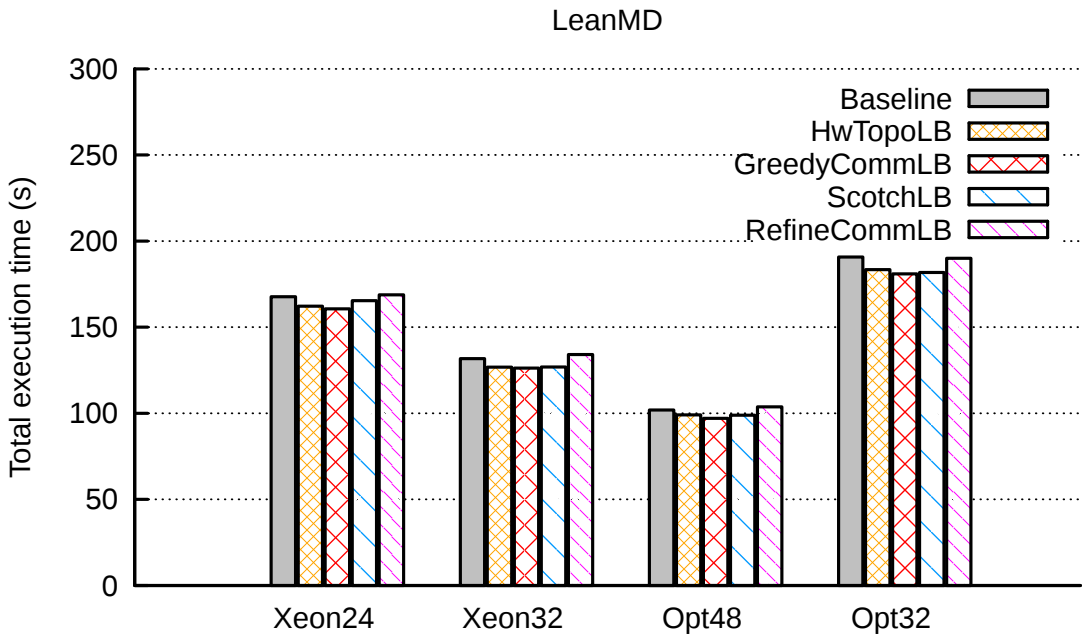

Figure 6.9: Total execution time of *LeanMD* with different load balancers.

6.2.3 NucoLB and HwTopoLB

This section provides a performance comparison between our centralized load balancing algorithms and GreedyLB, RefineCommLB, and ScotchRefineLB on machine Xeon32 using the seismic wave simulator *Ondes3D*. As mentioned in Section 6.1.2, *Ondes3D* was originally implemented using MPI, and was ported to Adaptive MPI (AMPI) to benefit from Charm++'s load balancing framework. As load balancing requires multiple tasks per PU to be effective, we overdecompose the application into 512 tasks (AMPI processes), which represents 16 tasks per PU on this platform. We will refer to this version as *overdecomposed*, or OD. This version does not employ any load balancing algorithm. Meanwhile, our *baseline* refers to a version running one task per PU (32 tasks total) without load balancing.

The overdecomposition of *Ondes3D* comes with an increase in the total load and execution time of the application, as can be seen in Figure 6.10. It shows the average timestep duration and the average PU load for the baseline and the OD versions of *Ondes3D* running for 580 timesteps. The horizontal axis represents the timesteps, while the vertical axis represents time in seconds. Each point represents the average time or load for 20 timesteps. The timestep duration represents the most loaded PU at a timestep, while the average PU load provides a lower bound for the timestep duration. It gives an idea of what would be the optimal timestep duration in a balanced work distribution. As the figure illustrates, the overdecomposed version increases the average load of *Ondes3D* by 13% and the timestep duration by 21%. The latter means that the OD version is more load unbalanced.

Other phenomena can be noticed in Figure 6.10. For instance, the average load and timestep duration of the first 20 timesteps are much larger for the OD version. This happens because the overdecomposition increases *Ondes3D*'s initialization time by 34 seconds. *Ondes3D*'s load dynamicity can be seen between timesteps 120 and 420 where increases on the average PU load and timestep duration happen. While the average PU

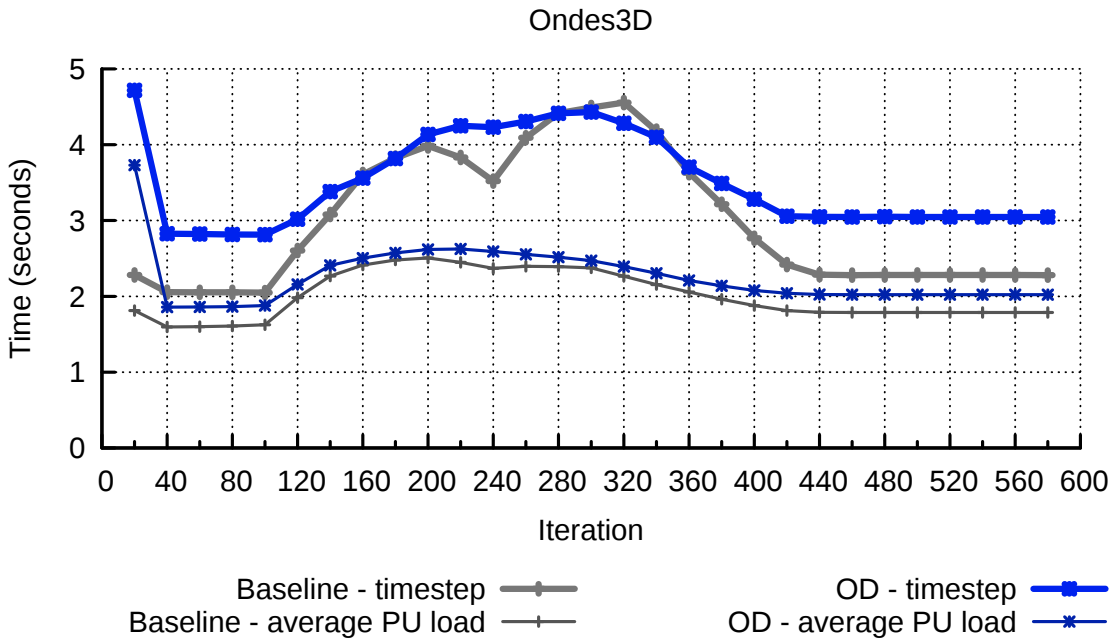

Figure 6.10: Average timestep and average PU load for *Ondes3D* on Xeon32. Values averaged at each 20 timesteps.

load achieves its maximum around timestep 200, the maximum timestep duration can be seen around timestep 320 for the baseline and timestep 300 for the OD version. Still, both stabilize after timestep 420.

The execution times measured for *Ondes3D* with different load balancers can be seen in Table 6.9. The baseline takes approximately 30 minutes to simulate 600 timesteps, while the overdecomposed version of *Ondes3D* increases the execution time by 17.5% to 35 minutes. Still, all load balancers were able to reduce the total execution time of the application by at least 18% when compared to the baseline. NucoLB achieved the best performance in this scenario, with a total execution time of 24 minutes approximately, which results in a speedup of 1.25 over the baseline.

Table 6.9: Total execution time in seconds of *Ondes3D* and speedups for NucoLB, HwTopoLB and other load balancers.

Load balancers	**Execution time**	**Speedup over baseline**	**Speedup over OD**
Baseline	1787 s	—	—
OD	2100 s	—	—
HwTopoLB	1471 s	1.21	1.43
NucoLB	1429 s	1.25	1.47
GreedyLB	1517 s	1.18	1.38
RefineCommLB	1456 s	1.23	1.44
ScotchRefineLB	1494 s	1.20	1.40

The results presented in Table 6.9 are strongly related to the trade-offs of each of the centralized algorithms. For instance, the three most aggressive algorithms consid-

ered, GREEDYLB, SCOTCHREFINELB, and NUCOLB, achieve the best timestep durations as they migrate more tasks to quickly mitigate load imbalance. Their timesteps are between one and two percent better than the ones achieved by REFINECOMMLB and HWTOPOLB. However, the elevated number of task migrations of GREEDYLB and SCOTCHREFINELB (496 and 117 on average, respectively) increases their migration overhead, which results in a longer load balancing time. In this scenario, NUCOLB attains the best equilibrium among the tested load balancers for *Ondes3D*.

Figure 6.11 shows a comparison between the timestep durations attained by HWTOPOLB and NUCOLB to the ones of the baseline, and to the average PU loads for the baseline and the overdecomposed version. Both load balancers are able to achieve timestep durations close to the average PU load of OD during the most static phases of the simulation. It is important to notice that this average PU load represents a limit to the performance gains achievable while overdecomposing *Ondes3D*. During the most load dynamic phase of the simulation (between timesteps 120 and 320), the timestep durations achieved by NUCOLB and HWTOPOLB are 10% to 15% longer, respectively. This performance difference between NUCOLB and HWTOPOLB highlights how the greedy approach of the former is able to handle load dynamicity.

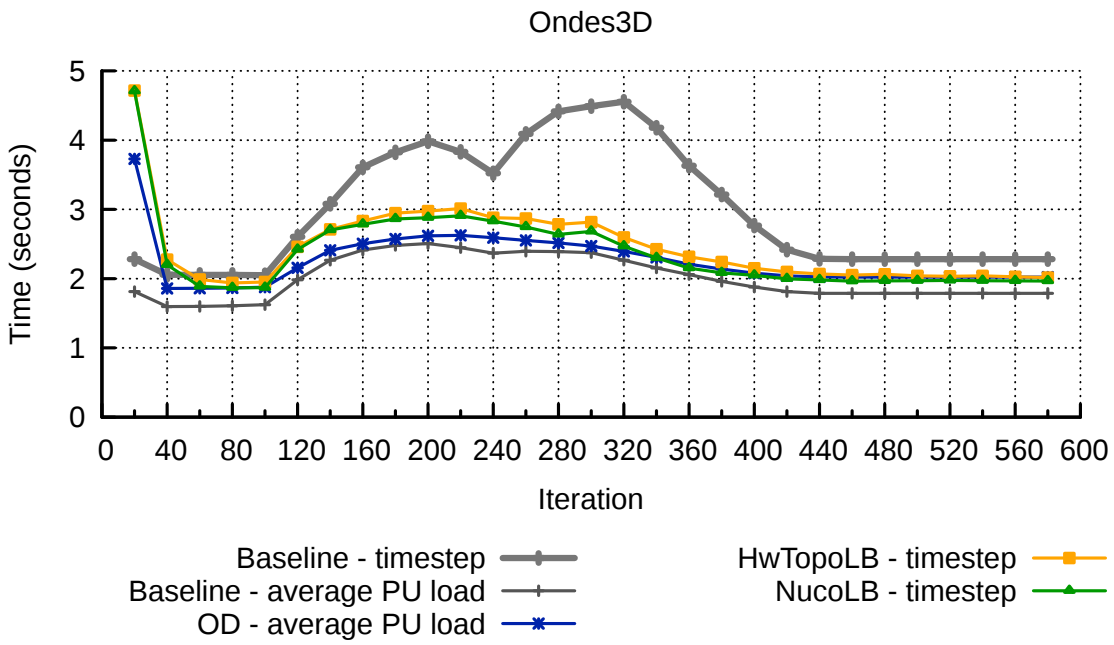

Figure 6.11: Average timestep duration for *Ondes3D* on Xeon32 with HWTOPOLB and NUCOLB. Values averaged at each 20 timesteps.

All results presented so far demonstrate how the proposed centralized topology-aware load balancing algorithms perform on multicore machines. In the next section, we show how our load balancers and others perform on platforms composed of multiple compute nodes.

6.3 Load balancing on clusters

The second part of our performance evaluation focuses on task mapping over parallel platforms composed of multiple compute nodes. These experiments increase in scale

when compared to the ones presented on Section 6.2 by including more PUs and a network level on the machine topology. The two platforms used in this evaluation involve 20 Opt4 CNs, and up to 16 Opt32 CNs, for a total of 80 and 512 PUs, respectively. These larger platforms are employed to assess the scalability of load balancing algorithms, which have to handle more tasks and PUs.

All experiments presented in this section use the molecular dynamics application *LeanMD*. This application was chosen due to its flexibility, as its parameters can be easily varied, and scalability, as it has been show to run on tens of thousands of PUs (KALE et al., 2012).

We organize this section in three parts. Firstly, we present the results obtained with NUCOLB on 20xOpt4. This is followed by the results with HWTOPOLB on up to eight Opt32 CNs. Lastly, we compare scalability of both algorithms to the one of HIERARCHICALLB on up to 16 Opt32 CNs.

6.3.1 NUCOLB

We compare the performance achieved by NUCOLB when load balancing *LeanMD* to the ones of GREEDYCOMMLB, SCOTCHLB, and TREEMATCHLB in Figure 6.12. These results were obtained on 20xOpt4, a cluster composed of 20 compute nodes interconnected through Gigabit Ethernet. This experiment with *LeanMD* involves the same parameters previously presented in Section 6.2.1, which comprises 1875 tasks running for 300 timesteps, with a load balancing call at each 60 timesteps. This results on an average of 23 tasks per PU in this platform, which is much less than what was seen on machines Xeon32 and Opt48.

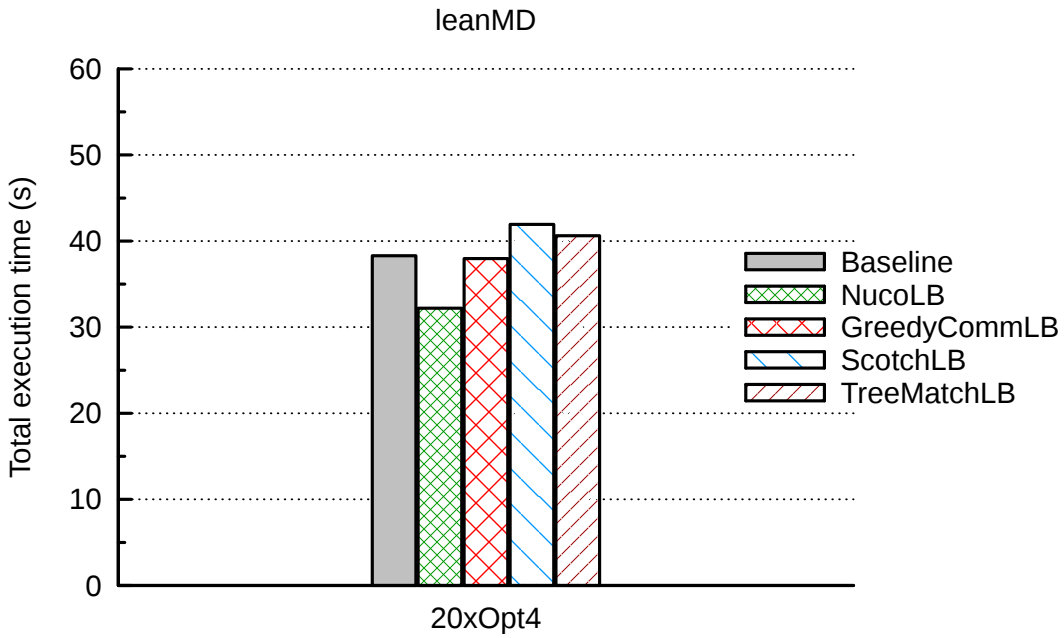

Figure 6.12: Total execution time of *LeanMD* on 20xOpt4.

As illustrated in Figure 6.12, NUCOLB is the load balancer that improves the performance of *LeanMD* the most. It is 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 topology-aware algorithm is able to balance the load over the available PUs

while reducing the communication costs. The timestep duration is reduced by 21% — from 115 ms to 91 ms, as can be seen in Table 6.10. This table presents the average timestep duration and load balancing time for the different load balancers. The load balancing time includes the time spent executing the load balancing algorithm and migrating tasks.

Table 6.10: Average timestep duration and load balancing time (ms) for *LeanMD* with NUCOLB and other load balancers on 20xOpt4.

Load balancers	Average timestep duration	Load balancing time
Baseline	114.94 ms	0.00 ms
NUCOLB	90.64 ms	104.45 ms
GREEDYCOMMLB	111.94 ms	96.05 ms
SCOTCHLB	123.76 ms	144.08 ms
TREEMATCHLB	116.50 ms	355.64 ms

The load balancing decisions of NUCOLB are able to achieve a PU usage of 93% in this configuration, implying an average PU idleness of 7%. The algorithm chooses to keep up to 30% more tasks than the average on some PUs. This leads to a better performance than spreading these tasks, 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 runtime system managing network communication.

NUCOLB migrates approximately 300 tasks 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 tasks. 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 timestep of *LeanMD*, as can be seen in Table 6.10. With a load balancing call after every 60 timesteps, NUCOLB's overhead is easily compensated by the performance improvements that it brings.

6.3.2 HWTOPOLB

This section presents a weak scalability evaluation of HWTOPOLB, GREEDYCOMMLB, SCOTCHLB, and REFINECOMMLB with *LeanMD* over 2, 4, 6, and 8 Opt32 compute nodes. The application parameters were set to vary the number of tasks according to the number of compute nodes. The cell array dimensions Y and Z were set to 10 and 5, respectively, while dimension X is equal to $5N/2$, where N is the number of compute nodes used in the experiment. The number of tasks goes from 3750 for 2 compute nodes to 15000 with 8 compute nodes. All configurations execute 301 timesteps, and include a load balancer call after the 20th timestep, and at each 100 timesteps thereafter, for a total of three load balancing calls.

The total execution times achieved with load balancing are shown in Figure 6.13. HWTOPOLB is the only algorithm able to achieve performance improvements on all situations. These improvements range from a speedup of 1.26 over the baseline with 2 compute nodes (64 PUs), to a speedup of 1.05 with 8 compute nodes (256 PUs).

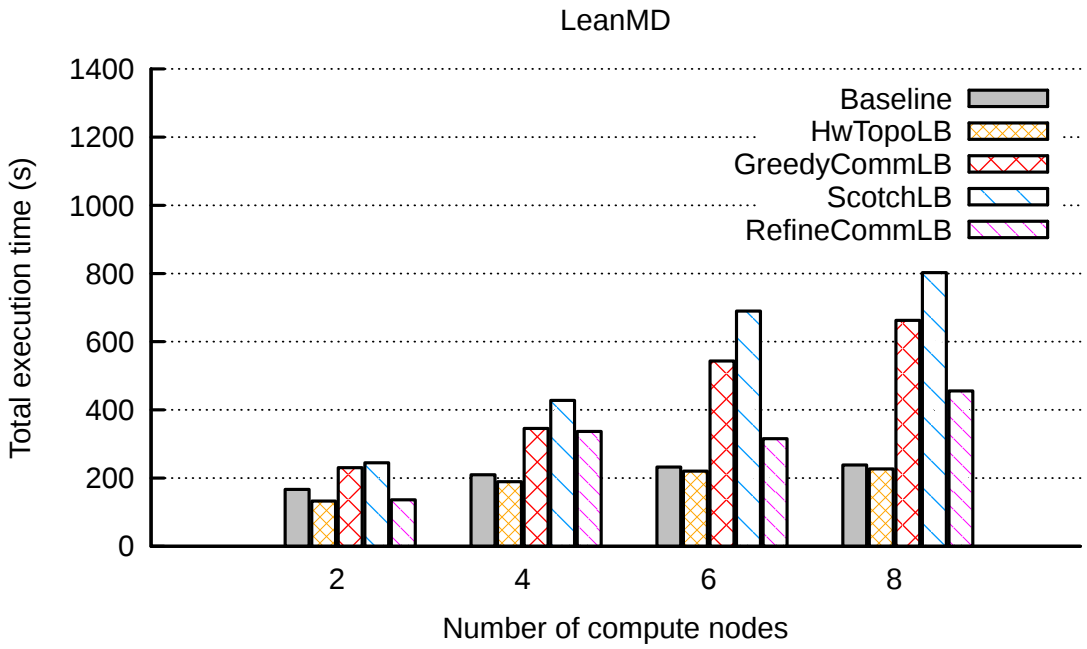

Figure 6.13: Total execution time of *LeanMD* with HwTopoLB and other load balancers on a varying number of Opt32 compute nodes. The problem size is increased with the number of compute nodes.

LeanMD starts in an unbalanced state in these experiments, where the slowest PU is twice as loaded as the average. Additionally, *LeanMD* does not fully exploit the machine topology initially, as no two tasks in the same PU exchange messages in these configurations. As in the situation with *stencil4D* presented in Section 6.2.2, HwTopoLB is able to quickly correct imbalance and improve over its previous mapping decisions. For instance, it decreases the average timestep duration on 2 compute nodes from 447 ms to 323 ms, a reduction of 38%. The average timestep duration's achieved by the different load balancers and their improvements over the baseline are listed in Table 6.11. Values inside parentheses mean a percentual increase in timestep duration (and a decrease in performance). HwTopoLB's timestep duration improvements range from 13% to 38%. Meanwhile, GreedyCommLB and ScotchLB degrade performance by up to 352%. This comes from the lack of knowledge of the machine topology, as they do not profit from the different levels of cache and shared memory inside each compute node. Both algorithms focus on increasing locality only at a PU level by mapping communicating tasks to the same PU. Although they increase the number of messages exchanged among tasks in the same PU, they also increase the number of messages exchanged through the network by 38% on 2 compute nodes, and up to 80% on 8 compute nodes.

When comparing the timestep durations in Table 6.11 and the total execution times presented in Figure 6.13, it can be noticed that there is a large gap between the performance improvements achieved in the first and in the second. For instance, RefineCommLB always improves the average timestep duration, but only improves the total execution time when running on 2 compute nodes. This difference can be explained by two sources of overhead, namely the setup time and the load balancing time.

The setup time of *LeanMD* includes the creation of all tasks, their distribution over different PUs, and the setup of initial simulation parameters. It increases with the size

Table 6.11: Average timestep duration of *LeanMD* with HWTOPOLB and other load balancing algorithms on a varying number of Opt32 compute nodes.

#CNs	Base	HWTOPOLB	GREEDYCOMMLB	SCOTCHLB	REFINECOMMLB
2	447 ms	323 ms	497 ms	429 ms	314 ms
(% vs. Base)	-	38%	(11%)	4%	42%
4	452 ms	385 ms	633 ms	942 ms	396 ms
(% vs. Base)	-	17%	(40%)	(108%)	14%
6	459 ms	398 ms	1174 ms	1576 ms	401 ms
(% vs. Base)	-	15%	(155%)	(243%)	14%
8	472 ms	416 ms	1518 ms	2133 ms	431 ms
(% vs. Base)	-	13%	(222%)	(352%)	9%

of problem and the parallel machine. Although the setup time is negligible in a smaller scale, this is not the case here. The setup times measured for *LeanMD* are 30 s, 66 s, 87 s, and 84 s for an increasing number of compute nodes. When subtracted from the total execution time, the total performance improvements of HWTOPOLB resemble the ones seen for the average timestep duration, with a maximum difference of 5%.

The second source of overhead discussed is load balancing time. It comprises the time load balancing algorithms spend computing a new task mapping. It does not include the overhead of applying a new task mapping, such as the migration time. The average load balancing times for the evaluated algorithms are shown in Table 6.12. HWTOPOLB presents the smallest overhead, spending less than 1 second to compute its decisions at each load balancing call. GREEDYCOMMLB and SCOTCHLB have similar load balancing times. However, REFINECOMMLB takes much longer to compute a new mapping, which compromises the scalability and performance portability achieved by the algorithm. For instance, the time spent by REFINECOMMLB on the three load balancing calls over the execution of *LeanMD* on 8 compute nodes is responsible for half of the total execution time of the application. This explains why the average timestep duration reductions obtained by REFINECOMMLB do not result in performance improvements at this scale.

Table 6.12: Average load balancing time of HWTOPOLB and other load balancing algorithms for *LeanMD* on a varying number of Opt32 compute nodes.

#CNs	HWTOPOLB	GREEDYCOMMLB	SCOTCHLB	REFINECOMMLB
2	0.04 s	0.13 s	0.14 s	2.05 s
4	0.82 s	1.34 s	1.10 s	39.68 s
6	0.54 s	0.73 s	0.64 s	32.93 s
8	0.99 s	1.28 s	1.00 s	76.24 s

These results illustrate how increases in the size of the parallel platform and application impact the performance of centralized load balancing algorithms. Even though HWTOPOLB improves the performance of *LeanMD* on all evaluated scenarios, these gains get smaller as the platform and application increase. This serves as a motivation for the experiments with HIERARCHICALLB presented in the next section.

6.3.3 HIERARCHICALLB

This section presents a comparison between our proposed hierarchical load balancing algorithm HIERARCHICALLB, our centralized algorithms NUCOLB and HWTOPOLB, and another hierarchical algorithm named HYBRIDLB. Other centralized algorithms were not considered in these experiments due to the poor performance they previously showed in Section 6.3.2.

As previously discussed in Section 4.2.1, the operation of HIERARCHICALLB is organized as a tree involving centralized load balancers in two levels. HWTOPOLB is employed to map tasks to CNs at the root level, while both HWTOPOLB and NUCOLB can be used at the leaf level to map tasks to PUs inside each CN. We present results using both versions of the load balancer.

The experimental evaluation of HIERARCHICALLB is split into two parts. The first one focuses on the weak scalability of the algorithms, while the second one tests their strong scalability. The difference between them is that weak scalability experiments increase the size of the problem (in our case, application tasks) when increasing the resources used (number of PUs or CNs), while the strong scalability ones keep the same problem size. Both weak and strong scalability experiments involve running *LeanMD* on 2, 4, 6, 8, 10, 12, 14, and 16 Opt32 compute nodes.

The weak scalability experiments were set to execute *LeanMD* for 501 timesteps, and include a load balancer call after the 40th timestep, and at each 100 timesteps thereafter (timesteps 140, 240, 340, and 440). The cell array dimensions Y and Z were set to 11 and 5, respectively, while dimension X is equal to $2N$, where N is the number of compute nodes used. The number of tasks varies from 6600 for 2 CNs to 52800 with 16 CNs. In this scenario, the total execution times achieved by the different proposed load balancers are presented in Figure 6.14. Results for NUCOLB are presented only for up to 6 CNs due to its increase in execution time.

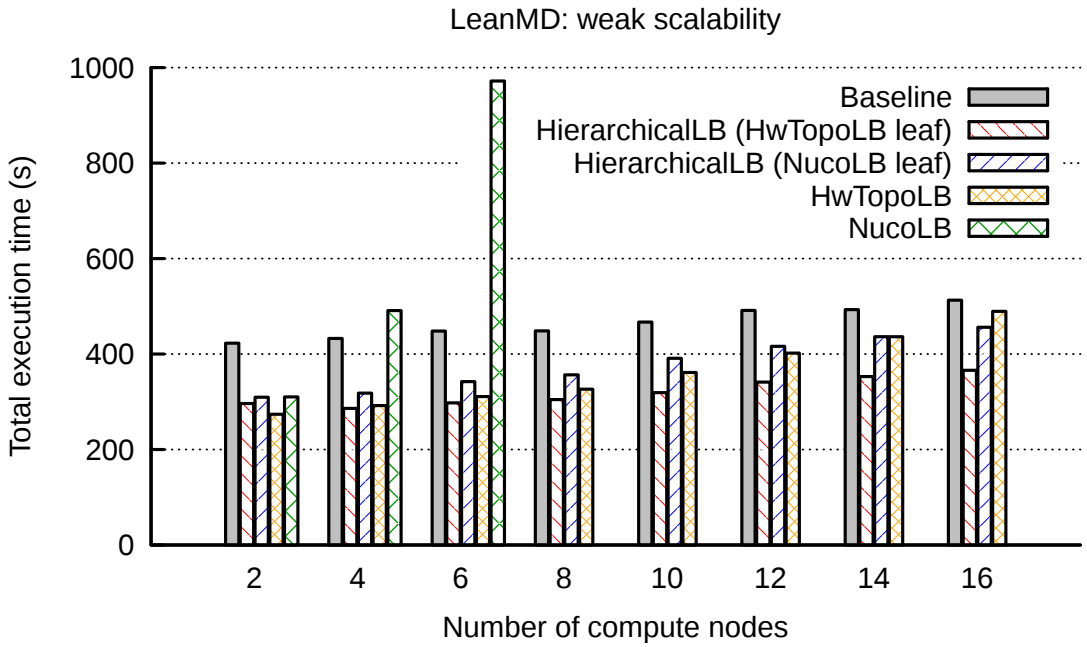

Figure 6.14: Total execution time of *LeanMD* with different load balancers on up to 16 Opt32 CNs. The problem size is increased with the number of compute nodes.

HWTOPOLB achieves the best performance improvement on 2 CNs, with a speedup of 1.54 over the baseline, and a total execution time 8% smaller than the second best load balancer in this scenario, HIERARCHICALLB with HWTOPOLB. Nevertheless, the latter presents the best total execution time for all other configurations tested, with speedups over the baseline varying from 1.51 on 4 CNs to 1.40 on 16 CNs. Meanwhile, HWTOPOLB speedups decrease from 1.48 to 1.04 on the same numbers of compute nodes. This performance difference comes from the time spent load balancing by the two algorithms. The hierarchical approach of HIERARCHICALLB with HWTOPOLB takes an average of 20 s on each load balancing call on 16 CNs, while using only HWTOPOLB in a centralized fashion takes 55 s on average. This overhead is so noticeable that even HIERARCHICALLB with NUCOLB was able to outperform HWTOPOLB on 16 CNs.

NUCOLB showed the worst scalability among the proposed algorithms. Although it achieves better timestep durations than HIERARCHICALLB when running *LeanMD* over 2 CNs, its load balancing time surpasses its benefits. Its average load balancing starts at 9 s for 2 CNs, increasing to 40 s on 4 CNs, and 126 s on 6 CNs. In this last scenario, the total load balancing time surpasses the total execution time of the baseline. Meanwhile, a better scalability is sustained when using NUCOLB as a leaf load balancer for HIERARCHICALLB. Both load balancers present performances on the strong scalability experiments that are similar to these results.

The strong scalability experiments use the same configuration than the weak scalability ones, with the exception of the number of tasks. *LeanMD* was set to execute 46080 tasks coming from a cell array with dimensions $24 \times 16 \times 8$. The total execution times achieved by the different proposed load balancers and the hierarchical load balancer HYBRIDLB are presented in Figure 6.15. Again, results for NUCOLB are presented only for up to 6 CNs due to its increase in execution time. Experiments on 2 and 4 CNs were run more than 10 times, but less than the minimum of 20 runs achieved in all other experiments and configurations. This choice was made due to their long execution times. Still, these results present the same statistical confidence as the other ones.

As shown in Figure 6.15, HYBRIDLB does not improve *LeanMD*'s performance in any of the evaluated configurations. Even though its hierarchical organization reduces its load balancing overhead, the use of GREEDYLB as a leaf load balancer proved to be a problem, as its decisions result in too many migrations (just like GREEDYCOMMLB in previous results), which overcome the benefits of the timestep reductions achieved by the algorithm. Meanwhile, NUCOLB, and HIERARCHICALLB using NUCOLB as a leaf load balancer are the algorithms that present the best performances for *LeanMD* on 2 CNs, with speedups of 1.44 and 1.47 over the baseline, respectively. Still, NUCOLB is not able to keep that performance when we increase the size of the parallel platform. When considering 6 CNs and beyond, HIERARCHICALLB with HWTOPOLB shows the best performance among all examined load balancers. This is clearly illustrated in Figure 6.16, where the speedup over the baseline running on 2 CNs is shown for the different load balancing algorithms and the baseline.

As Figure 6.16 exposes, when HWTOPOLB is used as a leaf load balancer for HIERARCHICALLB, a speedup of 9.4 is achieved on 16 CNs when compared to the baseline on 2 CNs. This performance gain, which is greater than the 8 times increase in resources, is only possible because the baseline is unbalanced. When comparing the performance of the load balancer on 16 CNs to its own on 2 CNs, the speedup is reduced to 7.

Figure 6.16 also shows that the total execution times achieved by HWTOPOLB and HIERARCHICALLB with NUCOLB are very similar between 6 and 14 CNs. There are

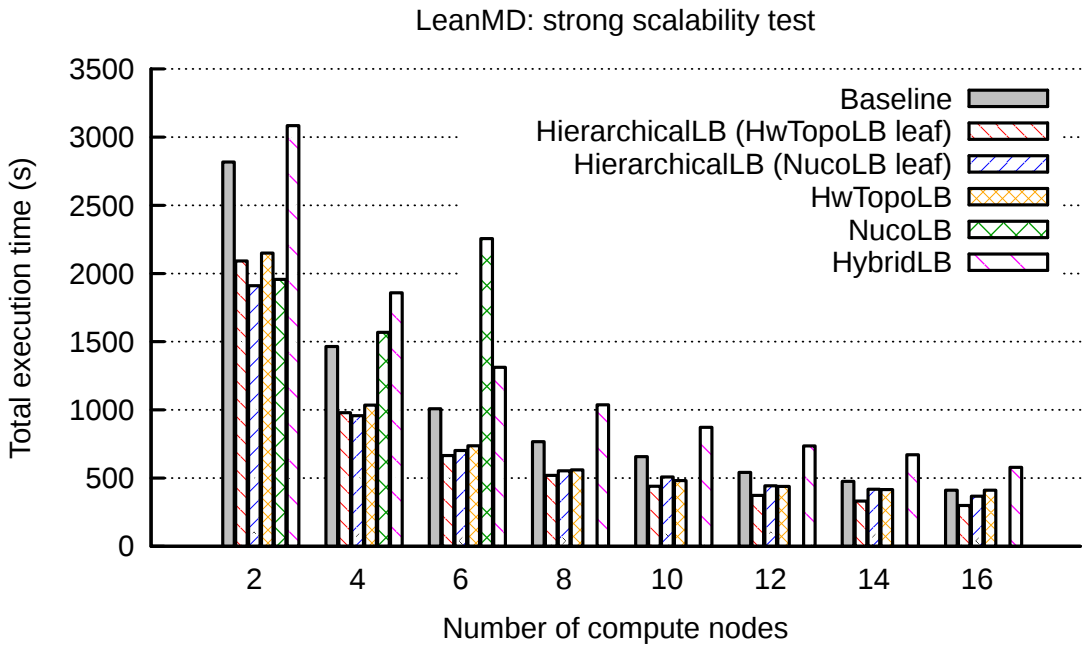

Figure 6.15: Total execution time of *LeanMD* with different load balancers on up to 16 Opt32 CNs. The problem size is the same for all numbers of compute nodes.

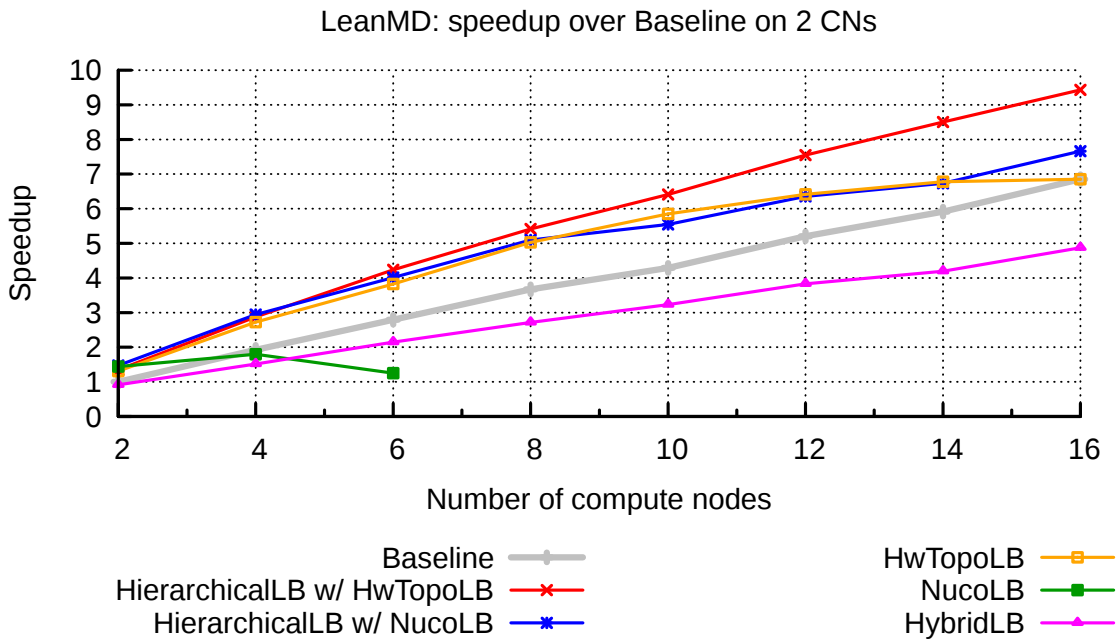

Figure 6.16: Speedup for different load balancers over the execution time of the baseline on 2 compute nodes.

two reasons behind that. The first one is that their load balancing times are similar in several cases, as the time it takes to run HwTopoLB in a centralized way is comparable to running NucoLB on a CN plus the overheads of the hierarchical decisions. The second reason is that by using NucoLB, HierarchicalLB is able to correct load imbalance more quickly. Still, the timestep duration achieved by the hierarchical algorithm is not as

good as the one achieved by the centralized algorithm. Nonetheless, HWTOPOLB takes more load balancing calls to achieve such timestep duration.

The results of both weak and strong scaling experiments show how our hierarchical load balancing algorithm is able to outperform centralized algorithms on larger scale parallel platforms. By using HWTOPOLB as a leaf load balancer for HIERARCHICALLB, we were able to achieve an average speedup of 1.22 over our best centralized algorithm when running *LeanMD* on 10 or more CNs. We discuss more of the benefits of our topology-aware load balancers in the next section.

6.4 Conclusion

In this chapter, we presented an evaluation of our topology-aware load balancing algorithms NUCOLB, HWTOPOLB, and HIERARCHICALLB. We assessed their performance with three benchmarks and two applications with varied characteristics, including load imbalance due to load irregularity and dynamicity, costly communications, different load balancing frequencies, and tasks in different numbers and sizes. Application were run in five parallel platforms with different machine topologies, which involved different numbers of processing units and NUMA nodes. The performance of the proposed algorithms were compared to other seven load balancers implemented with CHARM++, and to a baseline using no load balancer. In this context, our topology-aware load balancers were able to outperform other algorithms in most of the evaluated scenarios.

NUCOLB's greedy approach proved to be able to quickly mitigate load imbalance coming even from load dynamicity. It was able to reduce the PU idleness of some platforms down to 4%, and to improve the communication costs experienced by the application with a 6% reduction on the average load of the communication-intensive benchmark *kNeighbor*. NUCOLB presented a small load balancing overhead in the smaller platforms, with load balancing times similar to other centralized algorithms, but with a maximum number of task migrations limited to 30% of all tasks on average. Still, its load balancing time showed to a be a problem when working with larger scale platforms.

HWTOPOLB presented a better scalability than NUCOLB due to its lighter refinement algorithm and smaller number of task migrations. Although beneficial for scalability, the slower convergence of HWTOPOLB's algorithm makes it less appropriate to handle load dynamicity. HWTOPOLB showed an average performance improvement of 20% when compared to the absence of a load balancer, and of 10% when compared to other centralized algorithms without considering the weak scalability experiments of Section 6.3.2, which would increase this value to 40% due to other load balancers having total execution times larger than the baseline.

Our hierarchical load balancer HIERARCHICALLB showed the best scalability among the evaluated algorithms. It presented an average speedup over the baseline of 1.45 and 1.28 when using HWTOPOLB and NUCOLB as leaf load balancers, respectively. The former achieved performance improvements of 13% on average over HWTOPOLB when considering all tested scenarios, and improvements of 22% on average when considering ten or more compute nodes only. These improvements were achievable by HIERARCHICALLB thanks to its hierarchical design, which enables a smaller load balancing overhead.

In general, performance portability was achieved by our proposed load balancing algorithms in different levels, as they were able to reduce the PU idleness and the total execution times of the applications. For this reduction to happen, they had to handle both

load imbalance and costly communications, while maintaining a small load balancing overhead. Even though the latter proved to be a problem to centralized algorithms when scaling the platform and application, we are able to handle these scenarios by proposing hierarchical algorithms. We discuss the contributions of this thesis in more details in the next chapter.

7 CONCLUSION AND PERSPECTIVES

Modern science requires the use of simulations to better understand and predict complex phenomena. As these scientific applications increase their level of detail, so do increase their demands for computing power, and memory and network resources. These parallel applications are decomposed into tasks that can present complex, many times unpredictable, behaviors. Tasks with irregular or input-dependent loads, and dynamic communication patterns are examples of this. If the affinity between tasks and their load differences are not managed by some mechanism, then application performance will suffer.

Besides the performance obstacles imposed by the behavior of such applications, the HPC platforms where they run bring their own challenges. These parallel platforms composed of multicore compute nodes have hierarchical machine topologies that can present asymmetric and nonuniform communication costs. If ignored, these characteristics can damage the communication performance of an application due to costly communications over the memory and network topologies.

In this context, the employment of a task mapping algorithm becomes crucial to achieve scalable performances with applications over different parallel platforms. Such algorithms have to manage the distribution of tasks to mitigate the effects of load imbalance and costly communications, while introducing a low overhead to the total execution time of the application. Considering this, the main objective of this thesis was to provide performance portability and scalability to complex scientific applications running over hierarchical multicore parallel platforms.

The contributions of this thesis were centered on the hypothesis that precise machine topology information can help task mapping algorithms in their decisions. In this sense, our research was focused on organizing a detailed machine topology model of hierarchical platforms, and developing topology-aware load balancing algorithms that make use of it.

7.1 Contributions

In order to capture relevant information of parallel platforms composed of multicore compute nodes, we proposed a generic and unified machine topology model in Chapter 3, and implemented it as the HIESCHELLA library. Our model is able to expose asymmetry and nonuniformity at different topology levels, and represents communication costs as latencies and bandwidths. All communication costs are obtained by profiling both the memory and network hierarchy of the platform. Our approach is kept generic by using tools and benchmarks independent of application and system architecture, and can be used by multiple algorithms.

Based on our machine topology model, we proposed three different topology-

aware load balancing algorithms on Chapter 4, named NUCOLB (PILLA et al., 2012), HWTOPOLB (PILLA et al., 2012, 2014), and HIERARCHICALLB. They combine our topology model with application information gathered during execution time. NUCOLB is a refinement-based, greedy, list scheduling algorithm that focuses on the nonuniform aspects of parallel platforms. It employs an aggressive strategy to fix imbalance, which is effective for applications with dynamic behaviors, but can result in a larger load balancing overhead. Meanwhile, HWTOPOLB considers the whole machine topology in its decisions, and prioritizes moving tasks away from the most loaded processing unit in order to minimize the makespan. It resorts to a lighter strategy than can lead to a slower convergence to a task mapping with a low PU idleness, but is proved to asymptotically converge to an optimal solution. These two centralized load balancing algorithms have a complete view of the application and platform, which allows them to make better task mapping decisions at the cost of a longer execution time on larger platforms. On the other hand, our hierarchical algorithm HIERARCHICALLB is organized as a tree, splitting and parallelizing parts of the scheduling decisions. This was done to reduce its load balancing overhead. It benefits from our machine topology model in two moments by splitting the scheduling tree based on it, and by being able to use our centralized topology-aware algorithms for load balancing. Our three load balancing algorithms were implemented at runtime level using the CHARM++ runtime system, which kept them unattached to a specific application or architecture.

We evaluated our proposed topology-aware load balancing algorithms over five different multicore parallel platforms with three benchmarks and two applications, and compared their performances to other seven load balancers implemented with CHARM++ on Chapter 6. Experimental results with NUCOLB showed performance improvements of up to 19% over state of the art load balancers on different platforms composed of NUMA compute nodes. This performance was obtained while migrating a maximum of 30% of available tasks on average, which results in a migration overhead up to 11 times smaller than other load balancers. NUCOLB was able to reduce PU idleness down to 4%, and to reduce the communication costs experienced by an application (a 6% reduction on the average load of each PU for *kNeighbor*). Nevertheless, its load balancing time showed to a be a problem when working with larger scale platforms. Meanwhile, HWTOPOLB showed that our load balancing approach improves application performance by 20% on average when compared with the absence of a load balancer. Finally, our scalability experiments on a parallel system composed of multiple compute nodes demonstrated that HIERARCHICALLB is able to surpass other algorithms in scalability while improving application performance. Thanks to its hierarchical design, it showed performance improvements of 13% on average over HWTOPOLB when considering all tested scenarios, and improvements of 22% on average when considering ten or more compute nodes. In general, our topology-aware load balancers outperformed other algorithms in most of the evaluated scenarios.

The results presented in this thesis enforced that precise machine topology information helps task mapping algorithms to provide performance portability and scalability to complex scientific applications running over hierarchical multicore parallel platforms, as our topology-aware load balancers were able to reduce PU idleness and the total execution time of applications.

7.2 Perspectives

The research presented in this thesis can be extended in several ways, some of which are listed below.

- **Algorithms' development and spread of topology-awareness**: As parallel platforms grow in size, complexity, and heterogeneity, the impact that knowing the machine topology will have on performance will tend to grow too. Since our machine topology model is made available through an application and platform-independent library, we believe that it could be employed to provide detailed communication cost information to task mapping algorithms on current and future platforms. This would lead to the research and development of new work stealing algorithms, operating system scheduling algorithms, algorithms specific to legacy scientific applications, distributed load balancers, and more.

- **Automate the selection of scheduling algorithms based on application and machine information**: The results from our experimental evaluations with load balancing reinforce that different heuristics are better suited to different scenarios. In this sense, a research challenge lies on finding the most effective algorithm to balance the load of an application given its characteristics and current behavior. As this can be difficult for a user to decide beforehand, we believe that an automatic mechanism to select scheduling algorithms would both ease the burdens of users and developers of complex applications, and benefit from available scheduling algorithms. One alternative would be to use machine learning techniques to automatically select the best load balancing algorithm given an application and a parallel platform (whose detailed information could be obtained from our machine topology model).

- **Extend algorithms to focus on energy and other metrics**: The main objective of the load balancing algorithms proposed in this thesis was to reduce the total execution time of an application, or reduce PU idleness. Although an important objective, we see that current and future platforms shall struggle with energy and power constrains, such as is documented for Exascale platforms (KOGGE et al., 2008), or embedded parallel processors. In this sense, we believe that a research perspective lies on the development of new algorithms and the extension of our scheduling algorithms to consider energy and power constrains, or focus on reducing the total energy or power consumption of parallel applications running on hierarchical platforms.

REFERENCES

ALÓS-FERRER, C.; NETZER, N. The logit-response dynamics. **Games and Economic Behavior**, [S.l.], v.68, n.2, p.413–427, 2010.

An Introduction to the Intel QuickPath Interconnect. [S.l.]: Intel Corporation, 2009.

AOCHI, H. et al. Finite Difference Simulations of Seismic Wave Propagation for the 2007 Mw 6.6 Niigata-ken Chuetsu-Oki Earthquake: validity of models and reliable input ground motion in the near-field. **Pure and Applied Geophysics**, [S.l.], v.170, n.1-2, p.43–64, 2013.

BHATELE, A. et al. **NAMD: a portable and highly scalable program for biomolecular simulations**. [S.l.]: Department of Computer Science, University of Illinois at Urbana-Champaign, 2009. (UIUCDCS-R-2009-3034).

BHATELE, A.; KALE, L. V.; KUMAR, S. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In: CONFERENCE ON SUPERCOMPUTING (ICS 2009), 23., New York, NY, USA. **Proceedings...** ACM, 2009. p.110–116.

BLACKJACK. **Compiler Metrics and Evaluation**. `http://icl.cs.utk.edu/blackjack/`.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **J. ACM**, New York, NY, USA, v.46, n.5, p.720–748, Sept. 1999.

BOLZE, R. et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. **International Journal of High Performance Computing Applications**, [S.l.], v.20, n.4, p.481–494, 2006.

BOMAN, E. et al. **Zoltan home page**. `http://www.cs.sandia.gov/Zoltan`.

BRAMS. **Brazilian developments on the Regional Atmospheric Modelling System**. `http://brams.cptec.inpe.br/`.

BREMAUD, P. **Markov chains: gibbs fields, monte carlo simulation, and queues**. [S.l.]: Springer, 1999. v.31.

BROQUEDIS, F. et al. hwloc: a generic framework for managing hardware affinities in hpc applications. In: PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2010 18TH EUROMICRO INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2010. p.180–186.

BROQUEDIS, F. et al. Structuring the execution of OpenMP applications for multicore architectures. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING (IPDPS 2010). **Proceedings...** IEEE Computer Society, 2010. p.1–10.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.14, p.141–154, Feb. 1988.

CASTRO, M. **Improving the Performance of Transactional Memory Applications on Multicores: a machine learning-based approach**. 2012. 208p. Ph.D. Thesis — Université de Grenoble, Grenoble.

CATALYUREK, U. V. et al. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2007. IPDPS 2007. IEEE INTERNATIONAL. **Proceedings...** [S.l.: s.n.], 2007. p.1–11.

CHARM++. **Parallel Programming Laboratory**. `http://charm.cs.illinois.edu/`.

CHEN, H. et al. MPIPP: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In: SUPERCOMPUTING, 20., New York, NY, USA. **Proceedings...** ACM, 2006. p.353–360. (ICS '06).

CONCEPTUAL. **A Network Correctness and Performance Testing Language**. `http://www.ccs3.lanl.gov/~pakin/software/conceptual/`.

CRUZ, E. H. M.; DIENER, M.; NAVAUX. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. In: PARALLEL DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), 2012 IEEE 26TH INTERNATIONAL. **Proceedings...** [S.l.: s.n.], 2012. p.532–543.

CULLER, D. et al. LogP: towards a realistic model of parallel computation. **SIGPLAN Not.**, New York, NY, USA, v.28, n.7, p.1–12, July 1993.

DAGUM, L.; MENON, R. OpenMP: an industry standard api for shared-memory programming. **Computational Science & Engineering, IEEE**, [S.l.], v.5, n.1, p.46–55, 2002.

DANALIS, A. et al. BlackjackBench: portable hardware characterization with automated results analysis. **To be published in The Computer Journal**, [S.l.], p.1–13, 2012.

DEVINE, K. et al. Zoltan Data Management Services for Parallel Dynamic Applications. **Computing in Science and Engineering**, [S.l.], v.4, n.2, p.90–97, 2002.

DIENER, M.; CRUZ, E. H. M.; NAVAUX. Communication-Based Mapping Using Shared Pages. In: IEEE INTERNATIONAL PARALLEL DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.: s.n.], 2013.

DINAN, J. et al. Scalable work stealing. In: CONFERENCE ON HIGH PERFORMANCE COMPUTING NETWORKING, STORAGE AND ANALYSIS, New York, NY, USA. **Proceedings...** ACM, 2009. (SC '09).

DONGARRA, J.; MEUER, H.; STROHMAIER, E. **TOP500 Supercomputer Sites: november 2013**. 2013.

DUPROS, F. et al. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In: COMPUTATIONAL SCIENCE AND ENGINEERING, 2008. CSE '08. 11TH IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2008. p.253–260.

DUPROS, F. et al. High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v.36, n.5-6, p.308–325, June 2010.

FRANCESQUINI, E.; GOLDMAN, A.; MEHAUT, J.-F. A NUMA-Aware Runtime Environment for the Actor Model. In: PROCEEDINGS OF THE 42ND INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP 2013, Lyon, France. **Proceedings...** [S.l.: s.n.], 2013. p.250–259.

FRASCA, M.; MADDURI, K.; RAGHAVAN, P. NUMA-aware graph mining techniques for performance and energy efficiency. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 2012. (SC '12).

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. **SIGPLAN Not.**, New York, NY, USA, v.33, n.5, p.212–223, May 1998.

GAUTIER, T.; BESSERON, X.; PIGEON, L. Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PARALLEL SYMBOLIC COMPUTATION, 2007. **Proceedings...** ACM, 2007. p.23.

GONZALEZ-DOMINGUEZ, J. et al. Servet: a benchmark suite for autotuning on multicore clusters. In: PARALLEL DISTRIBUTED PROCESSING (IPDPS), 2010 IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.: s.n.], 2010. p.1–9.

GONZALEZ, T.; IBARRA, O. H.; SAHNI, S. Bounds for LPT schedules on uniform processors. **SIAM Journal on Computing**, [S.l.], v.6, n.1, p.155–166, 1977.

GROPP, W.; LUSK, E.; SKJELLUM, A. Using MPI: portable parallel programming with the message-passing interface. **MIT Press Cambridge, MA, USA**, [S.l.], p.371, 1999.

HERMANN, E. et al. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In: EURO-PAR CONFERENCE ON PARALLEL PROCESSING: PART II, 16., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.235–246. (Euro-Par'10).

HIESCHELLA. **Hierarchical Scheduling for Large Scale Architectures**. `http://forge.imag.fr/projects/hieschella/`.

HOEFLER, T. et al. Netgauge: a network performance measurement framework. In: HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, HPCC'07. **Proceedings...** Springer, 2007. v.4782, p.659–671.

HOEFLER, T.; SCHNEIDER, T. Runtime detection and optimization of collective communication patterns. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 21., New York, NY, USA. **Proceedings...** ACM, 2012. p.263–272. (PACT '12).

HOEFLER, T.; SNIR, M. Generic topology mapping strategies for large-scale parallel architectures. In: SUPERCOMPUTING, New York, NY, USA. **Proceedings...** ACM, 2011. (ICS '11).

HOFMEYR, S. et al. Juggle: proactive load balancing on multicore computers. In: HIGH PERFORMANCE DISTRIBUTED COMPUTING, 20., New York, NY, USA. **Proceedings...** ACM, 2011. p.3–14. (HPDC '11).

HUANG, C.; LAWLOR, O.; KALE, L. V. Adaptive mpi. **Lecture notes in computer science**, [S.l.], 2004.

HWLOC. **Portable Hardware Locality**. `http://www.open-mpi.org/projects/hwloc/`.

JEANNOT, E.; MERCIER, G. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In: D'AMBRA, P.; GUARRACINO, M.; TALIA, D. (Ed.). **Euro-Par 2010 - Parallel Processing**. [S.l.]: Springer Berlin / Heidelberg, 2010. p.199–210. (Lecture Notes in Computer Science, v.6272).

KALE, L. et al. **Migratable Objects + Active Messages + Adaptive Runtime = Productivity + Performance A Submission to 2012 HPC Class II Challenge**. [S.l.]: Parallel Programming Laboratory, 2012. (12-47).

KALE, L. V.; KRISHNAN, S. Charm++: a portable concurrent object oriented system based on c++. In: EIGHTH ANNUAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA 1993). **Proceedings...** [S.l.: s.n.], 1993. p.91–108.

KALE, L. V.; SINHA, A. Projections: a preliminary performance tool for charm. In: PARALLEL SYSTEMS FAIR, INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, Newport Beach, CA. **Proceedings...** [S.l.: s.n.], 1993. p.108–114.

KARYPIS, G.; KUMAR, V. METIS: unstructured graph partitioning and sparse matrix ordering system. **The University of Minnesota**, [S.l.], v.2, 1995.

KIM, C.; BURGER, D.; KECKLER, S. W. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.36, n.5, p.211–222, Oct. 2002.

KOGGE, P. et al. Exascale computing study: technology challenges in achieving exascale systems. **DARPA Information Processing Techniques Office, Washington, DC**, [S.l.], p.278, 2008.

KOMATITSCH, D.; MARTIN, R. An unsplit convolutional perfectly matched layer improved at grazing incidence for the seismic wave equation. **GEOPHYSICS**, [S.l.], v.72, n.5, p.SM155–SM167, 2007.

KUMAR, V.; GRAMA, A. Y.; VEMPATY, N. R. Scalable load balancing techniques for parallel computers. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.22, n.1, p.60–79, July 1994.

LEUNG, J. Y. T. **Handbook of scheduling: algorithms, models, and performance analysis**. [S.l.]: Chapman & Hall/CRC, 2004. (Chapman & Hall/CRC computer and information science series).

LIFFLANDER, J.; KRISHNAMOORTHY, S.; KALE, L. V. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In: HIGH-PERFORMANCE PARALLEL AND DISTRIBUTED COMPUTING, 21., New York, NY, USA. **Proceedings...** ACM, 2012. (HPDC '12).

LIKWID. **Lightweight performance tools**. `http://code.google.com/p/likwid`.

LMBENCH. **Tools for Performance Analysis**. `http://lmbench.sourceforge.net/`.

LOF, H.; HOLMGREN, S. affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-numa system. In: SUPERCOMPUTING, 19., New York, NY, USA. **Proceedings...** ACM, 2005. p.387–392. (ICS '05).

MCCALPIN, J. D. Memory Bandwidth and Machine Balance in Current High Performance Computers. **IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter**, [S.l.], p.19–25, Dec. 1995.

MENON, H. et al. Automated Load Balancing Invocation based on Application Characteristics. In: IEEE CLUSTER 12, Beijing, China. **Proceedings...** [S.l.: s.n.], 2012.

MERCIER, G.; CLET-ORTEGA, J. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In: ROPO, M.; WESTERHOLM, J.; DONGARRA, J. (Ed.). **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. [S.l.]: Springer Berlin / Heidelberg, 2009. p.104–115. (Lecture Notes in Computer Science, v.5759).

METIS. **Family of Graph and Hypergraph Partitioning Software**. `http://glaros.dtc.umn.edu/gkhome/views/metis/`.

NELSON, M. et al. NAMD - a Parallel, Object-Oriented Molecular Dynamics Program. **International Journal of High Performance Computing Applications**, [S.l.], v.10, n.4, p.251–268, 1996.

NETGAUGE. **A Network Performance Measurement Toolkit**. `http://www.unixer.de/research/netgauge/`.

OLIVIER, S. L. et al. Scheduling task parallelism on multi-socket multicore systems. In: INTERNATIONAL WORKSHOP ON RUNTIME AND OPERATING SYSTEMS FOR SUPERCOMPUTERS, 1., New York, NY, USA. **Proceedings...** ACM, 2011. p.49–56. (ROSS '11).

PAKIN, S. The Design and Implementation of a Domain-Specific Language for Network Performance Testing. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.18, p.1436–1449, 2007.

PELLEGRINI, F.; ROMAN, J. 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). **Proceedings...** [S.l.: s.n.], 1996. p.493–498.

PILLA, L. L. et al. A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems. In: PARALLEL PROCESSING (ICPP), 2012 41ST INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2012. p.118–127.

PILLA, L. L. et al. Asymptotically Optimal Load Balancing for Hierarchical Multi-Core Systems. In: PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2012 IEEE 18TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2012. p.236–243.

PILLA, L. L. et al. A Topology-Aware Load Balancing Algorithm for Clustered Hierarchical Multi-Core Machines. **Future Generation Computer Systems**, [S.l.], v.30, n.0, p.191–201, Jan. 2014.

QUINTIN, J.-N.; WAGNER, F. Hierarchical work-stealing. In: EURO-PAR CONFERENCE ON PARALLEL PROCESSING: PART I, 16., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.217–229. (EuroPar'10).

RIBEIRO, C. P. **Contributions on Memory Affinity Management for Hierarchical Shared Memory Multi-core Platforms**. 2011. Ph.D. Thesis — University of Grenoble.

RIGHI, R. et al. Observing the Impact of Multiple Metrics and Runtime Adaptations on BSP Process Rescheduling. **Parallel Processing Letters**, [S.l.], v.20, n.2, p.123–144, June 2010.

RODRIGUES, E. R. et al. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. **Computer Architecture and High Performance Computing, Symposium on**, Los Alamitos, CA, USA, v.0, p.71–78, 2010.

SCOTCH. **Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordening Package**. `http://www.labri.fr/perso/pelegrin/scotch/`.

SERVET. **The Servet Benchmark Suite Homepage**. `http://servet.des.udc.es/`.

SGI UV 2000 System User Guide. [S.l.]: Silicon Graphics, 2012.

STAELIN, C. lmbench: portable tools for performance analysis. In: IN USENIX ANNUAL TECHNICAL CONFERENCE. **Proceedings...** [S.l.: s.n.], 1996.

STAT perf. **Linux man page**. `http://linux.die.net/man/1/perf-stat`.

TANENBAUM, A. S. **Modern operating systems**. [S.l.]: Prentice Hall Englewood Cliffs, 2008. v.4.

TCHIBOUKDJIAN, M. et al. A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores. **Proceedings of the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)**, [S.l.], p.1–10, 2010.

TESSER, R. et al. Using Dynamic Load Balancing to Improve the Performance of Seismic Wave Simulations. In: TO BE PUBLISHED ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2014 22ST EUROMICRO INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2014. p.1–8.

THIBAULT, S.; NAMYST, R.; WACRENIER, P.-A. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the bubblesched framework. In: KERMARREC, A.-M.; BOUGé, L.; PRIOL, T. (Ed.). **Euro-Par 2007 Parallel Processing**. [S.l.]: Springer Berlin Heidelberg, 2007. p.42–51. (Lecture Notes in Computer Science, v.4641).

TREIBIG, J.; HAGER, G.; WELLEIN, G. LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS, 2010., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.207–216. (ICPPW '10).

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, New York, NY, USA, v.33, n.8, p.103–111, Aug. 1990.

WALKO, R. et al. Coupled atmosphere-biophysics-hydrology models for environmental modeling. **Journal of applied meteorology**, [S.l.], v.39, n.6, p.931–944, June 2000.

WULF, W.; MCKEE, S. A. Hitting the memory wall: implications of the obvious. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.23, p.20–24, Mar. 1995.

XUE, M.; DROEGEMEIER, K.; WEBER, D. Numerical prediction of high-impact local weather: a driver for petascale computing. **Petascale Computing: Algorithms and Applications**, [S.l.], p.103–124, 2007.

ZHENG, G. **Achieving high performance on extremely large parallel machines: performance prediction and load balancing**. 2005. Ph.D. Thesis — Department of Computer Science, University of Illinois at Urbana-Champaign.

ZHENG, G. et al. Periodic Hierarchical Load Balancing for Large Supercomputers. **International Journal of High Performance Computing Applications (IJHPCA)**, [S.l.], Mar. 2011.