A Simulation Workflow to Evaluate the Performance of Dynamic Load Balancing with Over-decomposition for Iterative Parallel Applications

Rafael Keller Tesser

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

RAFAEL KELLER TESSER

A Simulation Workflow to Evaluate the Performance of Dynamic Load Balancing with Over-decomposition for Iterative Parallel Applications

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

Advisor:
Prof. Dr. Philippe Olivier Alexandre Navaux

PhD Internship Advisor:
Prof. Dr. Arnaud Legrand

Porto Alegre
April 2018

CIP — CATALOGING-IN-PUBLICATION

Keller Tesser, Rafael

A Simulation Workflow to Evaluate the Performance of Dynamic Load Balancing with Over-decomposition for Iterative Parallel Applications / Rafael Keller Tesser. – Porto Alegre: PPGC da UFRGS, 2018.

114 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Philippe Olivier Alexandre Navaux
; PhD Internship Advisor: Arnaud Legrand.

1. Parallel computing. 2. High-performance computing. 3. Performance evaluation. 4. Dynamic load balancing. 5. Over-decomposition. 6. Simulation of distributed systems. 7. Iterative applications. 8. SimGrid. 9. AMPI. 10. Charm++. I. Olivier Alexandre Navaux
, Philippe. II. Legrand, Arnaud. III. Título.

ACKNOWLEDGMENTS

First of all, I would like to thank Prof. Philippe Olivier Alexandre Navaux, my PhD advisor, who has been my advisor since my masters. Thanks for believing in me, and picking me up as a PhD student, even after a somewhat troubled masters, and for the support during the five years of development of this thesis.

I would also like to thank Prof. Arnaud Legrand, who was my advisor on a Sandwich PhD Internship at *INRIA Grenoble*, as a member of the *Laboratoire d'Informatique de Grenoble*, in the context of the *LICIA* international cooperation project. A big portion of my work on load balancing simulation was done during this internship. Even after it ended, we continued our collaboration, resulting in the publication of a paper at Euro-Par 2017, and a submission to a research journal.

I am also grateful for the collaborators I had in several stages of this PhD. At the start of my PhD, my then colleague Laércio Lima Pilla collaborated on my paper on the benefits of dynamic load balancing applied to Ondes3D. In this same work I started a collaboration with Fabrice Dupros, who at the time worked at the *French Geological Survey* (BRGM). I also would like to thank Fabrice for receiving me for a short visit to BRGM, to work on Ondes3D related developments and on experiments using machines owned by BRGM. Besides that, Fabrice is a co-author on all papers related to this PhD.

I thank Prof. Lucas Mello Schnorr, from UFRGS, for his essential collaboration on the presented in this thesis. He was very helpfull in several stages of the process, and is my co-author on two papers on the simulation workflow proposed in this thesis.

I can't forget to thank Prof. Jean-François Méhaut, with whom I collaborated in the be the beginning of my PhD, as a potential internship advisor. Thank you for receiving me twice in Grenoble, and for referring me to Prof. Arnaud Legrand, when I decided to take the simulation route with my work.

I also would like to thank the members of the jury, Prof. Edson Norberto Cáceres, Prof. Márcio Bastos Castro, and Prof. Nicolas Maillard, for accepting our invitation to participate on my PhD defense.

Many thanks to my colleagues of the *Group of Parallel and Distributed Processing* (GPPD), for the companionship and patience.

Last, but not least, I would like to thank my family, especially my father and my sister, for always being there for me. In all my time in academia, they have never failed to believe in my capacity, and to provide the emotional support whenever I needed it.

ABSTRACT

In this thesis we present a novel simulation workflow to evaluate the performance of dynamic load balancing with over-decomposition applied to iterative parallel applications at low-cost. Its goals are to perform such evaluation with minimal application modification and at a low cost in terms of time and of resource requirements. Many parallel applications suffer from dynamic (temporal) load imbalance that can not be treated at the application level. It may be caused by intrinsic characteristics of the application or by external software and hardware factors. As demonstrated in this thesis, such dynamic imbalance can be found even in applications whose codes do not hint at any dynamism. Therefore, we need to rely on runtime dynamic load balancing mechanisms, such as dynamic load balancing based on over-decomposition. The problem is that evaluating and tuning the performance of such technique can be costly. This usually entails modifications to the application and a large number of executions to get statistically sound performance measurements with different load balancing parameter combinations. Moreover, useful and accurate measurements often require big resource allocations on a production cluster. Our simulation workflow, dubbed Simulated Adaptive MPI (SAMPI), employs a combined sequential emulation and trace-replay simulation approach to reduce the cost of such an evaluation. Both sequential emulation and trace-replay require a single computer node. Additionally, the trace-replay simulation lasts a small fraction of the real-life parallel execution time of the application. Besides the basic SAMPI simulation, we developed spatial aggregation and application-level rescaling techniques to speed-up the emulation process. To demonstrate the real-life performance benefits of dynamic load balance with over-decomposition, we evaluated the performance gains obtained by employing this technique on a iterative parallel geophysics application, called Ondes3D. Dynamic load balancing support was provided by Adaptive MPI (AMPI). This resulted in up to 36.58% performance improvement, on 288 cores of a cluster. This real-life evaluation also illustrates the difficulties found in this process, thus justifying the use of simulation. To implement the SAMPI workflow, we relied on SimGrid's Simulated MPI (SMPI) interface in both emulation and trace-replay modes. To validate our simulator, we compared simulated (SAMPI) and real-life (AMPI) executions of Ondes3D. The simulations presented a load balance evolution very similar to real-life and were also successful in choosing the best load balancing heuristic for each scenario. Besides the validation, we demonstrate the use of SAMPI for load balancing parameter exploration and for computational capacity planning. As for the performance of the sim-

ulation itself, we roughly estimate that our full workflow can simulate the execution of Ondes3D with 24 different load balancing parameter combinations in ≈ 5 hours for our heavier earthquake scenario and in ≈ 3 hours for the lighter one.

Keywords: Parallel computing. high-performance computing. performance evaluation. dynamic load balancing. over-decomposition. simulation of distributed systems. iterative applications. SimGrid. AMPI. Charm++.

Um Workflow de Simulação para Avaliação do Desempenho do Balanceamento de Carga Dinâmico com Sobre-decomposição para Applicações Paralelas Iterativas

RESUMO

Nesta tese é apresentado um novo *workflow* de simulação para avaliar o desempenho do balanceamento de carga dinâmico baseado em sobre-decomposição aplicado a aplicações paralelas iterativas. Seus objetivos são realizar essa avaliação com modificações mínimas da aplicação e a baixo custo em termos de tempo e de sua necessidade de recursos computacionais. Muitas aplicações paralelas sofrem com desbalanceamento de carga dinâmico (temporal) que não pode ser tratado a nível de aplicação. Este pode ser causado por características intrínsecas da aplicação ou por fatores externos de hardware ou software. Como demonstrado nesta tese, tal desbalanceamento é encontrado mesmo em aplicações cujo código não aparenta qualquer dinamismo. Portanto, faz-se necessário utilizar mecanismo de balanceamento de carga dinâmico a nível de *runtime*. Este trabalho foca no balanceamento de carga dinâmico baseado em sobre-decomposição. No entanto, avaliar e ajustar o desempenho de tal técnica pode ser custoso. Isso geralmente requer modificações na aplicação e uma grande quantidade de execuções para obter resultados estatisticamente significativos com diferentes combinações de parâmetros de balanceamento de carga. Além disso, para que essas medidas sejam úteis, são usualmente necessárias grandes alocações de recursos em um sistema de produção. *Simulated Adaptive MPI* (SAMPI), nosso workflow de simulação, emprega uma combinação de emulação sequencial e *replay* de rastros para reduzir os custos dessa avaliação. Tanto emulação sequencial como *replay* de rastros requerem um único nó computacional. Além disso, o *replay* demora apenas uma pequena fração do tempo de uma execução paralela real da aplicação. Adicionalmente à simulação de balanceamento de carga, foram desenvolvidas técnicas de *agregação espacial* e *rescaling a nível de aplicação*, as quais aceleram o processo de emulação. Para demonstrar os potenciais benefícios do balanceamento de carga dinâmico com sobre-decomposição, foram avaliados os ganhos de desempenho empregando essa técnica a uma aplicação iterativa paralela da área de geofísica (Ondes3D). Adaptive MPI (AMPI) foi utilizado para prover o suporte a balanceamento de carga dinâmico, resultando em ganhos de desempenho de até 36.58% em 288 cores de um cluster. Essa avaliação também é usada pra ilustrar as dificuldades encontradas nesse processo, assim justificando o uso de simulação para facilitá-la. Para implementar o workflow SAMPI, foi utilizada a interface SMPI do simulador SimGrid,

tanto no modo de emulação, como no de *replay* de rastros. Para validar esse simulador, foram comparadas execuções simuladas (SAMPI) e reais (AMPI) da aplicação Ondes3D. As simulações apresentaram uma evolução do balanceamento de carga bastante similar às execuções reais. Adicionalmente, SAMPI estimou com sucesso a melhor heurística de balanceamento de carga para os cenários testados. Além dessa validação, nesta tese é demonstrado o uso de SAMPI para exploração de parâmetros de balanceamento de carga e para planejamento de capacidade computacional. Quanto ao desempenho da simulação, estimamos que o workflow completo é capaz de simular a execução do Ondes3D com 24 combinações de parâmetros de balanceamento de carga em $\approx$5 horas para o nosso cenário de terremoto mais pesado e $\approx$3 horas para o mais leve.

Palavras-chave: Computação paralela. computação de alto desempenho. avaliação de desempenho. balanceamento de carga dinâmico. sobre-decomposição. simulação de sistemas distribuídos. aplicações iterativas. SimGrid. AMPI. Charm++.

LIST OF ABBREVIATIONS AND ACRONYMS

ABC	Absorbing Boundary Condition
AMPI	Adaptive MPI
AMR	Adaptive Mesh Refinement
API	Application Programming Interface
AS	Autonomous System
BC	Betweeness Centrality
BRAMS	Brazilian Regional Atmospheric Modeling System
CFL	Courant–Friedrichs–Lewy [condition]
CPML	Convoluional Perfectly Matched Layer
CPU	Central Processing Unit
CSP	Concurrent Sequential Processes
DAG	Directed Acyclic Graph
FDM	Finite-Difference Method
FDTD	Finite-Difference Time-Domain
FPU	Floating-Point Unit
HPC	High-Performance Computing
I/O	Input and Output
LB	Load Balancer
MPI	Message-Passing Interface
NP	Non-Polynomial
NUMA	Non-Uniform Memory Access
P2P	Peer-to-Peer
PAPI	Performance Application Programming Interface
PDE	Partial Differential Equations

PML	Perfectly Matched Layer
PMPI	MPI Profiling Interface
PSINS	*PMaC's Open Source Interconnect and Network Simulator*
PUP	Pack-and-Unpack
PU	Processing Unit
RTS	Runtime System
SAMPI	Simulated Adaptive MPI
SMPI	Simulated MPI
SST	Structural Simulation Toolkit
TCP	Transmission Control Protocol
TIT	Time-Independent Trace
VP	Virtual Processor
xSim	Extreme-Scale Simulator

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In this thesis, we propose a novel low-cost simulation workflow to evaluate the performance of dynamic load balancing with over-decomposition, with different parameters and execution platforms. Our focus, on this work, is on the load balancing of iterative MPI applications, which can be relatively easily modified to employ this load balancing approach, e.g., with the use of Adaptive MPI (AMPI) (HUANG; LAWLOR; KALÉ, 2004). This workflow aims to reduce the cost load balancing performance evaluation. To achieve this goal, we combine sequential emulation and trace-replay simulation to perform this evaluation with minimal application modification and at a highly reduced cost both in terms of time and in terms of resource allocation.

In next section, we introduce the research context of this thesis, including a description of the problems we are trying to solve. Following this, in Section 1.2 we present our proposal of a low cost simulation workflow to evaluate dynamic load balancing. After introducing our context and proposal, in Section 1.3 we summarize the main contributions of this thesis.

1.1 Context: Load imbalance in parallel applications

In the field of high-performance computing, there are a multitude of legacy *Message Passing Interface* (MPI) applications. Some of these applications suffer from load balancing problems, which can be either static or dynamic. In some cases, the load imbalance is intrinsic to the application itself. In other cases, it may be caused by components that are external to the application. For instance, optimizations at the compiler, operating system, and hardware level may cause some tasks to run faster than others. In this case, even very regular applications, whose control flow does not seem diverge, may still be affected by load imbalance problems. Static load imbalance is often predictable and solvable through application modifications. Dynamic load imbalance is often impossible to predict before the execution of the application. It may depend on several parameters, such as the input data, the domain decomposition, the number of parallel processes or threads, resource allocation, and hardware characteristics of the execution platform. Therefore, mitigating dynamic load imbalance through static application optimizations can be difficult, if not impossible.

To solve this problem, we need to employ one of the many existing dynamic load balancing approaches, to rebalance the load distribution at runtime. In this thesis we focus on one of such approaches, which aims to dynamically balance the load of the application by employing a combination of domain over-decomposition and task migration guided by a load balancer heuristic. Over-decomposition means the domain of the application is partitioned in more subdomains than the number of processing resources (e.g., processor cores) it will utilize. For simplicity, in the remainder of this thesis we will mostly call this approach *dynamic load balancing with*, or *based on*, *over-decomposition*, or even *over-decomposition based on dynamic load balancing*. Such approach is implemented by *Charm++* (KALÉ; KRISHNAN, 1993). More specifically for iterative MPI applications, one may employ *Adaptive MPI* (AMPI) (HUANG; LAWLOR; KALÉ, 2004), which is an MPI implementation built over the Charm++ runtime.

To illustrate the benefits of dynamic load balancing based on over-decomposition, in Chapter 3 we present performance improvements obtained by employing this technique to a real-world geophysics application called Ondes3D (DUPROS et al., 2010). This iterative MPI application possesses a code structure that is representative of many iterative MPI applications, and presents both static (spatial) and dynamic (temporal) load imbalance. These characteristics make it an interesting use case to test our simulation. A detailed analysis of both load imbalances is presented on Section 3.3. Our load balancing performance evaluation demonstrates a decrease in execution time up to 35.18% using 192 cores of a cluster, and up to 36.58% using 288 cores (with a larger dataset).

Besides demonstrating the performance gains which can be achieved with over-decomposition based load balancing, this initial work with Ondes3D also illustrates a series of difficulties in running this kind of experiment. First of all, we need to modify the application in ways that may need a serious understanding of its implementation. Second, there is a big evaluation cost, both in terms of time and of computational resources. Pertaining the former, evaluating the performance of dynamic load balancing means several executions, testing different parameter combinations, in order to find the one that results in the best performance. This can easily take a huge amount of time, especially if we want statistically sound results. As for the latter, in order to get useful and accurate performance results, the experiments should be executed in the real execution platform, which would usually be a production environment. Additionally, the experiments should be done at the same scale they would run in production. This presents a problem, since it is often very

difficult to get a big enough resource allocation in a production system for the amount of time needed for the evaluation.

1.2 Proposal: A low-cost simulation workflow to evaluate dynamic load balancing

To mitigate the problems described in the previous section, we developed a low-cost simulation workflow, which we decided to call *Simulated Adaptive MPI* (SAMPI). It combines two forms of simulation: sequential emulation and trace-replay. *Emulation* is used to obtain a complete characterization of the execution of the application in the form of a *time-independent trace* (TIT). This emulation executes the application sequentially, but emulates a parallel execution. All computation is really executed, but the communication is done in shared memory (i.e., network usage is emulated). The computation and communication costs are adjusted by the simulator according to an user configured platform specification. *Trace-replay* uses a TIT trace to quickly run several simulations of the same execution on a chosen execution platform, while exploring a variety of load balancing parameters, such as different load balancing heuristics and frequencies in which the load balancing process is triggered. Both sequential emulation and trace-replay require one core of a single computer node, thus presenting a great reduction in the amount of resources needed for such evaluation. Besides, since both computation and communication are simulated, trace-replay is really fast compared to an actual execution of the application. In our tests, an execution of Ondes3D that takes 12 minutes on a 16 core cluster takes $\approx$100 seconds to be replayed in a laptop computer. All of this is done with minimal application modification, as we only need to add a special call at the end of the outermost iteration loop of the application, to indicate when the application is ready for load balancing.

The main drawback of this basic simulation workflow is that for each level of over-decomposition we need a new sequential emulation. This represents a huge cost in terms of time, which would defeat the purpose of the simulation. For this reason, we developed two upgrades to the load balancing simulation, by adding two more steps to our workflow: *spatial aggregation* and *application-level rescaling*. Spatial aggregation allows us to obtain the computation costs for a coarser-grained (less subdomains) execution of the application by aggregating the computation costs (loads) of a finer-grained execution. At this coarser-grain, we then only need to emulate communications, thus speeding-up the emulation at different levels of over-decomposition. The second upgrade, application level rescaling, deals with carefully changing application parameters (e.g., reducing the resolution of the

simulation) in a way that makes it run faster, but still preserves the general behavior of its load distribution among its tasks, and then rescaling the recorded computation costs to the original parameters. While this will, undoubtedly, reduce the accuracy of our results, it can drastically reduce the execution time of the original emulation.

We implemented our workflow using the SimGrid (CASANOVA et al., 2014) distributed system simulator, with a few custom modifications. More specifically, we used the SMPI (CLAUSS et al., 2011) interface, which supports the simulation of MPI applications through direct emulation and through trace-replay (DESPREZ et al., 2011; CASANOVA et al., 2015). Our simulation was validated by comparing the load evolution and execution times of real-life and simulated executions of Ondes3D. The resulting load-distributions for the simulation and real-life are very similar for all the tested configurations. There were small miss-predictions in terms of total execution times, but the simulation was still successful in choosing the best load balancer. Besides these validation experiments, we demonstrate the use of SAMPI to explore two load balancing parameters (level of over-decomposition and load balancing frequency). We also present an use case of SAMPI for capacity planning, by using it to estimate the amount of resources needed to run Ondes3D under a execution time threshold. Last, we present rough estimates of the simulation performance of our workflow implementation. This includes an analysis of the gains from our spatial domain aggregation and application-level rescaling techniques.

1.3 Contributions of this thesis

In the previous two sections, we presented our research context, the problems we are trying to solve, and our proposed solution. We now summarize the *main contributions of this thesis*, which are:

1. *A novel low-cost simulation workflow to evaluate the performance of dynamic load balancing* strategies based over-decomposition applied to iterative parallel applications. This workflow combines *emulation and trace-replay* to perform this evaluation with *minimal application modification*, using a *single computer node* and in a *small fraction of the time* required to do the same with real-life executions.
2. Development of two *improvements to speed-up the gathering of the time-independent traces* received as input by our load balancing simulation. *Application-level Rescaling* extrapolates the original, fine-grain, computational behavior of the application

from a faster, coarser-grain, execution. *Spatial Aggregation* speeds-up the obtainment of traces with different coarser-grain domain decompositions by aggregating the computation costs of a fine-grain execution trace. As a consequence of these improvements, we roughly *estimate ≈75-80% shorter simulation times* when testing *24 different* load balancing *parameter combinations*.

3. An *analysis of the load imbalance* of a *real geophysics application* and subsequent *performance improvements up to 36.58%*, achieved by modifying the application to support *dynamic load balancing* through AMPI.
4. An *implementation* of the proposed *simulation workflow*, which was *validated* by comparing the outcomes of simulations with those of real life executions. Moreover, its usefulness is demonstrated by a series of *use-cases*, in which we *explore different load balancing parameters* and run some estimations for *computational capacity planning*.

1.4 Structure of this document

The remainder of this document is structured as follows. In the next chapter, we present the two main research contexts in which this thesis resides: dynamic load balancing of parallel applications (Section 2.1) and simulation of distributed parallel applications (Section 2.2). We also present Charm++ and AMPI (Section 2.3) and SimGrid (Section 2.4). After that, in Chapter 3 we present our study of the benefits of over-decomposition based dynamic load balancing to a real-life iterative MPI application (Ondes3D). Following this, in Chapter 4 we present the main proposal of this thesis, which is a low-cost simulation workflow to evaluate the performance of dynamic load balancing based on over-decomposition applied to iterative applications. Next, in Chapter 5 we validate our simulator (Section 5.2), demonstrate its usage in load balancing parameter exploration (Section 5.3) and in cluster capacity planning (Section 5.4), and analyze the overall performance of the SAMPI workflow (Section 5.5). Then, in Chapter 6 we present some related works. This includes works related to dinamic load balancing (Section 6.1) and also a few distributed system simulators other than SimGrid (Section 6.2). We finish this chapter with a discussion on why we choose to use SimGrid to implement our simulation workflow (Section 6.2.5). The last chapter is the conclusion, in which

we surmise the work developed in this thesis, the obtained results and how they fit our objectives, and discuss possible future developments.

2 RESEARCH CONTEXT

Our work is situated within two main research contexts: dynamic load balancing of parallel applications and simulation of distributed parallel applications. In this chapter, we present an overview of these two subjects, as well as characteristics of specific tools, libraries and programming systems we employ in our work. Namely, we present details of Charm++, Adaptive MPI, and SimGrid.

In the next section, we present some concepts related to the load balancing of parallel applications. This includes a more detailed discussion of dynamic load balancing based on over-decomposition, on Subsection 2.1.4. Next, on Section 2.2, we approach the subject of simulation of distributed systems and applications. We give special attention to the typical elements included in the design of such simulator. We also present the two main software tools we base our work upon. First, in Section 2.3 we present characteristics of the Charm++ runtime system and of Adaptive MPI (AMPI). Finally, on Section 2.4, we include a detailed presentation of SimGrid.

2.1 Load balancing of parallel applications

One of the main concerns in parallel computing is the efficient usage of computational resources. This means we want these resources to be busy for the largest fraction of the execution time as possible. One way to improve this efficiency is to reduce the amount of time spent synchronizing different tasks. Here is where load balancing becomes important.

In parallel computing, load balancing usually refers to the distribution of the workload among the available computing resources (e.g., processor cores). In an ideal scenario, all tasks would perform the same amount of computation during any specific period of time. In this case, we would say that they have the same workload (or simply load) or, in other words, that the application is load balanced.

In a perfectly load balanced execution, all tasks would start their synchronization at the same time. This would lead to a minimal loss of efficiency due to synchronization. On the other hand, if load imbalance is present, tasks will reach the synchronization point at different moments. This means that all tasks will have to wait for the most loaded one to reach the synchronization point. During this time the tasks are waiting for

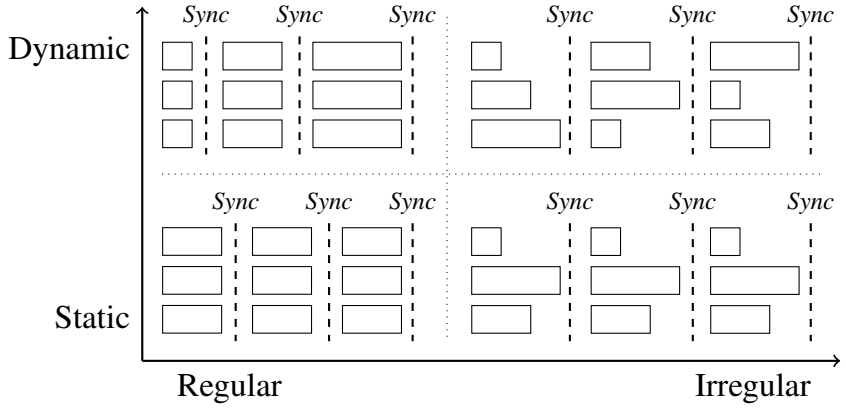
synchronization, they are wasting processing power, since they are not performing any computation. Therefore, load imbalance typically leads to lost computational efficiency.

Unfortunately, in real life it is very common for applications to have some kind of load imbalance. What this means is that given a period of time, their tasks may perform different amounts of computation. More than that, in some cases the loads of different tasks can vary dynamically throughout the execution of the application.

2.1.1 Types of load balance

The load of an application can be classified as static or dynamic and as regular or irregular, as illustrated on Figure 2.1. An application has a static load when the load of its tasks does not change during the whole application's execution. On the other hand, a dynamic load means that the load of the application's tasks changes as its execution progresses. It is important to point out that a dynamic load does not necessarily mean that the application is load imbalanced.

Figure 2.1: Classification of load balance, according to regularity and dynamicity



Source: (PILLA, 2014)

On the other axis of Figure 2.1, we see the classification of the load according to its regularity. A regular load means that all tasks have the same amount of work, in other words, the application is load balanced. An irregular load means the application's tasks perform different amounts of work, given a specified period of time. On other words, the application is load imbalanced.

When an application's load distribution is both *static and irregular*, we say it suffers from *static load imbalance* (or *spatial load imbalance*). In this case, it may be possible to solve the problem using static strategies, applied prior to the actual execution

of the application. That is not the case when the application's load distribution is both *irregular and dynamic*. In which case, we say that the application suffers from *dynamic load imbalance* (or *temporal load imbalance*). To mitigate this problem, we need to apply dynamic (or adaptive) load balancing techniques that are able adapt to the variations in load that happen in execution time.

In some cases, load balancing techniques also take into account other performance related costs implied in the execution of parallel programs. For example, some heuristics take into account the memory and network topology of the execution platform, in order to reduce communication costs. This is done by trying to keep tasks that exchange or share high amounts of data close to each other in the hardware topology.

Load balancing is the distribution of computation and communication loads of an application equally throughout a platform (PILLA, 2014). The main objective is to prevent the overloading of processing units (PU), which may cause the under-utilization of other PUs, due to synchronization.

Most dynamic load balancing techniques employ some kind of task remapping approach, based on measurements made in runtime. Load balancing is an NP-Hard problem (LEUNG, 2004). This means that every problem in NP can be reduced to the load balancig problem in polynomial time. In other words, it means this problem is at least as hard as the hardest problems in NP. It is suspected there is no polynomial time algorithm to solve NP-Hard problems, even tough this has not been proven. For this reason, finding an optimal solution for load balancing is impractical. Therefore, there are several different load balancing heuristics to decide on which tasks will be mapped to where. An heuristic is an algorithm that does not guarantee an optimal solution, but one that is sufficiently good to fulfill its goal. In the following subsection, we present the main general characteristics that should be taken into consideration in the development and analysis of load balancing algorithms.

2.1.2 Types of load balancing algorithm

Load balancing algorithms can be either static or dynamic. *Static* algorithms are based on previous knowledge of the application. In this case, task mapping is decided prior to the execution of the application. *Dynamic* load balancing algorithms make decisions in execution time. These algorithms have to dynamically adapt to variations in the loads of

the tasks. In this case, task mapping is dynamic and can change during the execution of the application.

The algorithms can also be classified as *generic* or *application specific*. *Generic algorithms* do not rely on previous knowledge of the application. Therefore, they can be applied to different applications. *Application specific algorithms* rely on prior-knowledge about the behavior of the application. These algorithms are able to implement load balancing approaches that take advantage of specific characteristics of the application. This way, they may be able to get better improvements than a generic algorithm. On the other hand, they are often useful only for the application for which they were developed.

A third classification of load balancing algorithms has to do with whether their decision process is *centralized* or *distributed*. In a *centralized algorithm*, a single process is responsible for all the task remapping decisions. The advantage of centralization is that the algorithm has data about all the tasks and processing resources, therefore leading to better decisions. On the other hand, this approach can lead to issues of scalability, since the cost of the algorithm (overhead) will tend to increase with the number of parallel tasks.

In a *distributed load balancing algorithm*, the decision process is divided among various distributed processes. One example of such would be a hierarchical algorithm, where the top level represents the whole system and lower levels represent subsets of its resources. In this case, the algorithm would remap tasks in each branch of the hierarchy and send only a subset of these tasks up in the hierarchy to potentially be moved to other branches. This type of strategy should have a lower overhead on large scale systems than a centralized one. On the other hand, it may lower the quality of the load balancing, since the local load balancer may not have enough information to make good decisions.

2.1.3 Load balancing techniques

In this subsection we present a few different strategies that attempt to improve the load balancing of parallel applications. In our work, we focus on *runtime level load balancing*. Other two possible approaches use *graph partitioning techniques* or implement *application specific* schemes. Below, we include a short description of each of these.

1. *Runtime level* load balancing refers to load balancing that is implemented as part of a *runtime system* (RTS). This technique usually relies on some kind of over-decomposition of the application and task remapping, often done through task

migration. One advantage of this approach is the ability to support dynamic load balancing using measurements collected during the execution of the application. The runtime can apply different heuristics to redistribute the tasks based on this information. One example of runtime that employs this technique is the *Charm++ Runtime System* (KALÉ; KRISHNAN, 1993).

2. *Graph partitioning* based load balancing, employs hypergraph partitioning algorithms (CATALYUREK et al., 2007) to map tasks in a way that improves load balancing and communication costs. This works by associating the task mapping to a hypergraph. Vertices represent computational loads associated to data and edges represent data dependencies. Edges between different partitions incur in communication costs. On a first phase, called *coarsening*, the algorithms *fuses* vertices with costly (high amount of communication) networks between each others vertices. On a second phase, each processing unit computes an hypergraph *growing* algorithm, which divides the graph into k partitions. On the third phase, called *refinement*, vertices are moved between partitions, to improve load balancing. After this first load balancing phase, one fixed local vertex is added to each partition. These fixed vertices can not be fused in the coarsening phase. Migration costs are represented by networks between the fixed vertex and the other vertices in the same partition.
3. *Application specific* load balancing refers to techniques that rely on application or platform specific knowledge. Therefore, they only work for one specific application or execution platform. These algorithms can take advantage of prior knowledge about the application (e.g. communication patterns, execution model) and about the execution platform (e.g. architecture, memory topology, interconnection network).

Besides load balancing, there are a few other techniques that may improve the efficiency and load distribution of parallel applications. Two of these techniques are *work stealing*, at the task level, and *process mapping*, at the operating system or processor architecture level.

Work-stealing (FRIGO; LEISERSON; RANDALL, 1998; BLUMOFE; LEISERSON, 1999) consists of having a pool of tasks for each processing unit (PU). Idle processing units, called *thieves*, try to steal tasks from PUs that have non-executed tasks in their pools. PUs whose tasks are stolen are called *victims*. There are several different work-stealing algorithms, which usually are either distributed or hierarchical. These algorithms mostly differ on whether the task poll is centralized or distributed and on their victim selection

heuristic. Work-stealing is specially useful for applications where the load is unknown beforehand and when there is dynamic creation of tasks.

Task mapping refers to techniques used mainly to define the initial mapping of the application. The main focus is to mitigate communication costs. These approaches usually consider that the number of tasks is the same as the number of processing units. Task mapping usually implies the employment of some kind of matching mechanism that takes into account the communication pattern of the application and the machine topology. This mechanism may be implemented either in software, e.g., in the operating system scheduler, or in hardware, e.g., inside a multicore processor.

2.1.4 Dynamic load balancing based on over-decomposition

In this work we focus on periodic load balancing based on the concepts of over-decomposition and task migration. Over-decomposition means that the application is decomposed in more parallel tasks than the number of processing units in which it will be executed. In other words, more than one task can be mapped to a single processing unit. An example of that would be running 128 MPI ranks on a 32 core machine. In this context, task means a parallel process or thread (e.g., an MPI rank), which may be implemented as at operating system or user level.

A series of benefits is derived from this increase in parallelism. One initial advantage is that it can improve efficiency by automatically overlapping computation and communication (or any other type of synchronization). This is achieved naturally, as a task that is ready for execution can be scheduled while another task is in the waiting state. Thus potentially hiding the latency implied in task synchronization. It's worth to note, however, that the amount of synchronization you can hide is heavily dependent on the dependencies between tasks. Another benefit is that as we split data into more tasks, we also improve memory locality and cache usage, which may lead to performance improvements. On the other hand, having more than one task per core is prone to reduce cache performance, as multiple tasks compete for it.

A possible issue when over-decomposing an application is the fact that a larger number of virtual processors results in a direct increase in communication volume. For instance, it is very common for applications to communicate with their neighbors in the simulated domain. In this case, if this domain is divided into more subdomains, more neighbors will be formed. Consequently, more communication will happen in the platform.

Therefore, if too many tasks are scheduled per core, communication contention may become a bigger problem than load imbalance.

An important requirement to enable over-decomposition to improve application performance is the use of an efficient runtime system (RTS). It must be able to efficiently schedule tasks and manage their communication. Otherwise, its overhead may surpass the benefits of over-decomposition. For that reason, we resort to the use of *Adaptive MPI* (AMPI) (HUANG; LAWLOR; KALÉ, 2004; HUANG et al., 2006), which allows to run MPI applications on top of the Charm++ RTS. Charm++ provides a well known and reliable dynamic load balancing framework based on over-decomposition and task migration (more details in Section 2.3).

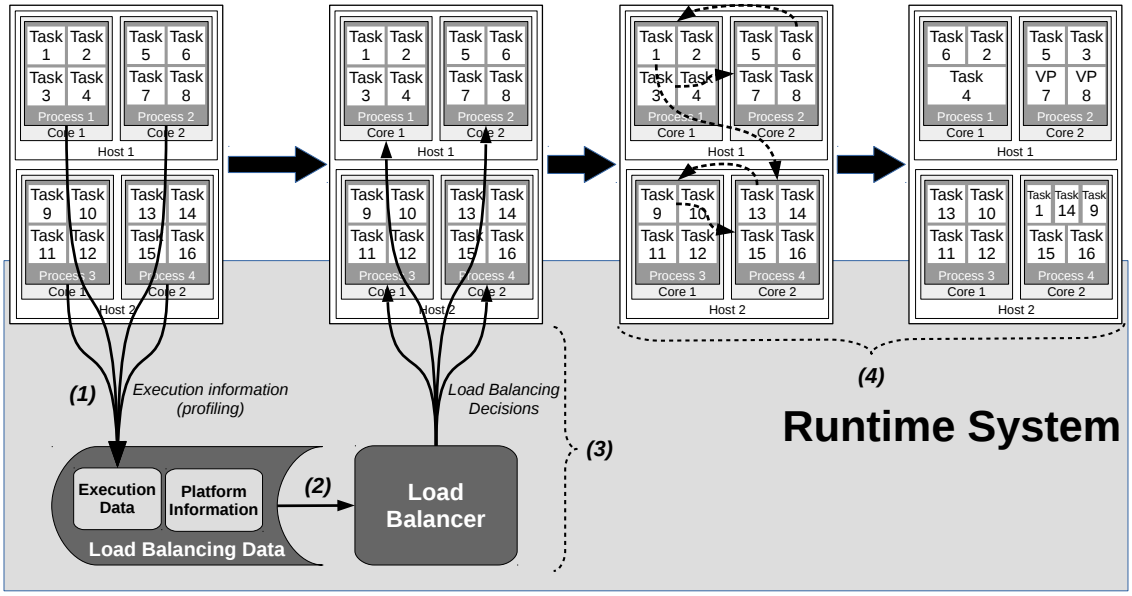
Over-decomposition becomes more interesting when coupled with a RTS capable of migrating tasks between processing units. With these two characteristics, it becomes possible to apply diverse task scheduling techniques at the level of the RTS, to improve resource usage and reduce communication costs. Among these techniques, we are most interested in dynamic load balancing.

Dynamic load balancing consists of using information about the applications execution and about the execution platform to remap tasks in a way that improves load balancing. Figure 2.2 illustrates the steps involved in this approach:

1. While the application is executing, the runtime collects load balancing information, such as the load per task and per processing unit, and the amount of communication between tasks.
2. The collected information is provided to a load balancing heuristic, which may also utilize information about the execution platform.
3. The heuristic uses the data to decide which tasks should be migrated between processing units. That means the heuristic tries to improve future state of the application based on its near past.
4. The runtime system performs the necessary task migrations, hopefully resulting in a more load balanced state.

The steps above may be performed periodically or based on a more dynamic heuristic. In our case, as we are dealing with iterative MPI applications, which mostly have well defined phases, we will focus on periodic load balancing. Once again, we rely on AMPI to support this funcitonality.

Figure 2.2: Dynamic load balancing strategy based on over-decomposition: *(1) collect runtime execution information; provide this information to a load balancer; (3) the load balancer makes load balancing decisions; (4) the runtime migrates processes based on these decisions.*



Source: The author

2.1.5 Load balancing parameters

In the context of load balancing based on over-decomposition, there are three main parameters that affect its effectiveness. They are the load balancing *heuristic*, its *frequency* and the *level of over-decomposition*.

For the first one, the *load balancing heuristic*, the best choice depends on characteristics of both application and platform. Besides that, some heuristics are more scalable than others. This means that an heuristic that performs well with a certain number of tasks may not be so good with a larger task count. This happens especially on centralized strategies, where the number of tasks tends to be strongly related to the overhead of the load balancing strategy. So, the level of parallelism is very important when choosing a heuristic.

The next two parameters present a trade-off between possible performance gains from load balancing and the overhead cause by it. It is important to choose these carefully, in order to strike the right balance between them.

The *level of over-decomposition* influences how well the the load balancer is able to redistribute the load. In this aspect, the more subdomains, the better. On the other hand, increasing the number of subdomains may also increase the amount of communication at

the application level. In some cases, the cost of this communication may surpass the gains from load balancing, thus defeating its purpose. Furthermore, the number of tasks also determines the amount of work done by the balancer, thus affecting its performance. If the load balancing process takes too long to finish, we may lose performance instead of gaining.

The third parameter is the frequency of the load balancing. The more frequent the load balancing, the more likely we are to detect dynamic imbalance. If the balancing is too frequent, however, it may incur in too much overhead. In other words, we want to avoid frequent calls to the load balancer when the imbalance is low. On the other hand, we also want to quickly detect significant increases in imbalance. One must find the right trade-off between detecting imbalance early and avoiding to call the load balancer too many times. This is key to achieve performance improvements without too much overhead.

2.2 Simulation of distributed systems and applications

The behavior of applications on distributed platforms can be extremely difficult to understand. Not only these systems are continuously becoming more complex, as the applications that run on them tend to have a high level of complexity of their own. The former is especially true in the parallel computing context. Nevertheless, there is the need to accurately asses the quality of competing algorithms and system designs with respect to precise objective metrics. This could be done with theoretical analysis, however, these tend to be tractable only by relying on stringent and unrealistic assumptions. As a result, most of the research in this field is empirical. That means that research questions are answered by comparing the objective metrics across multiple experiments.

We classify experiments to evaluate distributed applications on distributed systems in three classes: *real-life evaluation*, *on-line simulation* and *off-line simulation*. In our context, *real-life evaluation* refers to the execution of the real application in a real platform. One potential problem with this approach is its resource requirements, since it is usually hard to get access to a costly production system to run this kind of experiment. This would probably be deemed too expensive by the owner of the system. Besides that, due to the shared aspect of most of these systems, it may not always be possible to get enough resources. Therefore, limiting the exploration of different scenarios.

On-line simulation experiments refer to executing the application on a “virtual” platform. Examples of such platforms could be a series of virtual machines (on top of a

smaller/slower platform) or an emulator. One advantage of this approach is the potential reduction of the resource requirements. For example, an emulator could mimic the parallel execution of the application while actually running inside a single sequential process. An important aspect of both *real-life evaluation* and *on-line simulation* is that they can be significantly time consuming. This is due to the necessity to execute long running applications and to perform a large number of experiments to in order to test different configurations and to obtain statistically sound results.

The third kind of experiment is called *off-line simulation*, or simply *simulation*. This means simulating the execution of the application on a simulated environment. In this case, instead of running the real application, its execution is simulated based on some kind of abstract application specification. *Off-line simulation* experiments are typically less costly than the other two, both in terms of workload and in terms of hardware resources.

When simulating parallel applications, the key concerns are accuracy and scalability. The more accurate the model, more complex are its calculations, therefore less scalable it is. This consists of a very important trade-off. Finding the right balance between model accuracy and simulation scalability is of paramount importance in the design of distributed system simulators.

2.2.1 Design of a typical distributed systems simulator

A typical distributed system simulator is composed of three components: *simulation models*, *platform specification*, and *application specification*. In this subsection, we will present some details about each one of these.

Simulation models implement the evolution of simulated application activities using simulated resources through simulated time. Models differ in their trade-offs between model performance (in terms of time) and the level of detail of real resources that is incorporated in the simulation. The most common models found in distributed system simulators model *CPU*, *storage*, and *network*.

To simulate CPUs, it is usual to employ a simple analytical model of computation delay. These usually work by dividing computation cost by computation speed. In some cases, a random component maybe included. This simple model tends to yield reasonable results for simple CPU bound applications. It may even be used to simulate simple multicore parallelization. One drawback of this approach is that it does not account for

architecture specific features of the simulated CPU resources. Designing analytic models that capture these features is still an open research question.

In the field of distributed system simulation, storage resource simulation is rarely done. For this reason *storage models* are mostly ignored by distributed system simulators. Implementing these models is usually left to the users who really need this kind of simulation. The accurate modeling of storage resources is known to be extremely challenging.

Network models, on the other hand, are required by all parallel and distributed application simulators. The highest level of accuracy is obtained by employing packet-level simulation. The problem is that this accuracy comes at a high cost in terms of performance, which leads to a lack of scalability. In some cases, simulation times can be orders of magnitude larger than simulated time. Therefore, this technique is unsuited for the vast majority of users.

The only way to achieve fast simulation of networks is through analytic models. There is, however, a loss of accuracy compared to packet-level simulation. The evaluation of the accuracy of analytical network models is made by comparing their results with "ground truth". This could be done by comparing it with real experiments. The problem is that this requires access to the real platform to run the experiments, which is often difficult. This limited access to resources inevitably leads to a limitation in the number of different configurations that can be tested. A more feasible approach to validate analytic network models is to use packet-level simulation as the "ground truth". This approach has been used in the evaluation of some distributed system simulators. The problem, in most cases, is that the evaluation is only used to confirm use cases where the application performs well (VELHO et al., 2013). Instead, there should be a focus in checking how the model performs when simulating difficult cases.

The second component of a typical distributed system simulator is the *platform specification*. This consists of mechanisms and abstractions that allow the instantiation of different platforms (simulation scenarios) without modifying the simulation models or implementation. A typical platform specification would describe the computational resources available (including their computing power), as well as the network infrastructure that connects them (including bandwidths and latencies). To simulate the execution of the application on different platforms, the only thing that would need to be changed is this description.

The third component is the *application specification*, which consists of mechanisms and abstractions used to describe the nature and the sequence of activities to be simulated. There are three main approaches used to specify the application: *formal description*, *programmatic description*, and *offline simulation*.

The *formal description* of applications has the advantage of good scalability in terms of size. One example of this approach would be to describe the application as formal automata. The difficulty with this approach is that the formalisms may be too rigid, making it hard to describe complex application logic.

Programmatic description is the method of choice of most simulators. In this approach, applications are described as a set of functions or methods. These are used to describe *Concurrent Sequential Processes* (CSP). This technique tends to be more accurate than formal automata, while still retaining scalability in terms of size.

The third kind of application description, and the one we focus our present work, is *offline simulation*. In this approach, the application is described in the form of traces from an execution in a real platform. This has the advantage of presenting an accurate description of the execution of the application. There is, however, a scalability issue due to the potentially large size of these traces.

2.3 Charm++ and Adaptive MPI

Charm++ (KALÉ; KRISHNAN, 1993) is a parallel programming system that aims to be platform independent. Besides aiming at portability, it employs techniques such as over-decomposition, message-driven execution, task migration and dynamic load balancing to improve computational efficiency. Charm++ also supports checkpoint and restart of the execution of applications, using the same mechanism that allows the migration of tasks.

In Charm++, parallel objects are called *chares*. Their execution is message-driven, which means that all computations are initiated in response to messages being received. All communication is asynchronous and non-blocking. The combination of this feature with over-decomposition enhances the latency tolerance of Charm++. For the purposes of this work, however, we are most interested in its dynamic load balancing functionality.

2.3.1 Load balancing in Charm++

Charm++ implements dynamic load balancing based on the concept of over-decomposition. Each task is mapped to a *Virtual Processor* (VP), which is implemented in a user-level thread. These user-level threads have the advantage of being lightweight and possible to schedule by the RTS without interference or reliance on the operating system scheduler. In a typical execution, several VPs are mapped to the same system process, which is mapped to a physical processor. Another important aspect of VPs is that they can be migrated between processors.

The combination of over-decomposition and migration enables Charm++ to periodically redistribute tasks, in order to improve load balancing. This redistribution is done according to a load balancing heuristic. This heuristic uses load information from the near past of the application to decide which processes need to be migrated (*principle of persistence*). This gives the Charm++ runtime system the ability respond to variations in load that happen in execution time. Therefore, the load balancing approach in Charm++ is dynamic.

Charm++ provides its own programming model and API. For legacy applications, however, it may be too costly to rewrite the code to use Charm++. It is reasonable to infer that many, if not most, of these legacy applications are based on MPI. For these, at least, there is Adaptive MPI (AMPI), which makes possible for MPI applications to take advantage of features provided by the Charm++ runtime system.

2.3.2 Adaptive MPI

Adaptive MPI (AMPI) (HUANG; LAWLOR; KALÉ, 2004; HUANG et al., 2006) is an MPI implementation built upon Charm++. As such, it allows MPI applications to benefit from features provided by the underlying runtime system. Among these features, we are most interested in its dynamic load balancing capabilities.

A program written in AMPI is very similar to its MPI counterpart. In fact, AMPI can run most MPI programs without code modifications. To take advantage of VPs and the load balancing framework, however, the programmer must deal with two issues. The first one refers to the privatization of global and static variables. The second modification is related to aiding the runtime on the migration of dynamically allocated data.

2.3.3 Porting MPI applications to AMPI

In order to take advantage of AMPI's load balancing functionality, the application needs to be changed in three ways. First, to support over-decomposition, the application cannot have any global or static variables. This is necessary because tasks are implemented as user-level threads that share the same address space, thus these variables would become shared between all tasks (VPs) which are mapped to the same system process. On standard MPI, each MPI rank has its own private address space.

Second, to make a VP migratable, the programmer needs to insure that all necessary data can be migrated. Variables that are statically allocated in the stack are automatically migratable. To migrate dynamically allocated data (stored in the heap), however, the developer needs to implement Pack-and-Unpack (PUP) routines using helper functions provided by AMPI.

PUP functions instruct the runtime on how to serialize the application's dynamically allocated data. During migration, the runtime will use these functions to "pack" data on the origin and "unpack" it on the destination. Writing these functions can be a time consuming process, since it may require a profound understanding of the application's data structures and the way it handles memory allocation.

AMPI provides PUP functions for basic types, such as `int`, `float`, `double`, and `char`, as well as for arrays of these types. These can be used to construct more complex PUP functions. Besides packing-and-unpacking, the PUP function must free the allocated memory on the source after packing the data, and allocate it on the destination before unpacking. PUP routines must be registered with the runtime by calling the function `MPI_Register`. This function receives as arguments two pointers: one to the data and another to the PUP function. After that, the RTS will automatically call the routine to guide the storage of data before and after migration.

The third modification is the addition of a call to `MPI_Migrate`, to indicate to the runtime when to trigger the load balancing process. Examples of things the developer must make sure at this point are that there are no active communications or open files. In a typical use case, `MPI_Migrate` will be placed at the end of the application's outermost iteration, and will be called periodically every *N* iterations.

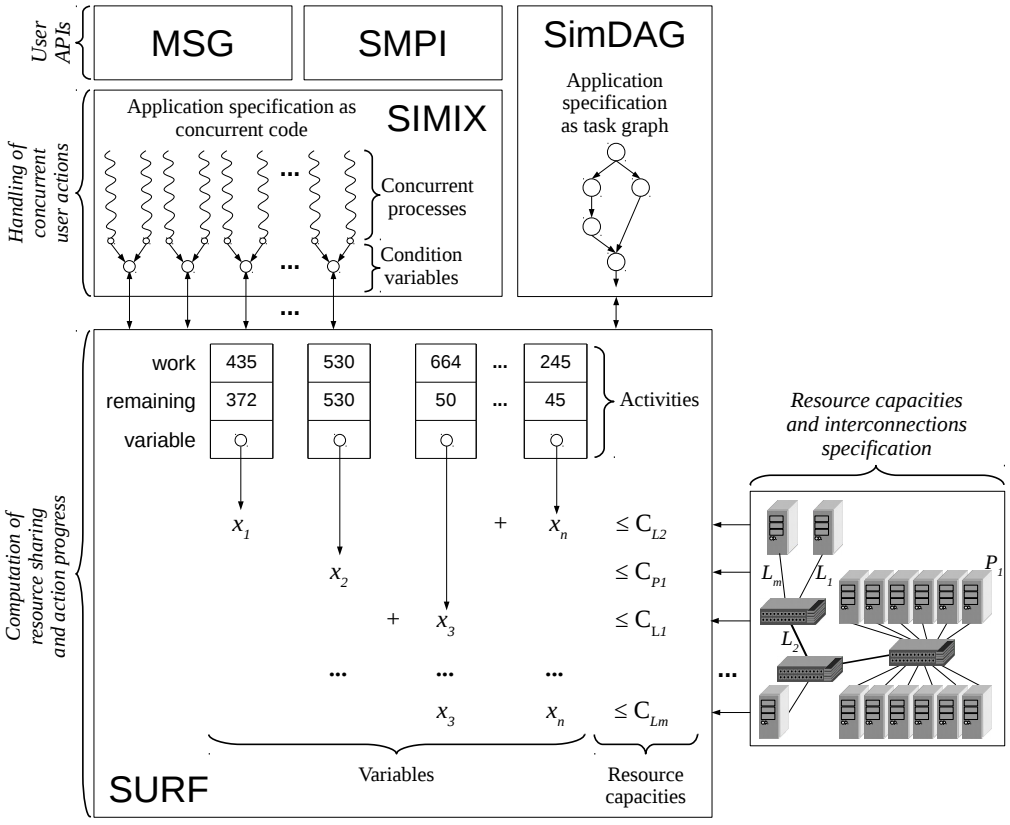
2.4 SimGrid

SimGrid (CASANOVA et al., 2014) is a discrete event simulation framework with more than 15 years of development. It aims to simulate distributed applications on distributed platforms. The simulator implementation aims at versatility. This means providing the necessary capability to run simulation from multiple domains, such as cluster, grid, cloud, volunteer and peer-to-peer computing. Key design aspects include the complete separation between resource specification, application execution, and resource sharing models.

2.4.1 Structure

Figure 2.3 illustrates the layered modular structure of SimGrid. On the top level, we have the user level APIs. An intermediary layer handles the simulation of concurrent processes. On the bottom, we have the simulation core.

Figure 2.3: Design and internal structure of SimGrid



Source: The Author

MSG is an user API used to describe a simulated application as a set of concurrent processes. These processes execute code implemented by the user in C, C++, Java, Lua, or Ruby. Inside this code, the user employs MSG calls to simulate computation and communication activities. *SimDAG* allows the users to specify the application as an abstract task graph of communicating computational tasks with non-cyclic dependencies. *SMPI* uses sequential emulation to simulate the execution MPI applications as a set of concurrent processes. It supports the direct simulation of unmodified MPI application codes written in C or Fortran. SMPI includes a runtime system that implements the necessary MPI-specific functionalities. In our work, we use SMPI to collect the input traces used on our simulation. The simulation itself is executed using a trace-replay simulator built on top of SMPI.

Under MSG and SMPI on Figure 2.3, we have a module called *SIMIX*, which implements mechanisms for simulating concurrent processes. This module provides functionalities used by both MSG and SMPI. On the bottom level, we have a module called *SURF*. Regardless of the API used, application simulation consists of a set of communication and computation activities which are executed on simulated hardware resources. SURF is the simulation core responsible for the execution of these activities on the resources.

2.4.2 Models

SimGrid relies on a unified analytical model for simulating the execution of activities on simulated resources. This model is based on solving a constrained *Max-Min* optimization problem. This problem is at the core of SURF, which implements efficient algorithms and data structures to quickly solve the corresponding linear system. This is a general model which can be used to simulate CPU, storage and network.

In fact, SimGrid's CPU and storage models use exactly this unified model, which guarantees fair sharing of these resources. This is enough to comply with the analytical models used by state-of-the-art simulators (CASANOVA et al., 2014).

Besides CPU and storage, SimGrid provides several analytical network models, including a *constant time* model, a *coordinate based* model and a *flow-level* model. For users that require more accuracy than provided by the analytical models, SimGrid supports packet-level network simulation through NS-3 bindings. NS-3 (RILEY; HENDERSON, 2010; NS-3 CONSORTIUM, 2018) is a discrete-event network simulator, targeted primarily for research and educational use.

Among the analytical network models, the default is the *flow-level model*, which provides more realistic results. This model is based on TCP, which does not implement Max-Min fairness. This problem is solved by improving SimGrid's unified model in several ways via the use of additional metrics (CASANOVA et al., 2014). This model was tested through a thorough validation study (VELHO et al., 2013), which focused on characteristics of TCP that are not captured by the simple Max-min model. The improved flow-level model can capture characteristics at the macroscopic level of TCP, such as slow start, protocol overhead, RTT-unfairness and flow control limitation.

2.4.3 Platform specification

SimGrid's platform description aims at being versatile, yet scalable. Its format exploits the hierarchical structure of real world networks. This is done using the concept of Autonomous Systems (AS), which can be represented recursively, thus allowing the exploitation of regular patterns. Each AS has a number of gateways, which are used to compute routes between ASes inside the same higher level AS.

This platform description assumes that routes are static over time. In the real world, routing changes are known to affect traffic on backbone links. But, usually these links are not communication bottlenecks. To specify the platform the user needs to create an input file in the XML format.

2.4.4 Application Specification

SimGrid supports three forms of application specification through it's user API's (MSG, SimDAG, SMPI), plus one more form through it's trace-replay functionality.

MSG provides an API to describe the behavior of the application programmatically, as a set of concurrent processes. It provides calls to simulate communication and computation activities. Another kind of programmatical specification accepted by SimGrid is provided by SMPI. In this case, the specification is the actual application code implemented using MPI. One key difference from MSG is that SMPI simulates only the communications, the computation is actually executed. See the next subsection for more details on SMPI. SimDAG allows the specification of the application in the form of a direct acyclic graph (DAG) representing the dependencies between communicating tasks.

Last, we have the time-independent trace replay mechanism built on top of SMPI. This mechanism supports the specification of MPI applications in the form of time-independent traces from a real execution of the application. Alternatively, this trace may also be obtained from an emulated execution using SMPI. More details on this trace-replay mechanism are presented on Subsection 2.4.6.

2.4.5 Emulation of MPI applications with SMPI

In the present work, SMPI is the main SimGrid user level component we employ. It consists of an implementation of MPI on top of SimGrid. It allows the sequential emulation (online simulation) of the execution of unmodified MPI applications. In this execution, the MPI ranks are folded into a single process. This allows to run the emulation on a single computer node, or even on a laptop. There is, however, the requirement that the computer have enough memory to fit the data belonging to all MPI ranks. As result of an SMPI emulation we can get an estimation of the execution time of the application, detailed execution traces (with optional resource usage traces), and time-independent traces (TIT). These TIT can be used as input for the trace-replay simulation described in the next subsection.

During the emulation all the application's computation is actually executed, whereas the communication is simulated. To simulate MPI point-to-point communications, SMPI includes a piece-wise linear simulation model. This model accounts for the different ways MPI implementations handle messages based on their length. When coupled with SimGrid's flow model, to simulate bandwidth sharing, this allows the simulation of collective operations. Collective communications are modeled as a set of point-to-point communications, which may contend on the network. SimGrid provides several collective communication implementations used by different MPI implementations, such as OpenMPI and MPICH.

All of the above characteristics make it possible to simulate the execution of MPI applications accounting for network topology and contention in high speed TCP networks. Besides, SMPI is the basis for the trace-replay mechanism employed in our work. In the next subsection, we present more details on this functionality.

2.4.6 Trace-replay of MPI applications with SimGrid

Trace-replay, also called *offline simulation* or *post-mortem* simulation, refers to the simulation of the execution of applications based on activities registered on traces from a previous execution.

SimGrid has the capability of performing *time-independent trace replay* (DESPREZ et al., 2011) for MPI applications. The difference of this technique from traditional trace-replay is that traces contain volumes of computation and communication instead of times. These quantities do not depend on the host platform, except for an small fraction of applications that adapt their execution path to the execution platform.

The employment of time-independent traces has the advantage of loosening the link between trace acquisition platform and target platform. A small drawback is that it is no longer possible to scale the timings by simply modifying them by a factor. Instead, one must re-execute the replay process to get the simulated times on a new platform.

SimGrid's time-independent trace replay functionality was first implemented using the MSG API (DESPREZ et al., 2011). This simulator was thoroughly validated, with encouraging results. There were, however, some issues in terms of accuracy and scalability. For this reason, an improved version of the time-independent trace replay was implemented (DESPREZ; MARKOMANOLIS; SUTER, 2012). This new version uses the SMPI API, leading to a simplification of the simulator code and better accuracy in the simulation of communications. The latter is a direct result of the piece-wise linear model for communication included in SMPI. Besides that, there were some modifications to reduce the size of the traces. This was done by registering only computations and MPI calls.

Initially, the instrumentation to obtain the traces was done using Tau (SHENDE; MALONY, 2006). Currently there are at least two other ways to get them. One of them is to use the MinI (CASANOVA et al., 2015) instrumentation library, to get the traces from a real parallel execution of the application. The other option is to get the traces from a sequential emulation using SMPI. The parallel execution has the advantage to be faster, but requires more resources. The emulation requires only one processor core to execute but takes much longer to finish, since all computation is sequential.

A time-independent trace is composed of a list of actions. Each action has an *id*, *type*, *volume* (instructions or bytes), and optional *action specific parameters* (e.g.,. rank

of the receiving process for one way communications). The simulator has each action associated to a function that uses SMPI to simulate the action.

The replay mechanism receives as input the time-independent traces and a platform specification file. The replay process consists of reading the actions from the traces and executing the associated function. This function will read the action arguments and execute the appropriate SMPI calls to simulate the communication or computation activity represented by the action. Besides the total simulated time, the output of the simulation can include execution traces and resource usage traces.

2.5 Conclusion

In this chapter, we presented the research context of our work and the main software tools we used to implement it. On the next chapter we will present a real-life study case on the employment of dynamic load balancing based on over-decomposition. We analyze the load imbalance in the Ondes3D, present its adaptation to AMPI, and evaluate the performance improvements obtained from dynamic load balancing.

3 A CASE STUDY OF OVER-DECOMPOSITION BASED DYNAMIC LOAD BALANCING APPLIED TO A REAL-LIFE APPLICATION

As explained in Chapter 1, dynamic load imbalance is intrinsic to many parallel applications. This problem is especially common in scientific applications that model real-world phenomena. Possible sources of this load imbalance at the application level may include the dynamic nature of the simulated process, characteristics of the model, and implementation aspects. Load imbalance may also come from factors external to the application, such as hardware and software characteristics of the execution platform (e.g., compiler and hardware level optimizations). As high-performance systems become increasingly larger and more complex, it gets increasingly harder to predict their behavior. In this work, we focus on improving the performance of legacy iterative MPI applications which present some form of dynamic load imbalance. More specifically we study the use of periodic load balancing techniques based on over-decomposition coupled with task migration.

In this chapter, we demonstrate the benefits of this load balancing approach through real-life experiments. This is done through the modification of a seismic wave propagation simulator, called Ondes3D (DUPROS et al., 2010), to support dynamic load balancing through AMPI, and the evaluation of the resulting performance gains. Ondes3D is a typical iterative MPI application with a fairly regular domain decomposition. Its implementation is very representative of many other legacy parallel applications that employ the message-passing paradigm. Even tough its code seems to have a very regular execution flow, we demonstrate that it presents both spatial (static) and temporal (dynamic) load imbalance, and present a hypothesis for its causes.

At the beginning of this chapter, we present the main aspects involved in using over-decomposition based dynamic load balancing to improve the performance of legacy iterative applications. In Section 3.1, we discuss the requirements for taking advantage of this technique, as well as the involved costs in terms of development, time, and resources. After that, we illustrate the application of this technique to Ondes3D. More specifically, on Section 3.2, we present the characteristics of the application and details of its implementation. Next, on Section 3.3 we present an analysis of the spatial and temporal imbalances present in the application. Then, on Section 3.4, we present the modification of Ondes3D to support dynamic load balancing with AMPI. Following that, on Section 3.5.1 and 3.5.2, we have the evaluation of the load balancing benefits. Finally, in the last section we discuss

the difficulties involved in the load balancing evaluation process, thus justifying the use of simulation for this purpose.

3.1 Employing load balancing based on over-decomposition on a legacy application

In order to benefit from dynamic load balancing based on over-decomposition, we need a runtime system that is able to: (1) efficiently schedule the over-decomposed tasks on their assigned processing units; (2) collect runtime information to be used as input for the load balancing algorithm; (3) support the execution of the load balancing algorithm; and (4) migrate tasks and their data between different processing units, according to the decisions made by the load balancing algorithm.

Besides having a suitable runtime system, it may be necessary to modify the application code. In some cases, changes may be necessary to support the over-decomposition of the application. These changes may be either due to application characteristics or due to requirements of the runtime system. We also need to make the application tasks migratable between processing units. This may involve serialization code used to guide the runtime system in the migration of data. We may also need some way to know when it is safe to perform the load balancing. For periodic load balancing, this may be made by adding a call to the runtime, for instance, at the end of every N application iterations.

Once the developer has a runtime system and the application is properly modified to take advantage of its load balancing features, he will need to run a series of experiments aimed at finding the best load balancing configuration. For this he will need to explore a series of combinations of parameters, such as different load balancing heuristics, different levels of over-decomposition, and different intervals between calls to the load balancer. These parameters were discussed in Subsection 2.1.5.

Running these experiments is costly, both in terms of time as in terms of resources. Especially since it is not uncommon for parallel applications to take hours, or even days, to complete their execution. In some cases, it may be sufficient to run only a fraction of its time-steps or reduce the size of the input data while still getting meaningful results. This may decrease the execution time to something more tractable, at the cost of a possible reduction in the accuracy of the performance measurements. Another fact that contributes to the cost in time is the need to run each experiment several times, in order to get statistically significant results.

Then, we need to consider the resource usage. To get realistic measurements of the benefits of load balancing, the experiments should to be run on the same system and on a similar scale it would be executed in production. In real-life this can be really difficult, since it is not easy to get access to a high-performance production system to run non-production experiments. The main reason being the high maintenance cost of such system. Even if we get access to the system, the system's owners will likely restrict the machine time, or the amount of computing resources that can be used for these experiments. This will, in turn, limit the number of experimental scenarios that can be tested, which may not allow the developer to find an optimal combination of parameters.

3.2 Ondes3D: a typical parallel iterative application

Ondes3D is an iterative parallel application developed by geophysics scientist at the BRGM, the *French Geological Survey* (*Bureau de Recherches Géologiques et Minières*). It implements a model to simulate the propagation of seismic waves. Understanding the wave propagation with respect to the structure of the Earth lies at the core of many analysis for quantitative seismic hazard assessment. Usually, seismic wave modeling considers the domain under study as an elastic linear material and the governing equations are given in three dimensions by:

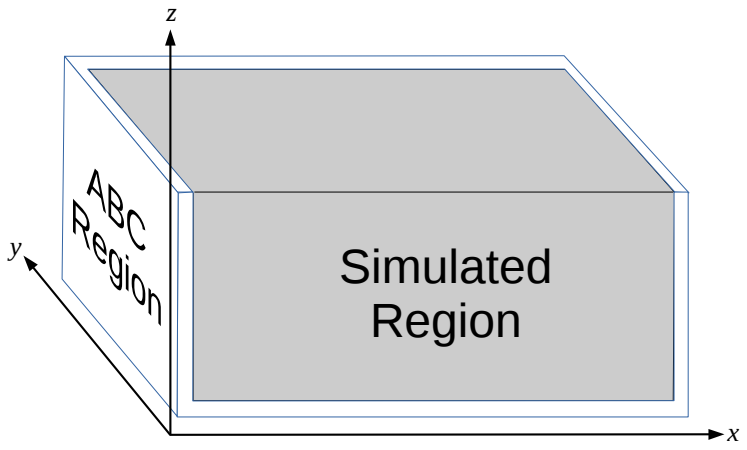
$$\begin{cases} \rho\frac{\partial}{\partial t}v_x &= \frac{\partial}{\partial x}\sigma_{xx} + \frac{\partial}{\partial y}\sigma_{xy} + \frac{\partial}{\partial z}\sigma_{xz} + f_x \\ \rho\frac{\partial}{\partial t}v_y &= \frac{\partial}{\partial x}\sigma_{yx} + \frac{\partial}{\partial y}\sigma_{yy} + \frac{\partial}{\partial z}\sigma_{yz} + f_y \\ \rho\frac{\partial}{\partial t}v_z &= \frac{\partial}{\partial x}\sigma_{zx} + \frac{\partial}{\partial y}\sigma_{zy} + \frac{\partial}{\partial z}\sigma_{zz} + f_z \end{cases} \tag{3.1a}$$

$$\begin{cases} \frac{\partial}{\partial t}\sigma_{xx} &= \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial x}v_x \\ \frac{\partial}{\partial t}\sigma_{yy} &= \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial y}v_y \\ \frac{\partial}{\partial t}\sigma_{zz} &= \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial z}v_z \\ \frac{\partial}{\partial t}\sigma_{xy} &= \mu\left(\frac{\partial}{\partial y}v_x + \frac{\partial}{\partial x}v_y\right) \\ \frac{\partial}{\partial t}\sigma_{xz} &= \mu\left(\frac{\partial}{\partial z}v_x + \frac{\partial}{\partial x}v_z\right) \\ \frac{\partial}{\partial t}\sigma_{yz} &= \mu\left(\frac{\partial}{\partial z}v_y + \frac{\partial}{\partial y}v_z\right). \end{cases} \tag{3.1b}$$

In the above equations, v and σ represent the velocity and the stress field, respectively, and f denotes a known external source force. ρ is the material density, while λ and μ are the elastic coefficients known as Lamé parameters. We consider an isotropic medium leading to a symmetric stress tensor. The dominant numerical scheme to solve the above equations is the explicit finite-difference method. A review can be found in (MOCZO; ROBERTSSON; EISNER, 2007). One of the key features of this scheme is the introduction of a staggered-grid for the discretization of the seismic wave equation. This improves the overall quality of the scheme in terms of numerical dissipation and stability, especially in the case of strong material heterogeneity.

One particularity of the three-dimensional simulation of seismic wave propagation is the consideration of a finite computing domain whereas the physical problem is unbounded. Additional numerical conditions are then required to absorb the energy at the artificial boundaries. On Figure 3.1 we have an illustration of the Absorbing Boundary Condition (ABC) region, situated at the lateral and bottom edges of the three-dimensional geometry. Inside this region a specific set of equations is implemented. For instance, the classical Perfectly Matched Layer conditions (PML) relies on the introduction of a sponge numerical function that provides exponential attenuation in the non-physical region (COLLINO; TSOGKA, 2001). In the PML zone, plane regions require damping in their normal direction. The number of standard equations that must be modified varies from one to three depending on the location of the grid point.

Figure 3.1: Vertical cross-section of the simulation domain, showing the absorbing boundary condition zone on the sides and on the bottom of the region.



Source: The author

The computational procedure is described in Algorithm 3.1 which corresponds to the time-dependent phase. The first loop is devoted to the computation of the velocity

components. We reuse these values in order to compute the stress field. The exchange phases correspond to explicit communication between neighboring subdomains in order to exchange boundary information.

Algorithm 3.1: Standard implementation of the elastodynamics equations based on MPI decomposition

```
1  for ∀ step of Δt simulated time do
2      for all x do
3          for all y do
4              for all z do
5                  Compute stress σ_xx, σ_yy, σ_zz, σ_xy, σ_xz, σ_yz at point p_xyz;
6              end
7          end
8      end
9      Exchange stress σ
10     for all x do
11         for all y do
12             for all z do
13                 Compute velocity v_x, v_y, v_z at p_xyz;
14             end
15         end
16     end
17     Exchange velocity v
18 end
```

The overall implementation could be described as follows. The first step is the initialization of the velocity, stress and external force components. The second step is the time-dependent computation. At this level, the time-step loop includes two triple nested loops over the three directions in space. The velocity and the stress components are evaluated with respect to the odd-even dependency between these fields (see Equations 3.1a and 3.1b). The Ondes3D software package conforms with this description (AOCHI et al., 2013). It implements the standard fourth-order in space numerical scheme and the Convolutional Perfectly Matched Layer (C-PML) method is used as absorbing boundary conditions because of its efficiency (KOMATITSCH; MARTIN, 2007). The standard thickness of this surrounding layer is ten grid points in each direction.

3.2.1 Parallel implementation

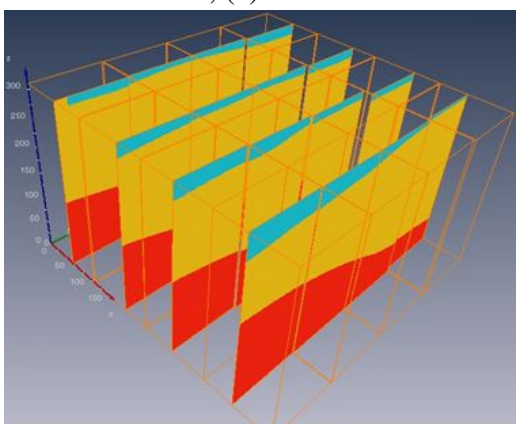
Ondes3D (DUPROS et al., 2010) approximates the partial differential equations (PDE) governing the elastodynamics of rock medium using finite-differences numerical methods (FDM) (MOCZO; ROBERTSSON; EISNER, 2007). More specifically, it implements the Finite-Difference Time-Domain (FDTD) method. Most parallel implementations of the FDTD method for the elastodynamics equations are based on a standard MPI Cartesian grid decomposition. The problem domain is statically partitioned in cuboids, as depicted in Figure 3.2a. Each iteration (see Figure 3.2b) corresponds to a given time-step and consists in calling three macro kernels (`Intermediates`, `Stress`, and `Velocity`) that apply a series of finite differences micro kernels (see the `CPML4` example in Figure 3.2c) to the whole domain. Message passing consists in asynchronous neighborhood communications intertwined with the three macro kernels. There is no global barrier, which enables a slightly asynchronous evolution of each process depending on the computational cost associated to a given part of the subdomain decomposition. Each process manages a cuboid with the same fixed geometry and runs the same regular code.

Despite this apparent regularity, Ondes3D suffers from major load imbalance that limits its scalability. The main source of imbalance has been previously identified (DUPROS; DO; AOCHI, 2013) as the extra-computation needed to deal with boundary conditions. A common characteristic of absorbing condition formulations is the introduction of different computational loads, especially at the lateral and bottom edges of the three-dimensional geometry. In terms of CPU-cost, the computation ratio between a grid point located in the physical domain or in the PML zone can vary from one to four depending on its location. The processes on the border of the domain have thus more work, which causes significant spatial imbalance when compared to processes that compute the inner part of the 3D region being simulated.

3.3 Characterizing spatial and temporal load imbalances in Ondes3D

Figure 3.3a depicts a 16×16 domain decomposition where each cell in the Cartesian grid represents one of the 256 processes in charge of a cuboid subdomain. In this heatmap, the color indicates the computational load per process during the first iteration of the execution of Ondes3D, before the triggering of the initial shock that originates in the 13×5 subdomain. Processes on the borders demonstrate a much higher computational load (red

Figure 3.2: The Ondes3D application: (a) an example of domain decomposition for 16 processes; (b) the three large kernels of the main loop, with intertwined neighborhood communications; (c) a detailed view of the small `CPML4` kernel (out of 24).



(a) 3D view of the heterogeneous rock medium, with a 4×4 domain decomposition; each process calculates a cuboid.

```
for (ts = 0; ts < N; ts++){
  Intermediates();

  Stress();
  //Intertwined Asynchronous
  //Neighborhood Communication

  Velocity();
  //Intertwined Asynchronous
  //Neighborhood Communication

}
```

(b) The main loop with three kernels: `Intermediates`, `Stress` and `Velocity`; no global synchronization.

```
static inline double CPML4 (double vp, double dump,
      double alpha, double kappa, double phidum, double dx,
      double dt, double x1, double x2, double x3, double x4) {
  double a, b;
  b = exp(-(vp * dump / kappa + alpha) * dt);
  a = 0.0;
  if (abs(vp * dump) > 0.000001)
    a = vp * dump * (b - 1.0) / (kappa * (vp * dump + kappa * alpha));
  return b * phidum + a *
    ((9. / 8.) * (x2 - x1) / dx - (1. / 24.) * (x4 - x3) /dx);
}
```

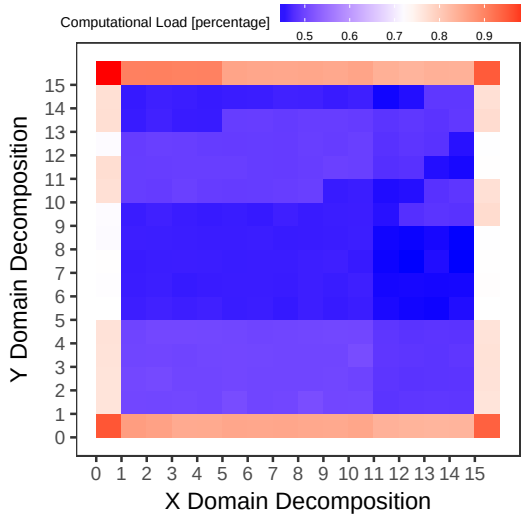
(c) The `CPML4` kernel, called nine times for each cell i, j, k in the `Intermediates` kernel when processing a subdomain. Variables `x1`, `x2`, `x3`, and `x4` represent the rock medium states (e.g., speed) that evolves along the iterations.

Source: The author

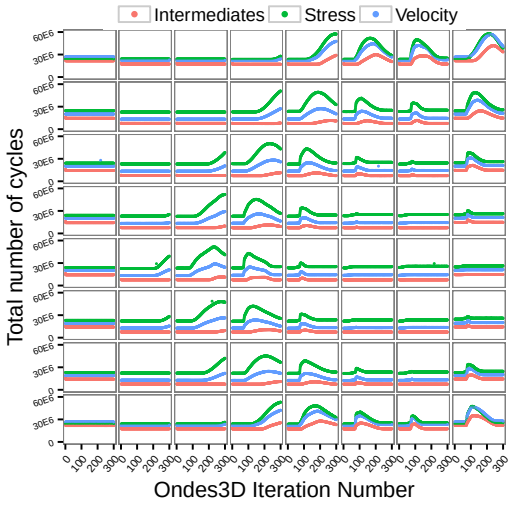
color) than those located inside the physical domain. Another, much more subtle, source of spatial imbalance (blue shades), depends mostly on the rock multi-layer configuration of the input (six layers for this scenario). Although minor, such effect exists and solely depends on the substrate geometry.

The code of Ondes3D does not exhibit any branching structure (convergence loops, refinements, thresholds) that could contribute to an evolution of computation load along simulation iterations. There are conditional branches, but they are related to absorbing conditions and thus solely to the fixed problem geometry. Yet, as illustrated in Figure 3.3b, one can observe a variability in computational cost along iterations that is even higher than the spatial variability incurred by the absorbing boundary conditions. This figure details the behavior of 64 processes (each box in the 8×8 grid), showing (in the vertical axis of

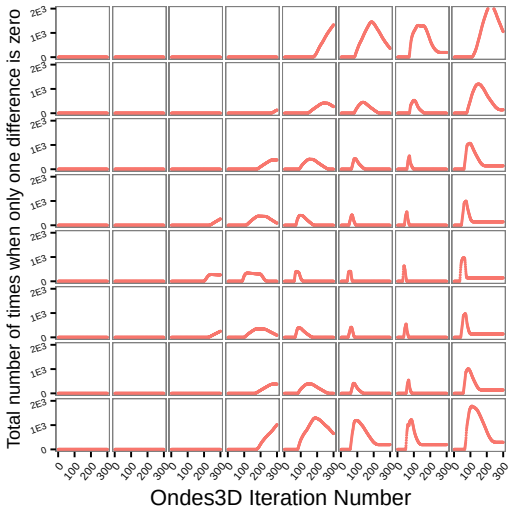
Figure 3.3: Load imbalance for the Ligurian workload: (a) spatial load imbalance; (b) temporal load imbalance for three kernels; (c) `CPML4` substrate values evaluating to distinct values; and (d) explaining temporal anomaly with `CPML4` argument analysis.



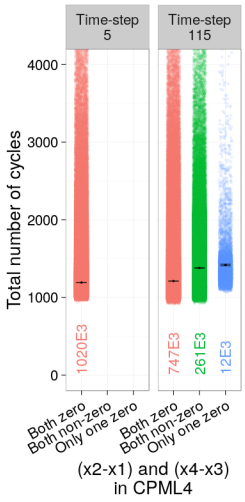
(a) Spatial imbalance for the first iteration represented by a color gradient for each rank in a 16×16 grid (256 processes).



(b) Total number of cycles for each kernel along iterations for each rank in a 8×8 grid (64 processes).



(c) How much the differences evaluate to contrasting values in the `CPML4` kernel along iterations, for each of the 64 ranks.



(d) Expected duration (cycles) and number of calls to `CPML4`.

Source: The author

each box) the total number of cycles measured with PAPI (the `PAPI_TOT_CYC` hardware counter) per macro kernel as a function of the iteration (horizontal axis). This metric seems to follow the earthquake shock progression, standing out around the eightieth iteration.

To explain the origin of this variable computational cost, we use the `CPML4` kernel shown in Figure 3.2c. This kernel is only one out of the 24 small inlined kernels but is perfectly representative of the other ones. It is called by the `Intermediates` macro kernel that iterates over the cuboid subdomain with three nested loops. For each cell of

the subdomain, the CPML4 kernel is called nine times with slightly different parameters, resulting in more than a million calls per process/iteration in a 64-process simulation, for this workload. In this code, variables dx and dt are simulation constants, while x1, x2, x3, and x4 are variables that describe the state of the rock medium (speed, pressure, and others), which unfolds along the iterations.

For completeness, we proposed and verified several hypothesis to explain the temporal load variation, among them: conditional branches in micro kernels (e.g., the if in CPML4 code of Figure 3.2c), variable duration of math functions depending on their input (e.g., the exp call in the CPML4 code), arithmetic exceptions at the architecture level, compiler level optimizations that could, for example, introduce conditional branches in the assembly code, data cache effects, higher cost of divisions when compared to multiplications, and so on. By either playing with compiler options, carefully looking at the assembly code, or finely instrumenting the different operations, we have ruled out all these assumptions. Nevertheless, we noticed a strong correlation between the evolution of the load and the instruction cache misses (PAPI_L2_ICM) and branch miss-predictions (PAPI_BR_MSP). As we will show next, this correlation is a consequence and not the cause of the evolution.

Let us consider the x1, x2, x3, and x4 arguments of the CPML4 kernel (still in Figure 3.2c). The return statement has the arithmetic expression that is evaluated in the FPU. So, we instrumented the CPML4 kernel to count how many times per time-step and per process these differences were equal to zero (let us name these numbers $n^0_{2,1}$for x2−x1 and $n^0_{4,3}$for x4−x3). Looking at them separately indicates no correlation with the temporal load evolution. However, the difference $|n^0_{2,1} - n^0_{4,3}|$, depicted in Figure 3.3c, is perfectly correlated with the computational load change as shown in Figure 3.3b, and with the growth of the branch miss-prediction counters. Intuitively, this value measures how often only one of the two differences is zero.

To confirm this hypothesis, we instrumented the CPML4 kernel to record its duration for each call (in cycles) along with the result of the two differences (x2−x1 and x4−x3). Since this kernel is called very often, we traced only the rank in the top-right corner of the domain decomposition, which demonstrates a strong variability in time, and only for two very different time-steps: iteration 5, which has low computational load, and iteration 115, which has maximal computational load. Figure 3.3d shows the number of cycles of a single call to the CPML4 kernel depending whether both differences are zero, non-zero or whether only one of them is zero. Just after simulation starts, in iteration 5, when the seism has not

yet reached this region, both differences are always zero (red points in Figure 3.3d) and the number of cycles is relatively small. In iteration 115, both differences are non zero (green points in Figure 3.3d) for more than 25% of the `CPML4` calls and the average execution time is then 168 cycles slower. The FPU optimization to speed up multiplications by zero is thus worthwhile. Interestingly, the situation where only one difference is zero (blue points in Figure 3.3d) is sporadic (about 1% of calls) but both the average and the minimum execution time are significantly longer. It is interesting to note that the corresponding cells i, j, k for which the condition is true are spatially organized but they do not relate at all to cache alignment and do not generate additional data cache misses but clearly more branch miss-predictions and L2 instruction cache misses that participate in the slow down of the code. The observed increased duration originates from the combination of both a speed-up of multiplications by zero and of branch miss-predictions in the FPU incurred by the irregular sequence of zeros and non-zeros.

Finally, even if we focused on the `CPML4` kernel here, all other small inlined kernels share the same structure. It is thus the aggregated contribution of all these small additional cycles that generates the temporal load variation.

3.3.1 The necessity of dynamic load balancing for Ondes3D

From the above it becomes clear that Ondes3D is, fundamentally, a spatially and temporally imbalanced code, despite the regularity of its computation kernels and data partitioning. Such imbalance is usually overlooked, as it requires a careful and fine analysis to be identified. Therefore, we believe that it may also be present in many other so-called regular applications.

Modeling and predicting the load imbalance of Ondes3D is difficult, as it strongly depends on the initial and evolving conditions of a given earthquake simulation. A possible solution to leverage this load balancing issue in Ondes3D could be to rely on Adaptive Mesh Refinement (AMR) (BERGER; COLELLA, 1989). It enables to have a heterogeneous partitioning of the problem domain and may thus provide a better performance. However, the actual variability of the Ondes3D computation kernels would still require some periodic data and computation re-balancing. Another difficulty is that the effort undertaken by the application developer to put AMR in place is usually very high. That is why we propose to use a simpler approach (from the application developer mindset) by mixing load balancing at runtime with over-decomposition, a strategy in the scope of the AMPI

framework (HUANG; LAWLOR; KALÉ, 2004; HUANG et al., 2006). Next section presents our AMPI-aware version of Ondes3D.

3.4 Modifying Ondes3D to support dynamic load balancing

To introduce load balancing based on over-decomposition to Ondes3D, we required an MPI-like programming language, and a runtime system with support for this load balancing technique. The latter is one of the main features of Charm++ (KALÉ; KRISHNAN, 1993). The requirement of an MPI-like programming language is covered by Adaptive MPI (HUANG; LAWLOR; KALÉ, 2004; HUANG et al., 2006), which we presented in Subsection 2.3.2. AMPI implements MPI processes as *virtual processors* (VPs), which are executed as user level threads. AMPI also separates the assignment of work to processes from the mapping of processes to processors. The former is defined explicitly by the programmer, while the latter is defined by the runtime system (RTS).

We started this work with an MPI implementation of Ondes3D. The porting of this code to AMPI consisted of modifying it to support two of its main features: over-decomposition through virtual processors (VPs) and migration. As previously explained, to support VPs we needed to remove or privatize global and static variables. Fortunately, Ondes3D did not have many global variables and most of them could be replaced by constants.

To support migrations, we need to be sure that all data belonging to the migrated process is moved along with it. The static variables are migrated automatically, but we need to guide the runtime to migrate the dynamically allocated variables. Ondes3D declares several data structures, as shown in Listing 3.1. In order to be able to migrate these data structures, we wrote Pack-and-Unpack (PUP) functions for each of them (as explained in Subsection 2.3.3. To illustrate this process, Listing 3.3 shows a snippet of the code to PUP `struct MEDIUM`. The definition of this data structure is shown in Listing 3.2.

The first function in Listing 3.3 is used to PUP a kind of tri-dimensional matrix included in the data structure. The second function (only a snippet is shown) is used to PUP the data structure itself. Besides packing and unpacking the data, the developer must also take care of allocating and freeing the memory used by the dynamic variables. This can also be observed on Listing 3.3. After packing the data, we free the memory used by it on the origin of the migration. In the same manner, on the destination, we first allocate the memory for the variable. Only after that, we can unpack the migrated data.

To simplify the PUP process, we created a new data structure, containing all the other data structures used by Ondes3D. This way, we can have a global PUP function that calls all the other PUP functions we implemented. The last step in this process was to register the PUP functions with the runtime system, thus associating the PUP routine with the created data structure.

Listing 3.1: Declaration of data structures in Ondes3D

```
/*From "main.c" (MPI version): */

/* Model parameters */
struct PARAMETERS PRM = {0};

/* Source : values */
struct SOURCE SRC = {0};

/* Velocity and stress */
struct VELOCITY v0 = {0};
struct STRESS   t0 = {0};

/* Medium */
struct MEDIUM MDM = {0};

/* Variables for the PML/CPML */
struct ABSORBING_BOUNDARY_CONDITION ABC = {0};

/* ANELASTICITIES */
struct ANELASTICITY ANL = {0};

/* OUTPUTS */
struct OUTPUTS OUT = {0};

/* Variables for the communication between the CPUs */
struct COMM_DIRECTION NORTH = {0},  SOUTH = {0}, EAST  = {0},  WEST  = {0};
```

Listing 3.2: Example of data structure in Ondes3D

```
/*From "struct.h" (MPI version): */
struct MEDIUM {
  int  ***imed;
  int numVoid;
  int numSea;
  char **name_mat;
  double  *laydep;
  int nLayer;
  int *k2ly0;
  double  *laydep2;
  int nLayer2;
  int *k2ly2;
  double *rho2, *mu2, *kap2;
  double  *rho0,  *mu0,  *kap0;
};
```

After the application was ready to support migration, we added a call to the function `MPI_Migrate`. This call tells the runtime that the VP is ready to be migrated. In our code, this routine is called every N time-steps, with N being defined at compile time. With this last modification, Ondes3D is ready to take advantage of AMPI's features such as over-decomposition and dynamic load balancing.

Although AMPI provides all features required to take advantage of dynamic load balancing on Ondes3D, there is still a question regarding its performance. As discussed before, it is important to guarantee that overhead coming from the runtime system and communication will not surpass the benefits of over-decomposition.

Listing 3.3: Example of PUP functions

```
/*From "pup.c" (AMPI version):*/

/*Function to serialize a 3d matrix (i3tensor).*/
void pup_i3tensor(pup_er p, int ****i3t, int x_start, int x_end,
                  int y_start, int y_end, int z_start, int z_end)
{
  if(pup_isUnpacking(p)){ // Allocate some memory to store the data.
    *i3t = i3tensor(x_start, x_end, y_start, y_end, z_start, z_end);
  }

  /*Calculate the start and the size of the data.*/
  int *real_start = (*i3t)[x_start][y_start] + z_start;
  int dim_x = x_end - x_start + 1;
  int dim_y = y_end - y_start + 1;
  int dim_z = z_end - z_start + 1;

  /*Pack or Unpack the data.*/
  pup_ints(p, real_start, dim_x * dim_y * dim_z);

  if(pup_isPacking(p)){// Free the memory used by the packed data.
    free_i3tensor(*i3t, x_start, x_end, y_start, y_end, z_start, z_end);
  }
}

/*A fragment of the function that serializes struct MEDIUM.*/
pup_medium(pup_er p, struct MEDIUM *MDM, struct PARAMETERS *PRM)
{
  int i;
  pup_int(p, &(MDM->nLayer));
  pup_int(p, &(MDM->nLayer2));
  if(model == GEOLOGICAL){
    if(pup_isUnpacking(p)){//Allocate memory for dynamic variables.
      MDM->name_mat = calloc(MDM->nLayer, sizeof(char*));
      for (i = 0 ; i < MDM->nLayer; i++){
              MDM->name_mat[i] = calloc(STRL, sizeof (char));
      }
    }
    pup_i3tensor(p, &MDM->imed, 1, PRM->mpmx + 2, -1, PRM->mpmy + 2,
                 PRM->zMin - PRM->delta, PRM->zMax0);
    pup_int(p, &(MDM->numVoid));
    pup_int(p, &(MDM->numSea));
    for (i = 0 ; i < MDM->nLayer; i++){
      pup_chars(p, MDM->name_mat[i], STRL);
    }
        ...
}
```

3.5 Evaluation of the benefits of dynamic load balancing

In the previous section, we presented details of the porting of Ondes3D to AMPI. With this done, we now needed to evaluate the actual benefits gained by employing AMPI's load balancing capabilities. In this section, we present such evaluation. In a first moment,

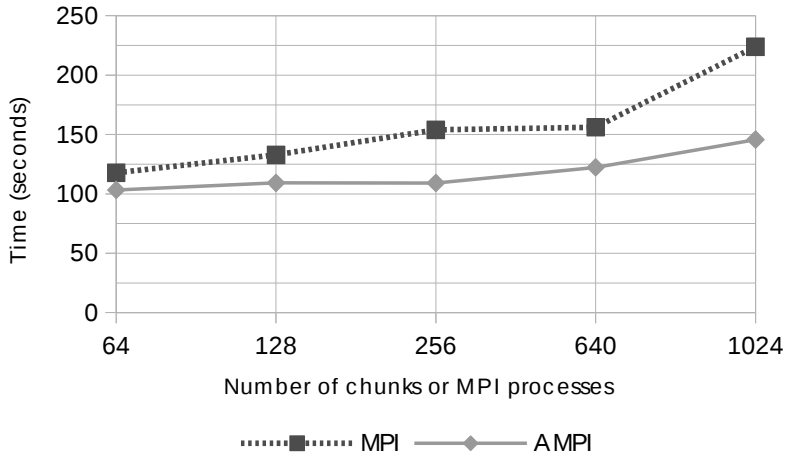
on Subsection 3.5.1, we evaluate how over-decomposition itself affects performance. For this purpose, we present a small evaluation of the performance of Ondes3D without load balancing while increasing the number of MPI ranks but keeping the number of actual processes fixed. Next, in Subsection 3.5.2, we present the actual load balancing evaluation. There we evaluate the performance of Ondes3D with different load balancers and levels of over-decomposition and on two different clusters of different scales.

3.5.1 Evaluation of Over-decomposition

To assert that running multiple virtual processors with AMPI would not harm Ondes3D's performance, we measured the total execution times of Ondes3D with MPI and AMPI on a cluster using one or more MPI processes (or, in the case of AMPI, VPs) per core. This cluster's nodes are equipped with two quad-core processors and are interconnected by an Infiniband network. For more details on this cluster, see Table 3.1, in Subsection 3.5.2. The MPI implementation we used in our tests uses threads instead of processes. With AMPI, we created a system process per core and a user thread for each VP.

Figure 3.4 shows the execution time of 100 time-steps of Ondes3D using MPI and AMPI on 8 nodes (64 cores). The vertical axis represents time in seconds (the smaller the time, the better the performance), while the horizontal axis represents the variation of Ondes3D chunks (MPI ranks or AMPI VPs). In this experiment, we varied the number of chunks per core from 1 (64 chunks) up to 16 (1024 chunks). From Figure 3.4 it becomes clear that when we increase the number of chunks, MPI loses performance much faster than AMPI. This is expected, since MPI is not designed for over-decomposition. For this reason, our baseline for comparison is the result for MPI with only one chunk per core. The results in Figure 3.4 show that AMPI starts with an execution time about 12% shorter than MPI. The AMPI execution time increases nearly 5% when running 2 VPs per core, but is still 7.27% faster than the baseline. With 4 VPs per core, the AMPI performance keeps almost steady. Still, we can see in Figure 3.4 that performance starts to decrease as the number of chunks per core increases to 10. Nevertheless, the execution time is still only 3.82% higher than the baseline. These results show that using virtual processors to over-decompose an application incurs in a low overhead over using real MPI processes. With 16 VPs per core, however, the execution times are 23.70% higher than the baseline. This may be linked to an increase in communication, and an initial VP distribution that is unbalanced.

Figure 3.4: Comparison of MPI and AMPI with Ondes3D on eight nodes.



Source: The author

These results help us justify the use of AMPI and its virtual processors to improve Ondes3D's performance. Additionally, the over-decomposition of Ondes3D enables the use of load balancing algorithms to further optimize its performance.

3.5.2 Evaluation of Load Balancing

This evaluation is based on the hypothesis that dynamic load balancing can improve the performance of Ondes3D. To assess that, we measured the execution time of Ondes3D over AMPI with six different load balancers. We also made measurements without load balancing, in order to provide a baseline for comparison.

For the experiments presented here, we ran a simulation based on the Mw 6.6 2007 Niigata Chuetsu-Oki, Japan earthquake (AOCHI et al., 2013). The full simulation is composed of 6000 time-steps. In our experiments, due to time constraints, the application was configured to simulate only the first 500 time-steps. The load balancing routine is called every 20 time-steps (excluding the last one), which results in a total of 24 calls. Besides evaluating different load balancers, we also vary the number of VPs per core. We ran tests with one, eight, and 16 VPs per core. This over-decomposition is required for the load balancers to redistribute the work.

In this section we present two sets of experiments, performed in two different systems, described on Table 3.1. In the first set, we have small scale experiments, which were performed on 8 nodes of *System #1*. The second set consists of larger scale tests,

performed on 8 and 12 nodes on *System #2*. Next, we present the load balancers we tested in these experiments.

Table 3.1: Configuration of the systems used in the experiments.

	System #1	**System #2**
CPU model	Intel Xeon E5520 (Nehalem)	AMD Opteron 6344
Last Level Cache	8 MB	16MB
Cores per CPU	4	12
CPUs per node	2	2
Number of nodes	8	12
Network	InfiniBand 40G (Mellanox ConnectX IB 4X QDR MT26428)	InfiniBand 40G (Mellanox MT27500 Family [ConnectX-3])

Source: The author

3.5.2.1 Load balancers employed in the experiments

Charm++ is distributed with many load balancing algorithms. For our experiments, we chose the following subgroup of centralized algorithms:

- **GreedyLB** is used to quickly mitigate load imbalance with aggressive scheduling decisions. It is a greedy algorithm that uses only VP loads for its decisions. GreedyLB iteratively maps the virtual processor with the biggest load to the least loaded core.
- **GreedyCommLB** is an extension of GreedyLB that includes communication loads computed using the amount of messages and bytes exchanged among VPs. Instead of simply mapping the VP with the biggest load to the least loaded core, GreedyCommLB also considers all other cores that have VPs that it communicates with.
- **RefineLB** is a less aggressive algorithm which tries to improve load balance by incrementally adjusting the current scheduling. It checks all possible VP migrations from the most loaded core to the cores below the average load, and migrates the VP that leaves its new core the closest to the average. This leads to less migrations than GreedyLB and GreedyCommLB.
- **RefineCommLB** adds communications costs to RefineLB. It considers that an overhead is present whenever a VP is mapped to a different core than the ones that contain VPs that it communicates with.

In addition to Charm++'s algorithms, we experimented with two topology-aware load balancing algorithms proposed by Pilla et al. (PILLA et al., 2012; PILLA et al., 2013). Besides the information provided by the load balancing framework, these algorithms use a machine topology representation that includes benchmarked communication costs. In our larger scale tests, we also included a hierarchical load balancer which makes use of the two topology aware algorithms. These three load balancers are detailed below:

- **NucoLB** was developed for parallel platforms involving non-uniform levels in their topologies (mainly NUMA nodes) (PILLA et al., 2012). It is a refinement-based greedy list scheduling algorithm that assigns the VP with the largest load to the core that presents the smallest cost. This cost is related to the current load on the core and the communication costs related to mapping such VP to it. NucoLB is an aggressive algorithm, like GreedyCommLB, but its scheduling decisions involve more parameters and result in less migrations (as it considers the current scheduling).
- **HwTopoLB** tries to find the best trade-off between mapping a VP to a less used core and mapping it closer to the other VPs it communicates with (PILLA et al., 2013). It considers the whole machine topology in its decisions, which includes caches, memory, and the network. HwTopoLB iteratively chooses a core and a VP that is assigned to it. The algorithm then evaluates all possible mappings to choose the one that has the highest probability of minimizing the makespan. Besides being lighter than NucoLB, HwTopoLB is proven to asymptotically converge to an optimal solution (PILLA et al., 2013).
- **HierarchicalLB** (PILLA, 2014) is a hierarchical algorithm that splits the scheduling into two levels. At platform level, it employs HwTopoLB to distribute VPs among compute nodes. After that, it employs HwTopoLB or NucoLB at compute node level, to map VPs to cores. By splitting the load balancing process, HierarchicalLB becomes more scalable than centralized algorithms.

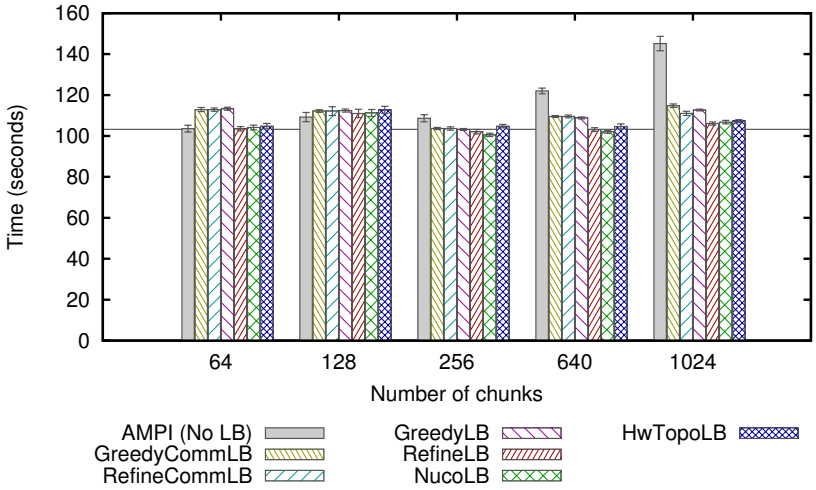
3.5.2.2 Small scale evaluation

First, we will present the results of the small scale experiments, with 64, 256, and 1024 chunks (or VPs). The grid dimensions used in these simulations were 600×600×200. The experiments presented in this section were performed on 8 nodes of *System #1* described in Table 3.1. The experiments on this subsection, along with their results, were previously published in the proceeding of a scientific conference (TESSER et al., 2014).

Our first batch of experiments was designed to provide initial insights on the effect of load balancing. The application was configured to simulate only the first 100 time-steps with a resolution of 122 million grid points. The load balancing routine is called every 20 time-steps (excluding the last one), which results in a total of 4 calls. Besides evaluating different load balancers, we also vary the number of VPs per core. We start with one VP per core, for a total of 64 VPs (or chunks), and go up to sixteen VPs per core (1024 chunks).

The average execution times are presented in a bar chart on Figure 3.5. The bars are grouped along the horizontal axis according to the number of chunks. The vertical axis represents the average execution time in seconds. On top of each bar we depict the corresponding 95% confidence interval.

Figure 3.5: Average execution time for 100 time-steps of Ondes3D with different load balancers on *System #1*.



Source: The author

As expected, load balancers were unable to provide performance improvements when only one VP is available per core. This also happened when running two VPs per core (128 chunks), as the workload was not partitioned enough to be redistributed. The first gains in performance happened with 256 chunks, or 4 VPs per core. When compared to the fastest configuration without load balancing, the improvements were of 0.38% with GreedyLB, 1.43% with RefineLB, and 2.94% with NucoLB. Only NucoLB can be said to have improved the average execution time with 95% confidence.

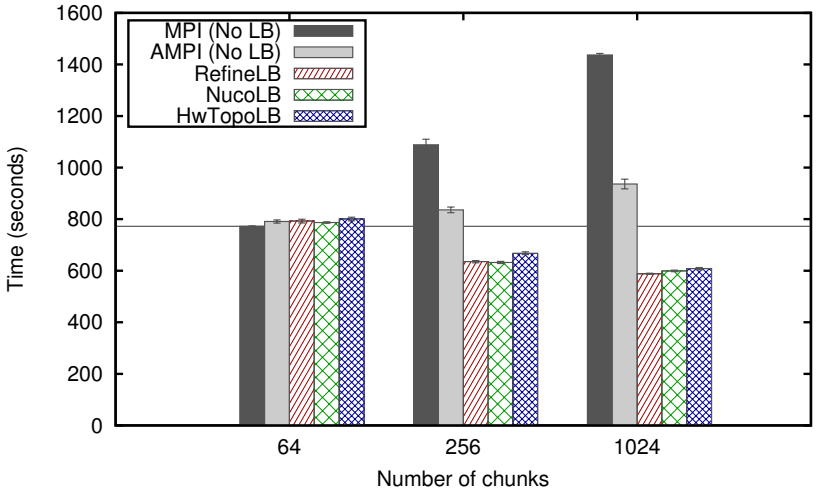
The experiments with a larger number of chunks, namely 640 and 1024, showed either a similar performance to the one with 256 chunks, or a decrease in performance. Although the communication increases with the amount of VPs, it was not responsible for the increase in execution time. The main reason for that lies on the application's initial

time-steps, which are more unbalanced due to the over-decomposition (as could be seen in Figure 3.4). The initial 20 time-steps before load balancing result in a 10 seconds increase in total execution time when executing 16 VPs per core (1024 total). Such overhead makes load balancing less effective in this scenario. To overcome that, our second batch of experiments includes more time-steps, as 100 represents only one sixtieth of the total simulation.

In the second batch of experiments we increased the number of simulated time-steps to 500. This gives us a more significant execution time, while keeping it short enough to enable experiments with multiple over-decomposition configurations and load balancers. We kept the same platform and interval between load balancing calls used in the previous tests. For these experiments, we selected only the load balancers that showed the best performances with over-decomposition in our initial measurements: RefineLB, NucoLB, and HwTopoLB. To have a baseline to compare our results, we also measured the execution times with MPI and with AMPI without load balancing.

The results with 500 time-steps are presented on Figure 3.6. With 64 chunks, MPI achieved the best performance among all configurations in this scenario. It outperforms AMPI without a load balancer by 2.3%. For this reason, we chose it as the baseline to analyze the remaining results.

Figure 3.6: Average execution time for 500 time-steps of Ondes3D with different load balancers on *System #1*.



Source: The author

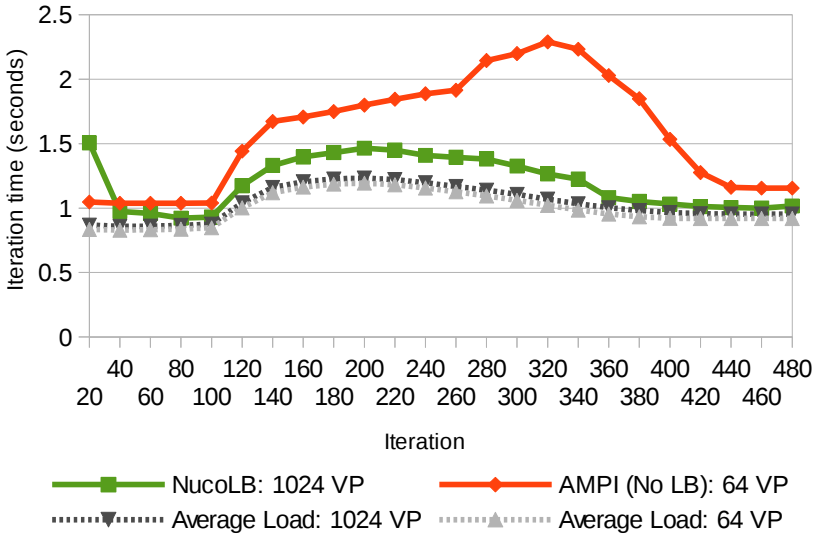
When we increased the number of VPs from 64 to 256, both MPI and AMPI's performances decreased. Meanwhile, all three tested load balancers were able to achieve performance improvements over the baseline. The average execution times were improved by 18.19% with NucoLB, 17.78% with RefineLB, and 13.53% with HwTopoLB. It is

worth noting that there is an intersection between the confidence intervals of RefineLB and NucoLB, which means that we cannot identify which one is better in this experiment.

The best performance improvements for Ondes3D were obtained when over-decomposing it into 1024 chunks (16 per core), as can be seen in Figure 3.6. When compared to the baseline, the simulation time was reduced by 23.85% with RefineLB, 22.44% with NucoLB, and 21.27% with HwTopoLB.

To illustrate the reason behind such improvements, Figure 3.7 shows the average iteration times measured in our experiments with AMPI and NucoLB. The horizontal axis represents the iterations, while the vertical axis represents time, in seconds. The two curves with average loads provide a lower bound for the iteration time. They represent what would be the best possible iteration time if the application was completely balanced, while keeping the same workload. Due to the unbalanced characteristics of Ondes3D, this ideal scenario is unlikely to be achievable in the real world.

Figure 3.7: Average iteration time and average load per core for Ondes3D with 64 and 1024 VPs and NucoLB.



Source: The author

Other two curves are present in Figure 3.7. One of them illustrates how the average iteration time varies when executing the application with 64 VPs and no load balancer (the baseline in this scenario), while the other shows the performance we obtained with NucoLB and 1024 VPs. The smaller the iteration time, the better the performance.

Four different phenomena can be seen in this figure. The first one is the average iteration time in the first 20 time-steps. NucoLB starts with an iteration time 0.45 seconds bigger (44%) than AMPI with no load balancer. This is due to the over-decomposition of

the application, as mentioned before. Still, NucoLB is able to quickly fix this difference and provide an improvement of 6.5% over the baseline.

The second phenomenon starts at the 120th time-step, when the average load and iteration times start to increase.

The unbalanced baseline iteration time rapidly increases, while NucoLB is able to stay close to the optimal bounds. This is a difficult period for load balancing algorithms, as the load of each VP changes at each time-step. Even so, dynamic load balancing is still able to provide performance improvements to Ondes3D. Static approaches, however, would not be able handle this behavior.

The third phenomenon happens as the average load starts to decrease while the load imbalance seen at the AMPI baseline increases. The same is not seen on the iteration times obtained by NucoLB, because the VPs from that most loaded core have already been distributed over the platform. Lastly, we can see that the average load starts to stabilize again after the 360th time-step. In this scenario, NucoLB achieves an average iteration time that is only 5% bigger than the average load with 1024 chunks, which represents its limit. These results corroborate our initial hypothesis by showing how dynamic load balancing can improve the performance of Ondes3D.

3.5.2.3 Larger scale evaluation

In this subsection, we present larger scale experiments executed on *System #2*, described on Table 3.1. In these experiments, we restricted our evaluation to the load balancers that presented the best results in our previous tests. The selected load balancers were RefineLB, NucoLB, and HwTopoLB. Besides these, we also include tests with HierarchicalLB.

The previous experiments were performed on a small scale system (64 cores). Next, we present results obtained on a larger scale platform, using 8 and 12 nodes of *System #2* (192 and 288 cores, respectively). Besides the load balancers we tested previously, these new experiments include measurements with HierarchicalLB in two different configurations. HierarchicalLB 1 uses HwTopoLB at platform level and NucoLB inside each node. HierarchicalLB 2 uses HwTopoLB at both levels. The problem sizes we used in these new tests are larger than the previous ones. For the tests with 192 cores (8 nodes), we used the dimensions 960×960×320, while with 288 cores the dimensions were 1152×1152×384. As the previous results demonstrate, over-decomposition only improves performance with load balancers, and vice versa. For this reason, measurements without load balancers

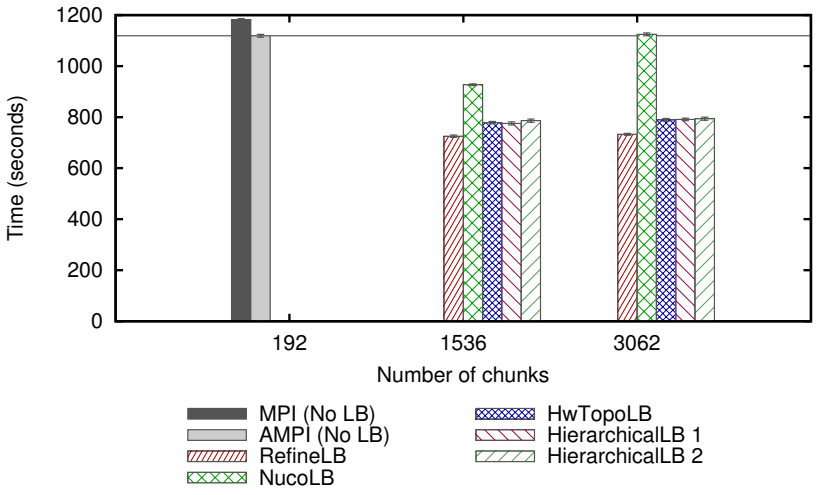
were limited to one chunk per core, while measurements with load balancers always use over-decomposition.

The results with 192 cores are presented in Figure 3.8. Without load balancing, AMPI executed 5.3% faster than MPI, so AMPI will serve as baseline for comparison.

In a standard situation (one chunk per physical core), AMPI is not designed to outperform MPI libraries. Nevertheless, results described in (PÉRACHE; CARRIBAULT; JOURDREN, 2009) with a three-dimensional Jacobi stencil demonstrate significant speedups over the native MPI library (a maximum of 12% on 512 processors) without any overloading strategy. The authors explain these results by the benefits coming from the internal mechanisms of the AMPI library. Additionally, as explained in (HUANG et al., 2006), the AMPI library can benefit from a better strategy for communications at the shared-memory-level in comparison with costly copies of buffers generally implemented by the native MPI library.

In this scenario, the best performance improvement was achieved by RefineLB. It improved the execution time by 35.18% with 8 chunks per core, and by 34.52% with 16 chunks per core. HwTopoLB, HierarchicalLB 1, and HierarchicalLB 2 achieved improvements of approximately 30% with 8 chunks per core and near 29% with 16 chunks per core. The worst result was obtained with NucoLB, which ended increasing the execution time by 0.5% with 16 chunks per core. This result illustrates the lack of scalability of NucoLB, which is caused by the large increase in execution time of its load balancing heuristic.

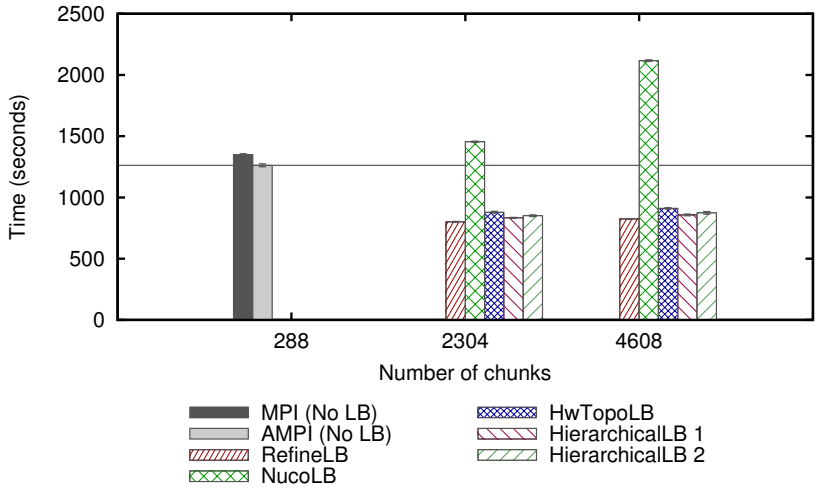
Figure 3.8: Average execution times of Ondes3D on 192 cores of *System #2*, with different load balancers.



Source: The author

The results of the tests with 288 cores are shown in Figure 3.9. Again, the baseline for comparison is the result for AMPI without load balancing, which was 6.29% better than MPI. RefineLB remained the best load balancer, gaining 36.58% performance with 8 chunks per core, and 34.72% with 16 chunks per core. The second best load balancer was HierarchicalLB 1, with an improvement of 33.97% with 8 chunks per core, and 32.12% with 16 chunks per core. Meanwhile, HwTopoLB presented slightly worse results, with gains of 30.35% with 8 chunks per core and 27.83% with 16 chunks per core. This shows that, in this larger scale, HierarchicalLB was more efficient in handling Ondes3D's imbalance than the centralized HwTopoLB algorithm.

Figure 3.9: Average execution times of Ondes3D on 288 cores of *System #2*, with different load balancers.



Source: The author

These last results show that, even on a larger scale, dynamic load balancing with over-decomposition improves performance of Ondes3D. These results were even better than the ones obtained on the small scale system, with gains in execution time up to 36.58%. This emphasizes how load imbalance becomes a bigger problem as application and platform scale up.

3.6 The cost of evaluating dynamic load balancing with over-decomposition

The focus of our work is to facilitate the evaluation of the benefits of over-decomposition based dynamic load balancing to legacy iterative parallel applications. In this chapter we demonstrated the traditional evaluation approach, through real-life experiments. For this purpose, we modified a seismic wave propagation simulation code to

take advantage AMPI for load balancing. In our experiments, we were able to improve the performance of the application up to 36.85%.

Besides evaluating the results of load balancing, this chapter illustrates the amount of work necessary to perform such an evaluation. First, we had to modify the code to be fully supported by the runtime system. This already represents an initial cost in terms of time, even before we can evaluate anything.

Next, we had to run experiments with different configurations. In our case, we tested different load balancing heuristics and levels of over-decomposition. We tested 45 configurations for the initial 100 time-step tests, 15 configurations for the 500 time-steps experiments on System #1, and 14 configurations for the experiments on System #2. This gives us a total of 74 different test configurations. Besides that, we had to test each configuration several times, in order to get statistically significant results. All these tests represent a major cost in terms of time. Which could be even larger had we tested different load balancing frequencies, which we avoided exactly because of this cost.

Besides the costs in terms of time, we also have the costs in terms of computational resources. In a ideal configuration, we would to run the experiments in the production system in a scale as close as possible as the one the application would run in production. In the real world, however, it is quite hard to get access to these resources for as long as we need to run our experiments. Even for the experiments presented on Subsection 3.5.2.3, we only had access to the system for a limited amount of time, which led us to limit our parameter exploration (i.e., we tested less load balancers and over-decomposition levels than in the previous experiments).

Aiming to mitigate these costs, we developed a low-cost simulation workflow to evaluate the performance of dynamic load balancing with over-decomposition for iterative parallel applications. This workflow is presented in next section, and aims at allowing this evaluation at a reduced cost in terms of time and using a single computer node. All of this with minimal application modification.

4 A SIMULATION WORKFLOW TO EVALUATE THE PERFORMANCE OF DYNAMIC LOAD BALANCING WITH OVER-DECOMPOSITION

As discussed at the end of the previous chapter, evaluating the performance dynamic load balancing can entail a big cost in both time and computational resources. To solve this problem, we propose a new simulation workflow, which we call *Simulated Adaptive MPI* (SAMPI). It aims to provide an estimation of the benefits of dynamic load balancing with over-decomposition at low cost, both in terms of computational resources and time (for both development and evaluation). To achieve these goals, the simulation should:

1. **Be fast:** simulations should last only a fraction of the time of a real execution, thus enabling the quick evaluation of several parameter combinations;
2. **Use few resources:** no need to run at large-scale or to use the production system for the evaluation;
3. **Require minimal modification of the application:** enabling the early evaluation of the potential benefits from load balancing, before implementing the changes needed to support of the runtime system (e.g., Charm++/AMPI).

In this work, we focus on the improvement of legacy iterative MPI-based applications such as Ondes3D. Therefore, we mimic the behavior of the dynamic load balancing algorithms implemented by Adaptive MPI (AMPI) (HUANG; LAWLOR; KALÉ, 2004).

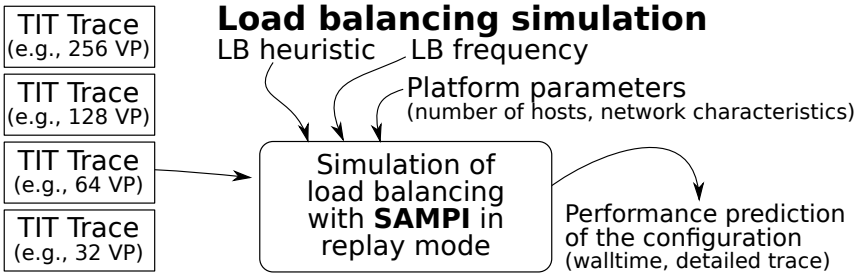
Our workflow relies on a combination of two simulation modes: *sequential emulation* and *trace-replay*. Sequential emulation enables the execution of the unmodified parallel application using a single processor core. During emulation, we record a *time-independent trace* (TIT) of its execution, which represents its computation and communication behavior. The load balancing simulation is done in trace-replay mode, receiving as input the trace generated in emulation mode. Since both computation and communication events are simulated, trace-replay takes a significantly shorter time to finish than an actual executions of the application.

4.1 Initial trace-replay based load balancing simulation workflow

Figure 4.1 shows the initial SAMPI simulator workflow, which exclusively uses trace-replay. The simulator takes as input a *Time-Independent Trace* (TIT) with a certain number of VPs. This TIT trace is generated as output of a previous emulation of the

application. SAMPI replays the trace to provide makespan predictions for a given load balancing configuration (heuristic and frequency) and platform (number of hosts and their computing power, network bandwidth/latency). The use of trace-replay allows to quickly and easily explore the load balancing parameter space to find the best combination of load balancer heuristic, frequency of load balancing, for a given application workload.

Figure 4.1: Initial version of the SAMPI simulation workflow: As input, `TIT` traces with different number of VPs (256, 128, 64, and 32 in this example); the `SAMPI` simulator replays these traces to provide makespan predictions for alternative load balancing heuristics, migration frequency, and platform characteristics (number of hosts, network capacity).



Source: The author

To use this load balancing simulator, the only modification you need to make to the application is to add a special call to trigger the load balancing procedure, in the appropriate place in the code. There is no need to remove global and static variables, or to write data serialization functions, as required by AMPI. In our implementation, inspired by AMPI, we named this function `MPI_Migrate`. When this function is called, the emulator generates an event in the TIT trace, to let the trace-replay simulator know when the load balancing must be triggered. Ideally, this call should be made at every iteration of the application. This way, we can change the frequency of load-balancing in simulation, by controlling when (i.e., in which iteration) these calls will actually result in the execution of the load balancer.

The main problem of our SAMPI simulator is that it takes as input TIT trace files with a specific number of VPs. That is, this parameter can not be changed during simulation. There are two alternatives to obtain them: (a) to allocate resources on a cluster to run the application in parallel, and trace its computing/communication behavior during the execution; and (b) to use a simulator to emulate and trace this execution using a single node. In both cases, these traces need to be generated in, or converted to, the TIT trace format used by the trace-replay simulation. Each of these alternatives has advantages/drawbacks: the first option requires an execution using the whole cluster, but trace collection is much faster. The second option is much cheaper in terms of resources (only one node), but it

is extremely time-consuming, since sequential emulation serializes the execution of the application.

Our goal here is to reduce the cost of obtaining multiple TIT traces by deriving all of them from a single emulation, i.e., a single TIT trace. To do so, we propose two performance modeling strategies that build upon the computational characteristics of the iterative applications with regular domain partitioning (e.g., Ondes3D). Even in the presence of multiple sources of load imbalance, these two techniques enable to faithfully capture the computational behavior of the application:

1. **MPI downscaling through spatial aggregation** (Section 4.2): we show how a coarse grain execution trace (e.g., 16 MPI processes) can be derived from a fine grain execution trace (e.g., 256 MPI processes) by spatially aggregating computational costs considering the fixed domain decomposition of the application. The only requirement for this is that the computational load of each subdomain during a specific iteration be dependent only of its position. Such procedure is straightforward, brings no information loss, and enables one to artificially reduce the number of MPI processes.
2. **Application upscaling through temporal/spatial extrapolation** (Section 4.3): we detail how we can derive fine grain execution traces from a coarser grained execution, by adjusting application specific parameters. Although some computational performance details might be lost, the bulk of its computational behavior may be kept, up to a certain extrapolation factor. If so, this procedure makes it much faster to obtain an initial trace of the behavior of the application.

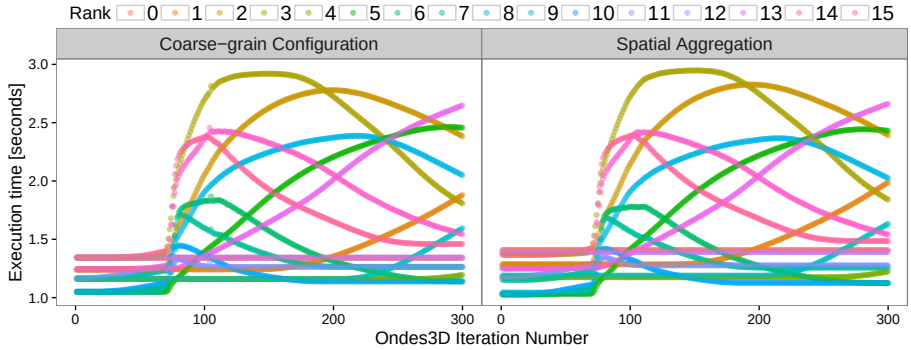
4.2 Spatial Aggregation to reduce the number of MPI Processes

In many applications that present dynamic load imbalance, such as the one found in Ondes3D (Section 3.3), it seems difficult to anticipate the performance by solely looking at the input data, without running the code at least once. Yet, as far as the application relies on a regular domain-based decomposition, we show that it is possible to derive a coarse grain execution trace from a fine grain execution trace, effectively creating a new trace with a reduced number of MPI processes. This process should work for any application in which, such as Ondes3D, the amount of work in each subdomain during a given iteration depends solely on its position in the domain. In this case, the amount of work induced by

a given area of the domain at a given time-step corresponds to the sum of the amount of work of the corresponding sub-areas.

To illustrate how this process works, we illustrate its application to Ondes3D. On Figure 3.3a we have shown a representative scenario with a 16×16 decomposition. When using instead a coarse grain 4×4 grid (a 8-factor spatial aggregation), the computational load of each process is the sum of the 16 corresponding processes in the fine grain 16×16 decomposition. To calculate this, we divide the X and Y coordinates of the original domain decomposition and then sum up the computational load of the 16 processes belonging to each combination of the X and Y coordinates of the target grid. Figure 4.2 shows the computing time (vertical axis) of each main loop iteration of Ondes3D as a function of the time-step (horizontal) for each rank (colors). The left facet shows the measurements from an execution with a coarse-grain 16 process (4×4) grid. The right one shows the resulting computing time for the same 16 process grid, but now aggregated from a fine-grain 256 process (16×16) grid. The difference between the coarse-grain trace and the aggregation from the fine-grain trace is minimal and allows to fully capture the temporal evolution for a given earthquake scenario after the spatial aggregation.

Figure 4.2: Execution time (vertical axis) along the time-steps (horizontal) for each rank (color) considering all three computation kernels of Ondes3D; the left facet is the behavior of a 16-process run, the right facet shows the same, but calculated from a 256-process run.



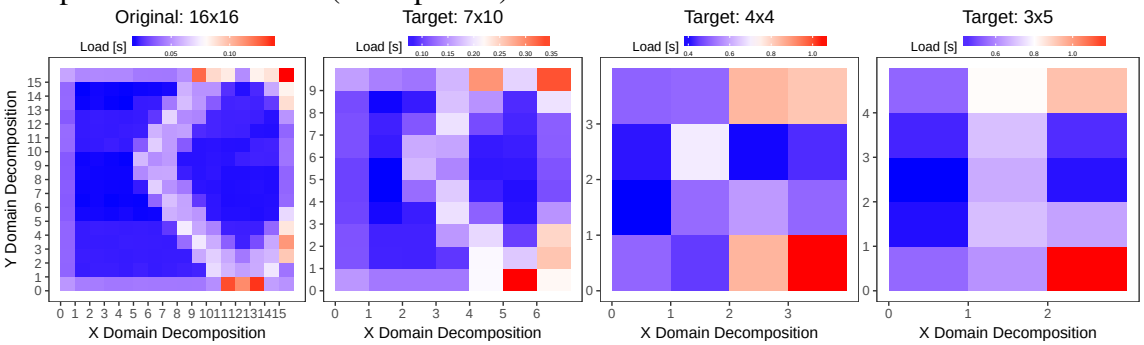
Source: The author

The procedure illustrated in Figure 4.2 is straightforward because it adopts division factors leading to rounded (in the integer space) downscaling, in both dimensions. However, very frequently we need to downscale the number of processes to a non-trivial configuration, i.e., from 256 to 21 processes. That would obviously lead to a non-integer repartition. Our approach to do spatial aggregation captures such non-trivial scenarios, enabling one to reconstruct any $P \times Q$ domain decomposition configuration. It works by creating a target grid and then calculating the computational intersection against the original grid, for every combination of new X and Y coordinates. Since the original grid is much more

fine-grained (smaller cells), we need to sum up the computational cost of the original cells proportional to the areas that intersect the larger (coarse-grained) cells.

This methodology is illustrated in Figure 4.3, showing the original domain decomposition 16 × 16 (left) and four derived grids for which the computational behavior has been aggregated: 7 × 10 (center left), 4 × 4 (center right) and 3 × 5 (right). We have chosen to depict the `Velocity` kernel of Ondes3D on iteration 115 of the Ligurian case since it is the moment with the largest load imbalance (see Figure 4.2). This mechanism allows us to generate coarse-grain computation traces at barely no cost.

Figure 4.3: Four domain decompositions: the original 16x16 scenario (left) and three spatial aggregation scenarios: 7x10 (center left), 4x4 (center right), and 3x5 (right); the computational load scale (color palette) is different for each of the four cases.



Source: The author

With the technique described above, we are able to aggregate the computation costs from a fine-grain execution into a coarse-grain computation trace. But we still have to solve the problem of estimating the *communication costs* and *migration payloads* for the coarse-grain execution. For this purpose, we perform a *new emulation* of the application with the coarse-grain decomposition, in which *we inject the aggregated computation costs* obtained in the previous stage. So, in this new emulation we do not actually execute the computations. Instead, we simulate them, at almost no cost, through a simple function call to the simulator. As a result, in this coarse-grain emulation, both computation and communication are simulated. Therefore, we only pay the full cost for application initialization (i.e., operations such as memory allocation, initialization of data structures, and reading of input files) and finalization.

4.3 Application-Level Rescaling

This improvement to our simulation workflow aims to decrease the time needed to trace the execution of the application, by modifying its input parameters (e.g., spatial

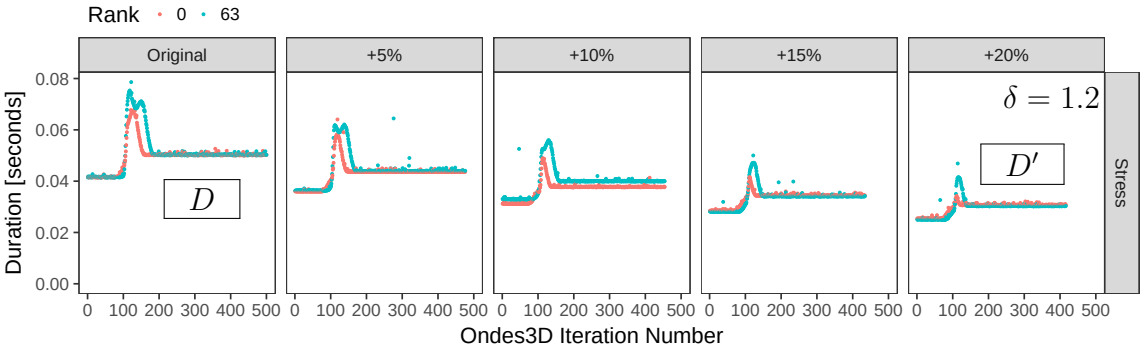
and temporal resolution). The execution of the application with modified parameters results in a coarse grain trace whose performance information can then be rescaled to the original configuration. For this to work, we need to analyze to what extent we can modify the parameter values without losing too much detail of the application's behavior. As this deals with application level parameters, the implementation of this technique is very application specific. Therefore in the remainder of this section, we will discus its application of application-level rescaling to Ondes3D. Even so, we believe this technique can easily be adapted to other applications.

The resolution of the domain that is simulated by Ondes3D is governed by two parameters: `ds`, which provides the spatial resolution, the distance between each element in the discretized domain; and `dt`, indicating the temporal resolution, the length of the time-step. From the geophysics perspective, both values can be slightly modified as long as their ratio remains in an acceptable range (according to the CFL numerical condition for stability criterion (MOCZO; ROBERTSSON; EISNER, 2007)). Increasing both values together by a certain factor, respectively allow the reduction of the number of cells in the cuboids and of the number of iterations and therefore decrease the total execution time at the cost of decreasing the precision of the results in term of physics. These values are set by geophysics investigators and are tailored for each workload.

For our performance evaluation, however, we are purely interested in the computation time profile of each process in the domain decomposition. Therefore, running at a coarser-grain may be sufficient to capture the evolution of the computational load. Figure 4.4 illustrates this scenario: each facet shows the duration of the computation (on the Y axis) of the `Stress` macro-kernel along the iterations (on the X axis). The left facet shows the computational behavior with the default `dt` and `ds` values, then the next four facets show the computational cost when we increase `dt` and `ds` together by the given percentage, up to 20%. We can see that as we lower the resolution (larger values of `dt` and `ds`), the computational cost becomes smaller and smoother (without the peak at the ≈100 iteration). The cost per iteration reduces, in average, from ≈0.0623 seconds, in the original version, to ≈0.0381 seconds (≈39% faster) in the case where `dt` and `ds` are increased by 20% (right facet). In this figure, we show only the `Stress` component, for only two ranks: rank 0 (red color, top-left corner in the fixed domain decomposition) and 63 (blue color, bottom-right corner in the domain) out of the 64-process run. The behavior of other ranks and of the other two macro-kernels are similar. This procedure only works up to a certain factor level, from which the computational behavior gets too smooth to be able to

reconstruct the original shape. For the Chuetsu-Oki scenario, $\delta = 1.2$ (i.e., an increase of 20%) was the largest scaling factor allowing to faithfully observe its load evolution.

Figure 4.4: The duration (on the Y axis of each facet) for the `Stress` macro-kernel along the Ondes3D iteration numbers (on the X axis of each facet) for two processes (0, red, and 63, blue) of a 64-node run. The horizontal faceting indicates the scaling factor used towards a coarse-grain simulation: Original (left column), then 5% to 20% increase in `ds` and `dt` (i.e., decrease in spatial and temporal resolution).



Source: The author

We now explain how the trace of a coarse-grain execution (larger `dt` and `ds` values) can be rescaled to obtain a faithful (in terms of computational behavior) approximation of a fine-grain execution (original `dt` and `ds` values). Let us denote by $D_{i,p}$ (respectively $D'_{i,p}$) the duration of computations at iteration i on rank p when using the original (respectively rescaled by a factor δ) `dt` and `ds`. Our goal is to reconstruct a trace $\widetilde{D}$ from D' such that $\widetilde{D}$ and D have the exact same number of iterations and $\widetilde{D} \approx D$. Since time is rescaled, the number of iterations needed to simulate T seconds is $N = T/\texttt{dt}$ (resp. $N' = T/\delta\texttt{dt}$). Therefore iteration i in D corresponds approximately to iteration $\lfloor i/\delta \rfloor$ in D'. From Figure 4.4, it is natural to define $\widetilde{D}$ as:

$$\forall i, p : \widetilde{D}_{i,p} = \alpha . D'_{\lfloor i/\delta \rfloor, p}, \text{ with a well chosen } \alpha.$$

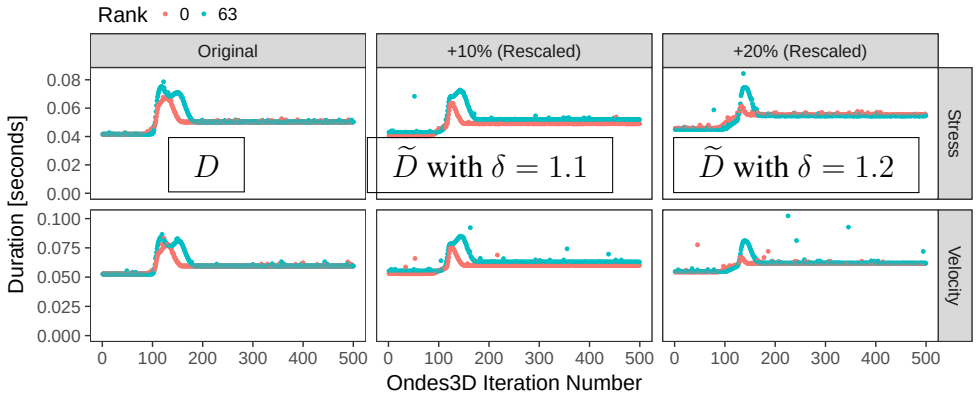
Obviously, we could fit a perfect α to that $\widetilde{D}$ and D are as close as possible from each others but it would require fully knowing D, which is precisely what we try to avoid. We claim that a correct α can be found solely from the first iteration of D and fit α so that $\sum_p \left(D_{1,p} - \widetilde{D}_{1,p} \right)^2$ is minimized, which is obtained with a simple linear regression (with no intercept).

To summarize, approximating the high resolution traces from a coarse grain configuration involves three steps: (1) obtaining the coarse grain trace; (2) running the first iteration of the high resolution setup to obtain an estimation over all processes of how much additional work is required compared to the coarse grain setup; (3) compute the

magnitude of the computational cost rescaling factor α and apply it to the fine-grain trace. Note that such rescaling should be performed kernel by kernel. For Ondes3D, for instance, the consequences of rescaling `dt` and `ds` by δ may be very different for each kernel.

Figure 4.5 shows a comparison of the original behavior of Ondes3D (left column) with two δ rescaling scenarios: when traces with `dt` and `ds` increased by 10% (center column) and 20% (right column) are rescaled back to the original `dt` and `ds`. We show the duration of the `Stress` and `Velocity` macro-kernels as a function of the iteration numbers for two ranks: rank 0 in red, and rank 63 in blue. We can see that the rescaling is imperfect but is sufficient to capture the magnitude of the computational cost of the original run. Although, for simplicity, we only show here the rescaling for two ranks (in the corners of the domain decomposition), similar results are obtained with other ranks, no matter their location in the domain decomposition.

Figure 4.5: The duration (on the Y axis) of two macro-kernels (`Stress` and `Velocity`) in two ranks (colors): 0 in red, 63 in blue, as a function of the Ondes3D iteration number (on X). The data shown in the left column is collected with the original spatial/temporal resolution; while the points shown in the center and right columns are obtained by rescaling measurements taken with lower (by 10% and 20%) spatial/temporal resolution.



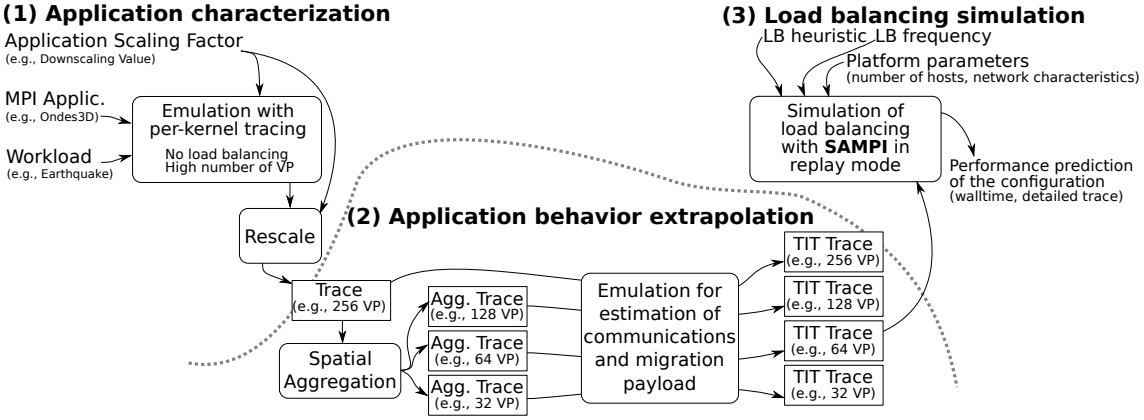
Source: The author

4.4 Low-cost simulation workflow to study dynamic load balancing of iterative applications with regular domain decomposition

The performance modeling strategies detailed in the previous section include *spatial aggregation* (Figure 4.2) and *application rescaling* (Figure 4.5). *Aggregation* enables the downscaling of the number of MPI processes used to execute the application, while *application rescaling* enables a faster execution/emulation of the original code. Together, one can create a computational load *profile* for each rank, kernel and iteration of the

application receiving as input only one execution trace of the application. Our upgraded workflow stands on these performance modeling strategies and on trace manipulation to enable a faster exploration of over-decomposition parameters. Thus, we don't need to run a new full emulation (or execution) for every over-decomposition level. Figure 4.6 presents our three-step simulation workflow, based on the previously described SAMPI simulator. Although we only evaluate our workflow with Ondes3D (more on Chapter 5), we believe that it is generic enough to be applied to any imbalanced MPI code that is iterative and relies on static domain decomposition.

Figure 4.6: Performance evaluation workflow. The irregular distribution of the load of the application over space and time needs to be carefully and finely captured in step (1). It can then be spatially aggregated in step (2) to study different levels of over decomposition. The final step (3) allows to quickly study the influence of load balancing parameters.



Source: The author

Step (1) Application Characterization: The execution of the original MPI application is traced while configured to use a given workload, a given application scaling factor, and the largest number of processes to be studied (e.g., 256 in the Figure 4.6). The scaling factor is used to change the resolution of the temporal/spatial domain resolution of that workload, towards a coarser grain definition. As previously described in Section 4.3, this procedure makes the execution much faster. This execution can be done through sequential emulation or in a real cluster, recording the number of cycles (e.g., with PAPI) instead of real timings as done in (CASANOVA et al., 2015). The trace must also contain events to mark the beginning and end of each main loop iteration, and the computational cost (in micro-seconds or cycles) of the application kernels, interleaved with the communication pattern (source, destination, payload) of the application. After this execution, the trace must be rescaled to the original temporal/spatial resolution. Note that the resulting trace is representative of a given input workload but not linked to a given platform.

Step (2) Application behavior extrapolation: The second step of the workflow aims at extrapolating the previous fine grain trace to any coarser decomposition level (i.e., using fewer processes). As shown in Section 4.2, we can spatially aggregate computational costs without information loss. As consequence, based on a fine-grain decomposition, one can compute the computational load for each rank, kernel and iteration of the application for any coarser decomposition level. In the example of Figure 4.6, the computational behavior of the 256-process trace is spatially aggregated to 128, 64, and 32 processes. As result of this procedure, we obtain traces that contain only the aggregation of the computational cost of each macro-kernel, for each iteration and rank. Extrapolating communication patterns can be more complicated, although techniques such as the ones in (NOETH et al., 2009) could be used. A simpler approach, involving a minor modification of the application, consists in using a emulation/trace-replay approach: the code is emulated again, as in the first step, but with fewer VPs and every computationally intensive part is skipped and replaced by the corresponding spatially aggregated computational profile read from the coarse grain profile. To do so, we have to manually modify the application code to inject the corresponding load in the simulator instead of actually running the CPU-bound code. As a result, we obtain a complete *time independent trace* (TIT) with both computations and communications for a reduced number of VPs. If building on the spatial aggregation property is not possible, then a trace for every envisioned level of decomposition should be obtained directly as in the first step.

Step (3) Load balancing simulation: Finally, the last step consists in running SAMPI simulation as previously described (Section 4.1), but now using TIT files that have been spatially aggregated and rescaled from one single emulation or real run.

4.5 SAMPI workflow implementation

Our workflow relies on SimGrid's SMPI (CLAUSS et al., 2011), which offers two key features on which we have built. (1) First, SMPI allows to study MPI applications either in *emulation mode* or through *trace-replay*. In emulation mode, unmodified MPI applications are sequentially executed, in a controlled way, on top of the simulator. Such approach can be costly in terms of simulation time (all the computations are executed) but faithfully captures the behavior of complex applications while using only one core.

In trace-replay mode, the events of an MPI application are replayed on top of the simulator, which is much faster than a normal execution at full scale. It typically takes one or two minutes on a single core compared to ≈15 minutes on a cluster for our Ondes3D simulations. Thus, we uphold our resource requirement goal. (2) Second, SMPI builds on the hybrid flow-level network models of SimGrid (BEDARIDE et al., 2013) that allow to faithfully model communications and contention, which is essential in the context or our study since load balancing in AMPI induces Virtual Process migration that can be costly in terms of communications.

In our workflow, we use SMPI emulation to capture the behavior of the application into time-independent traces (TIT). This has the advantage of the tracing being done in a controlled execution, which minimizes intrusion, as well as requiring a single node to execute the application. Capturing this trace from real parallel executions would likely be more susceptible to external perturbations, and to suffer from inaccuracies caused by intrusion from the tracing code. Moreover, this would require us to implement the generation of TIT traces, which is already done in SMPI. Alternatively, we could try to employ an existing tracing tool or library. Even so, we would still have to implement the conversion of its output traces to TIT.

Our load balancing simulation employs SMPI in trace-replay mode, using TIT traces obtained through SMPI emulation. Trace-replay allows us to quickly run several simulations from a single TIT trace, to test different load balancing configurations on different platforms. As with SMPI emulation, this also allows us to run all the simulations on a single computer node. In the following subsection, we describe modifications and extensions that were made to SMPI, in order to implement our load balancing simulator.

4.5.1 Modifications to SMPI to enable the SAMPI workflow

Three main modifications in SMPI have been necessary to make SAMPI possible:

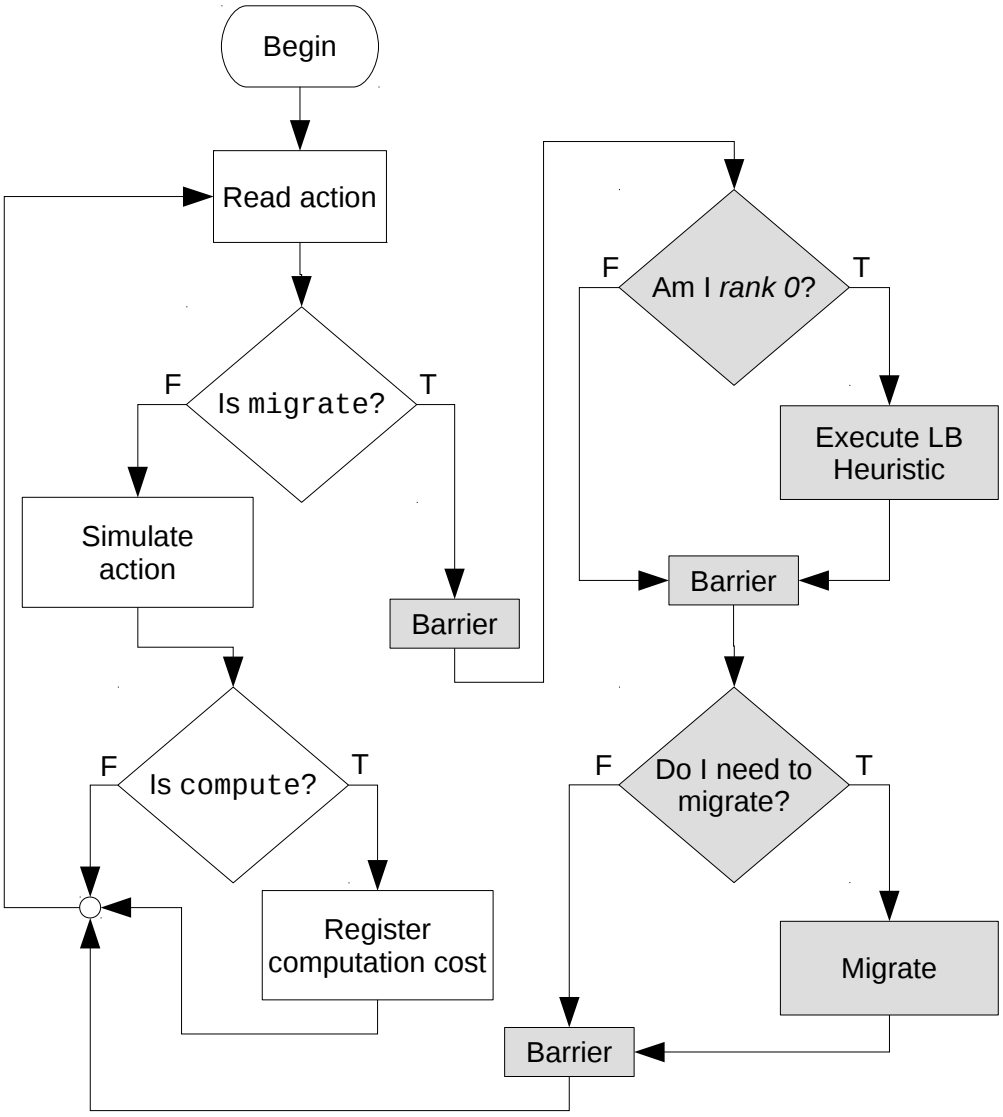
1. First, we had to augment the SMPI interface with the non-standard `MPI_Migrate` function both in the emulation mode (to generate an event in the trace) and in the trace-replay mechanism. In trace replay (the last step of the workflow), whenever the load balancing is activated, this function calls the `MPI_Barrier` function, the load balancing heuristic that defines the new process mapping, and simulates all resulting processes migrations. The main steps of the simulation process are illustrated in more

details in the fluxogram on Figure 4.7. On the right side of this figure, represented by gray blocks, we have the processing of an `action_migrate`. Inside this action there is a barrier that waits until all tasks have called this function. Once that is done, the task with *rank 0* will call the load balancer, passing the data that was collected since the last call. After the load balancer finishes, each process checks if it needs to migrate or not, and acts accordingly. After that, the load balancing information is reset and `action_migrate` is finished.

2. To obtain a faithful behavior of AMPI load balancing heuristics, it is essential to stay as close as possible to their real implementation. We investigated the possibility to directly plug Charm++'s load balancers into our simulator, but they heavily rely on Charm++ specific elements. Fully porting the Charm++ runtime ecosystem on top of SimGrid would require a significant development effort and a deep knowledge of the Charm++ internals. Instead, we have manually extracted and slightly adapted by hand two centralized load balancers: **GreedyLB** and **RefineLB**. Both of them have been presented in Subsection 3.5.2.1. This operation consisted mainly in removing internal references to Charm++, but making sure that the implementation of the heuristic remains intact. A few trace-replay routines also had to be modified to collect the load data that is fed to the load balancing heuristics.
3. Both GreedyLB and RefineLB, receive computation loads as inputs. To capture this load we overrode `action_compute` to capture the computation amounts, after simulating the action. This is shown at the bottom left of Figure 4.7.
4. To account for the migration cost in the simulated execution time, we rely on SimGrid's contention-aware network models when sending the data belonging to the migrated task from its original location to its destination. The migration payload is estimated by trapping `malloc` and `calloc` functions in emulation.

Unrelated to the LB simulation, we also made some modifications to improve the way the SMPI trace-replay mechanism handles concurrent asynchronous messages in the same task. Previously, it only matched the calls to `MPI_Wait` to the last asynchronous call. Now, it properly matches the call which generated the `MPI_Request` received as parameter to `MPI_Wait`.

Figure 4.7: Simplified flowchart of the simulation of an action by our load balancing simulator using SimGrid's MPI time-independent trace-replay. The blocks in grey represent the simulation of the `migrate` action.



Source: The author

4.5.2 Implementation of spatial aggregation and application-level rescaling

The implementation of the spatial aggregation is done by manually modifying the application. In a first fine grain emulation, the application needs to register the computational costs of its computationally intensive kernels for each rank and iteration. These computational costs are spatially aggregated to a coarser grain by an efficient R script. Then, in a second coarse grain emulation, instead of executing the computational intensive kernels, the application should just inject the costs which are registered in the coarse grain traces. This injection is done through a call to a function provided by SMPI.

For the application level rescaling, a very efficient computer program written in R is responsible to rescale the trace back to the original temporal/spatial resolution for the given workload, as if we had executed the program with the original resolution. Different from the aggregation, however, this code is very specific to Ondes3D. Therefore, it may require significant modifications to work with other applications.

4.6 Conclusion

In this chapter, we presented a novel low-cost simulation workflow to evaluate the performance of dynamic load balancing with over-decomposition. Initially, we presented the basic trace-replay based SAMPI load balancing simulation workflow. Then. we also presented two techniques, *spatial aggregation* and *application-level rescaling*, to speed-up the capture of the time-independent-traces used as input to the load balancing simulation. Our complete low-cost simulation workflow combines these two improvements with the original dynamic load balancing simulator. This enables the low-cost evaluation of different load balancing parameters (levels of over-decomposition, load balancer heuristics and frequencies) in different execution platforms. All of this is done from *a single full execution* (emulation) trace, using *a single computer node*, and with *minimal application modification*.

In the next chapter, we first present a validation of the load balancing simulation. Next we demonstrate a few simulation use cases for SAMPI. We also present an estimation of the overall performance of our implementation of the SAMPI workflow, including the performance improvements from spatial aggregation and application-level rescaling.

5 SAMPI EVALUATION AND WORKFLOW PERFORMANCE

In Chapter 4, we presented our low-cost simulation workflow to estimate the performance of dynamic load balancing for iterative parallel applications. This workflow consists of a trace-replay simulator of dynamic load balancing based on over-decomposition, coupled with techniques to speed-up the obtainment of the input trace. The first of these techniques uses *spatial aggregation* to obtain traces with different levels of over-decomposition from a single fine-grained execution of the application. The second technique, *application-level rescaling*, deals with extrapolating the performance of the application, from a (faster) execution, with slightly reduced precision. This is done by tuning application specific parameters in a way that reduces its execution time, and precision, but retains the behavior of the computational cost along time. Thus allowing us to estimate the original costs from a faster execution. We demonstrated this technique on Ondes3D (Section 4.3), by extrapolating original behavior of its computational load from executions with reduced spatial and temporal resolutions (increased `dt` and `ds`).

In this chapter, we initially validate the trace-based load balancing simulation, in Section 5.2. For this validation we simulate two earthquake scenarios with Ondes3D and compare the SAMPI simulation with real-life AMPI executions. Following this, in Section 5.3 we demonstrated the use of SAMPI to estimate the execution times of Ondes3D with different load balancing frequencies and different levels of over-decomposition. These initial results employ only the load balancing simulation component of our workflow (see Section 4.1). That is, we had to execute a full emulation for every level of over-decomposition. Next, in Section 5.4 we demonstrate the use of the simulation workflow for capacity planning. That is, to plan the amount of resources to allocate in order to execute the application, given an execution time threshold. The experiments in this section employ spatial aggregation to speed-up the tracing with different levels of over-decomposition. Last, in Section 5.5, we present an analysis of the performance of the complete simulation workflow (see Section 4.4), demonstrating the simulation performance improvements brought in by *spatial aggregation* (Section 4.2) and *application-level rescaling* (Section 4.3).

5.1 Experimental context

These experiments were executed on two systems. The SMPI emulations (to record the TIT traces) and the AMPI executions used 16 nodes of a cluster of computers equipped with two 12-core 1.7GHz AMD Opteron processors and interconnected though a 20G Infiniband network. The configuration of the nodes is described on Table 5.1. The trace-replay based load balancing simulations were executed on a laptop computer equipped with a 3Ghz Intel Core i7-4760M processor.

Table 5.1: Description of the execution environment

Processor Model	AMD Opteron 6164 HE
Cores per CPU	12
CPU per node	2
CPU frequency	1.7 GHz
Network	20G Infiniband 4x QDR
Charm++ version	6.6.1

Source: The author

Our simulation is based on our modified version of SimGrid, to which we added one function to simulate the migration of MPI tasks. The load balancing simulation itself is implemented in an external module, which links to the SimGrid library. For more implementation details, see Section 4.5. The AMPI traces, were obtained using Charm++'s Tau tracemode. Charm++ was configured to use MPI for communication.

In these experiments, we are simulating two different earthquake scenarios. The first one is the Mw 6.6 Niigata Chuetsu-Oki earthquake (AOCHI et al., 2013), which happened in Japan in 2007. This same scenario was previously used in the evaluation presented in Section 3.5. Running the full simulation (6000 time-steps) would take an unreasonable amount of time, especially because we need to do several executions. So, in order to keep a reasonable duration for the experiments, we limited this simulation to the first 500 time-steps. For the same reason, the number of cells in each direction was reduced to 300×300×150. The second simulated scenario is the Mw 6.3 Lugurian earthquake, which happened in north-western Italy in 1887. This scenario has longer iterations than the previous one. For this reason, we limited its execution to 300 time-steps and the number of cells to 500×350×130.

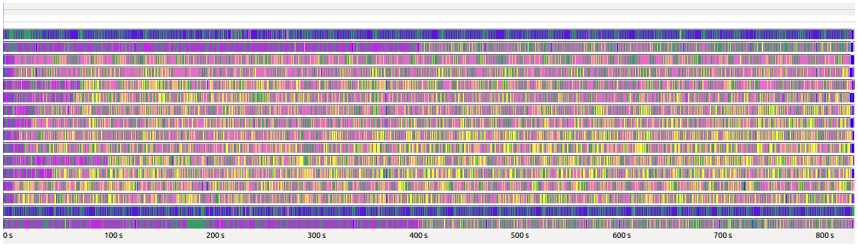
5.2 Validation of the SAMPI load balancing simulation

To test the validity of our simulation, we compare real executions of Ondes3D over AMPI with simulations of the same execution using SimGrid. For this purpose, we executed the application with AMPI on a cluster (see Table 5.1). After that, we emulated the same execution with SMPI on one of the cluster's nodes. The correct analysis of the results presents a challenge in itself. Therefore, before the results, we present the evaluation technique we devised in order to perform their analysis.

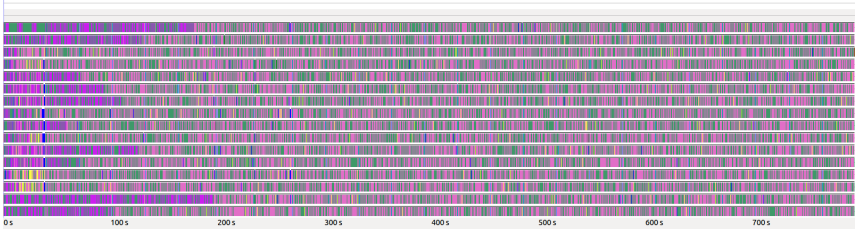
5.2.1 Evaluation technique

Several issues should be solved to correctly validate the accuracy of predictions obtained in simulation. Solely comparing the (predicted) makespan of simulation with the one of real-life executions on a few examples may be considered as insufficient to be fully trusted. Yet, comparing detailed execution traces of an application as complex as Ondes3D is also quite difficult. This can be illustrated through Figure 5.1, where we have Gantt chart visualizations of two executions of Ondes3D, one without load balancing (top), and another with load balancing (bottom). As we can see, at this level of detail, it becomes almost impossible to compare the two executions. Other ad hoc intermediate and aggregated representations are thus necessary.

Figure 5.1: Gantt chart visualization of two executions of Ondes3D



(a) Execution *without* load balancing

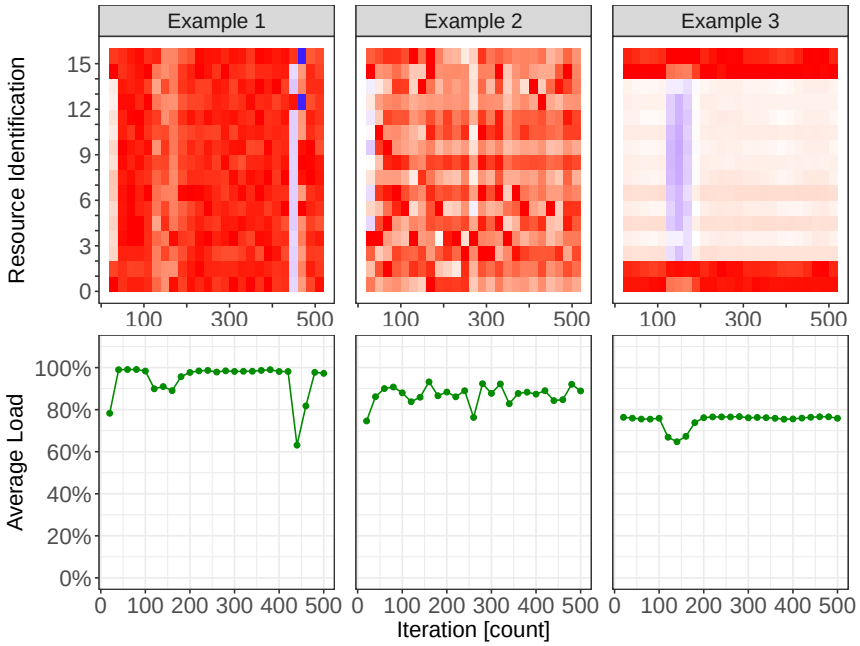


(b) Execution *with* load balancing

Source: The author

Since, in our context, iterations and load imbalance are of primary importance, we decided to track the resource usage per processor and per iteration and to study its evolution both temporally and spatially. Figure 5.2 depicts two possible representations of such information: a detailed view of resource usage with heatmaps (top row) or a spatially aggregated view (bottom row). Heatmaps associate color intensity with the load of each resource on the Y axis (a reddish color representing more load, white for medium, and bluish for lower loads) along the execution time (on the X axis). In the other representation, we spatially aggregate the data by calculating the average load among all resources between calls to the load balancer. This way, we can plot the load evolution of the application using a line plot, with the average load on the Y axis, and the iteration count on the X axis, and points representing the calls to the load balancer.

Figure 5.2: Illustration of the two evaluation techniques



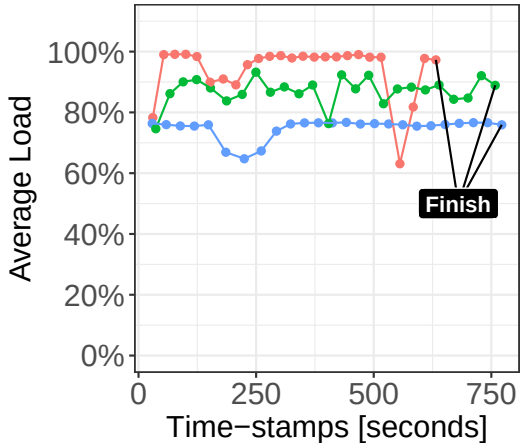
Source: The author

The three examples on Figure 5.2 show the correspondence between the heatmap approach and the aggregated version. *Example 1* has a well balanced execution, with a resource utilization close to one most of the time; *Example 2* has a more significant and varying load imbalance along time; and *Example 3* depicts a very poor load balance, which worsens between iterations 100 and 200.

Although the heatmaps allow to see the spatial details, in comparison to the spatially aggregated view, they are harder to use for comparing multiple executions. This is not a problem for the aggregated view, as demonstrated in Figure 5.3. Besides showing multiple executions, this alternate version of the aggregated view was modified to show timing

information. In this version, the horizontal axis is indexed by time and the points are placed at the time-stamp where each load balancing call has occurred. The labels in the charts point to the finishing time-stamp of each execution. This is useful when there's more than one execution per chart, as also illustrated on Figure 5.3.

Figure 5.3: Visualization of multiple executions with the aggregated view



Source: The author

In our evaluation we focus on analyzing the evolution of the average resource usage along time as the main metric to compare simulations with real executions. Therefore, although we also carefully inspected the heatmaps, we selected to use the aggregated view, as seen on Figure 5.3, for our analysis. Next, we present the validation results obtained with Ondes3D using two load balancers (GreedyLB and RefineLB).

5.2.2 SAMPI validation experiments

In this first set of experiments, aimed at validating our load balancing simulation, we compare real life executions, using AMPI, with our simulated executions, done with SAMPI. From AMPI, we got Tau traces and the time-stamps of each iteration of the application. From SMPI, we got time-independent traces, which were then used as input to our load balancing simulation with SAMPI. In both real and simulated executions, we measured the loads and time-spans, with and without load balancing. For the load balancing part we tested two different heuristics: GreedyLB and RefineLB (see Subsection 3.5.2.1).

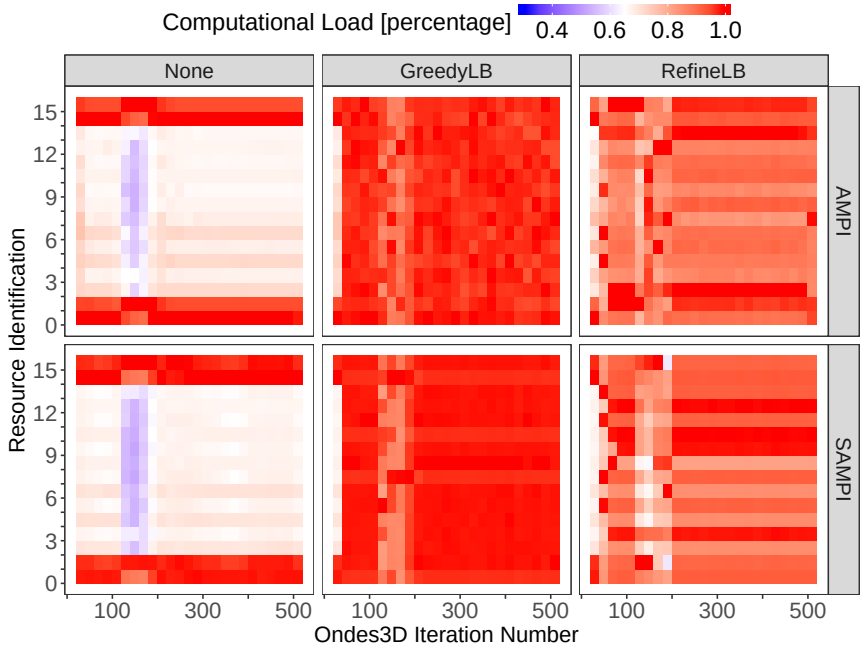
In all these experiments, we decomposed our domain into 64 tasks (VPs) which where mapped to 16 processes. In addition to this, `MPI_Migrate` was called every 20 time-steps of the application. In the results below, we use the techniques presented

on Section 5.2.1 to compare results without load balancer, with GreedyLB, and with RefineLB.

5.2.2.1 Results for the Chuetsu-Oki earthquake scenario

First, we have the results for the Chuetsu-Oki simulation. On Figure 5.4, we have the detailed view for one execution of each configuration. From this, we can see that without LB we have lots of underutilized resources, and that both LB seem to significantly improve this situation. From this single execution, it seems that GreedyLB would be the best choice for this configuration. It is not as easy to compare simulation with real life using this approach. It may seem that in simulation the loads were slightly better balanced, but the difference is not so easily visible. That's where the space aggregated view comes to our aid.

Figure 5.4: Detailed load evolution of the Chuetsu-Oki earthquake simulation executed with AMPI and SAMPI.



Source: The author

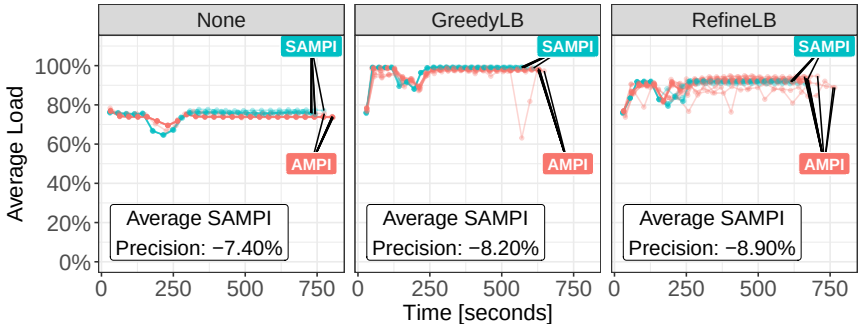
On Figure 5.5 we have the space aggregated view of the evolution of the average load distribution throughout our real-life and simulated executions. This view is much more useful to compare different configurations, while also allowing us to plot different executions of the same configuration on a single chart. On the vertical axis we have the average load, on the horizontal axis we have time. Each point represents the average load at the end of a load balancing interval. The colored labels (AMPI and SAMPI) point to the end of the last iteration of each execution. We can clearly see that, in this scenario,

GreedyLB performed better than RefineLB, both in simulation as in real life. We expected GreedyLB to perform worse than RefineLB, due to the amount of migrations it performs. It seems, however, that in this case the default overload tolerance (1.05) used by RefineLB was too high. There is still some inaccuracy in terms of total makespan, which is noted in the charts as *Average AMPI Precision*, which is calculates as:

$$\frac{100 \times ((avg.\ SAMPI\ exec.\ time) - (avg.\ AMPI\ exec.\ time))}{(avg.\ AMPI\ exec.\ time)}$$

In this scenario, SAMPI was pessimistic in relation to AMPI. Even so, this did not prevent SAMPI from successfully predicting that GreedyLB is best choice of load balancer for this scenario.

Figure 5.5: Load comparison of the Chuetsu-oki earthquake simulation executed with AMPI (real, red) and SAMPI (simulated, blue).



Source: The author

5.2.2.2 Results for the Ligurian earthquake scenario

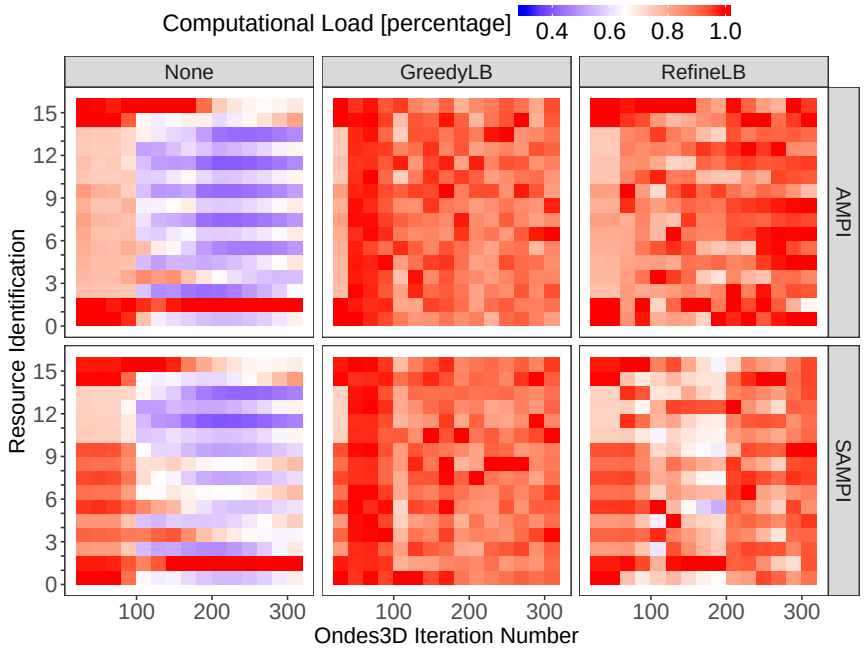
For the Ligurian scenario, due to higher variation in the load distribution, it is even harder to use the detailed view to compare simulation and real life, as seen on Figure 5.6. As on the previous scenario, it seems that both LB improve the load distribution (at least the single execution being displayed). On a first glance, we also may speculate that GreedyLB will be the best choice, but this is not as well defined as in the Chuetsu-Oki scenario. Besides that, not much else can be said only by looking at these graphics.

So, once again, we employ spatial aggregation to more easily compare different configurations. On Figure 5.7 we can see that, as on the previous scenario, both simulation and real life have similar behaviors in terms of load evolution. But, once again, there is some discrepancy in terms of makespan. In this case, the average simulated execution time without LB and with GreedyLB were very close to real life, but RefineLB lags behind real

life by ≈7.5%. Even so, SAMPI successfully led to the correct choice of load balancer, since RefineLB performed best in both simulation and real life.

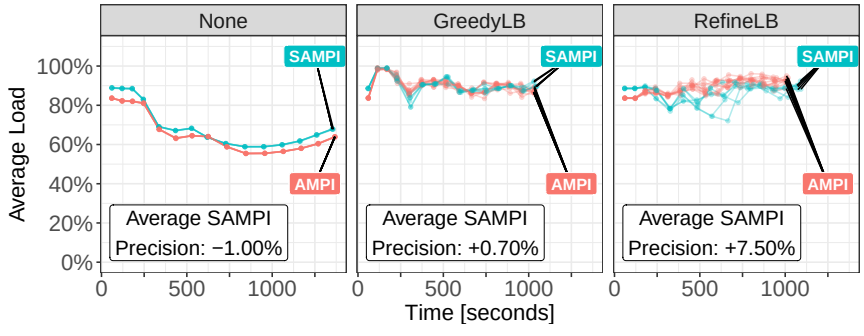
The results above demonstrate that our simulation is able to almost replicate the evolution of the load distribution of real life executions. These results are very encouraging, since this is one of the main aspects we are trying to analyze. There are still minor inaccuracies in terms of total execution time, but they did not affect our choice of load balancer, which remains the same as in real-life in the two investigated scenarios. This ends our validation experiments. In the next section, we demonstrate use of SAMPI to explore two different load balancing parameters..

Figure 5.6: Detailed load evolution of the Ligurian earthquake simulation executed with AMPI and SMPI.



Source: The author

Figure 5.7: Load comparison of the Ligurian earthquake simulation executed with AMPI (real, red) and SMPI (simulated, blue).

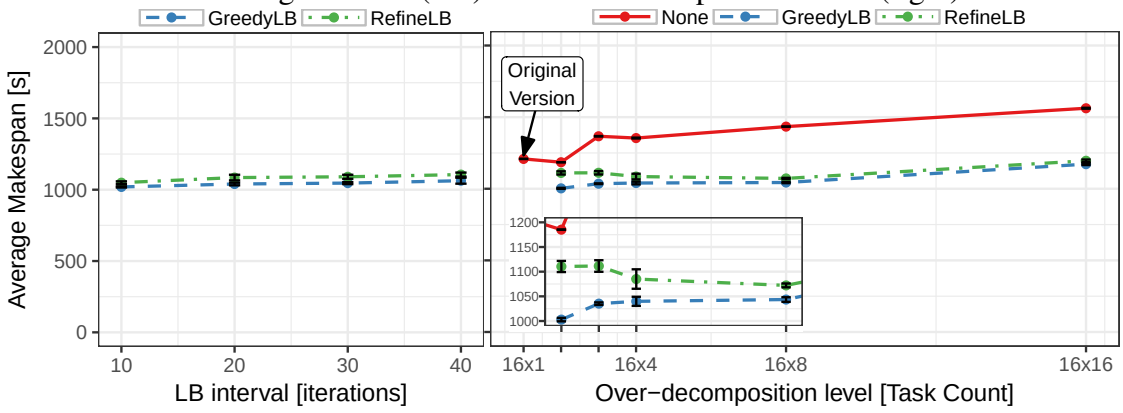


Source: The author

5.3 Exploration of load balancing parameters with SAMPI

In this section we present two use-cases to demonstrate of the use of SAMPI to estimate the performance of an application while varying load balancing parameters. For this purpose, we present below the results of SAMPI simulations of the Ligurian earthquake scenario on Ondes3D. The scenario configuration and execution platforms are the same as on the validation experiments above. First, we present an exploration of the performance of Ondes3D with different load balancing intervals (or frequencies). Next, we explore different levels of over-decomposition. The results of both explorations are on Figure 5.8.

Figure 5.8: SAMPI exploration of the performance (average makespan) of Ondes3D with different load balancing intervals (left) and over-decompositon levels (right).



Source: The author

In the first set of experiments in this section, we use SAMPI to predict the total makespan (execution time) of the application with different load balancing frequencies. During tracing, we call `MPI_Migrate` at the end of every time-step. This way, we can change the LB frequency in simulation by simply removing the unnecessary calls from the traces. Using this scheme, we vary the load balancing frequency at almost no cost (provided the order of magnitude of the simulation time). For these tests, the level of over-decomposition was 16×4 (meaning 64 VP mapped to 16 processes), the same as in the validation experiments.

The results are on the *left* side of Figure 5.8, in which we have the makespan on the vertical axis and the interval between `MPI_Migration` calls on the horizontal axis. According to our predictions, for the tested scenario and over-decomposition level, there's surprisingly little variation in the total makespan when changing load balancing frequencies.

In these experiments, we traced 10 executions of the application, using `smpirun` (sequential emulation). Each execution took approximately 5 hours to finish, which gives us an approximate 50 hours for tracing, using a single computer node. Each simulation takes about 200 seconds to finish for each of the 10 input traces. We simulated 3 load balancing heuristics with 4 different frequencies. This gives us a total of approximately 6 hours and 40 minutes for the load balancing simulations.

Another important parameter to analyze is the over-decomposition level and its impact on the performance of the application. For this, we did SAMPI simulations with five different levels of over-decomposition. For all these executions the number of processes was fixed to 16, and the load balancing interval was set to 20 iterations.

On the *right* side of Figure 5.8, we show the predicted makespans obtained with our simulation approach. On the horizontal axis, the levels of over decomposition are denoted as $np \times vpp$. Where np denotes the number of processes and vpp the number of virtual-processors (MPI ranks) assigned to each process at the start of the execution. In the vertical axis we have the estimated average makespan. The results without over-decomposition (16×1) are only shown for the case without load balancing, as it does not make sense to apply load balancing in this case. In a small rectangle inside the chart, we show a zoomed in view of part of the results, in order to give a better notion of the difference between the makespans with GreedyLB and RefineLB.

From the results, we conclude that the best choice for the tested scenarios should be to use GreedyLB with a 16×2 over-decomposition level. Interestingly, for RefineLB the best over-decomposition level is 16×8. This indicates that the RefineLB takes better advantage of a higher over-decomposition level than GreedyLB. The probable cause is the higher number of migrations done by GreedyLB. In this case, however, it seems it is still better to run GreedyLB at a lower over-decomposition level than running RefineLB. This may be due to a load imbalance threshold used by RefineLB to decide if the application needs load balancing. It may be possible that, since the cost of migrations does not seem to matter in this case (as evidenced by the gains with GreedyLB), RefineLB would perform better with a lower load imbalance threshold.

These simulations predate our implementation of spatial aggregation (Section 4.2), therefore we had to capture one full TIT trace for each over-decomposition level. This was needed because the number of tasks (MPI ranks) on the traces needs to be the same as in the simulation. It takes around 5 hours to capture one of these traces, and we captured about 5 traces for each of the 5 levels of over-decomposition. So, we spent approximately

5 days on a single host, to capture all the input traces. The tracing time can be shortened by using more resources, to trace the different configurations in parallel, on different nodes. It takes us about 200 seconds to run one simulation on a current laptop equipped with a Intel Core i7 processor (the same one used in the validation experiments). We ran 5 simulations with 5 VP counts and 3 load balancing heuristics. This amounts for a total of approximately *4 hours and 10 minutes* for simulation.

This aspect illustrates what used to be the main weakness of our approach, which is the large amount of time needed to obtain the input traces. Which made it especially difficult to simulate different levels of over-decomposition. This problem was solved by the integration of spatial aggregation to our simulation workflow. This integration allows us to pay the full computation cost only once, to obtain a fine grained trace. For the remaining (coarser grain) over-decomposition levels we pay only the initialization and communication cost of the emulation. More details, on the performance of the fully upgraded simulation framework will be presented on Section 5.5. But, before that, we demonstrate of the use of the SAMPI workflow for capacity planning, which contains the first experiments where we employed spatial aggregation to speed-up the simulation process.

5.4 Using SAMPI for capacity planning

The use of SAMPI in *trace-replay* mode brings several advantages. For instance, using a single representative TIT file with the behavior of a certain amount of processes, SAMPI can be used to verify what would be the number of hosts that respect a certain efficiency requirement. Plus, by employing our spatial domain aggregation technique, we can efficiently expand this exploration to different VP counts (levels of over-decomposition). This is especially useful for the simulations without load balancing (No LB), as for these we needed to do a new emulation for every host count.

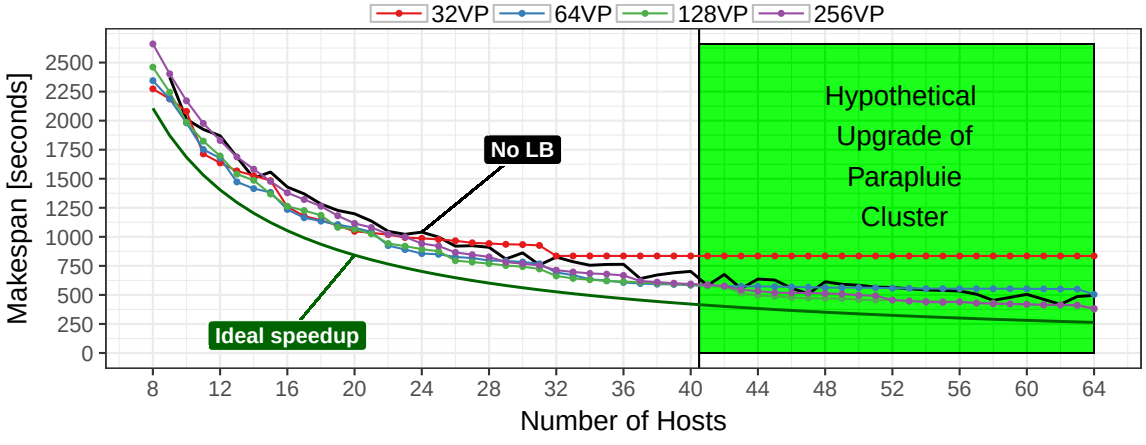
This *capacity planning* is illustrated in Figure 5.9, where we show the makespan given by SAMPI when it replays the computational behavior of the Ligurian scenario with 256, 128, 64 and 32 VPs, employing GreedyLB every 20 iterations. These makespans (Y axis, in seconds) are represented as a function of the number of hosts (X axis) available from a calibrated SimGrid platform description file that represents the Parapluie cluster of Grid'5000 (the same one used in our previous experiments). We tested every possible host count, from 8 to 64 hosts.

The results indicate a surprisingly good scalability of the application up to 64 hosts. The performance gains seem much more significative when moving from 8 to 24 nodes, where the makespan is reduced from ≈2273 seconds with 32 VPs to ≈856 seconds with 64 VPs (≈62% shorter execution time). The addition of 16 extra hosts to get the total capacity of Parapluie (40 hosts) reduces the makespan to ≈26% of the best result for the original case with 8 hosts (≈74% shorter execution time).

We have artificially increased the number of hosts to 64 (24 more hosts than the real cluster) to check whether this upgrade to Parapluie would be interesting from the Ondes3D Ligurian workload point of view. Figure 5.9 shows that makespan reduction from 40 nodes (the real capacity of Parapluie) to 64 nodes (the new hypothetical capacity) would reduce the execution time about 207 seconds (≈83% shorter than with 8 nodes). Although the gain is of much smaller scale than when going from 8 nodes to 40 nodes, these few minutes may be precious and worth the investment when conducting this kind of earthquake hazard assessment.

To exemplify the usefulness of this kind of exploration, with this simulation we can predict that to run the earthquake simulation in no more than 10 minutes we need to allocate precisely 40 nodes. Plus, our simulations show that the use of load balancing with AMPI on this machine allows to obtain a nearly perfect speed-up (green curve).

Figure 5.9: Capacity planning using SAMPI with the Ligurian case, employing GreedyLB with multiple VP counts; `TIT` traces for all but the 256 VP simulation were obtained by spatial aggregation; all trace-replay simulations for a given VP count use the same TIT trace; the chart shows the resulting makespan as a function of the number of hosts (8 to 64). The ideal makespan (with a perfect speedup) is derived from the execution time on a single host with only one VP (not shown). The black line shows the estimated execution times without load balancing (No LB).



Source: The author

5.5 Simulation workflow performance

As previously stated, the workflow must be fast to rapidly find the best load balancing parameters. Table 5.2 presents estimations of the performance of the workflow when it is run in a modern desktop, using exclusively emulation for the first step and two workloads: Ligurian and Chuetsu-Oki. We present the performance for two workflow configurations: the `Upgraded` version contains the changes brought by acceleration strategies presented in Sections 4.2 and 4.3 and has all the three steps; the `Raw` version has only the first and last steps, since the spatial aggregation is optional. For the estimations presented in this table, we assume that the goal is to define the best load balancing parameters considering one platform, four over-decomposition levels (256, 128, 64, and 32 processes), two heuristics (GreedyLB and RefineLB), and three load balancing frequencies (every 10, 20, and 30 iterations), for a total of 24 configurations. The estimation for the third step (SAMPI) are for a single simulation through trace-replay.

Table 5.2: Estimated per-step workflow performance, in seconds.

Workload	**Workflow**	**Step #1**	**Step #2**	**Step #3**	**Total**
Ligurian	`Raw`	4 × ≈21000s	`NA`	24 × ≈200s	≈88800s
	`Upgraded`	1 × ≈10000s	≈3650s	24 × ≈200s	≈18450s
Chuetsu-Oki	`Raw`	4 × ≈9500s	`NA`	24 × ≈200s	≈42800s
	`Upgraded`	1 × ≈4000s	≈1400s	24 × ≈200s	≈10200s

Source: The author

The results of Table 5.2 indicate that the `Upgraded` version of the workflow is faster than the `Raw`. The first step in `Raw` is costlier because it involves the trace collection in all four decomposition levels in SMPI emulation mode. This could be made much faster if traces are gathered from a parallel execution in a real cluster, but with a larger cost in terms of resource reservation. Moreover, the `Upgraded` version uses application-level scaling with a factor of 20% (see Section 4.3), requiring the collection of only one single very fine-grain trace (with the maximum number of ranks we want to study, in this example, 256 processes) plus the minor rescaling overhead. We estimate that with a 20% application scaling factor, we are capable to reduce the emulation time by ≈50% of the original emulation without scaling. Despite the necessity of executing Step #2, its cost is minor because of the hybrid emulation/trace-replay approach we adopted to downscale the communication pattern after spatially aggregating the computational load (see Section 4.2).

At this step, the Ligurian case is much slower than Chuetsu-Oki (not shown) because Ligurian has a geological model in its input files and also has higher memory usage. So the time for its hybrid emulation comprises also the very expensive I/O operations during the model instantiation. The computational load injection comprises a minor fraction of the cost in this step, for both cases. Finally, the SAMPI simulations of Step #3 consist in replaying the TIT traces (from Step #1 in `Raw`, or from Step #2 in `Upgraded`) for all 24 parameters combinations. The cost of these simulations are stable for both approaches with minor changes between the two workloads. For this evaluation, they are considered to be executed sequentially in a single computer.

To summarize, Table 5.2 shows that our Ondes3D modeling strategies are capable to reduce the workflow performance from ≈25 to ≈5 hours, an improvement of 80% using the `Upgraded` version considering the Ligurian Workload. For Chuetsu-Oki, the gains are of 75%, from ≈12 to ≈3 hours. Additionally, since the simulations on Step #3 are all independent, the cost of this step could be further reduced by running all of them in parallel.

5.6 Conclusion

In this chapter we presented experiments using our low-cost simulation workflow for estimating the performance of dynamic load balancing with over-decomposition, dubbed Simulated Adaptive MPI (SAMPI).

In a first set of experiments, to validate our simulator, we compared simulation with real-life while running without load balancing and with two different load balancing heuristics. The results show that our simulation behaves very similar to real-life in terms of load distribution. There are still, however, small imprecisions in the estimation of the total execution time. Even so, this imprecision was not enough to prevent SAMPI from successfully predicting the best load balancer, which was the same as in the real-life AMPI executions.

In a second set of experiments, we demonstrate the use of our simulation for load balancing parameter exploration. More specifically, we simulated the application with four different intervals between load balancing calls and with five different levels of over-decomposition. The former proved very practical, due to the potential for reuse of a single execution trace to test all the scenarios. This is not the case with the latter, since they were done before we developed our spatial aggregation technique (Section 4.2). Therefore

we needed a new full execution (emulation) to generate the TIT trace for each level of over-decomposition. Thankfully, we ere now able to mitigate this problem with *spatial aggregation*, since collecting these traces with full emulations would seriously hinder the ability to save time trough trace-replay.

In a third set of experiments, we demonstrate the use of SAMPI for capacity planning. That is the problem of choosing the ideal number of resources to execute the application given a certain objective (e.g., meet a execution time threshold or plan a platform upgrade). For this purpose, we simulated the execution of the Ligurian scenario in Ondes3D employing GreedyLB, while varying the number of hosts from 8 to 64. We simulated four VP counts, being 256, 128, 64, and 32 VPs. The emulations with different VP counts were sped-up by employing spatial aggregation of computational loads (Section 4.2). The simulation revealed surprisingly good scaling of the application, especially when going from 8 to 24 hosts ($\approx$61% performance gain). These simulations also indicate a potential, albeit small, performance gain could be obtained through the expansion of the platform from 40 to 64 nodes.

Last, we presented a rough estimation of the overall performance of our simulation workflow. This includes a comparison of our original *Raw* version of the workflow (without spatial aggregation), with the *Uprgraded* version, sped-up by spatial aggregation. With the upgraded workflow we where able to reduce the time to simulate the Ligurian scenario of Ondes3D with four over-decomposition levels, three load balancing frequencies and two load balancing heuristics, from $\approx$25 to $\approx$5 hours, an improvement of 80%.

With our evaluation done, in the next chapter we will analyze related works in the area of dynamic load balancing and distributed system simulation. We also include a justification of our choice to use SimGrid as the basis for the implementation of our simulation workflow.

6 RELATED WORK

To our knowledge, at the time of this writing, there is no other published work that implements simulation of dynamic load balancing based on over-decomposition. Even tough BigSim (ZHENG; KAKULAPATI; KALE, 2004) allows the simulation of previous executions of Charm++/AMPI programs on different platforms, it does not allow to change the load balancing parameters in simulation. Nevertheless, in the two sections below, we present works related to dynamic load balancing and distributed system simulators. We also included a small discussion about our choice of using SimGrid to implement our simulation workflow.

6.1 Dynamic load balancing

As mentioned before, load imbalance can be seen as an intrinsic property in many applications. As mentioned in Section 2.1, load balancing is an NP-Hard problem (LEUNG, 2004), which makes it impractical to find an optimal solution for it. Therefore, different approaches have been proposed to mitigate its effects. Below, we present previous attempts to balance the load of seismic wave simulations (such as Ondes3D) and other applications.

Several approaches have been proposed to tackle the particular load balancing issues of the elastodynamics equation. One classical way to ensure the load balancing of grid-based computations is to use mesh partitioning techniques (KARYPIS; KUMAR, 1995) for initial distribution or dynamic re-distribution during the computation. One could also exploit a quasi-static load balancing algorithm based on zone costs (SEGUIN; CRACRAFT; DREWNIAK, 2004). The main limitation comes from difficulties to accurately evaluate a priori the execution time of various parts of a program, because of hardware related effects (cache, branch prediction, and other optimizations), arithmetic considerations, or compiler behavior that can lead to unpredictable situations for such a fine-grained computation.

Other techniques that have been evaluated are based on domain overloading mechanisms that rely on the efficiency of thread scheduling to balance the computation (DUPROS et al., 2008). At the shared-memory level, OpenMP directives were added at the outer loop of a triple nested loop, to honor the unit-stride and reduce the number of costly OpenMP barriers. This induces a one-dimensional decomposition that is usually not the most efficient strategy for a three-dimensional problem.

These approaches could not efficiently tackle the different levels of imbalance that need to be considered for elastodynamics equations on multicore architectures. Besides imbalance coming from the algorithm, and from the machine hierarchy (concurrent access to various levels of cache memory, Non Uniform Memory Access) can represent a significant part of the imbalance (DUPROS et al., 2009). The programming model is also a critical issue. In the context of a large code devoted to industrial and research applications and mainly developed by geophysicists, it is important to rely on a standard programming model. Next, we discuss related works aimed at improving the load balance of applications other than seismic wave simulation.

Rodrigues et al. (RODRIGUES et al., 2010) discussed different strategies to reduce load imbalance and remote communications in weather forecast models (more specifically, BRAMS). These models can suffer from load imbalance due to irregular workloads assigned to processes which depend on input data, and due to dynamic phenomena moving through the simulated area (e.g., thunderstorms). The authors's approach tries to preserve the spatial proximity between neighbor processes by traversing them with a Hilbert curve, and recursively bisecting the curve according to the load of each part. This is repeated until the number of segments becomes equal to the number of cores in the platform. This approach, together with a graph partitioning load balancer based on METIS (KARYPIS; KUMAR, 1995), was one of the few algorithms able to improve the performance of BRAMS.

Graph mapping and partitioning tools, such as METIS, are commonly used to mitigate load imbalance. For instance, Catalyurek at al. (CATALYUREK et al., 2007) use hypergraph partitioning on dynamic applications using the Zoltan toolkit (DEVINE et al., 2002). Their approach tries to improve load balance while avoiding the migration of tasks. The application is represented as a hypergraph, where vertices are tasks, and nets (hyperedges) represent communication and migrations costs. Their approach involves a graph coarsening phase, a recursive bisection phase, and a refinement phase. Migrations are avoided by representing cores as vertices that cannot be merged during the coarsening phase, and by adding nets between a core and the tasks currently mapped to it.

Another kind of application that can suffer from load imbalance are molecular dynamics simulations, such as NAMD (NELSON et al., 1996). NAMD is decomposed in tasks using 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, affect the

workload distribution of the application. Another source of load imbalance is the dynamic nature of the simulated phenomenon, where atoms can move from one cell to another. In such a scenario, Bhatele et al. (BHATELE; KALE; KUMAR, 2009) discussed the use of static, dynamic, and topology-aware strategies to improve NAMD's performance. Unfortunately, some of the strategies are strongly related to the characteristics of NAMD and cannot be applied to Ondes3D.

One last method to mitigate load imbalance on dynamic applications involves the use of work stealing algorithms (BLUMOFE; LEISERSON, 1999). For instance, Frasca, Madduri and Raghavan (FRASCA; MADDURI; RAGHAVAN, 2012) investigated methods to improve the performance and power consumption of the graph mining algorithm Betweenness Centrality (BC) on multicore machines. Their approach differs from the traditional work stealing, as a victim is chosen according to its NUMA distance, and not randomly. The NUMA distance is an integer value that represents the distance between two cores in the machine. By guiding the work distribution and improving locality, the authors were able to improve the performance and energy consumption of BC by approximately 50%. However, work stealing would be an impractical approach to attack Ondes3D's load imbalance problem, as it is more appropriate for applications where the workload can be dynamically partitioned or is already split into many different tasks.

6.2 Distributed system simulators

In this section, we will present a few distributed system simulators found in the literature. We divided the discussion in four subsections. In the first subsection, we present simulators that only support off-line simulation. In the second, we present simulators that only support on-line simulation. In the third, we have the simulators that, like SimGrid, support both off-line and on-line simulation. Finally, in the last subsection we present a few simulators which are interesting for executing simulator itself in parallel.

6.2.1 Simulators that only support off-line simulation

There are a few simulators that support *offline simulation*, through the replay of traces from previous executions, but *do not support on-line simulation*. Among them are *LogGOPSim* and *PSINS*, which we presented in more detail below:

LogGOPSim (HOEFLER; SCHNEIDER; LUMSDAINE, 2010) is a simulation framework for large scale parallel applications. It is based on the LogGOPS model, which is an extension of the LogGPS (INO; FUJIMOTO; HAGIHARA, 2001) model to capture send and receive operation overhead per bytes. As our simulator, it employs trace based simulation to capture the characteristics of the application. Their goal is to use trace extrapolation to simulate the execution of parallel applications at large scale. Besides the different goals from our work, their traces are not time-independent as the ones employed by SimGrid's trace-replay mechanism. Another difference is that, due to limitations of the LogGOPS model, the simulation does not account for contention in the network.

PSINS *PMaC's Open Source Interconnect and Network Simulator* (TIKIR et al., 2009), is a trace driven performance modeling tool for MPI applications. It is composed of a tracer and a simulator. The first collects traces during the execution of the application. The second replays the execution through simulation. This simulation uses the traces and parameters of the target system to simulate the performance of the application on the target. The tracer library provides wrappers that intercept MPI calls, in order to collect information about the execution. This data is stored locally, using a very compressed binary encoding. It is possible for the user to add new trace decoders, in order to incorporate information provided by other tools. PSINS includes three communication models and provides the capacity for the developer to add new models.

6.2.2 Simulators that only support on-line simulation

Other simulators only support *online simulation*. One of the most well known is *GridSim*, which was developed with the purposed of simulating Grid systems and was employed in many works and even as the basis for other simulators.

GridSim (BUYYA; MURSHED, 2002) is a discrete event simulation toolkit widely used in the simulation of Grid systems. It supports the modeling and simulation of a wide range of heterogeneous resources, users, applications, resources brokers and schedulers. It can simulate application scheduling on various classes of distributed systems, like cluster, Grid and P2P systems. Despite being a very popular simulator,

its communication models have been demonstrated to be flawed (VELHO et al., 2013). This demonstrates the need for good validation of the simulation models.

6.2.3 Simulators that support both on-line and off-line simulation

Besides SimGrid, there are a few other simulators that support both *on-line and off-line simulation*. Among them are *BigSim*, *xSim*, and *SST/Macro*, which are presented below:

BigSim (ZHENG; KAKULAPATI; KALE, 2004) is a simulator for predicting the performance of large-scale parallel machines. It is based on the Charm++ (KALÉ; KRISHNAN, 1993) parallel programming system. Similar to our work, its prediction process has two phases: emulation and simulation. In the emulation phase the application is executed in a virtualized environment, in order to collect execution traces. In the simulation phase, the data contained in these traces is adjusted according to parameters of the target machine. For more accurate results, one can use a technique that uses machine learning to generate statistical models of the execution of sequential blocks of the application (ZHENG et al., 2010). To build these models, a smaller scale version of the application needs to be executed in a smaller version of the target machine or in a cycle accurate simulator. These models are then used to adjust the duration of the sequential blocks in the traces collected by the emulator. Different from our simulator, BigSim does not support changing the load balancer parameters in the trace-replay phase. In other words, you can only run simulations with the same load balancing configuration that was used in emulation. Therefore, BigSim requires a new emulation every time the load balancing parameters are changed.

xSim the *Extreme-Scale Simulator* (BöHM; ENGELMANN, 2011), is a parallel performance investigation toolkit. It allows to run an application at extreme scale in a controlled environment, without the need of a respective extreme-scale system. xSim aims to identify application scaling issues and to assist on HPC hardware/software co-design. This simulator uses lightweight parallel discrete-event simulation. The applications executed with a virtual wall clock, which is scaled from actual time, based on computation and network models. xSim executes MPI applications on a highly over-subscribed (more than one process per processor core) fashion, in a

much smaller HPC system. It relies on the MPI performance tool interface (PMPI) to interpose itself between over-subscribed application and MPI. This wraps the application in a virtual execution environment controlled by xSim. In other words, xSim provides an MPI layer for the application while the native (not over-subscribed) MPI layer is used by xSim itself. Besides running full MPI applications, xSim also supports simulations using MPI application communication traces. Different from SimGrid's SMPI, instead of replaying the trace in the simulator, it runs a benchmark application generated from these traces. To record the trace and generate the benchmark, the framework uses ScalaTrace (NOETH et al., 2009). Its parallel execution over MPI is one of the greatest advantages of xSim, since it enables it to scale to process counts which would not have been possible in a similar sequential emulator. On the other hand, this increases the complexity of the simulator, which can lead to big execution overheads. There have already been some works on improving these overheads (ENGELMANN; NAUGHTON, 2014). Other improvements include models for different network topologies (JONES; ENGELMANN, 2011) and a new network model which accounts for contention and bandwidth sharing (ENGELMANN; NAUGHTON, 2015). xSim was demonstrated to scale to 2^{27} MPI ranks on a 960 core cluster (ENGELMANN, 2014) with an empty MPI application (only `MPI_Init()` and `MPI_Finalize()`). It also was used to simulate 2^{24} MPI ranks of a basic, embarrassingly parallel, Monte-Carlo π solver application on the same cluster.

SST/Macro (ADALSTEINSSON et al., 2010) is a macroscale simulator that aims to allow the coarse-grained study of distributed memory applications. It is developed as part of the same project as the *Structural Simulation Toolkit* (SST) (RODRIGUES et al., 2011). Unlike SST, which aims for high fidelity, SST/Macro is concerned with lower-fidelity but lower-cost simulation techniques to enable simulation at larger scale. The stated goals of SST/Macro are to assist system design and application development. The simulator is modular, allowing to switch between different computation and communication models. It focuses in a extremely lightweight, single thread implementation, to avoid synchronization overheads present in parallel simulators. Parallel tasks are modeled as lightweight threads, allowing the simulator to maintain their states and manage their scheduling. It also includes a detailed MPI model that converts high-level MPI events into the corresponding lower-level communication operations. SST/Macro supports two network modeling approaches.

One provides a congestion free network, while the other models congestion on a link-by-link basis through a flow-based network model. SST/Macro does not support direct execution of MPI applications. Instead the simulator can be driven by either a trace file or a skeleton application. It supports its own detailed MPI trace format, called DUMPI, which can be recorded by linking the application with a library which relies on PMPI. A skeleton application is a simplified version of the simulated application, which uses specific simulator calls to provide enough information for it to model both its computation and communication. Despite the amount of work needed to write the skeleton application, it may be useful to simulate applications where the control flow depends on the order of the received messages (which is impossible to represent with traces). One disadvantage of the skeleton application approach is that it depends on the ability of modeling the application's computation costs prior to it's execution. This makes it difficult to use it to simulate applications with a dynamic load distribution.

6.2.4 Parallel simulators for MPI applications

In this section, we present a few simulators for MPI programs which are interesting for employing a parallel simulation approach. That is, the simulator itself is a parallel application. This is often achieved through direct execution of the application code, with the communication (and maybe I/O) calls being intercepted by the simulator. This way, the duration of these operations can be adjusted to the target execution platform by employing simulation models. One of these simulators is xSim (BöHM; ENGELMANN, 2011), which was already presented on Subsection 6.2.3. Below, present a few other parallel simulators for MPI applications:

Riesen (2006) developed a parallel MPI simulator in which the application is executed, normally, using MPI. The difference is that it uses the PMPI interface to intercept communications in order to employ a network simulator to adjust the communication delays according to the parameters of a simulated network. Therefore, this simulator simulates only the communications not the computations. From the point of view of the application, the MPI execution seems to be the same as without the simulation (except for the different timings).

MPI-NetSIM (PENOFF et al., 2009) uses a similar approach to the one employed by *Riesen*, but using a packet level simulator for the network. Besides, the simulator interface is built into the MPI middleware instead of using PMPI. One important aspect of this simulator is that the packet-level simulator can be much slower than the actual communications. For this reason, MPI-NetSIM slows down the execution of the application in order to preserve its communication behavior.

MPI-SIM (PRAKASH; BAGRODIA, 1998; BAGRODIA; DEELMAN; PHAN, 2001) is another parallel simulator which relies on the direct execution of the application. Different of the previous two simulators, it transforms MPI ranks in to threads, in order to simulate more tasks than the number of available processors. Although, it is not clear why this modification is needed. This simulator package includes a preprocessor responsible for modifying the program in order to support its execution with threads. Similar to other parallel MPI simulators, communication and I/O primitives are trapped by the simulator. MPI-SIM will then use models to predict the execution time of the activity in the simulated (target) architecture. One important aspect that can lead to imprecise results is that this simulation does not account for contention in the network.

6.2.5 Why did we choose SimGrid?

The simulation workflow we proposed in this thesis mostly depends on two factors. First, a faithful model of modern HPC networks and MPI implementations is essential since communications play a crucial role in the load balancing trade-offs. Second, the ability to run simulations both in trace-replay and emulation modes is more than helpful to select the approach most suited to the resources at hand. There is a plethora of simulation frameworks and tools to study MPI applications (CASANOVA et al., 2014) and at least four of them support both emulation and trace-replay and could thus have been modified: BigSim (ZHENG; KAKULAPATI; KALE, 2004), SST/Macro (ADALSTEINSSON et al., 2010), xSim (ENGELMANN, 2014), and SimGrid (CASANOVA et al., 2014) (through SMPI (CLAUSS et al., 2011)).

BigSim is part of Charm++, thus supporting the simulation of AMPI applications, such as our Ondes3D implementation. Although it is linked to Charm++, it does not allow to change the load balancing parameters during trace-replay, since its main concern is to analyze the influence of different network topologies. Adding this feature would require

major modifications to the BigSim simulator. Besides, in a visit to the laboratory that develops Charm++, at the end of 2013, it became clear that BigSim was not in active development at the time. Besides, its main developer had left the laboratory. These aspects would probably make it harder to work with this code.

SST-Macro allows both trace-replay through the DUMPI module and on-line simulation (emulation) through skeletonization. Although SST-macro is quite flexible, and has many network models, including flow-based ones, its emulation support requires to modify Ondes3D, which we wanted to avoid. Finally, *xSim* mostly focuses on extreme-scale executions. Its sources are not currently available and it is unclear whether it would be able to emulate unmodified full-fledged MPI applications.

For this work, we therefore chose to rely on the free software *SimGrid*, whose SMPI (CLAUSS et al., 2011) interface allows both emulation and trace-replay of MPI applications. SMPI leverages SimGrid's thoroughly validated flow communication models (VELHO et al., 2013), while also accounting for specific characteristics of MPI implementations (CASANOVA et al., 2014). Hence, SMPI allows us to collect accurate execution traces from emulation, and its replay mechanism allows us to quickly simulate this execution as many times as needed. Finally, traces are in pure text enabling an easy transformation through spatial aggregation. Besides these scientific an technical reasons, this thesis benefited from collaboration with developers of SimGrid. This included a Sandwich PhD Internship at INRIA Grenoble, as a member of one of the research teams responsible for SimGrid's development. This internship was advised by professor Arnaud Legrand, who is one of the main developers of SimGrid. This collaboration with developers who are familiar with SimGrid's code base significantly facilitated the implementation of our simulation workflow.

6.3 Conclusion

This ends our presentation of related works. At first, we presented works related to dynamic load balancing, including some early attempts to balance the load of Ondes3D. Next, we presented a few distributed system simulators, and justified our choice of using SimGrid to implement our simulation workflow. In the next chapter, we conclude this thesis, summarizing its proposal and results and discussing a few future developments.

7 CONCLUSION

In this thesis we proposed *Simulated Adaptive MPI* (SAMPI), a low-cost simulation workflow to evaluate the performance of dynamic load balancing based on over-decomposition applied to iterative MPI applications. The *goal* of this workflow is to *reduce the cost of this evaluation in terms of* development and evaluation *time*, and in terms of computational *resource requirements*. As such, SAMPI aims to enable this evaluation (1) with minimal application modification, (2) allowing the fast evaluation of different load balancing parameters, and (3) with small resource requirements. At the time of this writing, we are not aware of the existence of any other simulator that implements this kind of load balancing simulation.

The SAMPI workflow combines two models of simulation: *direct emulation* and **trace-replay**. Emulation is used for application execution characterization through detailed computation and MPI communication traces. Trace-replay allows to quickly replay the emulated execution as many times as required to test different load balancing parameter combinations. The full simulation workflow has three steps:

1. *Application characterization*: This comprises a full execution of the application using sequential emulation. The result will be a *time-independent trace* (TIT) which will serve as an input for the simulator. Optionally, this step can be sped-up by employing *application-level rescaling* as discussed on Section 4.3.
2. *Application behavior extrapolation:* In this step we may apply *spatial domain aggregation* (Section 4.2) techniques to estimate the computation costs of a coarser-grain execution from a finer-grain execution. These aggregated computation costs are injected into a new emulation to estimate the communication costs and migration payloads. This generates a new time-independent trace (TIT) for the coarser domain decomposition. Without this step, we would need to do a new full emulation for every different domain decomposition.
3. *Load balancing simulation:* The last step is the actual load balancing simulation. In this step we use trace-replay to simulate the execution of the application. Using the same TIT trace, we can evaluate different load balancing heuristics, load balancing frequencies, and execution platforms.

Both sequential emulation and trace-replay require a single computer node. Thus, *this workflow fulfills our resource requirement goals*. Besides, *trace-replay is very fast*,

taking only *a small fraction of the actual execution time* of the application. Additionally, *spatial aggregation* allows to obtain *traces at different levels of over-decomposition* from a *single full execution* (emulation) trace. Thus mitigating the extra cost incurred by using sequential emulation compared to real parallel executions. Moreover, this initial *emulation can be sped-up by applying application-level rescaling*. Therefore, *we also fulfill our goal of reducing the cost of the evaluation in terms of execution time*.

To implemented the SAMPI workflow, we used SimGrid (CASANOVA et al., 2014), which is a versatile and well validated distributed system simulator. It includes a flow-level network models that models contention and bandwidth sharing. It also implements a piece-wise communication model that accounts for specific characteristics of MPI implementations. More specifically in our work we used SimGrid's SMPI interface (CLAUSS et al., 2011), which provides both direct emulation and trace-replay (CASANOVA et al., 2015) of MPI applications. Besides SimGrid, we developed GNU R scripts to deal with application-level rescaling and the spatial aggregation of subdomain computation costs.

Our simulator was validated by comparing real AMPI (HUANG; LAWLOR; KALÉ, 2004) executions *with simulated SAMPI* executions of the *Ondes3D* seismic wave propagation simulator (DUPROS et al., 2010). As shown in Section 3.3, this application presents *both static and dynamic load imbalance*. In a *real-life evaluation using AMPI* (Section 3.5.2), *dynamic load balancing improved its performance up to 36.58%* on a 288 core cluster. To validate our simulation, we tested two real earthquake scenarios, with two different load balancing heuristics. The results were surprisingly good, showing a *simulated load balancing evolution very similar to real life*. Despite a small imprecision in the estimated execution times, *the simulator was successful at selecting the best load balancing heuristics* for both earthquake scenarios.

Besides validation experiments we also demonstrated *two SAMPI use cases*. In the first one, we demonstrated the use of SAMPI for *load balancing parameter exploration*. For this purpose, we used our simulator to estimate the execution time of Ondes3D with *different load balancing frequencies* and *different levels of over-decomposition*. On the second use case, we used *SAMPI for capacity planning*, that is, to estimate the ideal number of computational resources to run and application, given a certain objective or constraint. For this purpose, we simulated the execution of Ondes3D at four different over-decomposition levels while varying the number of hosts from 8 to 64. SAMPI's trace-replay based load balancing simulation allows to quickly run simulation for a given over-decomposition level with different resource counts. This use-case also benefited

from our spatial aggregation techniques to speed-up the evaluation with different over-decomposition levels.

To finalize our results, we presented a rough estimate of the performance of the SAMPI workflow. For this purpose, we estimated the total *execution time to simulate Ondes3D* with four over-decomposition levels, two load balancer heuristics, and three load balancing heuristics (*24 configurations in total*). *We compared the performance of the* `Raw` *and* `Upgraded` *versions* of the simulator. The original `Raw` simulator does not include spatial aggregation and application-level rescaling. We estimate that for the *Ligurian earthquake scenario* the `Upgraded` simulator improved the *total simulation execution time from ≈25 hour to ≈5 hours* (a reduction of ≈80%). For the *Chuetsu-oki earthquake scenario*, the improvement was *from ≈12 hours to ≈3 hours* (a reduction of ≈75%).

7.1 Future work

As future work, we hope to continue to improve our simulator both in terms of performance and accuracy. It would also be interesting to have more validation scenarios, in the form of different earthquake simulations as well as different applications. We already have a set of input files for a third earthquake scenario called *Sichuan*, which we hope will present a significantly different workload than the two we already tested. In the application front, we need to identify possible candidates that fit the requirements of our simulation. That is, iterative MPI applications with a regular domain decomposition in which the loads of the subdomains in each iteration depend only on their position. In order to compare SAMPI with real-life, we will need to port these applications to AMPI or find applications that already support it. We hoped to include some of this extended validation in this thesis but, due to time restrictions, we decided to prioritize the development of spatial aggregation and application level rescaling.

Another possible development on the validation front would be to validate the results of the simulation-only use cases presented in Sections 5.3 and 5.4. It would be interesting to find out if the predicted best configurations are indeed the best in real life. Even if they are not, this would open-up the opportunity to analyze the causes of the differences, potentially leading to improvements to our simulation workflow.

Other interesting developments would be to simulate different execution platforms and do larger scale simulations. This way we would reinforce the validity of our simulation,

by running the applications at a scale more similar to what would be used in real-life. Besides that, this is an opportunity to test the scalability of our simulation.

We would also like to gather more accurate simulation performance measurements to replace the rough estimates presented on this thesis. To calculate these estimates, we looked independently at the performance of the three components of our workflow (*load balancing simulator*, *spatial aggregation*, and *application-level rescaling*). We would like, instead, to measure the performance of the fully integrated workflow.

There is also the opportunity of future developments on the Ondes3D front. More specifically, it would be interesting to perform a more detailed analysis of the causes of its dynamic imbalance. As we suspect this imbalance comes from CPU-level optimizations, a first step would be to evaluate the application on different CPU architectures. It would also be interesting to investigate possible solutions to eliminate this imbalance.

7.2 Publications

The work presented on Chapter 3, except for the larger scale experiments, was published in the proceedings of the *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2014)* (TESSER et al., 2014).

We presented a very preliminary version of the work that constitutes Chapter 4 and Chapter 5 of this thesis (outdated versions of the results on Sections 5.2 and 5.3), at the *14th Workshop on Charm++ and its Applications*, on April 2016. There results have since then been improved. The slides of the presentation are available for download in the event's website.

A paper on using simulation to evaluate and tune the performance of dynamic load balancing with over-decomposition applied to Ondes3D was published in the Proceedings of the *23th International European Conference on Parallel Processing (Euro-Par 2017)* (TESSER et al., 2017). In this paper, we presented an initial version of our analysis of the load imbalance on Ondes3D (Section 3.3), justified and presented our basic load balancing simulation workflow, its validation (Section 5.3), and parameter exploration simulation results (Section 5.3).

We submitted an extended version of this paper in to the *"Concurrency and Computation: Practice and Experience"* journal. This paper was *accepted with minor revisions*. At the time of this writing, we were finishing the requested revisions. The original Euro-Par 2017 paper was extended in several ways. First, we expanded our load-imbalance

analysis (Section 3.3). Next, we added a *expanded* presentation of our basic SAMPI load balancing simulation workflow, presented at the beginning of Chapter 4, as well as the improved version of the workflow (Section 4.4), including spatial-aggregation (Section 4.2) and application-level rescaling (Section 4.3). In the experimental side, we added a demonstration of the use of SAMPI for capacity planning (Section 5.4), as well as an analysis of the overall performance of the upgraded simulation workflow (Section 5.5).

REFERENCES

ADALSTEINSSON, H. et al. A simulator for large-scale parallel computer architectures. **International Journal of Distributed Systems and Technologies**, IGI Global, Hershey, PA, USA, v. 1, n. 2, p. 57–73, abr. 2010. ISSN 1947-3532.

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**, v. 170, n. 1-2, p. 43–64, 2013. ISSN 0033-4553.

BAGRODIA, R.; DEELMAN, E.; PHAN, T. Parallel simulation of large-scale parallel applications. **International Journal of High Performance Computing Applications**, v. 15, n. 1, p. 3–12, 2001.

BEDARIDE, P. et al. Toward better simulation of MPI applications on Ethernet/TCP networks. In: **PMBS13 - 4th Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems**. [S.l.: s.n.], 2013.

BERGER, M. J.; COLELLA, P. Local adaptive mesh refinement for shock hydrodynamics. **Journal of Computational Physics**, Elsevier, v. 82, n. 1, p. 64–84, 1989.

BHATELE, A.; KALE, L. V.; KUMAR, S. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In: **Proceedings of the 23rd international Conference on Supercomputing (ICS 2009)**. New York, NY, USA: ACM, 2009. p. 110–116.

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

BUYYA, R.; MURSHED, M. M. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. **Concurrency and Computation: Practice and Experience**, v. 14, n. 13-15, p. 1175–1220, 2002. DOI 10.1002/cpe.710.

BöHM, S.; ENGELMANN, C. xsim: The extreme-scale simulator. In: **2011 International Conference on High Performance Computing Simulation**. [S.l.: s.n.], 2011. p. 280–286.

CASANOVA, H. et al. Simulation of MPI applications with time-independent traces. **Concuncurency and Computation: Practice and Experience**, Wiley Online Library, v. 27, n. 5, p. 1145–1168, 2015.

CASANOVA, H. et al. Versatile, scalable, and accurate simulation of distributed applications and platforms. **Journal of Parallel and Distributed Computing**, Elsevier, v. 74, n. 10, p. 2899–2917, jun. 2014.

CATALYUREK, U. V. et al. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In: **International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2007. p. 1–11.

CLAUSS, P.-N. et al. Single node on-line simulation of MPI applications with SMPI. In: **International Parallel Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2011. p. 664–675. ISSN 1530-2075.

COLLINO, F.; TSOGKA, C. Application of the PML absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media. **Geophysics**, v. 66, n. 1, p. 294–307, 2001.

DESPREZ, F. et al. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. In: **Second International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2011) Held in conjunction with ICPP 2011, the 40th International Conference on Parallel Processing**. Taipei, Taiwan: [s.n.], 2011.

DESPREZ, F.; MARKOMANOLIS, G. S.; SUTER, F. Improving the accuracy and efficiency of time-independent trace replay. In: **High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:**. [S.l.: s.n.], 2012. p. 446–455.

DEVINE, K. et al. Zoltan data management services for parallel dynamic applications. **Computing in Science and Engineering**, v. 4, n. 2, p. 90–97, 2002.

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**. [S.l.: s.n.], 2008. p. 253–260.

DUPROS, F.; DO, H.-T.; AOCHI, H. On scalability issues of the elastodynamics equations on multicore platforms. In: **International Conference on Computational Science**. Barcelone, Spain: Elsevier, 2013. (Procedia Computer Science), p. 9 p.

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

DUPROS, F. et al. Parallel simulations of seismic wave propagation on numa architectures. In: **Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, Lyon, France**. [S.l.: s.n.], 2009. p. 67–74.

ENGELMANN, C. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. **Future Generation Computer Systems**, Elsevier, v. 30, p. 59–65, 2014.

ENGELMANN, C.; NAUGHTON, T. Improving the performance of the extreme-scale simulator. In: **2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications**. [S.l.: s.n.], 2014. p. 198–207. ISSN 1550-6525.

ENGELMANN, C.; NAUGHTON, T. A network contention model for the extreme-scale simulator. In: **Proceedings of the 34^{th} IASTED International Conference on Modelling, Identification and Control (MIC) 2015**. Innsbruck, Austria: ACTA Press, Calgary, AB, Canada, 2015. ISBN 978-0-88986-975-2.

FRASCA, M.; MADDURI, K.; RAGHAVAN, P. NUMA-aware graph mining techniques for performance and energy efficiency. In: **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. Los Alamitos, CA, USA: 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.**, ACM, New York, NY, USA, v. 33, n. 5, p. 212–223, may 1998.

HOEFLER, T.; SCHNEIDER, T.; LUMSDAINE, A. LogGOPSim: simulating large-scale applications in the loggops model. In: **Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing**. New York, NY, USA: ACM, 2010. (HPDC '10), p. 597–604. ISBN 978-1-60558-942-8.

HUANG, C.; LAWLOR, O.; KALÉ, L. Adaptive MPI. In: RAUCHWERGER, L. (Ed.). **Languages and Compilers for Parallel Computing**. [S.l.]: Springer Berlin Heidelberg, 2004, (Lecture Notes in Computer Science, v. 2958). p. 306–322. ISBN 978-3-540-21199-0.

HUANG, C. et al. Performance evaluation of Adaptive MPI. In: **Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2006. (PPoPP '06), p. 12–21. ISBN 1-59593-189-9.

INO, F.; FUJIMOTO, N.; HAGIHARA, K. Loggps: A parallel computational model for synchronization analysis. In: **Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming**. New York, NY, USA: ACM, 2001. (PPoPP '01), p. 133–142. ISBN 1-58113-346-4.

JONES, I. S.; ENGELMANN, C. Simulation of large-scale hpc architectures. In: **2011 40th International Conference on Parallel Processing Workshops**. [S.l.: s.n.], 2011. p. 447–456. ISSN 0190-3918.

KALÉ, L.; KRISHNAN, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: PAEPCKE, A. (Ed.). **Proceedings of OOPSLA'93**. [S.l.]: ACM Press, 1993. p. 91–108.

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

KOMATITSCH, D.; MARTIN, R. An unsplit convolutional perfectly matched layer improved at grazing incidence for the seismic wave equation. **Geophysics**, Society of Exploration Geophysicists, v. 72, n. 5, p. SM155–SM167, 2007.

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).

MOCZO, P.; ROBERTSSON, J. O.; EISNER, L. The finite-difference time-domain method for modeling of seismic wave propagation. In: WU, V. M. R.-S.; DMOWSKA, R. (Ed.). **Advances in Wave Propagation in Heterogenous Earth**. [S.l.]: Elsevier, 2007, (Advances in Geophysics, v. 48). p. 421 – 516.

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

NOETH, M. et al. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. **Journal of Parallel and Distributed Computing**, v. 69, n. 8, p. 696 – 710, 2009.

NS-3 CONSORTIUM. **NS3 website**. 2018. <https://www.nsnam.org/>. Accessed in April 2018.

PENOFF, B. et al. Mpi-netsim: A network simulation module for mpi. In: **Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on**. [S.l.: s.n.], 2009. p. 464–471. ISSN 1521-9097.

PÉRACHE, M.; CARRIBAULT, P.; JOURDREN, H. MPC-MPI: an MPI implementation reducing the overall memory consumption. In: **Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings**. [S.l.: s.n.], 2009. p. 94–103.

PILLA, L. L. **Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems**. Thesis (Thesis) — UFRGS; Université de Grenoble, abr. 2014.

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**. [S.l.: s.n.], 2012. p. 118–127.

PILLA, L. L. et al. A Topology-Aware Load Balancing Algorithm for Clustered Hierarchical Multi-Core Machines. **Future Generation Computer Systems**, 2013.

PRAKASH, S.; BAGRODIA, R. L. Mpi-sim: Using parallel simulation to evaluate mpi programs. In: **Proceedings of the 30th Conference on Winter Simulation**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998. (WSC '98), p. 467–474. ISBN 0-7803-5134-7.

RIESEN, R. A hybrid mpi simulator. In: **2006 IEEE International Conference on Cluster Computing**. [S.l.: s.n.], 2006. p. 1–9. ISSN 1552-5244.

RILEY, G. F.; HENDERSON, T. R. The ns-3 network simulator modeling and tools for network simulation. In: WEHRLE, K.; GÜNEŞ, M.; GROSS, J. (Ed.). **Modeling and Tools for Network Simulation**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. chp. 2, p. 15–34. ISBN 978-3-642-12330-6.

RODRIGUES, A. F. et al. The structural simulation toolkit. **SIGMETRICS Performance Evaluation Review**, ACM, v. 38, n. 4, 2011.

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**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 71–78, 2010.

SEGUIN, S.; CRACRAFT, M.; DREWNIAK, J. Static and quasi-dynamic load balancing in parallel FDTD codes for signal integrity, power integrity, and packaging applications,. In: **2004 IEEE International Symposium on Electromagnetic Compatibility, Santa Clara, CA, USA**. [S.l.: s.n.], 2004. p. 107–112.

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. **Int. J. High Perform. Comput. Appl.**, v. 20, n. 2, p. 287–311, 2006. ISSN 1094-3420.

TESSER, R. K. et al. Improving the performance of seismic wave simulations with dynamic load balancing. In: **Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on**. [S.l.: s.n.], 2014. p. 196–203. ISSN 1066-6192.

TESSER, R. K. et al. Using simulation to evaluate and tune the performance of dynamic load balancing of an over-decomposed geophysics application. In: RIVERA, F. F.; PENA, T. F.; CABALEIRO, J. C. (Ed.). **International European Conference on Parallel and Distributed Computing (Euro-Par)**. [S.l.], 2017. p. 192–205.

TIKIR, M. M. et al. PSINS: An open source event tracer and execution simulator for mpi applications. In: **Proceedings of the 15th International Euro-Par Conference on Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2009. (Euro-Par '09), p. 135–148. ISBN 978-3-642-03868-6.

VELHO, P. et al. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. **ACM Transactions on Modeling and Computer Simulation**, Association for Computing Machinery, v. 23, n. 4, oct. 2013.

ZHENG, G. et al. Simulating large scale parallel applications using statistical models for sequential execution blocks. In: **Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on**. [S.l.: s.n.], 2010. p. 221 –228. ISSN 1521-9097.

ZHENG, G.; KAKULAPATI, G.; KALE, L. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In: **Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International**. [S.l.: s.n.], 2004. p. 78.