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

EDUARDO ROCHA RODRIGUES

Dynamic Load-balancing: A New Strategy for Weather Forecast Models

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

Prof. Dr. Philippe O. A. Navaux
Advisor

Dr. Jairo Panetta
Coadvisor

Prof. Dr. Laxmikant V. Kale
Partial Doctoral Fellowship advisor at UIUC

Porto Alegre, September 2011

CIP – CATALOGING-IN-PUBLICATION

Rodrigues, Eduardo Rocha

Dynamic Load-balancing: A New Strategy for Weather Forecast Models / Eduardo Rocha Rodrigues. – Porto Alegre: PPGC da UFRGS, 2011.

101 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Advisor: Philippe O. A. Navaux; Coadvisor: Jairo Panetta.

1. High Performance Computing. 2. Dynamic Load Balancing. 3. Weather Forecast Models. 4. Processor virtualization. I. Navaux, Philippe O. A.. II. Panetta, Jairo. III. Título.

“Blessed be the Lord my strength, which teacheth
my hands to war, and my fingers to fight.”
— PSALMS 144:1

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the Brazilian Council for Scientific and Technological Development (CNPq) and *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (Capes) for their financial support. Also, I am grateful to my advisors, Professors Philippe Navaux and Jairo Panetta, for their guidance and encouragement. In addition, I would like to thank Professor Laxmikant Kale for the productive one-year visit to the University of Illinois at Urbana-Champaign. I am also grateful to Celso Mendes, who was fundamental to the ideas shown here, and Alvaro Fazenda, for his invaluable help. I want to thank my colleagues at UFRGS and, finally, I would like to thank my family for always believing in my potential.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 7

LIST OF FIGURES 8

LIST OF TABLES 10

ABSTRACT 11

1 INTRODUCTION 12
1.1 The problem 12
1.2 Motivation 14
1.3 Hypotheses 14
1.4 Objectives 15
1.5 Contributions 15
1.6 Text organization 16

2 STATE OF THE ART 17
2.1 Load balancing classification 17
2.2 New classification 21
2.2.1 Embedded Load Balancer 21
2.2.2 Load Balancing Frameworks 23
2.2.3 Process Migration 24

3 BALANCING METHODOLOGY 28
3.1 Processor Virtualization 29
3.2 Charm++ 29
3.3 AMPI 30
3.4 Adaptations to AMPI 31
3.5 Preserving the original MPI semantics 33
3.5.1 Thread Local Storage 33
3.5.2 TLS for user threads 34
3.5.3 TLS for virtual processors 36
3.5.4 TLS support for Fortran 37
3.6 Model for Ideal Balancing 38
3.7 Communication pattern 39
3.8 Balancing Algorithms Employed 40
3.9 New Load Balancer 41
3.10 Adaptive strategy 45
3.11 Fully distributed strategies 48

3.11.1 Hilbert-curve-based load balancer . . . 48
3.11.2 Diffusion-based load balancer . . . 50

4 EXPERIMENTAL RESULTS . . . 53
4.1 Brams model . . . 53
4.2 First set of experiments: privatization strategy . . . 54
4.3 Second set of experiments: rebalancing granularity . . . 56
4.4 Third set of experiments: centralized load balancers . . . 57
4.4.1 Virtualization Effects . . . 58
4.4.2 Migration for Load Balancing . . . 62
4.4.3 Adaptive Balance Period . . . 65
4.5 Fourth set of experiments: automatic imbalance threshold . . . 66
4.6 Fifth set of experiments: distributed load balancers . . . 70
4.6.1 Centralized vs. Distributed load balancers . . . 72
4.6.2 DiffusionLB vs. HilbertLB . . . 73

5 FINAL REMARKS AND CONCLUSIONS . . . 80

REFERENCES . . . 83

APPENDIX A PUBLISHED ARTICLES . . . 91

APPENDIX B *RESUMO EM PORTUGUÊS* . . . 93
B.1 Introdução . . . 93
B.2 Método . . . 94
B.2.1 Virtualização de processadores . . . 94
B.2.2 Charm++ and AMPI . . . 94
B.2.3 Adaptações para o AMPI . . . 95
B.2.4 Algoritmos de balanceamento empregados . . . 95
B.2.5 Novo balanceador de carga . . . 96
B.2.6 Estratégia adaptativa . . . 96
B.2.7 Balanceador de carga distribuído . . . 97
B.3 Resultados experimentais . . . 97
B.3.1 Modelo Brams . . . 97
B.3.2 Primeiro conjunto de experimentos: estratégia de privatização . . . 98
B.3.3 Segundo conjunto de experimentos: granularidade de rebalanceamento . . . 98
B.3.4 Terceiro conjunto de experimentos: balanceadores de carga centralizados . . . 99
B.3.5 Quarto conjunto de experimentos: limiar automático de desbalanceamento . . . 100
B.3.6 Quinto conjunto de experimentos: balanceadores de carga distribuídos . . . 100
B.4 Conclusões . . . 101

LIST OF ABBREVIATIONS AND ACRONYMS

AMPI	Adaptive Message Passing Interface
Brams	Brazilian developments on the Regional Atmospheric Modelling System
CATT	Coupled Aerosol and Tracer Transport model
CFL	Courant-Friedrichs-Lewy
CPTEC	Center for Weather Forecast and Climate Studies
DLBL	Dynamic Load Balancing Library
DTV	Dynamic Thread Vector
ELF	Executable and Linkable Format
GCC	Gnu Compiler Collection
GDT	Global Descriptor Table
GLIBC	Gnu C Library
GOT	Global Offset Table
HPC	High-Performance Computing
LB	Load Balancer
LDT	Local Descriptor Table
MIQP	Mixed Integer Quadratic Programming
MISD	Multiple Instruction, Single Data
MM5/MM90	Penn State/NCAR Mesoscale model
MPI	Message Passing Interface
NCAR	National Center for Atmospheric Research
NCAR/CAM3	NCAR Community Atmosphere Model
PAPI	Performance Application Programming
PCCM2	Parallel Community Climate Model for Scalable Message Passing Systems
PE	Processing Element
POSIX	Portable Operating System Interface
RAMS	Regional Atmospheric Modelling System
SPMD	Single Process, Multiple Data
TLS	Thread-Local Storage
TP	Thread pointer
UIUC	University of Illinois at Urbana-Champaign
VP	Virtual Processor
WRF	Weather Research and Forecasting

LIST OF FIGURES

1.1 Dynamic source of load imbalance 13

2.1 A local process and a process that has migrated. 25

3.1 Domain decomposition 31
3.2 TLS structure. 34
3.3 Call graph of the TLS structure initialization. 35
3.4 Communication speed comparison. 39
3.5 Communication pattern of Brams in an execution with 36 processes (darker tones represent larger amounts of communication). 40
3.6 Hilbert curve for the case of 16 threads 42
3.7 16 processes and 256 threads after rebalancing. 44
3.8 Hilbert curve for domains of arbitrary size 45
3.9 Parallel prefix sum. 50
3.10 Diffusion-based load balancer. 51
3.11 DiffusionLB issues. 51

4.1 Context switch time comparison. 55
4.2 Artificial thunderstorm 56
4.3 Effect of virtualization on Brams performance 59
4.4 CPU utilization for various virtualization ratios 60
4.5 CPU usage under different load balancers 64
4.6 Cross-processor communication volume 65
4.7 Comparison between prediction and actual execution. 67
4.8 News about the thunderstorm used as case study. 68
4.9 Precipitation 69
4.10 Execution time with different imbalance thresholds. 70
4.11 Execution time of each timestep. 71
4.12 Comparison between the centralized and distributed algorithms. . . . 72
4.13 Execution time of each load balancer component. 74
4.14 Execution-time of two distributed load balancing approaches. 75
4.15 Number of neighbor processors after 50 invocations of the load balancer. 76
4.16 Rebalancing speed. 77
4.17 Load “trapped”. 77
4.18 Balancing speed with 16K threads and virtualization ratio of 64. . . . 78
4.19 Balancing speed with 64K threads and virtualization ratio of 64. . . . 79

B.1 Comparação do tempo de troca de contexto. 98
B.2 Artificial thunderstorm 99

LIST OF TABLES

2.1 Load balancing classification. 18

3.1 Number of global and static variables in two meteorological models. . 32
3.2 New Fortran attributes and compiler option. 37

4.1 Execution time (seconds) of the Brams application. 55
4.2 Execution time of the artificial thunderstorm case. 57
4.3 Brams execution time on 64 processors 58
4.4 Total number of cache misses on 64 processors in Brams 61
4.5 Load balancing effects on Brams (all experiments were run on 64 real processors) 62
4.6 Observed cost of load balancing 63
4.7 Brams execution time (in seconds) with adaptive load-balancing invocation and *HilbertLB* balancer 65

B.1 Execution time of the artificial thunderstorm case. 99
B.2 Load balancing effects on Brams (all experiments were run on 64 real processors) 100

ABSTRACT

Weather forecasting models are computationally intensive applications and traditionally they are executed in parallel machines. However, some issues prevent these models from fully exploiting the available computing power. One of such issues is load imbalance, i.e., the uneven distribution of load across the processors of the parallel machine. Since weather models are typically synchronous applications, that is, all tasks synchronize at every time-step, the execution time is determined by the slowest task. The causes of such imbalance are either static (e.g. topography) or dynamic (e.g. shortwave radiation, moving thunderstorms). Various techniques, often embedded in the application's source code, have been used to address both sources. However, these techniques are inflexible and hard to use in legacy codes.

In this thesis, we explore the concept of processor virtualization for dynamically balancing the load in weather models. This means that the domain is over-decomposed in more tasks than the available processors. Assuming that many tasks can be safely executed in a single processor, each processor is put in charge of a set of tasks. In addition, the system can migrate some of them from overloaded processors to underloaded ones when it detects load imbalance. This approach has the advantage of decoupling the application from the load balancing strategy.

Our objective is to show that processor virtualization can be applied to weather models as long as an appropriate strategy for migrations is used. Our proposal takes into account the communication pattern of the application in addition to the load of each processor. In this text, we present the techniques used to minimize the amount of change needed in order to apply processor virtualization to a real-world application. Furthermore, we analyze the effects caused by the frequency at which the load balancer is invoked and a threshold that activates rebalancing. We propose an automatic strategy to find an optimal threshold to trigger load balancing. These strategies are centralized and work well for moderately large machines. For larger machines, we present a fully distributed algorithm and analyze its performance.

As a study case, we demonstrate the effectiveness of our approach for dynamically balancing the load in Brams, a mesoscale weather forecasting model based on MPI parallelization. We choose this model because it presents a considerable load imbalance caused by localized thunderstorms. In addition, we analyze how other effects of processor virtualization can improve performance.

Keywords: High Performance Computing, Dynamic Load Balancing, Weather Forecast Models, Processor virtualization.

1 INTRODUCTION

Weather and climate forecasting models are undoubtedly an important class of applications. Currently, they are receiving even more attention because they are an indispensable tool to study climate change. These applications are also computationally intensive and the computer power demand is expected to increase due to higher resolutions, longer simulated periods and more complex models of the atmospheric processes (KINTER III; WEHNER, 2005). Therefore, to run these models in a feasible amount of time, they are usually executed in parallel machines. However, load imbalance is a major obstacle to obtain maximal efficiency.

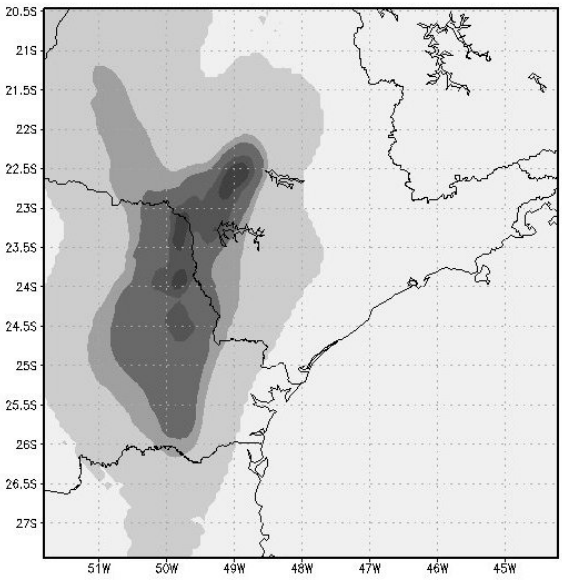
1.1 The problem

Load imbalance corresponds to the uneven distribution of load across tasks of a parallel application. In a synchronous parallel application, in which the tasks synchronize periodically, the total execution time is dictated by the heaviest processor. Weather models are an example of synchronous parallel application and therefore load imbalance delays execution of the entire model.

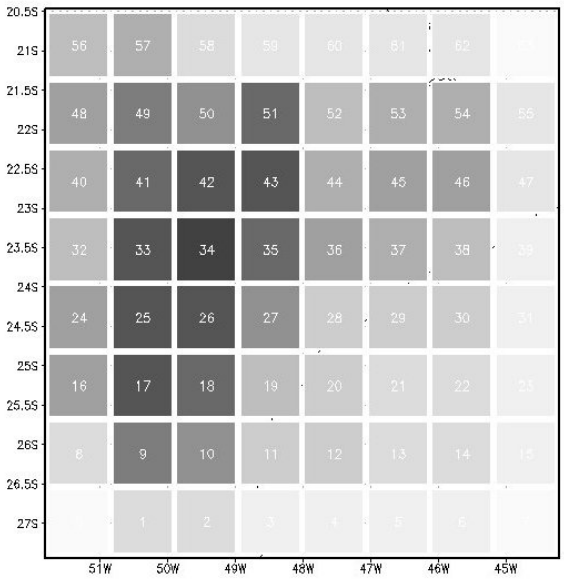
The causes of load imbalance in weather models can be either static or dynamic. One example of a static factor is topography. Many weather models represent the atmosphere with a three-dimensional grid of points, and distribute those points across the processors according to a horizontal domain decomposition (latitude/longitude plane). Each processor receives the full set of atmospheric columns corresponding to the horizontal points in its partition. If the vertical component of the atmosphere is represented by a shaved-cell method, locations with a high topography will have fewer grid points, hence less work to be computed by the model. Although such imbalance changes with the input dataset, it is conceivable that one could derive in advance an ideally balanced decomposition if the model is routinely used on the same region.

The dynamic sources of load imbalance in weather models are much more involved. Some of those sources are predictable, while others are unpredictable. Consider, for example, the effects caused by earth rotation. Two domains with the same latitude receive different values of solar incidence at a given time (e.g. day and night); this results in

distinct amounts of computation for the radiative components of the model on the two underlying processors. This kind of imbalance repeats with a periodicity of twenty-four hours in simulated time.



(a) Precipitation forecasted by a real weather model



(b) Grayscale-coding of observed computational load on 64 processors

Figure 1.1: Dynamic source of load imbalance

Meanwhile, other imbalance factors lack such predictability. As an example, running a certain weather model (described in Section 4.2) on 64 processors resulted in the precipitation forecast depicted in Figure 1.1a. Execution time instrumentation revealed the computational loads in the 8×8 set of processors indicated by the grayscale-coded distribution of Figure 1.1b, in which darker regions correspond to higher loads. There is a clear

correlation between rain and computational load: domains containing rain correspond to overloaded processors.

If the simulation of Figure 1.1a is allowed to proceed, the rain may "move" across domains, changing the distribution of overloaded and under-loaded processors. This movement is unknown a priori (since predicting where the rain will go is precisely what the model is designed for!). Responding to such unpredictable sources of load imbalance in weather models remains mostly an open problem.

1.2 Motivation

Currently, there is an increasing demand for higher resolution weather forecasting simulations. Some weather forecast centers are already running models at resolutions of a few kilometers and those are soon expected to increase further. However, increasing resolution is not just a matter of running the same model code with a finer mesh. As resolution increases, the executed code changes to simulate new phenomena that were previously in a sub-grid scale. Higher resolution allows the representation of localized phenomena that cannot be explicitly treated at larger scales. One example is small to medium scale cloud formation, which is treated by statistical methods at scales larger than the cloud itself and by explicit methods at finer scales.

A concrete instance of this fact is cumulus convection. At lower resolution, this phenomenon is usually parameterized (GRELL; DÉVÉNYI, 2002). Meanwhile, at resolutions of a few kilometers, it is possible to use cloud microphysics. This component is concerned with the formation, growth and precipitation of raindrops and snowflakes. This atmospheric process does not have horizontal data dependences, but it suffers from load imbalance. Indeed, it is well known that thunderstorms cause this problem (XUE; DROEGEMEIER; WEBER, 2007). Other sources of load imbalance are chemical and biological processes, such as those involved in biomass burning.

Therefore, as a consequence of increasing resolution and complexity, weather forecast models face load imbalance. There has been some research on the usage of load balancing strategies in meteorological models, but virtually no production code has this feature.

1.3 Hypotheses

In order to deal with the load balancing issues presented in the previous sections, we must consider some hypotheses about weather models:

- Real-world weather models are very large programs that require many man-hour to design and implement (typically more than 100.000 lines of source code). Therefore, embedding a load balancing mechanism into an existing model is a very complex and error-prone task;
- Typically, meteorological models have a fixed communication pattern. Hence the

load balancing strategy should take advantage of this characteristic when it moves load around to perform rebalancing;

- A rebalancing frequency and imbalance threshold should be established so that the load movement takes place only when it is beneficial to the overall performance;
- The load balancing algorithm should be very computationally inexpensive even if it does not produce the optimal load distribution, otherwise it will actually degrade performance.

1.4 Objectives

The objective of this thesis is to investigate both existing and new load balancing strategies to be applied to weather models. These strategies must take into account the stated hypothesis. In other words, they have to be as as non-intrusive as possible so that they can be applied to real-world models. Furthermore, the strategies must consider the communication pattern of this type of application, the cost incurred by the movement of load in a parallel machine and the cost of the load balancing algorithm itself. *My thesis is* to achieve this objective using:

- Processor Virtualization to decouple application and load balancing strategy;
- Strategies that minimize the amount of changes to the application in order to use Processor Virtualization;
- Load balancing heuristics that are fast to execute;
- Heuristics that consider spatial locality of the tasks;
- Information on the impact of moving load to achieve load balancing and use this information to decide when it is worthy to move that load;
- Techniques to automatically select the best imbalance threshold for an adaptive load balancing scheme;
- Distributed load balancers that can be used on machines with a large number of processors.

1.5 Contributions

In this text, we propose an approach that can handle both predictable and unpredictable sources of load imbalance in a uniform fashion. Moreover, minimal changes to the original application code are required. We present experimental results with an existing weather forecasting model, using real atmospheric data, to show that it is possible to improve the model's performance by employing appropriate load balancers.

One contribution of this thesis is to demonstrate that processor virtualization can be effectively used to improve performance of MPI applications that suffer from load imbalance, which is typically the case in weather and climate models. We also investigate other benefits of virtualization.

Another contribution is the development of a new load balancing strategy based on the Hilbert curve, that takes into account the characteristics of the application considered. This strategy uses the spatial locality of the virtual processors to decide where to move load. This strategy is very lightweight and effective. We also analyze the frequency at which the load balancer should be invoked and when it is worthy to perform rebalancing.

In addition, we propose a new technique to privatize data in user-level threads, so that an existing application can more easily be ported to virtual processors based on this type of thread. We evaluated this technique with benchmarks and a real application to show the merits of our idea.

Finally, it is important to emphasize that, although our focus here is to balance load in weather models, the developed strategies can also be applied to other applications. The importance of weather applications was the main motivation for the ideas presented in this text. However, many other applications share the same structure and present similar load imbalance behavior. Even applications that do not use only horizontal decomposition can apply our load balancer, because it can be easily extended to higher-dimensional decompositions.

1.6 Text organization

The reminder of this text is organized as follows. In Chapter 2, we review previous work in load balance. Chapter 3 presents the tools that we used and describes our methodology. Chapter 4 contains the results from our experiments, and Chapter 5 presents our final remarks, conclusions and potential avenues for future work.

2 STATE OF THE ART

Load balancing is not a new theme. For a long time it has been studied and many different strategies have been created. However, this theme is still relevant as it can be seen from the many articles written recently. The reason for this interest is two-fold: (1) the recent advances in architectures and (2) the demand that some applications pose these days. Advances in network have emerged and made possible the use of some techniques that were not viable before. An example of this fact is task migration. Currently, migrating a task image can be achieved in a feasible time so that the overall performance of an application can be improved. In the application side, the imbalance may grow as some applications increase in complexity, as described in the previous chapter. Furthermore, load balancing strategies are rarely used on production code and that presents questions about the proposed strategies.

This chapter presents several studies in load balancing in order to contextualize our own proposal. It starts with a broad load balancing classification. We also present two new classes, Embedded Load Balancers and Load Balancing Frameworks. In addition, we discuss process migration, which is in the domain of Load Balancing Frameworks.

2.1 Load balancing classification

Load balancing can be considered a problem of task scheduling and, as such, the task scheduling taxonomies of the solutions for this type of problem can be used to study and compare load balancing strategies. In this section, we present different criteria to classify task scheduling and examples of each class.

Casavant and Kuhl (CASAVANT; KUHL, 1988) present a task scheduling taxonomy for general-purpose distributed computing systems. It encompasses a very broad range of load balancing characteristics. However, since this taxonomy is intended to be general, it misses some specificities that can be found in more specialized strategies. For example, this taxonomy ignores hybrid approaches, that combine two or more simple strategies.

Rotithor (ROTITHOR, 1994) deals exclusively with dynamic task scheduling. He treats task scheduling as a two-component system. These components are state estimation and decision making. System state estimation refers to the dissemination of state

information throughout the parallel system in order to construct an estimate of the system load based on the state of individual processors. Decision making is concerned with task assignment (to processors) based on the estimate provided by the system state estimation.

As for dynamic scheduling, the classification presented in (ROTITHOR, 1994) is more complete than the one found in (CASAVANT; KUHL, 1988). However, the two-component division described in the first taxonomy creates some classes that have no instances, similar to the MISD class found on the Flynn taxonomy for computer architectures. For example, a fully distributed state estimator does not combine well with a centralized decision making strategy. Nonetheless, this is a valid class in the Rotithor's taxonomy.

Plastino *et al.* (PLASTINO et al., 2004) propose a taxonomy for load balancing of SPMD applications. They argue that most load balancing taxonomies deal only with functional decomposition, therefore, their focus is on domain decomposition. However, many of the classes presented can be employed both to functional and domain decomposition.

None of these taxonomies consider load balancing strategies that are either embedded or not into the applications' code. Since this property is important for practical use we include it in our own classification. Frameworks to decouple the load balancing strategies from the applications' code are also presented. We discuss this issue in the final two sections of this chapter. The remainder of this section presents a table with load balancing classes and examples found in the literature. At the end of this chapter, we classify our own strategy in this taxonomy.

Table 2.1: Load balancing classification.

Classes	**Description**	**Examples**
Local or Global	Load balancing can be local, when it is concerned with the scheduling of tasks in a single node, or global, when more nodes are available and the job of the load balancer is to define in which node the tasks must execute.	(SIDDHA; PALLIPADI; MALLICK, 2005) (BOKHARI, 1979)

Continued on the next page.

Classes	Description	Examples
Static or Dynamic	Static load balancing takes place prior to the application execution. Therefore, the load balancer must know the behaviour of the application in advance in order to produce task scheduling decisions. Meanwhile, dynamic load balancing makes few assumptions about the task characteristics and obtains information of the application load before making a rebalancing decision. It is suitable for applications whose behaviour changes during execution.	(SHEN; TSAI, 1985) (STONE, 1978) (DEVINE et al., 2005) (DOBBER; KOOLE; MEI, 2005) (ZHENG et al., 2010)
Optimal or Sub-optimal	As an optimization problem, load balancing can be solved to optimality, i.e. to find the best solution given a certain number of restrictions. However, since there is no efficient solution for this problem, a sub-optimal solution may be used. There are two categories inside the sub-optimal class: approximation algorithms and heuristics.	(ALTMAN; AYESTA; PRABHU, 2008) (BOKHARI, 1981)
Approximation algorithms or Heuristics	Approximation algorithms run in polynomial time and find solutions that are guaranteed to be close to optimum. The result is ideally distant from an optimum solution by a constant factor. On the other hand, heuristics are procedures that find solutions rapidly but they do not guarantee that the solution is optimal or even close to optimum.	(ICHIKAWA; YAMASHITA, 2000) (BILLIONNET; COSTA; SUTTER, 1992) (BRAUN et al., 2001) (FRANCESCHELLI; GIUA; SEATZU, 2007)

Continued on the next page.

Classes	Description	Examples
Centralized, Decentralized or Hybrid	The centralized load balancer has a dedicated central entity that collects information about the entire system and performs load balancing decisions. Conversely, in the fully distributed strategy each processors exchanges state information with a subset of processors and takes its own load balancing decisions. For large machines, Zheng (ZHENG, 2005) argues that neither centralized or fully distributed load balancing strategies is appropriate. For this type of machine, a hybrid approach, which combines features of both previous strategies, may be used.	(ZHENG, 2005)
Synchronous or Asynchronous	A synchronous load balancer executes simultaneously in all processors. Points of synchronization are specified in the application code so that the processors stop their regular execution and start the load balancing code. Meanwhile, asynchronous approaches can be executed by any processor at any moment.	(ZHENG et al., 2010) (WILLEBEEK-LEMAIR; REEVES, 1993) (GUIL; ZAPATA, 1997) (BARKER et al., 2004)
Cooperative or Non-Cooperative	Within the realm of distributed dynamic scheduling, the literature distinguishes between those mechanisms that require cooperation among the distributed components (cooperative) and those in which the individual processors make decisions independent of the actions of the other processors (non-cooperative).	(GROSU; CHRONOPOULOS, 2005) (GROSU; LEUNG, 2002) (KHAN; AHMAD, 2006)
Adaptive or Non-Adaptive	Being adaptive in the context of load balancing means that the scheduler changes its scheduling policy dynamically to adjust to the environment and previous scheduling decision.	(SHAH; VEERAVALLI; MISRA, 2007)

Continued on the next page.

Classes	Description	Examples
Collective or Individual	In the collective algorithms, the balancing decisions are taken by a group of processors. In contrast, these decisions are taken by individual processors in the individual approach. Individual algorithms balance load of a single processor by selecting processors that can receive or send extra load.	(LEE et al., 2005)
Sender, Receiver or Symmetric initiated	In a sender initiated strategy, overloaded processors seek to find underloaded processors to send the extra load. In the receiver initiated strategy, underloaded processors seek overloaded ones to receive the extra load. A symmetric initiated policy combines both previous strategies.	(BLUMOFE et al., 1995)
Periodic or Event-driven	Periodic algorithms are activated regularly, independently of the current workload distribution. On the other hand, the event-driven strategy is activated when a given condition is satisfied.	(ZHENG, 2005) (ICHIKAWA; YAMASHITA, 2000)

2.2 New classification

The previous taxonomies do not distinguish load balancing strategies that are embedded into the application code from load balancing frameworks, which isolate the application code from the load balancing mechanism itself. However, in the literature it is common to find these two classes. In this section, we present these two classes and show examples.

2.2.1 Embedded Load Balancer

The most common way to implement load balancers is embedding the strategy into the application itself. Many examples of such approach can be found in the literature. The Gordon Bell winner of 2005 (STREITZ et al., 2008) is an example of embedded load balancer. That article presents a new molecular dynamics application, which is a type of computer simulation. This simulation consists of computing the trajectories of particles, typically atoms and molecules, subject to a potential in order to study some macroscopic properties of the matter. Each processor in the algorithm presented in the paper is in

charge of a sub-domain that can be adjusted so that load balancing can be achieved.

The general structure of a molecular dynamic simulation is presented in (RAPAPORT, 2004). Since this structure is usually simple, the load balancing strategy tends to be simple as well. Meanwhile, implementing a load balancing strategy in other applications may be much more complex. Weather forecast and climate models are examples of such applications.

Koziar *et al.* (KOZIAR; REILEIN; RUNGER, 2005) present a study about load imbalance on a regional weather model named Gesima. That article points out that, although weather models have a regular structure, atmospheric processes can cause load imbalance throughout the domain. The objective of this study was to select criteria to activate microphysics so that the results were correct, but also to balance load across the processors. However, this work considers only one source of load imbalance; it does not deal with the composition of effects (for example, microphysics and the remainder of the model). In addition, that article does not perform actual load balancing, but it only evaluates possible directions to adapt the application to deal with this problem.

MM90 is an example of meteorological model that contains dynamic load balancing. It is a Fortran 90 parallel implementation of the Penn State/NCAR Mesoscale model (MM5). The article (MICHALAKES, 1997) presents how the MM90 was parallelized, the dynamic load balancing strategy and performance results. The domain decomposition is done in two dimensions (north/south and east/west). The sub-domains may be irregular and the processing unit is a mesh point. This approach makes load balancing easier, since any mesh point can migrate from one processor to another in order to rebalance load, even if the sub-domains become irregular.

The MM90 code is instrumented so that the load balancing strategy makes an estimate of the imbalance. This instrumentation basically measures the computational cost of the vertical atmospheric columns. Periodically, a new mapping of the domain is computed and its efficiency is compared with the previous mapping. The article, however, does not describe how the new mapping is done nor how the performance results are compared. According to Rotithor (ROTITHOR, 1994), these two issues are critical to the efficiency of the load balancing strategy.

The spatial resolution of meteorological models is limited by the computing power available (with the exception of purely meteorological factors such as inadequate observation data). In order to avoid this restriction, the user can use downscale techniques. The article (GHAN et al., 2002) presents a new downscale technique that uses orography to improve the resolution of the model NCAR/CAM3 (National Center for Atmospheric Research/ Community Atmosphere Model). This scheme improves resolution but causes load imbalance.

Ghan and Shippert (GHAN; SHIPPERT, 2005) present a load balancing algorithm to the downscale technique based on orography. That article shows that a static load balancer can be used, because the elevation classes do not change. The proposed algorithm not

only considers load but also communication. Since this is a static load balancer, the strategy cannot be used in a dynamic context. Furthermore, this algorithm is specific to the downscale technique used.

Foster and Toonen (FOSTER; TOONEN, 1994) identified that physics computation also causes load imbalance in climate codes. Examples of physics computation are radiation, which changes with the movement of the planet, and cloud and moisture, which are transported with the movement of the atmosphere. They proposed a dynamic scheme to balance the load based on a carefully planned exchange of data across processors at each timestep. Their rationale was that the model employed three types of timesteps, with varying degrees of radiative calculations, and a good decomposition for one kind of timestep was not as good for the other kinds. When applying their scheme to the PCCM2 climate model, they achieved an overall improvement of 10% on 128 processors, but that improvement degraded with more processors. This technique requires a significant amount of data exchange between processors at each timestep. As the model is scaled, this overhead may dominate execution and offset any potential gains provided by the load balancing scheme. Also, implementing this scheme requires intimate knowledge of the application's code, to determine which variables must be exchanged between processors.

Xue *et al.* (XUE; DROEGEMEIER; WEBER, 2007) stated that sub-domains assigned to some processors may incur 20%-30% additional computation due to active thunderstorms. They also claimed that the complexity of the associated algorithm and the overhead imposed by the movement of load prevent the use of load balancing techniques.

2.2.2 Load Balancing Frameworks

The load balancing strategies presented in the previous section are intimately linked to the specific application considered. In this section, we describe some load balancing frameworks. The objective of these frameworks is to isolate the application from the load balancing strategies. Some of them must be considered at code development time making them less suitable for legacy code. Meanwhile, other strategies can be used with existing applications. These strategies are based on process migration. We present process migration in the next section.

Dynamic Load Balancing Library

In general, loops are the structures that offer the best opportunities for parallelization; their iterations can be divided among the available processors for parallel execution as long as the dependencies are satisfied. Nonetheless, many factors can cause load imbalance in this type of parallelization, for example, heterogeneous architectures (even heterogeneous multi-core (KUMAR et al., 2004)) or iterations with different loads. Much research has been done in loop scheduling, whose objective is to distribute load equally to the processors of a parallel machine (HUMMEL; SCHONBERG; FLYNN, 1992) (BANICESCU; VELUSAMY, 2002). Banicescu *et al.* (BANICESCU et al., 2005) present the

Dynamic Load Balancing Library (DLBL), which employs an object migration mechanism to rebalance load in loops.

The DLBL library aims to parallelize loops in distributed memory architectures. It employs data migration in order to rebalance load. The strategy used in this library is a master/slave approach in which the master schedules loop iterations. Initially, the iterations are equally divided among the processors. The first processor that finishes its iterations requests more iterations from the master. This event triggers a decision making mechanism in the master processor to determine which processor has excessive load. The overloaded processor then sends some of its iterations to the underloaded one.

Zoltan

Zoltan (DEVINE et al., 2002) is an open source library developed at Sandia National Laboratories that provides load balancing and partitioning services. This library is not restricted to a specific application and does not impose any restriction to the application. However, the Zoltan usage must be considered at code development time. Therefore, for legacy codes, this means that the original code has to be modified.

Zoltan interacts with the application by means of callback routines, which are written by the application developer and that return data to the Zoltan core system. One of the services provided by this library is dynamic load balancing, which has a set of predefined rebalancing algorithms. The user can easily change the algorithm used and hence evaluate which one provides better performance. In order to help moving load around, Zoltan also provides routines to migrate data among the processors. In addition, a directory service is provided to locate application objects. Finally, Zoltan provides an unstructured communication package to simplify communication.

There is no performance evaluation of Zoltan in large machines. Teresco *et al.* (TERESCO; FAIK; FLAHERTY, 2006) present an evaluation with a small cluster of 16 processors. In addition, a critical limitation of this system is that it has to be taken into account at design time; it cannot be easily integrated into an existing application.

2.2.3 Process Migration

All previous load balancing frameworks are intended to be used at code development time. This approach is particularly inconvenient for large legacy applications, like weather models. A more flexible alternative to deal with the load balancing problem involves process migration. In this way, the user develops his/her application as usual. On the occurrence of load imbalance, the system can migrate some processes from more loaded processors to less loaded ones.

In this section we describe three systems that employ the concept of process migration: Mosix, Kerrighed and AMPI/Charm++.

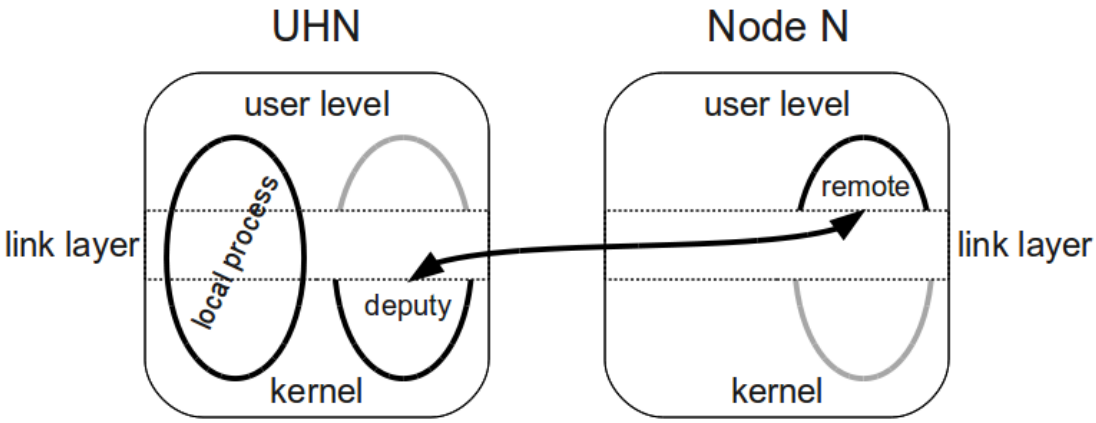


Figure 2.1: A local process and a process that has migrated.

Mosix

Mosix is a tool intended for cluster of workstations that offers load balancing by means of heavy process migration (BARAK; LA'ADAN; SHILOH, 1999). This system is implemented in kernel level and allows migrations preemptively and transparently. Mosix basically consists of two parts: (1) a Preemptive Process Migration (PPM) mechanism and (2) a set of algorithms to adaptively share resources.

The Mosix system is intended for distributed memory machines. A process is created in a node called Unique Home Node (UHN) of the parallel machine. If the load of a node exceeds a certain threshold, the PPM mechanism moves some processes from that node to others. Even after migrating, a process still keeps part of its code running at the UHN. In addition, the decisions concerning the destination of a process are done autonomously by each node. The algorithm that controls these decisions can be changed by the user. A fully distributed algorithm can be used to reduce the impact of communication of the load balancing scheme.

In order to migrate, the Mosix system divides a process in two contexts: (1) the user context, called *remote*, and (2) the system context, called *deputy*, that refers to the process executing in kernel mode (Figure 2.1). The user context is what actually migrates and has the application code and data. The *deputy* stays at the UHN and communicates with the *remote* through the interconnect network whenever a syscall is invoked.

The Mosix system is implemented as kernel extensions. As a consequence, its usage implies changes in the kernel of each cluster node. The open source version of this system (OpenMosix) is only available to the Linux kernel 2.4.46, therefore, its users cannot use the newer version of this kernel. Lottiaux *et al.* (LOTTIAUX et al., 2004) show that applications running with Mosix have lower performance if they use *sockets*. That is

because all communications must go first to the UHN node before going to their target. This issue is a severe limitation for highly coupled parallel applications.

Kerrighed

Kerrighed is a Single System Image operating system for clusters (MORIN et al., 2003) that provides support for both MPI and shared memory programming models. Its main design goals are to provide high-level services to high performance parallel and sequential applications on clusters of computers. Kerrighed is implemented as an extension to the Linux kernel.

This system is similar to Mosix. Therefore, the user can use migration to rebalance load. It offers better performance with respect to communication, as described in (LOTTIAUX et al., 2004). Nonetheless, the user still has to install a specific kernel in his/her cluster nodes. Furthermore, it is still in development.

Charm++ / AMPI

The idea of using process migration is very attractive, since it means fewer changes to existing applications. That is because the user can, in principle, develop his/her application as usual. During execution, each processor is typically in charge of more than one process. On the occurrence of load imbalance, the system can migrate some processes from more loaded processors to less loaded ones.

Nonetheless, the cost of migrating an entire process usually discourages the widespread use of this technique. An alternative is to implement the tasks of a parallel application as user-level threads. Doing so, the migration cost is dramatically reduced. This strategy can be found in AMPI, an implementation of MPI developed over the Charm++ framework.

We chose this framework to develop our load balancing strategy. We describe it thoroughly in the next chapter. The use of AMPI, however, does not guarantee the solution for the load balancing problem. It only means that we employ processor virtualization based on user threads, as we shall describe in the next chapter. We developed a load balancer that is:

- *global* - since we are interested in distributed memory machines;
- *dynamic* - because it is not possible to predict the imbalance of most weather models;
- *heuristic-based* - because even for modest sizes, the optimum solution takes too long to find;
- *centralized* - in a first moment we will use this approach and afterwards a distributed one;
- *synchronous* - because weather models are iterative applications and the load balancer can be easily invoked between time-steps;

- *non-adaptive* - although an adaptive scheme is proposed to further improve performance;
- *collective* - because the objective is to balance the entire set of processors and
- *periodic* - even though we use a threshold to trigger load balancing.

3 BALANCING METHODOLOGY

In this chapter, we describe our methodology to balance load in meteorological models. We rely on the concept of processor virtualization and its implementation on AMPI (HUANG et al., 2006). The chapter starts with a description of processor virtualization and a description of Charm++ and AMPI. Following that, we present the changes needed to use the migration capabilities of AMPI in a real-world application.

The changes to the AMPI environment include a new technique to privatize data in user-level threads. That is because AMPI implements virtual processors as user-level threads. The reason for this choice is that the migration cost is dramatically reduced with the use of this type of thread instead of heavy processes or kernel threads. However, global and static variables are shared by user-level threads located in a single processor. Therefore, MPI processes based on user-level threads restrict the type of MPI program which they can execute, i.e. those programs that do not have private global and static variables. We propose a new way to privatize data in user-level threads in order to enable MPI processes based on this type of thread to execute a larger class of parallel programs. This technique was crucial for the results we obtained, because the other options would require many changes to the application (something that we were trying to avoid) or they would be excessively coslty.

Still, enabling an application to run on AMPI is not enough to guarantee load balancing. We tried some existing load balancers that could in principle be useful for meteorological models, but, as we present in Chapter 4, they were not. That is the reason why we implemented a new load balancer. This load balancer is based on a heuristic that implicitly incorporates the communication pattern that typically is found in meteorological models. Furthermore, the algorithm is cheap enough to be invoked very frequently. We investigated an adaptive scheme that invokes the load balancing algorithm more frequently, but only migrates work across processors when the imbalance is beyond a certain threshold. With this adaptive scheme, we observed a considerable improvement in performance as we shall present in Chapter 4. All these strategies are centralized, i.e., a single processor takes load balancing decisions. This approach works well for moderately large machines. For larger machines, we propose a distributed load balancer, which is presented at the end of this chapter.

3.1 Processor Virtualization

Processor Virtualization (OTTO, 1994) refers to the ideia that the programmer decomposes a problem into a set of VP entities that will execute on P processors. Ideally, VP is much larger than P, so that, on the occurrence of load imbalance, the system migrates some entities from overloaded processors to underloaded ones. These entities are called virtual processors, because they emulate what a single processor would do in a conventional parallel execution, in which the number of tasks is equal to the number of processors.

One possible issue with Processor Virtualization is binary reproducibility, which means that the numerical results computed in a time-step are independent of the number of tasks used. This is because the user has to employ more tasks than he/she would normally use in order to benefit from the load balancing capabilities that this technique allows. If the results vary with the number of tasks, that may undermine the usefulness of these results.

The ratio between the number VP and the number of physical processors (P) available for the program is called the virtualization ratio. The ideal values for this ratio depend strongly on the underlying application and machine characteristics. Users cannot increase the number of VPs indefinitely, because the application may impose limits to the maximum number of tasks. Moreover, an excess of VPs can actually hurt performance, as we will see on Chapter 4.

In addition to enabling load balancing, Processor Virtualization has also some extra benefits: (1) automatic overlap of computation and communication and (2) better use of cache. A virtualized execution can benefit from automatic overlap of computation and communication, even without explicit use of nonblocking message passing calls: when a certain virtual processor blocks on a receive, another virtual processor can execute. In addition, this approach allows for better cache use, because each sub-domain is smaller than it would be in a non-virtualized environment. Consequently, these smaller sub-domains can more easily fit in cache.

3.2 Charm++

Charm++ (KALÉ et al., 2008) is an object-oriented parallel programming system aimed at improving productivity in parallel programming while enhancing scalable parallel performance. A guiding principle behind the design of Charm++ is to automate what the system can do best, while leaving to the application programmers what they can do best. It is assumed that programmers can specify what to do in parallel relatively easily, while the system can best decide which processors own which data units, as well as which work units each processor executes.

At its core, Charm++ employs the idea of processor virtualization based on migratable objects. In this approach, the programmer decomposes a problem into a set of N objects that will execute on P processors, where typically $N \gg P$. The programmer's view of

the execution is of N objects and their interactions. Meanwhile, the underlying runtime system, implemented by Charm++, maps those objects to the P processors. This mapping is dynamic and objects can migrate across processors during execution, under control of the runtime system. That over-decomposition scheme effectively decouples the partitioning of the problem from the physical machine where the program will run. This, in turn, provides many opportunities for runtime optimizations, such as better overlap between computation and communication, or improved communication strategies.

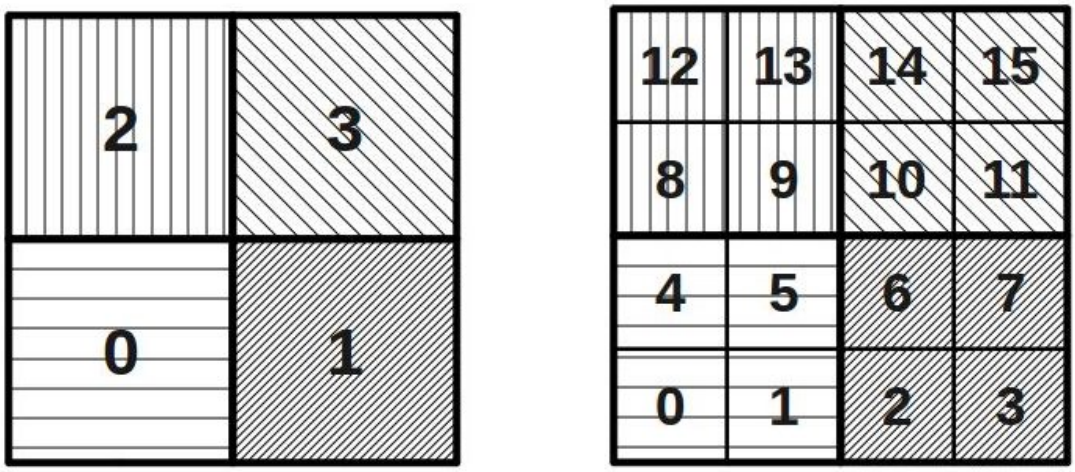
Object-based virtualization leads to programs that automatically respect locality, in part because objects provide a natural encapsulation mechanism. At the same time, it empowers the runtime system to automate resource management. The combination of features in Charm++ has made it suitable for the expression of parallelism over a range of architectures, from multi-core desktops to existing petaFLOP-scale parallel machines. Moreover, it has enabled scaling real applications to thousands of processors on several scientific areas, such as molecular dynamics (BHATELE et al., 2008), quantum chemistry (BOHM et al., 2008), computational cosmology (JETLEY et al., 2008), rocket simulation (JIAO et al., 2005) and others.

3.3 AMPI

Adaptive MPI (AMPI) is an implementation of the MPI standard based on Charm++ (HUANG et al., 2006). In AMPI, each MPI task is embedded in a Charm++ object and implemented as a user-level thread. Differently from kernel-threads, these user-level threads are lightweight and result in very short context-switch times (ZHENG; LAWLOR; KALÉ, 2006). Like any Charm++ object, those threads can migrate across processors as well. Hence, by using AMPI, many of the benefits from processor virtualization become available to *legacy* MPI applications, written in C/C++/Fortran.

In AMPI, N Charm++ objects are used to implement the original N MPI tasks. Thus, each of those tasks has the "illusion" of owning an AMPI virtual processor. Many VPs can share a physical processor during execution: each VP is associated to one of the user-level threads comprising the process that is running on that processor. Only one thread executes at a time. When the current thread blocks, on a receive for example, another thread resumes execution.

A potential problem that may arise from the sharing of a physical processor by multiple VPs is a conflict in the access to global and static variables in the application. This is because the VPs are implemented as threads, therefore they share the same address space. With the original MPI, this conflict does not exist because each task has its own address space, hence a given global or static variable can be accessed by only one task. To resolve that conflict in AMPI, it is necessary to *privatize* those variables. There are a few different mechanisms for such privatization, with varying degrees of automation. Although the privatization process can result in some overhead due to context-switching during execution,



(a) 4 MPI tasks

(b) 16 AMPI threads

Figure 3.1: Domain decomposition

there are techniques that keep this overhead low, regardless of the number of globals or statics in the application's code (RODRIGUES et al., 2010). We demonstrate, in the next sections, possible ways to handle this privatization issue.

3.4 Adaptations to AMPI

For a given MPI code to benefit from the advantages of AMPI, it must exploit processor virtualization. In this approach, the idea is to replace MPI's task decomposition with a new scheme that over-decomposes the same domain into a larger number of AMPI threads. Figure 3.1 illustrates that: on the left, we represent a regular domain decomposition with four MPI tasks and four processors, whereas the right side corresponds to a possible decomposition of the same domain into sixteen AMPI threads. With AMPI, there will be sixteen ranks, each associated to one thread. In this particular case, AMPI would have VP=16 and a resulting virtualization ratio of four. Other values of VP and virtualization ratio could be used as well. From the application's perspective, the execution with AMPI will behave similarly to an execution under sixteen MPI ranks.

Two issues have to be considered when using the over-decomposition scheme: (1) the application must have the property of binary reproducibility, which means that the numerical behaviour of the code is independent of the number of MPI ranks in use; and (2) the user has to consider the increase in memory usage due to the over-decomposition. This is because the amount of ghost cells and stack increases with the virtualization ratio. In our experiments, we measured an increase in memory usage. However, the benefits were still enough to justify the use of this technique.

In the example of Figure 3.1, AMPI will start the execution with threads {0,1,4,5} mapped to the first processor, threads {2,3,6,7} mapped to the second processor, and so on. As the execution progresses and the load balancer is invoked, threads may migrate

Model	Globals	Statics
Brams	10237	519
WRF-v.3	8731	550

Table 3.1: Number of global and static variables in two meteorological models.

across the four processors. Actual migrations depend on the observed behavior prior to load balancing and on the policy of the particular load balancer in use. Given the iterative behavior of meteorological applications, a natural place to invoke the load balancer is between a certain number of timesteps in the simulation. To perform that invocation at every K timesteps, we can simply add the following line at the end of the main loop in the application source code:

if (mod(iteration,K) == 0) call MPI_Migrate()

When using the AMPI decomposition shown on the right of Figure 3.1, four threads share the same physical processor. As we observed in the previous section, this may create problems for global and static variables. On platforms that support the ELF format, AMPI provides a build-time flag that can automatically handle the privatization of global variables. This flag (*-swapglobals*) ensures that each thread will have its own version of a given global. At thread context-switch time, those versions are automatically switched by the Charm++ runtime system. Unfortunately, this method does not work for static variables. A possible workaround is to create a module, insert all static variables into that module, and replace their original declarations in the source code. This scheme would effectively transform the statics into globals, which could then be handled with *-swapglobals*.

While the *-swapglobals* scheme effectivelly privatizes global variables, it may not be very efficient in some cases, because it makes the time of thread context switch proportional to the number of globals in the code. The reason for this fact is that the *-swapglobals* flag forces the code to be compiled as shared library. Consequently, the linker creates a global offset table (GOT) that contains pointers to all global variables in the application. To privatize globals, at context switch the Charm++ runtime system changes every entry of the GOT to the corresponding data of the resuming thread. In some applications, however, the number of static and global variables is very large. Table 3.1 shows those numbers for Brams and for WRF, two popular weather forecasting codes. For codes like these, with thousands of globals and statics, the context switch time becomes excessively large, as many operations are needed to update the GOT.

To eliminate this context switch overhead, we developed a new privatization strategy (RODRIGUES et al., 2010) based on Thread-Local Storage (TLS). We discuss in details this strategy in the next section.

3.5 Preserving the original MPI semantics

MPI programs assume that each task has its own private address space. Therefore, global and static data is private to each task. However, the use of user-threads to implement these tasks breaks this assumption. In order to privatize global and static data in user-level threads and preserve the original MPI semantics, we adapted the mechanisms that compilers, linkers and run-time system have to enable the specifier *__thread*. This specifier is specifically employed to privatize data in kernel-level threads and relies on a mechanism called Thread Local Storage (TLS). However, there was no user-level thread library that enables the functionality intended for this specifier. In this section, we describe how TLS works and how we implemented the support of this mechanism on user-level threads. Moreover, this section shows how the TLS was integrated to the AMPI environment and how the support to TLS was introduced to Fortran.

3.5.1 Thread Local Storage

The increasing interest in threads motivated developers to create ways to manage private data held by threads. The POSIX standard defines the interfaces *pthread_getspecific* and *pthread_setspecific* for this purpose, but their usage is very complex. Recently, the C and C++ languages were extended with the specifier *__thread*, which simplifies the management of private data (DREPPER, 2003). Variables declared with *__thread* are automatically allocated local to each kernel thread.

This new mechanism is known as Thread Local Storage (TLS). In order to make it possible, compilers, linkers and thread library have to cooperate. The compilers must issue references to private data through a level of indirection. The linker has to initialize special sections of the executable that hold thread-local variables. Furthermore, the thread library must allocate new thread-local data segments for new threads during execution.

In addition to that, a new data structure was designed to keep track of private data. Figure 3.2 presents how this data structure is organized. In this example, there are three private variables, *x*, *y* and *z*. TP is a pointer to the current thread's data and DTV means Dynamic Thread Vector.

There are different modes of access to private data in the TLS and these modes are supported by the TLS data structure. This is basically necessary (1) to guarantee efficient access to the private data held by the main executable and (2) to allow private data in dynamically loaded libraries. In Figure 3.2 these modes are represented by the different tones of gray.

The light gray area corresponds to private variables that are in the main executable.

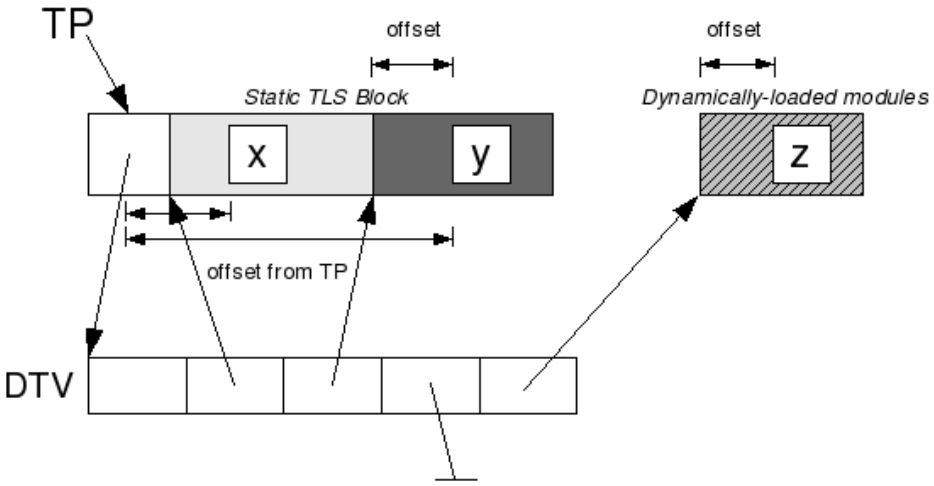


Figure 3.2: TLS structure.

The dark gray area holds private variables declared in dynamic libraries loaded before the main executable starts running. These two areas are stored in a contiguous region of memory known as Static TLS Block (OLIVA; ARAÚJO, 2006). Lastly, the hatched area corresponds to private data in dynamic libraries loaded after the main executable starts, for example by means of the *dlopen* function.

The private variables in the main executable can be accessed through offsets from TP, therefore the only overhead is the additional level of indirection. Conversely, variables loaded after the main executable starts (hatched area) can only be accessed through the DTV vector. Similarly, private data of libraries loaded before the main executable starts are also accessed by means of DTV, because a library does not know in advance whether it will be loaded before or after the main executable begins running.

3.5.2 TLS for user threads

The support to TLS in user threads is basically enabled by (1) the creation of the TLS structure during thread initialization and (2) the change of pointer TP during a context switch. Kernel threads implement steps (1) and (2). However, since we are using migration to enable load balancing, we cannot employ the default initialization routines. That is because the memory allocated to the TLS structure must have some properties in order to migrate. Therefore, the structure described in the previous section must be allocated appropriately in our implementation. In addition, there are some performance issues related to the form that some architectures set the TP pointer.

The allocation and initialization of the TLS structure are done by the dynamic linker. For example, the dynamic linker supplied by *glibc* comes with the routine *_dl_allocate_tls*, which allocates and initializes the TLS Static Block and the DTV. We cannot use this routine, because the allocated memory is not suitable for migration. Copying the allocated

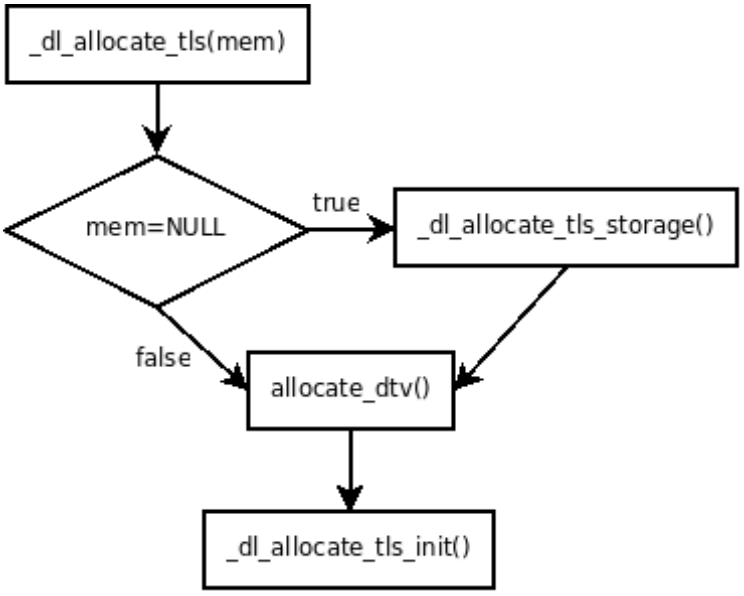


Figure 3.3: Call graph of the TLS structure initialization.

memory from one processor to another is not enough because the addresses would change and that could break existing pointers in the application. Since migration is central for load balancing, new routines must be created that use allocation routines that preserve addresses.

In our implementation, we used the *isomalloc* memory allocation routine as described by (HUANG; LAWLOR; KALÉ, 2003) to preserve the addresses during migration. We rewrote the TLS initialization routines of *glibc*. Figure 3.3 represents the call graph of the TLS structure initialization. The functions *_dl_allocate_tls_storage* and *_allocate_dtv* are the only ones which actually allocate memory. Thus, we substitute the regular allocation calls for *isomalloc* ones. We still call the function *_dl_allocate_tls_init* since it does not allocate memory but just initializes the memory allocated by the previous routines. This function receives as input the TLS structure uninitialized but with memory properly allocated. As a result, since there is no conceptual difference between the usual memory and the memory allocated by isomalloc, *_dl_allocate_tls_init* does not notice any change.

The performance issue we mentioned is related to the architecture we used. Our prototype was implemented in an x86 architecture due to its wide availability (however, all the other architectures that support TLS can be used in a similar way). The x86 architecture is short of registers, therefore the TP pointer is not implemented with a regular register, like in other architectures, but with the segment register `gs` (or `fs` on x86-64). Moreover, there are two alternatives to use this register, known as *direct access* and *indirect access*. To illustrate what that means, we use a small code example which has a global variable in the main executable:

```
__thread int i;
```

```
int main() {
  i = 42;
}
```

In the direct access, the attribution in this code is compiled as:

```
mov $42, %gs:-0x4
```

in which the offset to the variable *i* is `-0x4`. This alternative relies on Global Descriptor Table (GDT) or Local Descriptor Table (LDT) to select a segment where private variables are located. In the indirect access, the same attribution is translated as:

```
mov %gs:0x0 , %eax
mov $42 , -0x4(%eax)
```

In this alternative the TP pointer is stored in the first address of the segment pointed by `gs`.

We decided to use indirect access, because that makes the thread context switch lighter. The reason for this fact is that the GDT and LDT are managed by the Operating System (OS) and, therefore, a change to them would require an OS intervention. On the other hand, the indirect access requires only one write to the address `%gs:0x0`. Besides, the translation as two instructions, like that in the example, will usually happen in a few occasions, because the compiler usually caches the address of the TLS variables. However, this scheme implies that all code is compiled with indirect access. The user cannot mix different types of access in the same application.

3.5.3 TLS for virtual processors

We applied our new developed approach to the AMPI environment. As said before, this environment relies on the concept of virtual processors to enable load balancing. The virtual processors are implemented as user-level threads and AMPI provides routines to pack and migrate these threads.

AMPI is built on top of the Charm++ infrastructure. The threads of AMPI are based on TCharm, a thread library that provides a common environment to multiple parallel programming frameworks. In turn, Charm++ is implemented on top of Converse, a low level run-time system that presents an uniform parallel programing view of the underlying machine. We decided to implement our own strategy on Converse. We did that for

	default	*-allprivate*
no attribute	shared	private
__thread	private	-
__shared	-	shared

Table 3.2: New Fortran attributes and compiler option.

two reasons: (1) it makes the implementation simpler since it is implemented in plain C language and not in Charm++, and (2) this implementation can occasionally benefit other languages built on top of the Converse run-time system.

During thread creation, our new Converse thread library allocates the TLS structure using the isomalloc routine. Since the static TLS block may require alignment, we adapted the isomalloc to align data too. In addition, we enhanced the data structure that represents thread in the Converse run-time system so that TP, size of the blocks and alignment are stored there. Finally, we included the entire TLS in the process of packing that is done when a thread migrates. In this way, our TLS structure is transported along with other data such as stack and heap.

3.5.4 TLS support for Fortran

Fortran is arguably one of the most adequate languages for HPC (LOH, 2010). Many scientific programs are written in this language and it has evolved to support many advanced features that are found in more modern programming languages. Still, this language does not support TLS in the same way as C does. Therefore, we had to adapt a Fortran compiler so that it could produce appropriate code for our strategy.

GFortran was chosen for our experiments, because it is a high quality and open source compiler. We had to augment the syntax of the Fortran language to include the attribute *__thread*; in this way, the compiler behaves like the C compiler. This new attribute is included into the syntactic analyzer so that in a latter phase the appropriate code can be generated. In addition, we included a new compiler option (*-allprivate*) that makes all global, static and common variables to be stored in the thread local storage area. Doing so makes porting existing applications to a virtualized environment easy, since the user does not have to manually include the attribute *__thread* in his/her old application. On the other hand, we included a new attribute called *__shared* in case the user wants to make some variables shared. This new attribute is useful for controlling threads of a certain process, for performance measurements for example (we used that in some of our experiments presented in Chapter 4). Table 3.2 summarizes the new attributes and option.

Finally, the compiler back-end had to be changed in order to generate code with the proper form of access to the variables. Given that the user included the new attribute on certain variables (by using *__thread* or-*allprivate*) the compiler back-end generates code

with the level of indirection as described in Section 3.5.2. That was simpler to perform on GFortran, since its back-end (which is the same as in GCC) already had routines to generate this type of code. That is because the OpenMP *threadprivate* specifier is implemented by TLS.

3.6 Model for Ideal Balancing

Once we had an application enabled to use processor virtualization, the balancing policy received our attention. The problem of balancing N communicating threads among M processors can be modeled by Mixed Integer Quadratic Programming (MIQP). There are two objectives: (1) minimize the imbalance among processors and (2) minimize communication between any two processors. The second objective is necessary in order to guarantee that threads that communicate frequently are mapped close to each other and therefore the communication cost is reduced.

The MIQP model is the following:

$$minimize \qquad f : \sum_{i=0}^{M-1} \left[\left(\sum_{j=0}^{N-1} w_j x_{ij} \right) - W_{mean} \right]^2 \tag{3.1}$$

in which w_j is the weight of thread j and W_{mean} is the average load. The variables x_{ij} are binary and represent the placement of thread j on processor i. This objective penalizes processors that have load above and below average. The second objective function is:

$$\begin{aligned} minimize \quad g : \sum_{k=0}^{M-1} \sum_{l=k+1}^{M-1} D x_{ka} x_{lb} + \sum_{k=0}^{M-1} S x_{ka} x_{kb}, \\ \forall\, a, b \;/\; a \text{ communicates with } b \end{aligned} \tag{3.2}$$

where D represents the communication cost when two threads that communicate with each other are placed in different processors, while S represents the cost when these threads are placed in the same processor. Again, x is a binary variable. This function penalizes communicating threads that are placed in separate processors. The constraint of this model is:

$$\sum_{i=0}^{M-1} x_{ij} = 1, \forall\, j \tag{3.3}$$

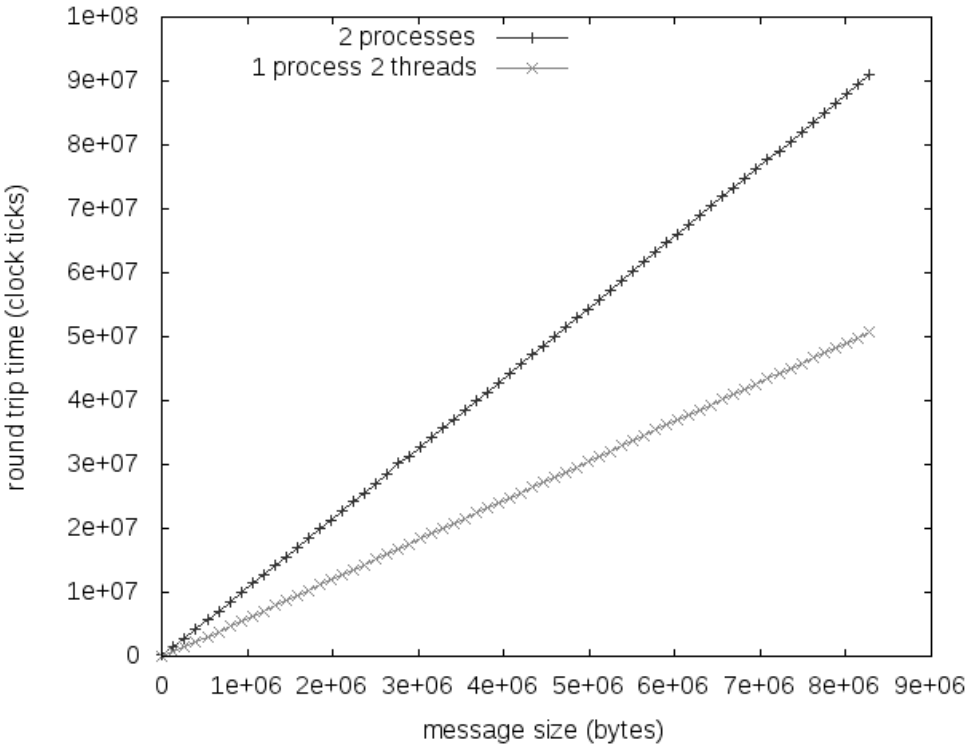


Figure 3.4: Communication speed comparison.

The problem expressed by the first objective function is a generalization of the multiprocessor scheduling problem (FOX; WILLIAMS; MESSINA, 1994) and is known to be NP-complete.

Solving this model to optimality can take a very long time on current machines, even for small cases. Indeed, solving the case $\{M = 4, N = 16\}$ takes several minutes with the state of art solver (CPLEX). Consequently, heuristics must be used to deal with realistic cases and obtain a result in a feasible amount of time.

3.7 Communication pattern

Before presenting the used heuristics in this thesis, we discuss communication in the context of a virtualized environment. An important observation about processor virtualization is that threads in the same processor communicate much faster than threads that are placed in different processors; a comparison is shown in Figure 3.4. Applications with a stable communication pattern may take advantage of this characteristic. Weather and climate models are examples of applications with this type of communication, as it can be seen in Figure 3.5 for the Brams model (WRF also has a similar behavior (BHATELE, 2010)).

In order to evaluate the benefits of placing threads that communicate more frequently

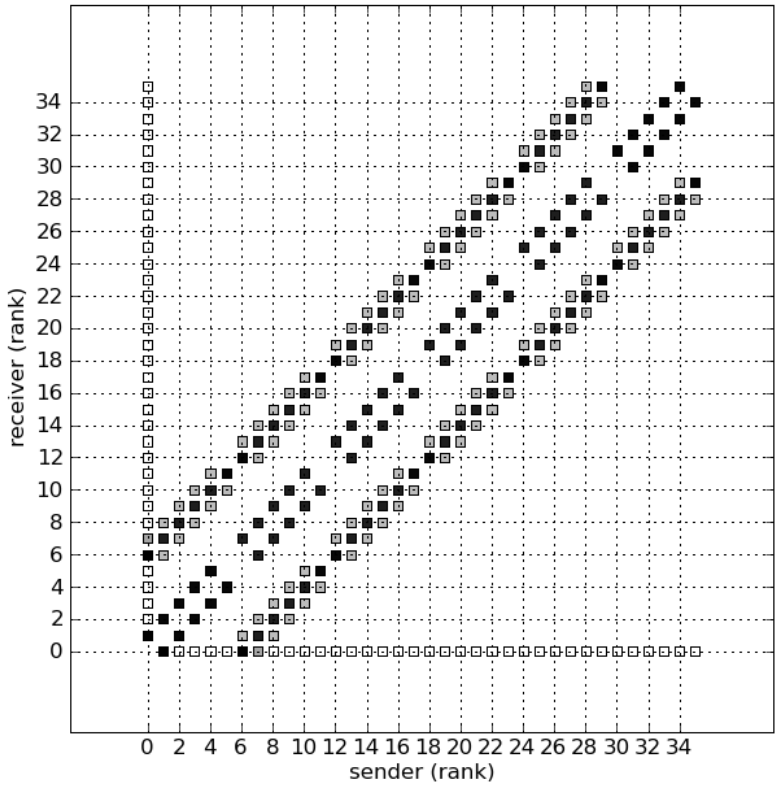


Figure 3.5: Communication pattern of Brams in an execution with 36 processes (darker tones represent larger amounts of communication).

closer together, we run an experiment that maps neighbor sub-domains of the Brams model to the same processor. We measured the amount of communication in a first execution, and used a technique of graph mapping to distribute the threads to the available processors in a second execution.

The performance improvement was almost 9%. That is similar to the results we obtained in a similar evaluation we performed, in which we used the same technique and the MPI with shared-memory intra-node communication in a cluster of multi-core machines (RODRIGUES et al., 2009).

Consequently, the load balancing strategies must consider the communication pattern so that the inter-processor communication is also minimized. Some of the load balancers presented in the next section, in principle, consider this aspect of the application.

3.8 Balancing Algorithms Employed

We investigated the use of various load balancers available in Charm++: *GreedyLB*, *RefineCommLB*, *RecBisectBfLB* and *MetisLB*. *GreedyLB* is a load balancer that has simplicity as its major feature; the thread with the heaviest computational load is assigned to

the least loaded processor, and this continues until a balance is reached. Hence, no communication information is considered, which can lead to a situation where two threads that communicate intensely are placed in distinct processors. However, given the simplicity of this policy, the balancing process is often very fast.

RefineCommLB is a balancer that takes both computational load and communication traffic into account. It attempts to move objects away from the most overloaded processors to reach average, but also considers how that movement would affect locality of communication. In addition, it limits the number of migrations, regardless of the observed loads. In general, this balancer is used for cases when moving just a few threads is sufficient to achieve balance.

The *RecBisectBfLB* balancer recursively partitions the communication graph of threads with a breadth-first enumeration; the partitioning is done based on the computational loads of the threads, until the number of partitions is equal to the number of processors. Although communication is considered by this scheme, there is no explicit guarantee that the resulting communication volume across partitions is minimized.

Meanwhile, *MetisLB* is a balancer that uses Metis (KARYPIS; KUMAR, 1995) to partition the thread communication graph. Both the computational load and communication pattern are considered. All of these Charm++ balancers employ a centralized approach, which works well for a moderate number of processors. However, as we show in the next chapter, none of these balancers fits well the two-dimensional spatial domain decomposition that is typically found in meteorological models.

In the weather model we used in our experiments (Brams), like in many other weather forecasting models, the atmosphere is represented with a three-dimensional grid of points. Those points are distributed across the MPI ranks according to a domain decomposition of the latitude/longitude plane. Each rank receives the full atmospheric columns corresponding to the points in its domain. Because the ranks are implemented by threads in AMPI, there is a high volume of communication between threads associated to ranks from sub-domains that are neighbors. Hence, mapping two threads from neighbor subdomains to the same physical processor will ensure that their communication is local to that processor, which minimizes the communication overhead.

3.9 New Load Balancer

We developed a new Charm++ balancer based on a space-filling curve. We place the various threads with their loads into a two-dimensional Hilbert space-filling curve (HILBERT, 1891), and then iteratively cut that curve until the number of segments is equal to the number of processors. Because the Hilbert curve preserves spatial locality, threads corresponding to sub-domains that are close in space are likely to be assigned to the same processor. This implies that a significant amount of communication between threads will be local to the same processor, which benefits application performance. Fig-

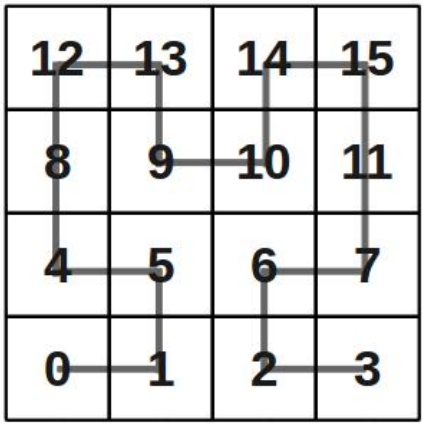


Figure 3.6: Hilbert curve for the case of 16 threads

ure 3.6 shows the Hilbert curve for a $4X4$ domain decomposition.

Firstly, the new load balancer has to compute the mapping between the Hilbert sequence and the 2D domain decomposition. An efficient coding scheme described in (LIU; SCHRACK, 1996) was used as a subroutine of our load balancer. For a domain decomposition of size $2^r \times 2^r$, the sub-domain at location (x, y), $((x_{r-1}...x_1x_0)_2, (y_{r-1}...y_1y_0)_2)$ in binary, can be encoded to the Hilbert sequence that is represented by a quaternary digit string $h = (q_{r-1}...q_1q_0)_4 = \sum_0^{r-1} 4^i q_i$ where $q_i \in \{0, 1, 2, 3\}$. Each quaternary digit h_k in h is represented by two bits h_{2k+1} and h_{2k}. These two bits are computed by the following recursive formulas:

$$\begin{aligned} h_{2k+1} &= \bar{v}_{0,k}(v_{1,k} \oplus x_k) + v_{0,k}(v_{1,k} \oplus \bar{y}_k) \\ h_{2k} &= x_k \oplus y_k \end{aligned}$$

where $k = 0, 1, ..., r-1$ and the values of $v_{0,k}$ and $v_{1,k}$ can be computed by:

$$\begin{aligned} v_{0,r-1} &= 0 \\ v_{1,r-1} &= 0 \\ v_{0,j-1} &= v_{0,j}(v_{1,j} \oplus \bar{x}_j) + \bar{v}_{0,j}(v_{1,j} \oplus \bar{y}_j) \\ v_{1,j-1} &= v_{1,j}(x_j \oplus y_j) + (\bar{x}_j \oplus y_j)(v_{0,j} \oplus \bar{y}_j) \end{aligned}$$

where $j = r-1, ..., 2, 1$.

To decode the Hilbert sequence h back to a sub-domain (x, y) the following formulas are used:

$$x_k = (v_{0,k}\bar{h}_{2k}) \oplus v_{1,k} \oplus h_{2k+1}$$

$$y_k = (v_{v0,k} + h_{2k}) \oplus v_{1,k} \oplus h_{2k+1}$$

where $v_{0,k}$ and $v_{1,k}$ are computed by:

$$\begin{aligned} v_{0,r-1} &= 0 \\ v_{1,r-1} &= 0 \\ v_{0,j-1} &= v_{0,j} \oplus h_{2j} \oplus \bar{h}_{2j+1} \\ v_{1,j-1} &= v_{1,j} \oplus (\bar{h}_{2j}\bar{h}_{2j+1}) \end{aligned}$$

According to Liu and Schrack (LIU; SCHRACK, 1996) this algorithm is $O(r)$. Since the dimension of the domain is $2^r \times 2^r$, the complexity with respect to the number of threads is $O(logN)$.

After computing the Hilbert sequence, the load balancer labels the sequence of threads with their loads. Afterwards, the sequence is cut in $numPEs$ (total number of processor) segments of similar load. This routine is described in Algorithm 1.

```
input  : float load[N]
         int numPEs
output: int cutVec[N]

prefixSum[0] = load[0];
for (i = 1; i < N; i++) do
    prefixSum[i] = prefixSum[i-1] + load[i];
    cutVec[i] = False;
end

idealLoad = prefixSum[N-1] / numPEs;
for (i = 1, j = 0; i < numPEs; i++) do
    inner:  for (; j < N; j++) do
        if (idealLoad * i - prefixSum[j+1])^2 > (idealLoad * i - prefixSum[j])^2 then
            cutVec[j + 1] = True;
            break inner;
        end
    end
end
```

Algorithm 1: Cut algorithm.

Algorithm 1 receives as input the load index (*load[N]*) of each one of the N threads and the total number of processors (*numPEs*). It computes the prefix sum of the load, that

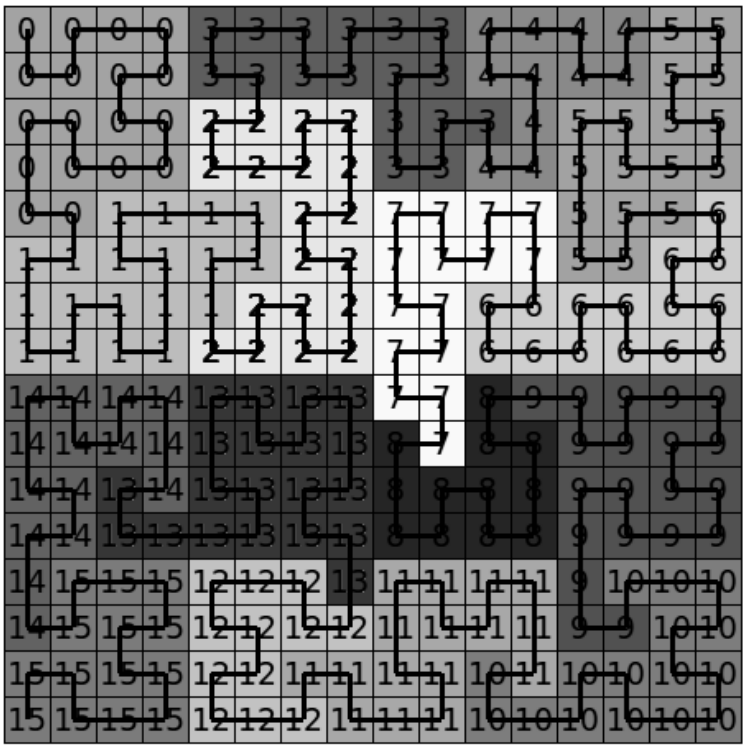
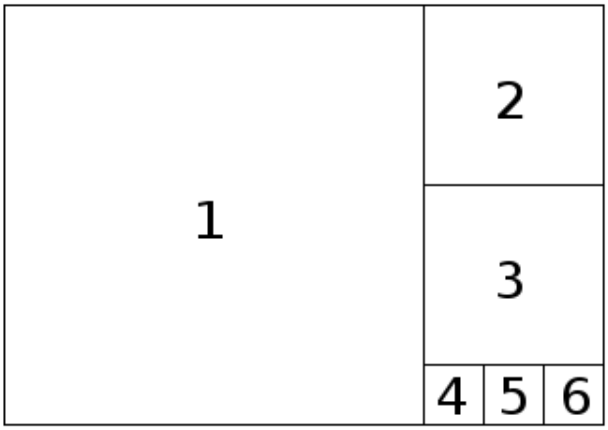


Figure 3.7: 16 processes and 256 threads after rebalancing.

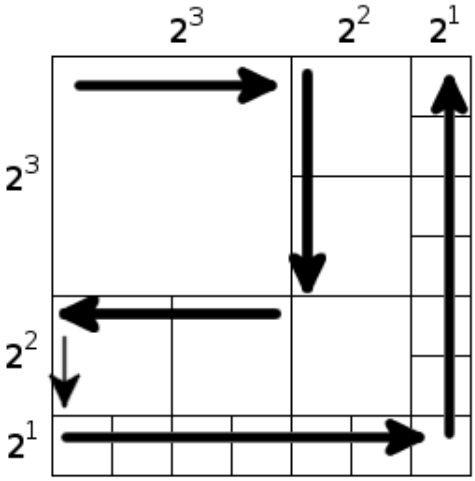
is used to find the best cut of the Hilbert sequence, i.e. which is closest to the ideal load balancing. The output is the vector $cutVec$ that marks the places on the Hilbert sequence that delimit each segment. These segments are assigned sequentially to the available processors. Figure 3.7 shows an example of a possible assignment. As it can be seen, the threads in each segment are close together, therefore, the external communication is reduced.

Despite being simple, the load balancer based on the Hilbert curve was very effective, as demonstrated by the performance shown in the next chapter. However, this original algorithm has a quite strong restriction concerning domain geometry: the domain must be a square and its side must be a power of two. Chung, Huang and Liu (CHUNG; HUANG; LIU, 2007) proposed an algorithm to overcome this limitation, allowing the use of rectangular domains of any size. Those authors used that algorithm in image processing. Here, we implemented the same algorithm to perform load balancing.

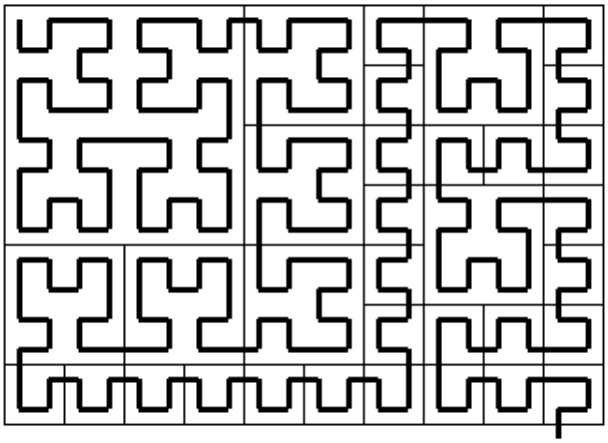
The algorithm starts by finding the largest square inside the original domain. This square is placed at the upper left corner of the domain. This step is applied recursively to the remaining area of the original rectangle, as illustrated in Figure 3.8a. Each square of the previous step is further decomposed into smaller squares whose side is a power of two. In order to do that, a "snake-scan" approach is used, as shown in Figure 3.8b. Finally, each smaller square is filled with the regular Hilbert curve following the direction used in the previous step (Figure 3.8c).



(a) Divide the original domain into squares



(b) For each square, apply the “snake-scan” approach



(c) Use the conventional Hilbert algorithm

Figure 3.8: Hilbert curve for domains of arbitrary size

3.10 Adaptive strategy

The centralized load balancing strategy we developed can be divided in four separate tasks. The first one is to gather load information at the processor that takes migration

decisions. This task may represent a bottleneck for large systems, since every processor must send its load to a single target. However, according to Zheng (ZHENG, 2005), centralized load balancers can achieve good performance in modestly large systems. Still, in the next section, we discuss a way to decentralize our strategy so that this gather is eliminated.

The second task in our load balancer is the execution of the algorithm described in the previous section, i.e. the Hilbert curve algorithm. This algorithm is $O(N)$, since cutting the curve is proportional to the number of threads, and computing the Hilbert sequence needs to be done only once, because it does not change in a single execution. Moreover, the execution time of this algorithm is small even for large systems, consequently we can call it very frequently, however, the algorithm depends on the load information, which is more expensive to obtain.

We are using past load as an indication of future load. This means that we must perform the first two tasks of our load balancer as frequently as possible. This is because if we let the weather model continue running for a long period, the load may "move" across sub-domains undermining the load index. In an extreme case, we can call these two tasks every time-step. However, since the cost of gathering load information may not be neglectable, the invocation should be less frequent.

A possible approach to adjust the frequency of the load balancer invocation starts with a comparison between the load estimation and the actual load of each processor. Since we are using load history as a predictor for future load, what we actually compare is the load at the current iteration of the load balancer and the load of the previous iteration. While the difference of these two values is smaller than a predefined ϵ for any processor, the frequency of invocation is allowed to decrease. If the difference increases beyond ϵ, then the frequency has to increase as well. The rationale of this approach is that the load estimation must be performed in a frequency that does not cause much overhead, but captures the movement of load from one processor to another.

A second approach takes advantage of the application behaviour. In some applications, all tasks exchange information with a master task at every time-step. For example, the Courant-Friedrichs-Lewy (CFL) condition must be met for each sub-domain while solving certain partial differential equations (HOFFMAN, 2001). Therefore, the CFL number must be sent to a master task which may adjust the time-step in order to guarantee numerical stability. The load balancer strategy can take advantage of this inherent communication to send the load index to the processor responsible for the load balancing decisions. In this way, the impact of the gather is reduced to the time it takes to send a few extra bytes in a communication that would exist anyway. As we will see, weather forecast models have this behaviour; specifically, they typically exchange CFL numbers with a master processor. Therefore, we decided to use this approach since it simplifies the load balancing implementation. However, a more generic application would require the first approach.

The third task corresponds to a scatter. The migration decisions must be sent to the corresponding processors. This task may also represent a bottleneck, since a single processor has to send a specific information to each of the other processors. Again, the fully distributed approach described in the next section is intended to deal with this issue in large machines.

The fourth task in our load balancer refers to the actual migration of threads. Migrating threads has an associated cost; the thread image must be copied to the target processor using the interconnect network. For applications with a large memory footprint, this cost can be pretty large. Hence, the load balancer must take into account how long it takes to migrate threads so that the rebalance process actually does not hurt performance.

There are two mechanisms to control the cost-benefit relationship with respect to the migrations: (1) establishing a load imbalance threshold beyond which migration will occur, and (2) establishing the number of threads that can migrate to rebalance load. The threshold is meant to prevent that small imbalances trigger migrations that will cost more than the benefit they would produce. Meanwhile, migrations may always be triggered, but the number of threads that actually moves is kept bellow a certain number, so that the overhead is smaller than the benefit obtained. Although this last approach allows a finer way to rebalance load, it may incur in more overhead due to network latency. That is because at every load balancer invocation only a few threads are allowed to migrate. On the other hand, the mechanism based on a threshold does not migrate threads at every invocation but, when migrations do occur, they are done in larger groups of threads. We decided to use the first mechanism.

Our approach to control the mechanism based on a threshold will be based on measurements taken during execution. The CPU time is used as load index, i.e. the longer the CPU time of a thread, the more loaded this particular thread is. We assume the principle of persistence, that is, the recent load is a good predictor for future load. In the particular case of meteorological models, this principle holds, because sources of load imbalance (like rain) does not move abruptly. In addition to the load index, we need a way to estimate how long does a certain number of threads take to migrate. In order to do that we measure the memory footprint of each individual thread. Furthermore, for each load balancing invocation, we use the simple model presented in the following equation (FOSTER, 1995) to estimate the migration overhead.

$$T_{msg} = T_s + t_w L \tag{3.4}$$

where T_{msg} is the time a certain number of threads takes to migrate, T_s is the startup cost, t_w is the cost of sending a single word and L is the memory footprint of the migrating threads.

In order to choose an inbalance threshold, we developed a strategy that stores the loads of all threads for a certain number of time-steps. This information is then used to evaluate how the performance would be if the load balancer had applied different thresholds. We use the estimate migration overhead and the new load distribution to choose a new threshold. We assume again that the principle of persistence will hold and for the next few time-steps this new threshold is used.

The next chapter has some experiments in which a threshold is manually established so that only when the imbalance reaches that threshold the load balancer actually migrates threads. In addition the period of invocations is also manually chosen. For these experiments, we did not take advantage of the preexisting communication to send the load index. We also present in the next chapter, the experiments in which the automatic threshold selection strategy is used.

3.11 Fully distributed strategies

This section describes two fully distributed load balancers that we developed and evaluated in this thesis. The first one is an extension of the load balancer that we presented in Section 3.9. The second strategy is based on the principle of diffusion.

3.11.1 Hilbert-curve-based load balancer

A central entity that receives all load information and distributes migration decisions is a bottleneck of the strategy described in the previous sections. For moderately large machines, this is not a problem, because the load balancer may not be called very frequently. However, with many thousands of threads and a higher invocation frequency, the load balancer may represent a very large overhead. Fortunately, the approach we developed can be fully distributed and, therefore, this bottleneck can be eliminated.

The first step of the distributed load balancer is to compute the Hilbert sequence. Liu and Schrack's algorithm (described in Section 3.9) can be used, since each thread can encode and decode the Hilbert sequence independently. All threads need to use this algorithm only once, because this process is necessary only at the beginning of the execution. Moreover, each thread does not need to compute the whole sequence, but only its position on the sequence and the position of those threads that this thread communicates with.

The next step is to compute the prefix sum of the loads of all threads. The prefix sum is an operation that takes as input a list and produces a result list in which each element is obtained from the sum of the elements in the operand list up to its index. The recursive doubling algorithm can be used to perform this list operation in $log_2(N)$ steps (where N is the number of threads) (JÁJÁ, 1992). This algorithm is illustrated in Figure 3.9. At the end of it, each thread will have its corresponding element of the result list.

The third step is a broadcast. The last thread (thread $N - 1$) has to send the total load to all the others; this thread has this information as a result of the previous step. This

operation can also be performed in $log_2(N)$ operations. This step is needed so that each thread can compute the ideal load, which is given by the total load divided by the number of processors.

The final step is to execute a routine corresponding to the Algorithm 2. This routine can be performed by each thread independently, because a thread needs only its own prefix sum element, its load, the total load and the number of processors. The result is the processor (*DestPE*) to where the thread must migrate.

With this distributed algorithm, the load balancer can scale to much larger machines than those used in this text. However, there is no meteorological model that scales to the level of parallelism in which this strategy is worthy - many thousands of processors (ZHENG, 2005). To the best of our knowledge, the largest meteorological model run is presented by Michalakesi *et al.* (MICHALAKES et al., 2007) with 65,536 processors, which includes only the dynamics portion (that does not suffer from load imbalance) of the WRF model. Therefore, we used a fixed load to test our distributed approach.

input : **float** myPrefixSum
float myLoad
float totalLoad
int numPEs
output: **int** destPE

idealLoad = totalLoad / numPEs ;
destPE = $\lfloor$ myPrefixSum / idealLoad $\rfloor$;
destPELeftNeighbor = $\lfloor$ (myPrefixSum - myLoad) / idealLoad $\rfloor$;
if *destPE* $\neq$ *destPELeftNeighbor* **then**
if *(idealLoad * destPE - myPrefixSum)*^{2} $\leq$
*(idealLoad * destPE - (myPrefixSum - myLoad))*^{2} **then**
destPE = destPE - 1;
end
end

Algorithm 2: Distributed cut algorithm.

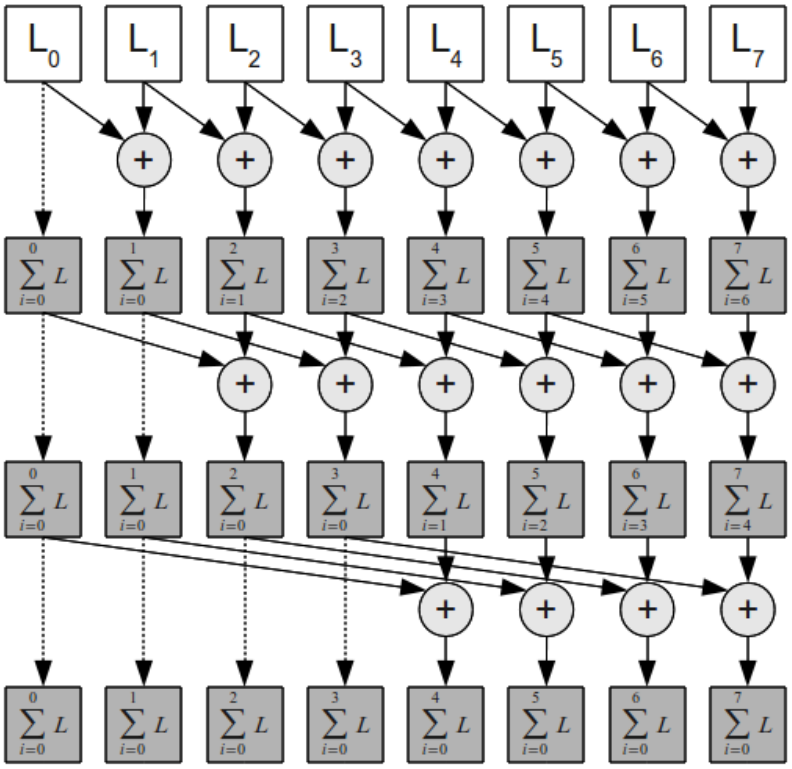


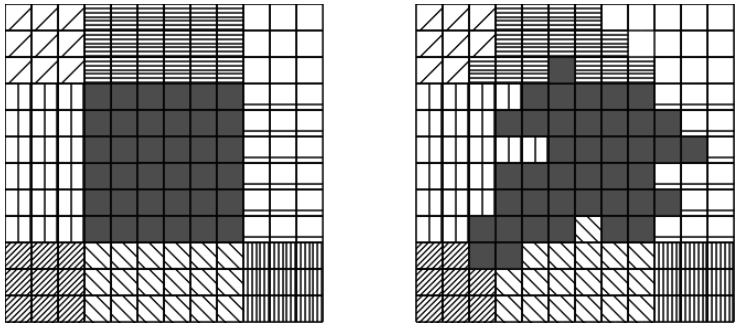
Figure 3.9: Parallel prefix sum.

3.11.2 Diffusion-based load balancer

A common distributed load balancing strategy is based on the principle of diffusion. This principle states that energy or matter flows from higher concentrations to lower concentrations, leading to an homogeneous distribution. The flow happens between contiguous regions, i.e. energy or matter flows from one region to another that is adjacent. This principle can be applied to load balancing, so that load moves from overloaded processors to underloaded ones in the same manner. In this strategy, the processors are only required to communicate with their neighbors. This amount of communication is smaller than that required by the Hilbert-based load balancer.

Figure 3.10a illustrates an initial thread distribution of a processor and its neighbors, while Figure 3.10b shows a possible configuration after some load balancing invocations. In this load balancer, threads migrate from one processor to a neighbor in order to equalize the load between them. For each individual invocation, the load is balanced locally, but, in the long run, the whole system tends to become balanced.

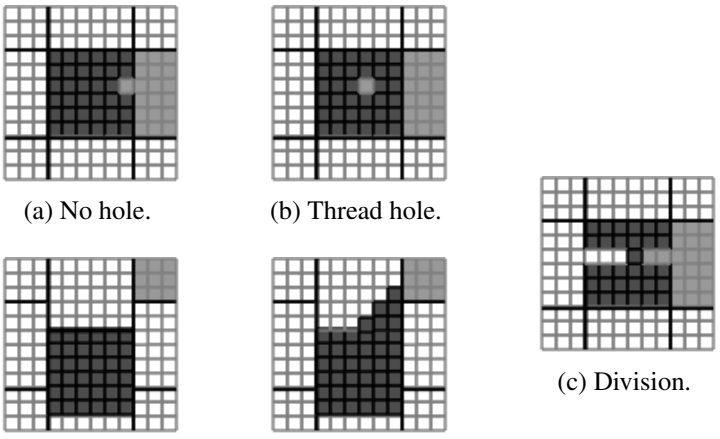
For this strategy to work efficiently, some issues must be properly handled. As we stated before, neighbor threads in weather models communicate frequently, because of the exchanging boundaries. Therefore, the load balancer must take into account this natural communication when it selects a thread to migrate. The diffusion load balancer has to keep track of the threads that directly communicate with neighbors processors. This is



(a) Initial state.

(b) State after a few migrations.

Figure 3.10: Diffusion-based load balancer.



(a) No hole.

(b) Thread hole.

(c) Division.

(d) Six neighbors.

(e) Seven neighbors.

Figure 3.11: DiffusionLB issues.

because they are the first candidates to migrate (Figure 3.11a). The objective is to avoid that holes appear in the set of threads of a processor, like that shown in Figure 3.11b.

A second issue is related with connectivity. Threads in each processor can be viewed as a directed task communication graph, where the regular communication determines the edges. The load balancer should avoid to break this graph apart, because the disconnected sub-graphs will not benefit from the local communication. One can achieve that by removing a candidate thread to migrate and running a depth-first traversal algorithm to the remaining threads. If the number of visited threads is equal to the number of remaining threads in that processor, then the candidate thread can migrate. Figure 3.11c has one thread that would not pass this test. Ideally, the thread graph in a single processor should be as connected as possible. In this way, the external communication is minimized.

A third issue with the diffusion-based strategy is that neighbors can move in and move out to the vicinity of a processor. Figures 3.11d and 3.11e show one example. The dark gray processor migrates some threads to the processor immediately above it. As a result,

the light gray processor gains a new neighbor. This event is hard to deal with, because the light gray processor does not know when a processor enters its vicinity. The incoming processor could send this information to its new neighbor, but the receiving processor does not know how long it has to wait for this message. One solution is to embed the neighbor information into the natural communication. We employed this scheme and the results are presented in Section 4.6.2.

4 EXPERIMENTAL RESULTS

This chapter presents our results. It starts with a description of Brams, the meteorological model we used as a case study. Following that, five sets of experiments are presented. The first one shows the performance of our privatization strategy, described in Section 3.5. The second set of experiments was intended to check the feasibility of the use of Processor Virtualization. An artificial thunderstorm that does not move was used. The remainder of the experiments uses real forecast data. The third set of experiments is divided in three groups. In the first group, we consider only the virtualization effect in the application used, namely overlap of communication and computation and better cache usage. The second group is a comparison among the load balancers used. We show that our approach produces the best results. In the third group, an adaptive scheme is evaluated. This scheme was implemented manually in a first moment. We developed an automatic mechanism for this scheme and evaluated its performance in a fourth set of experiments. Finally, we analyzed the performance of the fully distributed strategy in the last set of experiments.

4.1 Brams model

Brams (Brazilian developments on the Regional Atmospheric Modeling System, RAMS) is a multipurpose regional numerical prediction model designed to simulate atmospheric circulations at many scales. It is used both for production and research world wide. It has its roots on RAMS (WALKO et al., 2000), which solves the fully compressible non-hydrostatic equations described by Tripoli and Cotton (TRIPOLI; COTTON, 1982), and is equipped with a multiple grid nesting scheme that allows the model equations to be solved simultaneously on any number of two-way interacting computational meshes of increasing spatial resolution. It has a set of state-of-the-art physical parameterizations appropriate to simulate important physical processes such as surface-air exchanges, turbulence, convection, radiation and cloud microphysics.

Brams started as a research project aimed to tailor RAMS to the tropics and to modernize its software structure. Brams modeling features extended the original RAMS to include cumulus convection representation as part of an ensemble version of deep and

shallow cumulus scheme based on the mass flux approach (GRELL; DEVENYI, 2002), daily soil moisture initialization data (GEVAERD; FREITAS; LONGO, 2006) and a specific surface scheme that allows the representation of important tropical phenomena. More recently, a coupled aerosol and tracer transport model (CATT-Brams (FREITAS et al., 2009)) was developed to allow the study of emission, transport and deposition of gases and aerosols associated with biomass burning, such as those originated at the Amazon. CATT-Brams has been used in daily production mode at CPTEC to forecast air quality for the entire South America (see http://meioambiente.cptec.inpe.br/).

Brams uses Fortran 90 features to eliminate dusty deck software constructs from the original RAMS code, including static memory allocation and the heavy use of Fortran 77 commons, achieving production quality code while maintaining research flexibility. Brams is open source code freely available at http://brams.cptec.inpe.br/, supported and maintained by a modest software team at CPTEC that continuously transforms research contributions into production quality code to be incorporated at future code versions. It is also a platform for computer science research in themes such as grid computing (ELIANE et al., 2005; SOUTO et al., 2007).

This work uses the current research version (Brams 5.0) that has enhanced parallelism when compared to the current production version. Up to the current production version, Brams used the original RAMS master-slave parallelism that partitions the horizontal projection of the 3D domain into rectangles as close to squares as possible, assigning one rectangle to each slave process. The current research version eliminates the master-slave parallelism to avoid memory contention on the master process, using all processes on the original domain decomposition. Resulting code eliminated the master memory bottleneck, enhancing parallel scalability up to O(1000) processors (FAZENDA et al., 2009). Load balancing became the major scalability bottleneck, partially due to rectangular domain decomposition but mainly due to the dynamic load variation during integration.

4.2 First set of experiments: privatization strategy

In this section, we present some performance experiments using our privatization strategy, which was described in Section 3.5. We employed a micro-benchmark and the Brams model in these experiments.

Firstly, we compare the context switch time of our strategy (TLS) and the existing strategy (GOT, which is the existing privatization strategy of AMPI). This is an important metric, because whenever a virtual processor blocks, in *MPI_Recv* for example, a context switch happens. Therefore, in a communication bound application, a longer context switch may increase the total execution time substantially. In these experiments, we vary the total number of global data items so that it is possible to verify the impact of this factor to both strategies.

The results are shown in Figure 4.1. As it can be seen, the context switch of the

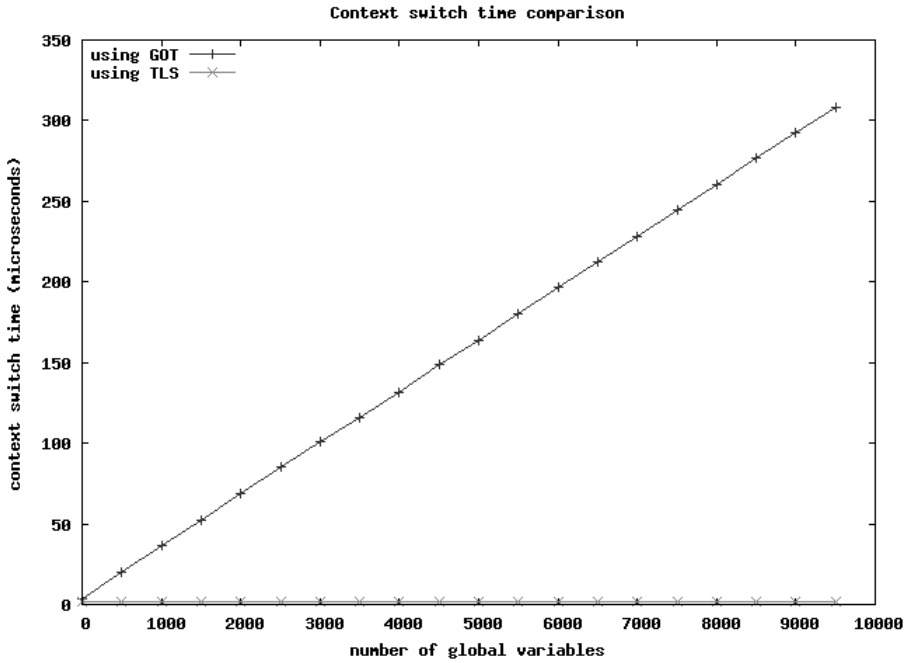


Figure 4.1: Context switch time comparison.

strategy based on GOT is proportional to the number of global variables. On the other hand, the performance of our strategy does not change as the number of global data items increases. This is because we do not have to change an entire table (the GOT table) as the original strategy does. Therefore, our context switch is more efficient. As shown in Table 3.1, meteorological models have many global and static data, therefore, the use of our privatization scheme is beneficial for this type of application.

We examine now the impact of our strategy on Brams. We ran a forecast over the southern region of Brazil. We employed a 40-level 64x64 grid and 2Km resolution. In this experiment, we ran the model on one processor and five different number of threads: 1, 2, 4, 8 and 16. We had to replace every static data with globals so that the original GOT-based solution could privatize them. The results are presented in Table 4.1

forecast length	1 thread	2 threads		4 threads		8 threads		16 threads	
		got	tls	got	tls	got	tls	got	tls
1h	509.35	497.71	491.09	505.50	476.11	554.15	474.36	661.72	490.14
6h	4241.10	4159.41	4047.24	4149.99	4005.09	4424.92	3967.77	5064.85	4054.84
12h	9148.00	8982.40	8860.85	9100.91	8689.95	9588.67	8719.21	10866.66	8831.14
24h	18583.50	18180.12	18033.24	18341.61	17631.42	19385.24	17654.10	21957.30	17880.50

Table 4.1: Execution time (seconds) of the Brams application.

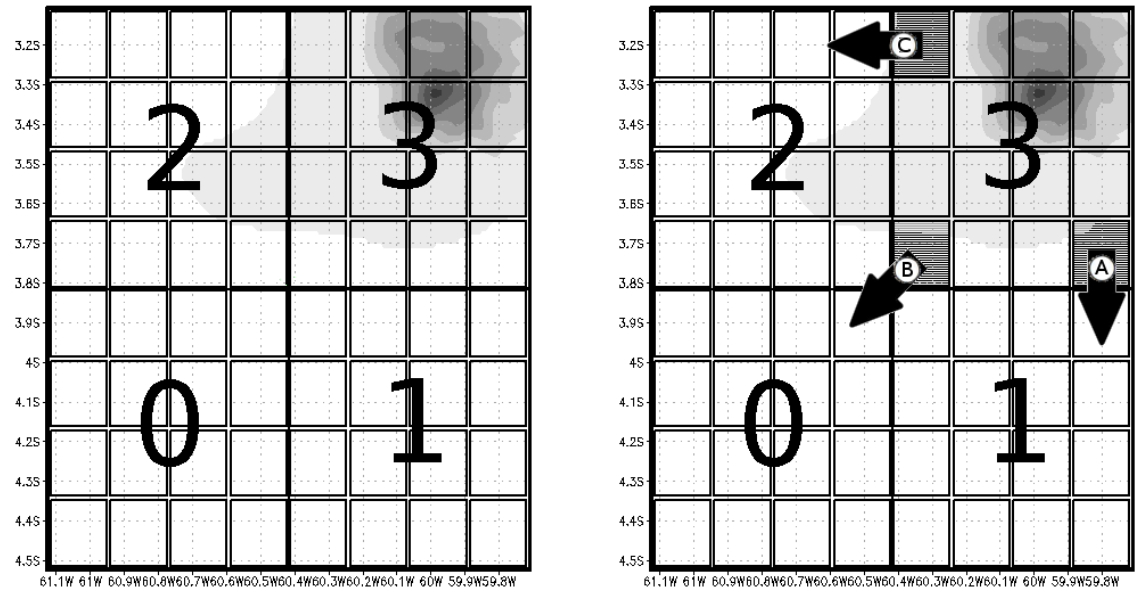
In these experiments, our privatization strategy reaches a maximum speedup of 25% over the default GOT strategy for 16 threads and 1 hour of execution. As the forecast length increases, this benefit decreases to approximately 18% with 16 threads. The reason for this fact is that the imbalance increases execution time and that hurts the improvements obtained with the over-decomposition (we will discuss the effects of over-decomposition on Section 4.4.1). That result points us again to the need for load balancing.

4.3 Second set of experiments: rebalancing granularity

The rebalancing granularity of our strategy is a single virtual processor, i.e. the minimum amount of load that can migrate to rebalance the application is one virtual processor. That may be too large for weather models, because of the large memory footprint; migrating a sub-domain associated to a virtual processor may take too long to be beneficial. In order to check that, we performed a forecast with the Brams model in which we used Processor Virtualization to rebalance load.

This experiment uses artificial atmospheric data so that there is a localized thunderstorm that does not move. Processors in charge of the thunderstorm have more load than the others. To rebalance load, we explicitly migrate some VPs from the over-loaded processors to underloaded ones. Then, we compare the execution time of this scenario with another in which the simulation is allowed to proceed without migrations.

The experiment is a 30-minute simulation of 128×128 points and 40 vertical levels. The input data is created so that a thunderstorm develops in the northeastern part of the



(a) Localized thunderstorm and domain decomposition

(b) Thread migration

Figure 4.2: Artificial thunderstorm

	execution time (s)	reduction	migration cost (s)
without load balancing	850.13	-	-
2 migrations (A and B)	780.65	8%	2.64
3 migrations (A, B and C)	747.55	12%	4.77

Table 4.2: Execution time of the artificial thunderstorm case.

domain. We used a Dell PowerEdge 1955, equipped with two 2.33Ghz quad core Xeon and 8Gb of DDR3 memory. In addition, we used four processors each of which ran 16 virtual processors. Figure 4.2a shows the precipitation and the decomposition.

In a first moment, we executed the model without any migration. In this way, Processor 3 is overloaded and delays the execution of the entire forecast. In a second experiment, we moved explicitly the threads with labels *A* and *B* in Figure 4.2b away from the overloaded processor. In a third experiment, three threads were moved from Processor 3, the threads *A*, *B* and *C*.

The execution times of these experiments are presented in Table 4.2. As it can be seen in this table, the cost to migrate these few threads is pretty low. And, indeed, the rebalance improves the performance of this case. Nonetheless, the interconnect network in this experiment is the bus that connects the processors in this shared memory machine and the atmospheric data is artificial. Thus, we performed other experiments in a distributed memory machine using real atmospheric data. Next sections describe these experiments in detail.

4.4 Third set of experiments: centralized load balancers

Our case study in this section is a forecast of a moving thunderstorm in the Southeast region of Brazil. We configured Brams to use a grid of 512×512 horizontal points and 40 vertical levels. The resolution was 1.6 Km and the timestep was 6 seconds. We conducted forecasts of 4 hours, corresponding to executions with 2,400 timesteps. These experiments were run on a Cray XT5 system at Oak Ridge National Lab., whose nodes have two six-core AMD Opteron processors at 2.6 GHz. The network connection is a SeaStar2+. We used 64 physical processors and up to 2048 virtual processors. To analyze in detail the executions, we used *Projections* (KALÉ et al., 2003), a performance analysis tool from the Charm++ infrastructure.

The experiments are divided in three parts. We start evaluating the impact of applying processor virtualization to Brams. We simply varied the virtualization ratio, without introducing any thread migration. Next, migration is used to balance the load across processors. The load balancers presented in the previous chapter are used; subsection 4.4.2

Configuration	Wall clock time (s)
No Virtualization	4970.59
256 virtual processors	3857.53
1024 virtual processors	3713.37
2048 virtual processors	4437.50

Table 4.3: Brams execution time on 64 processors

presents the execution times for each algorithm and an analysis of the reasons for these results. Finally, in subsection 4.4.3, we investigate how the frequency of balancing impacts performance of a selected algorithm. We also establish an imbalance threshold beyond which migration will occur.

4.4.1 Virtualization Effects

We started our experiments by analyzing the impact of using solely processor virtualization in Brams, i.e. we simply varied the number of virtual processors for executions on a fixed number of physical processors. As the number of virtual processors increases, some overhead is expected to occur, because over-decomposition also means that control code (e.g. ghost zone exchange) will increase as well. On the other hand, a virtualized execution can benefit from automatic overlap of computation and communication, even without explicit use of nonblocking MPI calls: when a certain virtual processor blocks on a receive, another virtual processor can execute. In addition, this approach allows for better cache use, because each sub-domain is smaller than it would be in a non-virtualized environment. Consequently, these smaller sub-domains can more easily fit in cache.

Using 64 physical processors, we conducted executions of Brams with AMPI employing, respectively, 64, 256, 1024 and 2048 virtual processors. These executions corresponded to virtualization ratios of 1, 4, 16 and 32, respectively. The mapping of virtual processors to physical processors was in a blocked fashion similar to what had been shown in Figure 3.1b.

Table 4.3 shows the results of these experiments. As it can be seen from this table, the execution time decreases 22.4% with 256 virtual processors. The decrease with 1024 virtual processors is slightly better: 25.3%. However, with 2048 virtual processors the reduction is only 10.7%. Therefore, there seems to exist a stagnation point beyond which adding more virtual processors does not improve performance.

Figure 4.3 compares the performance of two of those executions, one without virtualization and another with a virtualization ratio of 16. Both executions present peaks that occur with a period of 100 timesteps: these correspond to timesteps in which radiation modeling is active. For the virtualized execution, there is a wide “amplitude” in the observed duration of the timesteps; this occurs because those timing measurements are made

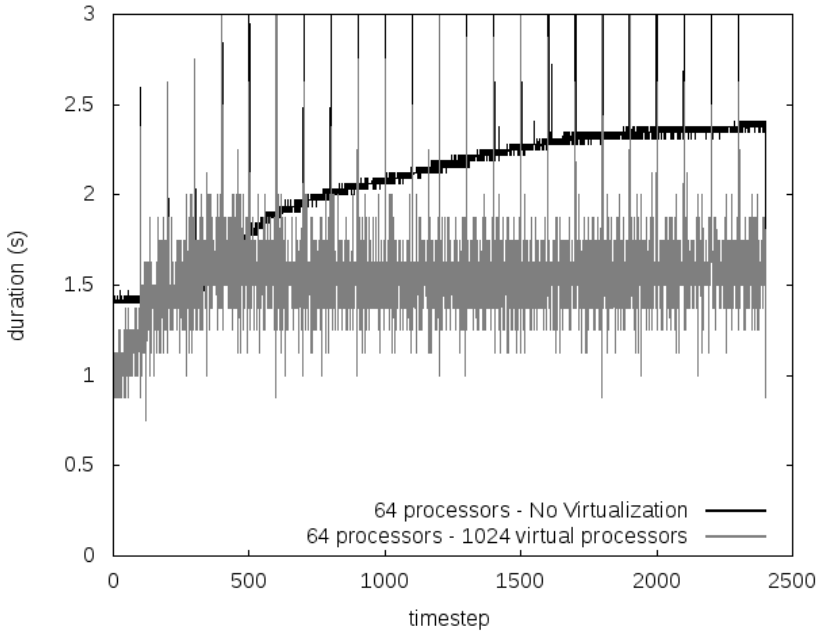


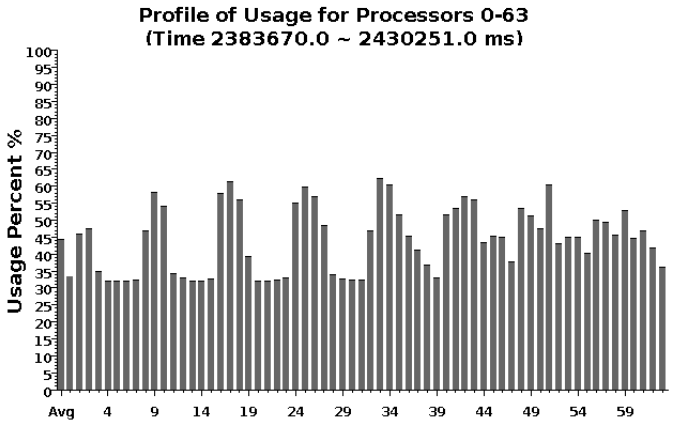
Figure 4.3: Effect of virtualization on Brams performance

on thread zero. In the virtualized execution, that thread shares the physical processor with fifteen other threads. The order of thread execution is determined by the Charm++ scheduler, and may vary across the simulation. Hence, thread zero's slot of execution fluctuates as the simulation progresses.

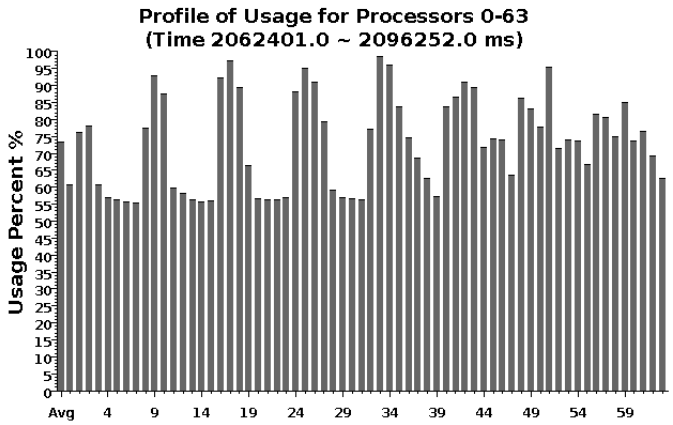
To determine the reasons for the performance improvements as we raise the number of virtual processors, we compared the cases of 256 and 1024 virtual processors, respectively, to the non-virtualized configuration. We enabled the automatic Charm++ instrumentation to capture detailed performance data during a section of the simulations (i.e. between timesteps 1250 and 1270), and analyzed the obtained data with the *Projections* performance analysis tool (KALÉ et al., 2003).

Figure 4.4 shows CPU usage for the various configurations. The bars represent the amount of CPU used during the measured period. There is one bar for each physical processor and the first bar is the average CPU usage. Without virtualization (Figure 4.4a), the average CPU usage was 44%. This CPU use is fairly low and could indicate that the sub-domains were too small. However, this experiment represents a typical size of a Brams simulation, where each sub-domain has $64 \times 64 = 4096$ columns of the atmosphere and the ghost-zone contains only 256 more columns. Furthermore, the experiments were run on a machine with a fast interconnection. Therefore this CPU usage is well representative of what an average user would experience with this application.

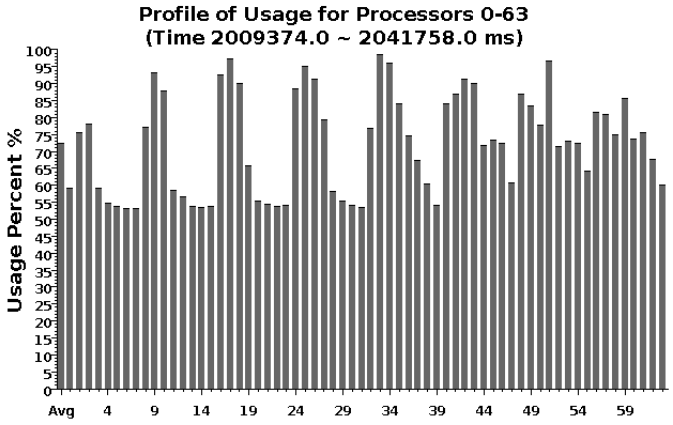
When we use four virtual processors per real processor (Figure 4.4b), the average CPU usage improves to 73%. The reason for this improvement is that the waiting time that each processor would experience is filled with computation of other virtual processors that are ready to execute. As we increase the virtualization degree further to sixteen virtual



(a) 64 processors - no virtualization



(b) 64 processors - 256 virtual processors



(c) 64 processors - 1024 virtual processors

Figure 4.4: CPU utilization for various virtualization ratios

processors, however, we do not see any additional improvement (Figure 4.4c). In this case, the average CPU usage is still 73%. This is because the bottleneck is no longer

Configuration	L2 cache misses	L3 cache misses
No Virtualization	12,416M	8,448M
256 virtual processors	10,560M	4,416M
1024 virtual processors	9,408M	3,904M
2048 virtual processors	13,696M	5,056M

Table 4.4: Total number of cache misses on 64 processors in Brams

idle time, but load imbalance (notice that some processors in Figure 4.4b were already near 100% utilization). Nevertheless, this higher degree of virtualization allows for more flexibility when using migration for load balancing. The average CPU usage for the case of 2048 virtual processors (not shown here) is also 73%. As we will show, another reason must exist to explain the performance loss for this case in Table 4.3.

We also analyzed cache utilization, by reading the hardware performance counters of the AMD processors. We used the Performance Application Programming Interface (PAPI) library (BROWNE et al., 2000) to access those counters. While using this library in a non-virtualized environment is trivial, one cannot use the same instrumentation in a virtualized execution, because the runtime system does not guarantee that the threads (virtual processors) are executed in an appropriate order. Therefore, we developed a scheme to ensure that the first thread entering the code section started the PAPI counters and the last thread leaving that section read those counters. Hence, our measured values account for the execution of all threads on a given processor. We used global variables forced to be shared among threads to control this scheme.

The performance gains of the cache should arise from the Brams code structure. In a time-step of Brams, similarly to other meteorological models, the various physical processes are called in sequence. Each of those processes performs its associated computation for the entire local sub-domain, hence the second physical process can benefit from the fact that the sub-domain is still in cache due to the computations of the first physical process. This effect repeats for the remaining physical processes within the same time-step. This performance gain is maximized when the local sub-domain matches the size of cache.

The measured amounts of cache misses for the period corresponding to the timesteps of interest in the application are presented in Table 4.4. A consistent decrease in cache misses can be seen in both L2 and L3 caches for the cases of 256 and 1024 virtual processors. That confirms the improvement in spatial locality that virtual processors allow.

The cache misses for the case of 2048 virtual processors, however, increased. The reason for this increase, and the corresponding increase in execution time observed in Table 4.3, is that the sub-domains for this case are too small and cannot benefit from all cache space available. Since context switch among threads occurs in this virtualized

Configuration	Execution Time (s)	Execution Time Reduction
No virtualization	4987.51	-
No load balancer - 1024 VP	3713.37	25.55%
GreedyLB - 1024 VP	3768.31	24.45%
RefineCommLB - 1024 VP	3714.92	25.52%
RecBisectBfLB - 1024 VP	4527.60	9.23%
MetisLB - 1024 VP	3393.12	31.97%
HilbertLB - 1024 VP	3366.99	32.50%

Table 4.5: Load balancing effects on Brams (all experiments were run on 64 real processors)

execution, the data of a sub-domain cannot stay in cache for a long period, because it has to give room for data from other virtual processors. The same fact happens with 256 and 1024 virtual processors, but in those cases more cached data is used between context switches. We confirmed this by running a new experiment with 1024 virtual processors but using only 32 physical processors. The resulting numbers of L2/L3 cache misses were 9,372M and 4,038M, respectively. Those numbers are very close to the original results with a virtualization ratio of 16, despite employing a new ratio of 32. Hence, we can conclude that the lower cache utilization measured in the last row of Table 4.4 is due to the smaller sub-domain size combined with the use of virtualization. In summary, there is a sweet spot in performance that is reached when the size of the sub-domain assigned to each virtual processor best matches the underlying cache sizes, in particular the size of the L3 cache, which accounts for the most expensive misses on the AMD Opteron.

4.4.2 Migration for Load Balancing

In this group of experiments, we executed Brams with a load-balancing invocation at the end of every forecast hour, corresponding to timesteps 600, 1200 and 1800, respectively. The same grid as in the previous experiment was employed. Table 4.5 presents the Brams execution time on these experiments and the corresponding execution time reduction in comparison to the case without virtualization.

The only load balancers that produced performance gains were *HilbertLB* and *MetisLB*. Although the other load balancers produced better performance in comparison to the non-virtualized case, they actually lost part of the gains obtained from over-decomposition, i.e. there was a reduction in performance when compared to the “No load balancer” case. There are three potential reasons for these results: (a) the cost of executing the balancing algorithm and the migration cost were excessive; (b) the load balancer was unable to rebalance load completely; and (c) the cross-processor communication increased after

Load Balancer	Balancing Time (s)
GreedyLB	80.81
RefineCommLB	10.81
RecBisectBfLB	78.33
MetisLB	81.00
HilbertLB	51.45

Table 4.6: Observed cost of load balancing

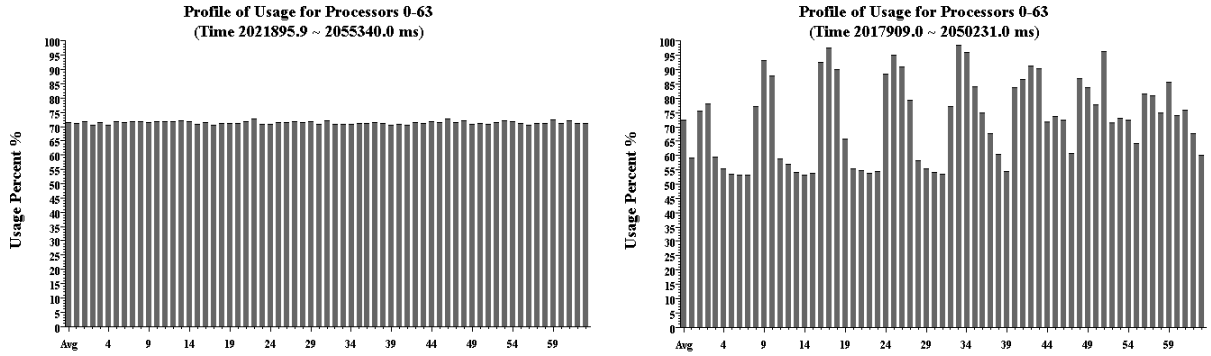
rebalancing.

To investigate the first reason, Table 4.6 presents the time each algorithm took to rebalance load. These values correspond to the sum of the timestep durations for the three timesteps where load balance occurred (i.e. timesteps 600, 1200 and 1800); they include both the execution of the balancing algorithm itself and the thread migrations. As it can be seen, there is not much difference among these values, except that *RefineCommLB* was much faster, as expected, since it limits the amount of migration. One of the most expensive algorithms, *MetisLB*, had one of the best application execution times. Therefore, another reason must exist to explain the poor application performance caused by the the first three load balancers in Table 4.6.

Let us consider the *GreedyLB* load balancer. This balancer was able to rebalance load quite well, as shown in Figure 4.5(a). This figure plots CPU usage of each physical processor and the first bar is the average CPU usage. The load is well balanced across all processors but the CPU usage is low, with an average near 70%. The reason for this fact is communication, as a CPU becomes idle when it waits for data from its neighbors. Since *GreedyLB* does not consider communication in its balancing decisions, the external (i.e. cross-processor) communication can increase as a consequence of rebalancing. We confirmed this by comparing the cross-processor communication in this experiment with the one from the "No load balancer" case. We found that the cross-processor communication volume increases by a factor of nearly five with *GreedyLB* (see Figure 4.6).

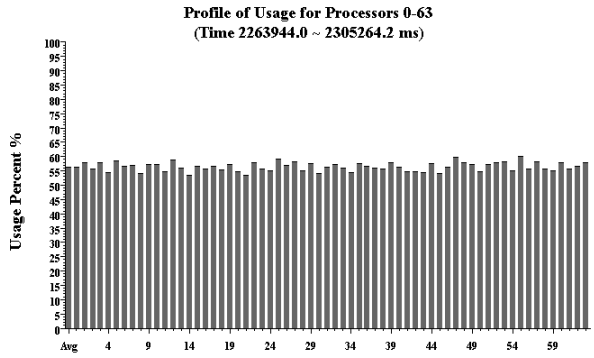
In turn, *RefineCommLB* kept much of the communication similar to the pattern in the original mapping. However, it did not migrate enough threads to fully rebalance load. Consequently, the load was still imbalanced after its use, as confirmed by the utilization plot of Figure 4.5(b). The problem with this load balancer is that it assumes the load is almost balanced and it will just perform a refinement (as its name implies). The imbalance of our experiment, in contrast, was much larger than what *RefineCommLB* could effectively handle.

With *RecBisectBfLB*, a good balance was achieved, as shown in Figure 4.5(c). However, the utilization was quite low, with an average near 55%. We noted that this load balancer caused some processors to have more external communication than others, simi-

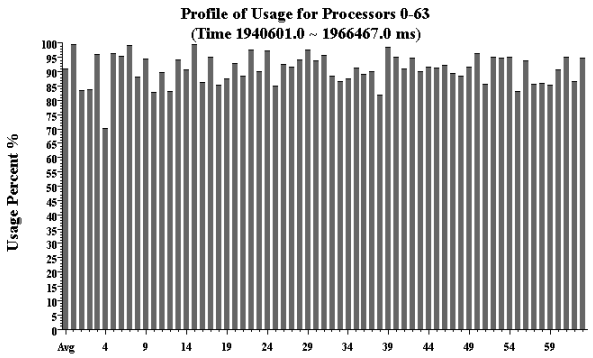


(a) GreedyLB

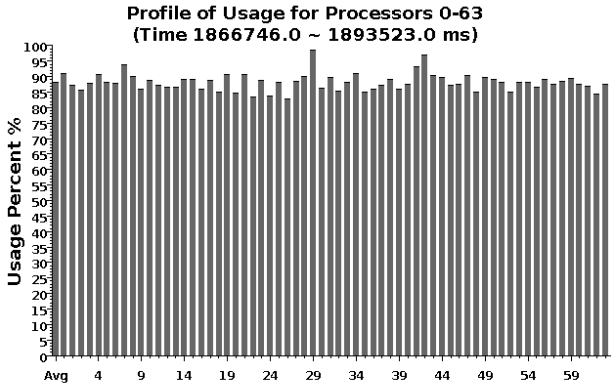
(b) RefineCommLB



(c) RecBisectBfLB



(d) MetisLB

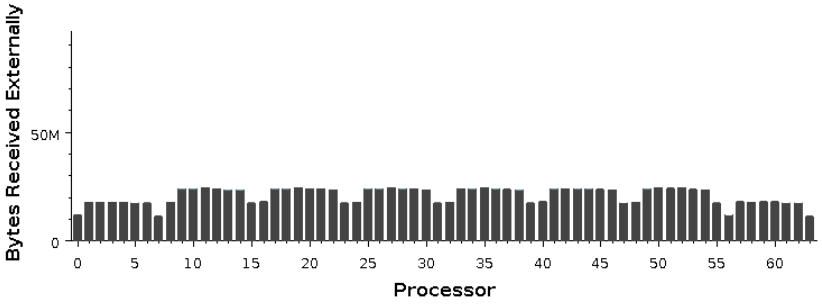


(e) HilbertLB

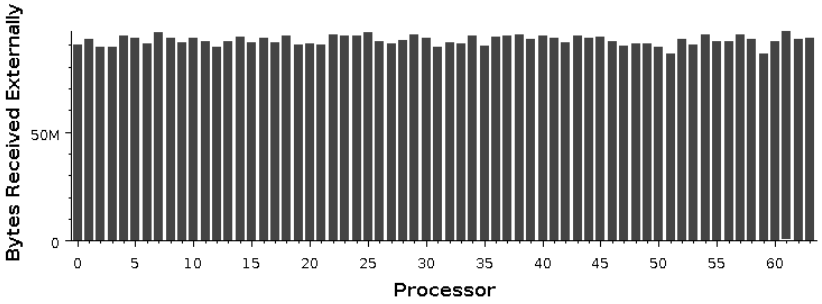
Figure 4.5: CPU usage under different load balancers

larly to what happend to the GreedyLB. Because the execution in Brams proceeds with an implicit synchronization caused by the exchange of boundary data between sub-domains at each timestep, delays in one processor slows down the entire execution. We conducted additional checks and confirmed that this was indeed the cause for the poor performance of *RecBisectBfLB* observed in Table 4.5.

Finally, for *MetisLB* and *HilbertLB*, which achieved the best performance in Table 4.5,



(a) No load balancer



(b) *GreedyLB* balancer

Figure 4.6: Cross-processor communication volume

LB Interval	**Imbalance Threshold**			
(Timesteps)	50%	20%	10%	5%
100	3639.54	3290.72	3211.10	-
10	3554.07	3179.31	3128.54	3245.03
1	-	3248.85	3872.11	-

Table 4.7: Brams execution time (in seconds) with adaptive load-balancing invocation and *HilbertLB* balancer

the balance was good and the average utilization was quite high; Figure 4.5(d) and Figure 4.5(e) show those details for *MetisLB* and *HilbertLB*.

4.4.3 Adaptive Balance Period

In the previous set of experiments, a fixed load-balancing invocation scheme was used and migrations would occur whenever there was imbalance, regardless of the amount of imbalance. Since the cost of executing the load balancing algorithm is usually low but the thread migration cost may be high, an adaptive balancing scheme can be more effective. In this new scheme, the load balancer is invoked more frequently and migrations occur only if the observed imbalance is beyond a given imbalance threshold.

In this subsection, we evaluate if the performance of Brams can be improved by using this adaptive scheme. We conducted executions invoking the load balancer every 100, 10

or 1 timestep(s), respectively. Also, we selected the following imbalance thresholds to trigger migrations: 50%, 20%, 10% and 5%. The imbalance corresponds to the difference between the load of the most loaded processor and the average load. Here, only the *HilberLB* load balancer was used. Table 4.7 presents the Brams execution time for these experiments.

It is possible to see that a high imbalance threshold hurts execution time. The threshold of 50% produced worst results than the fixed invocation scheme used previously. That is because this threshold was too high and did not trigger enough migrations to neutralize the imbalance. Decreasing the load balancer invocation interval in this case reduces execution time because the few occasions where the imbalance reaches 50% are detected sooner. However, a shorter invocation interval should not produce any benefit, because the load is highly unlikely to reach an imbalance of 50% or more in such a few timesteps after rebalancing.

An imbalance threshold of 20% improves performance even with a frequency of one invocation per timestep. The reason for this fact is that the cost of simply executing the *HilbertLB* algorithm without any migration is very low. Furthermore, a threshold of 20% is a good trade-off between imbalance and migration cost, i.e. migration (with its high cost) will not occur so frequently even if the balancing algorithm is being called at every timestep.

With a imbalance threshold of 10%, the performance reaches its best result, but only with an invocation interval of 10 timesteps. Lowering the invocation interval actually causes performance to be worse than in the case of fixed invocations (Table 4.5). This is because there are too many migrations and their cost surpasses the benefits from load balance. In addition, for these experiments the gathering of load indexes is carried out in a separate communication; in the next section, we coalesce this data with preexisting communication. A similar but milder effect occurs with the threshold of 5% and a balancing interval of 10 timesteps.

In summary, there is an optimal point with this adaptive scheme that is reached when the load balancer invocation frequency is high enough to detect the load variation. In addition, the imbalance threshold must be tuned according to that frequency, considering the typical imbalance that may arise within the invocation period. In our experiments, invoking the *HilbertLB* balancer every 10 timesteps and enabling migrations when the imbalance was higher than 10% resulted in a performance gain of 37.3% over the non-virtualized Brams execution reported in Table 4.5.

4.5 Fourth set of experiments: automatic imbalance threshold

In the previous experiments, the imbalance threshold had to be selected manually for a given forecast. As it can be seen from the results, there is potential for improvements. However, the user does not know in advance which threshold will produce the best perfor-

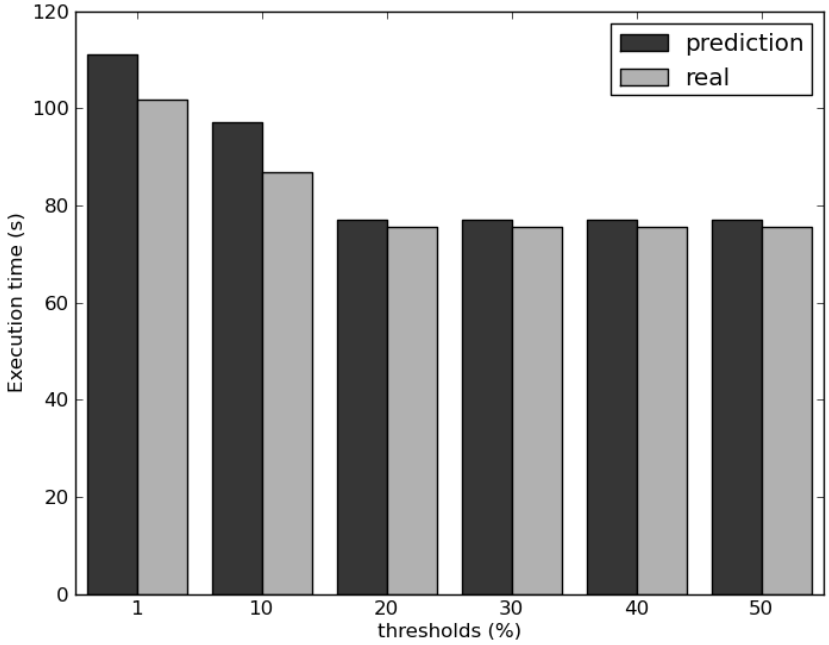


Figure 4.7: Comparison between prediction and actual execution.

mance. Trial and error is not a choice, even if the forecast is done routinely over the same region, because the dynamic sources of load imbalance may change from one execution to another. In this section, we describe the results of the strategy we developed to find a threshold automatically. In addition, we coalesced the communication of the load index with preexisting communications, as we described in Section 3.10, thus we can invoke the load balancer more frequently.

Our strategy for automatically choosing an imbalance threshold is based on the principle of persistence, i.e. we believe that the best threshold for the last few time-steps will be the best one for the next few time-steps. This hypothesis can be confirmed with an experiment that is described by the flowing steps: (1) we run the weather model for a certain number of time-steps, 50 time-steps in this case, and store the load indexes of this period; (2) for each threshold, we predict how long the execution time would be. This prediction takes into account the migration cost estimate and the execution time with the new assignment of threads to processors; (3) we continue running the model with each one of the thresholds (one different execution for each threshold) for other 50 time-steps. We perform rebalancing whenever the imbalance threshold is reached; (4) finally, the prediction and the actual execution time can be compared. Figure 4.7 shows that comparison for a forecast similar to the one presented in the previous section. As it can be seen, the prediction is quite accurate with the actual execution time.

Our automatic strategy uses the idea described in the previous paragraph to choose the best imbalance threshold for a certain number of time-steps. Periodically, this strategy is executed again to verify if a new threshold will produce a better execution time. In order

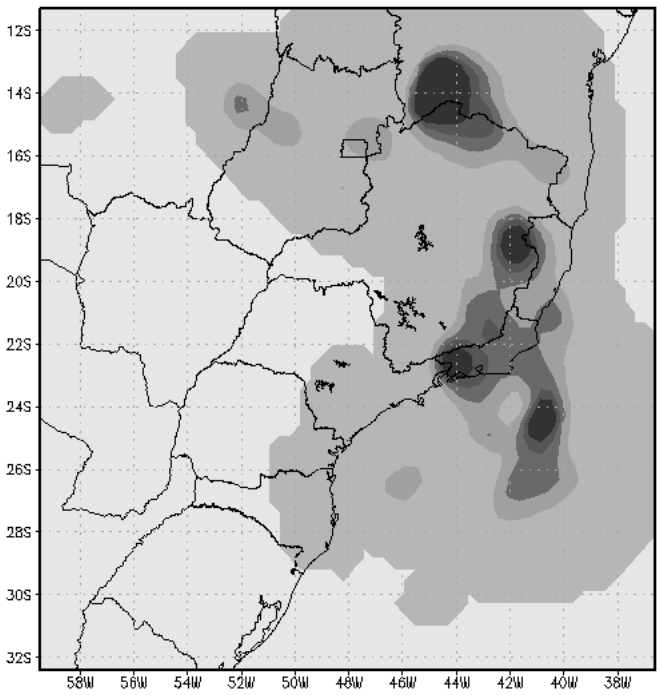
to test this approach, we ran a new set of experiments. We used the same case study as presented in the previous section. Moreover, two new forecasts were used as shown in the Figure 4.9. The forecast illustrated by the Figure 4.9b represents a recent thunderstorm that caused much destruction in the Southeast part of Brazil (Figure 4.8).

Figure 4.8: News about the thunderstorm used as case study.

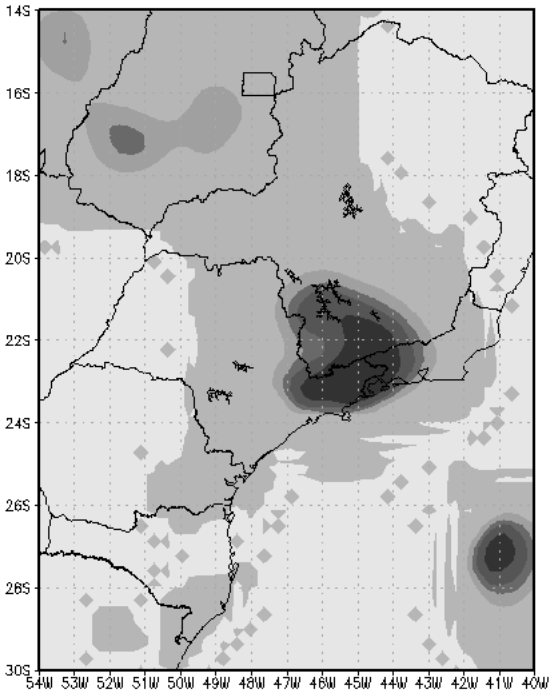
In these experiments, we compared the execution time of the Brams model with the default approach (the approach in which the load balancer is invoked at pre-established and fixed moments), with pre-established imbalance thresholds (1%, 10%, 20%, 30%, 40% and 50%), and with our automatic strategy. These experiments correspond to a execution of 2400 time-steps, each of which represents 6 seconds of real time. In our automatic strategy, we updated the threshold every 100 time-steps. Figure 4.10 shows the result for one of the experiments (the others have a similar result).

According to the result presented in the Table 4.7, choosing an appropriate imbalance threshold improved performance in comparison to the default approach. In this case, the best fixed threshold was 20%. The benefits of load balancing tend to diminish when smaller thresholds are used. Similarly, thresholds larger than 20% have lower performance, because they do not cause migrations at the right moment. Consequently the system executes longer periods imbalanced. However, as in the previous case, the best threshold is not known until the user run the model with these different values.

Our automatic strategy (presented in the Figure 4.10 as “variable”, meaning that the imbalance threshold is not fixed in addition to being found automatically) found the best threshold without any *a priori* knowledge about the execution. The performance of the presented case was better than the default approach. Figure 4.11a shows a comparison between the execution of the first 1200 time-steps of the default approach and the automatic strategy. In this figure, the measured time was taken by the first thread to execute in the master node (not necessarily rank 0). In this way, we eliminated the “amplitude” in the measures (like the one we showed in the Figure 4.3). Since we are calling the load bal-



(a) Thunderstorms over the states of Rio de Janeiro, Minas Gerais and Bahia



(b) Thunderstorms over the states of São Paulo and Rio de Janeiro

Figure 4.9: Precipitation

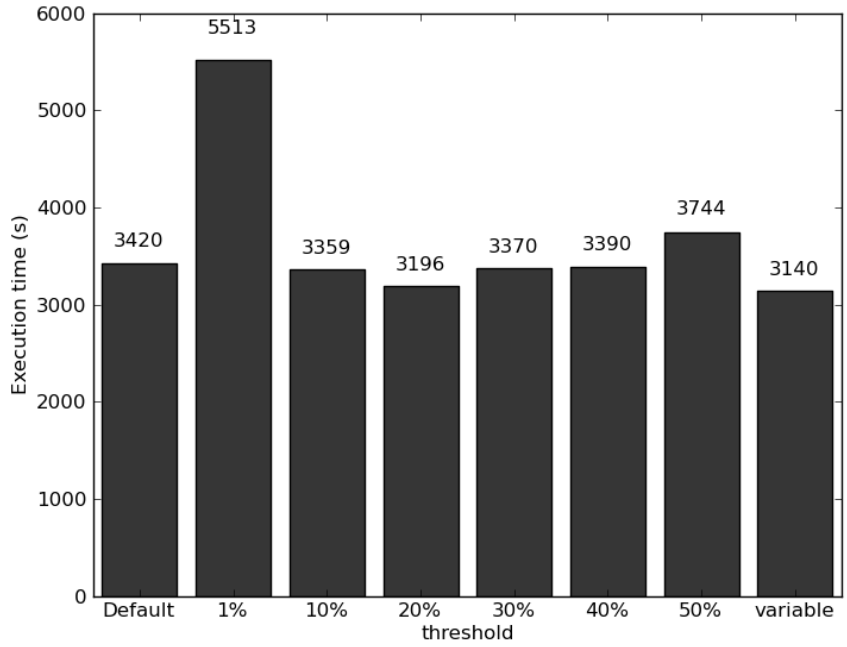


Figure 4.10: Execution time with different imbalance thresholds.

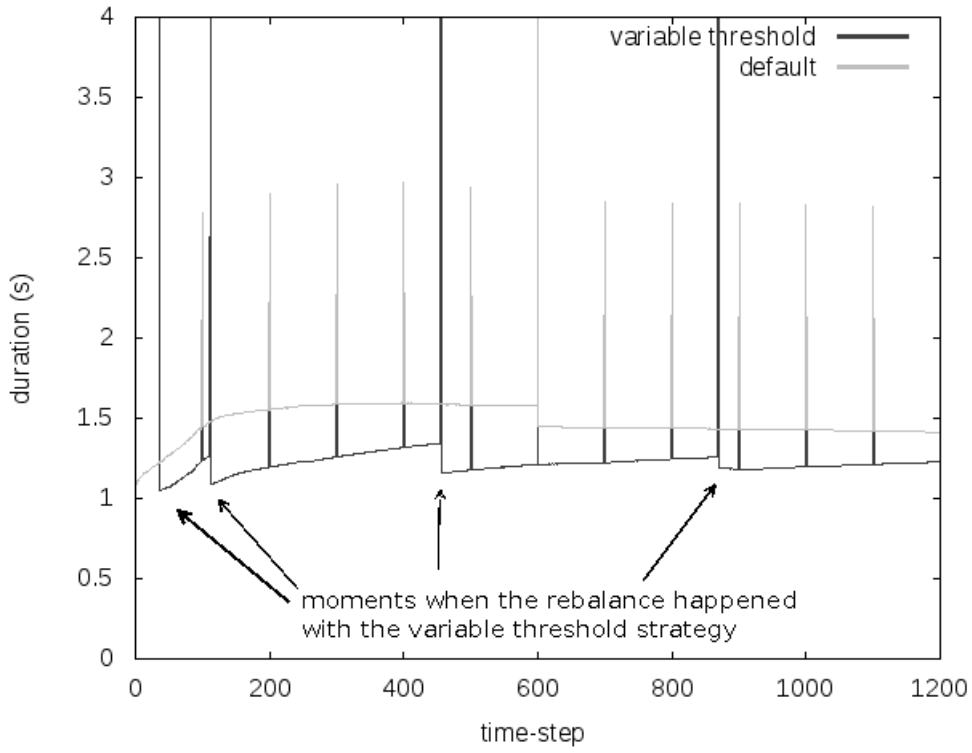
ancer more frequently, the automatic approach found the load imbalances more rapidly. This can be seen in this figure, which shows that the first migrations occurred in the very beginning of the execution (time-step 36).

Figure 4.10 shows that the variable imbalance threshold had a slightly better performance than a threshold of 20%. That happened because our strategy found some time-steps in which a threshold of 10% is more efficient. This can be seen on Figure 4.11b. The arrow in this figure points to the moment in which the threshold changed from 20% to 10%, and, as a consequence, it reduced the execution time of the next time-steps.

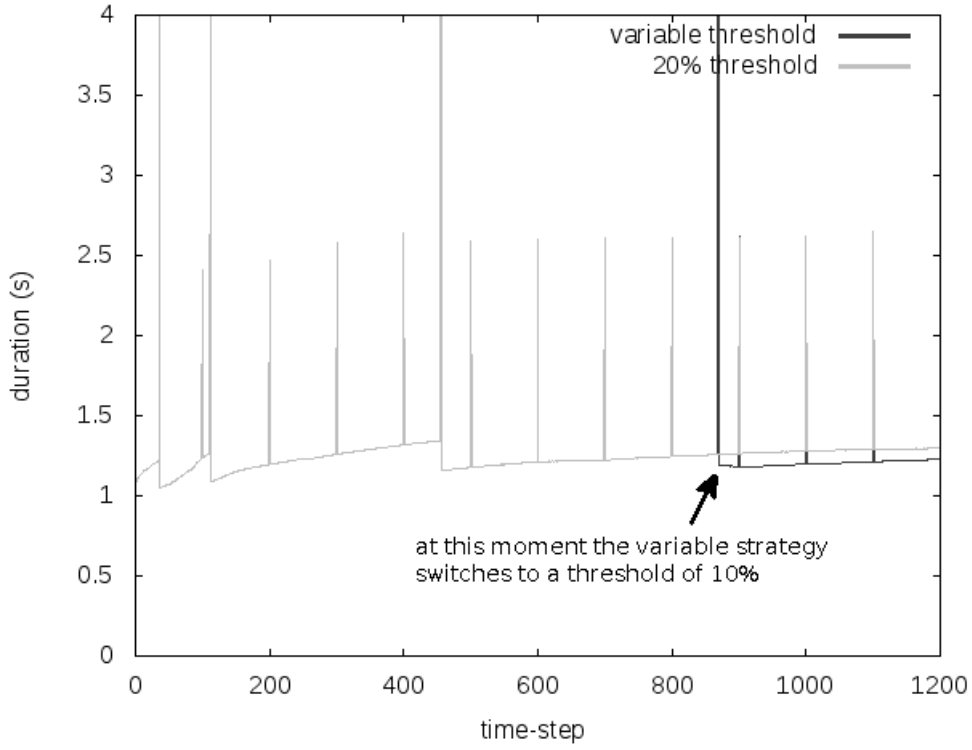
The execution time of the automatic strategy itself is short. In all the experiments, the total time of each invocation of the algorithm did not exceeded 0.1 second. One reason for this performance is the simplicity of the migration estimate. As it was shown, we use a simple latency model. However, this model was quite accurate for our experiments. A more complex and time-consuming model can be used. In this case, the user can run the strategy asynchronously, i.e. to compute the new imbalance threshold while the application executes. This is possible because the threshold does not need to be put in place immediately. The application typically can afford to run with the previous threshold for a few more time-steps.

4.6 Fifth set of experiments: distributed load balancers

So far, we have dealt exclusively with centralized load balancers. This design has the advantage that one central entity has all the information needed to take the best load



(a) Comparison between automatically variable imbalance threshold and a fixed threshold



(b) Comparison between automatically variable imbalance threshold and the default approach

Figure 4.11: Execution time of each timestep.

balancing decision. However, a major drawback is scalability. That is because all threads have to send their load indexes to the load balancer; the load balancer has to compute the destination of each thread; and it sends that decision to the appropriate thread. All these tasks may represent a bottleneck as the number of processors increases.

In Section 3.11, we described a distributed load balancer that takes advantage of the regular communication pattern typically found in meteorological models. In fact, the distributed load balancer only cuts, in a distributed way, the Hilbert curve. This curve naturally keeps neighbor threads close together. In this section, we compare this strategy with the centralized strategy and a distributed diffusion based approach.

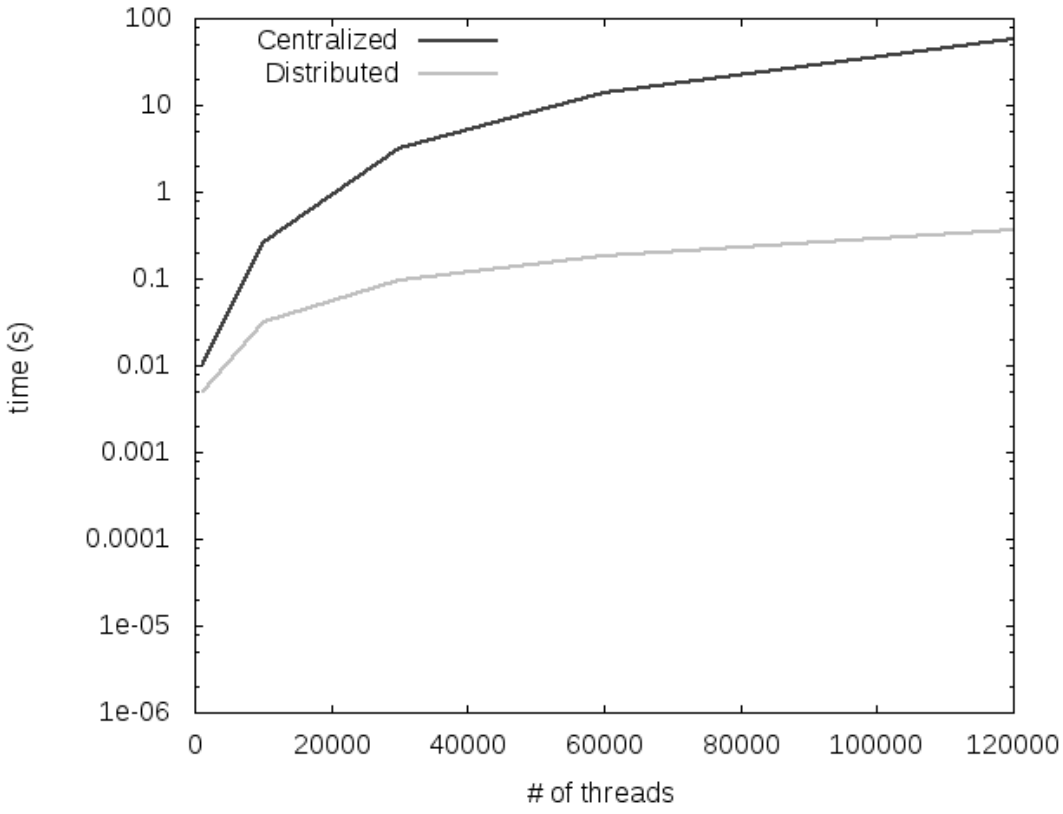


Figure 4.12: Comparison between the centralized and distributed algorithms.

4.6.1 Centralized vs. Distributed load balancers

Firstly, we compare the execution time of the centralized and distributed load balancers with different number of threads. Figure 4.12 shows this comparison. For small number of threads, the centralized load balancer is fast enough, since the load balancer is invoked only a few times for the whole execution. However, its execution time increases very rapidly as the number of threads goes up. Conversely, the distributed algorithm has much slower execution time increase. Even for 120,000 thread its execution time is under one-half of a second.

In order to analyze the reasons behind these results, we can decompose the execution

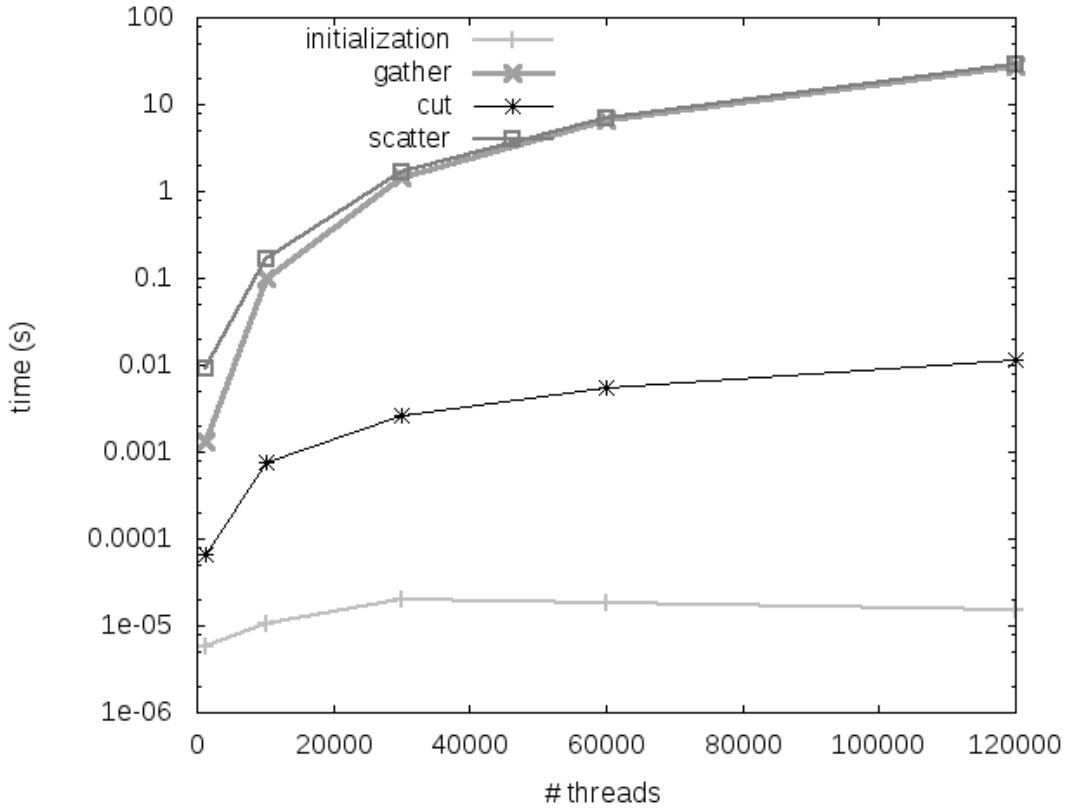
time of both approaches in their components. For the centralized load balancer, the components are four: (1) the initialization phase, which corresponds to setting up the required buffers, but only needs to be done once; (2) the gathering of load in which all threads send their loads to the load balancer; (3) the algorithm that cuts the Hilbert curve according to the received loads; and (4) the scatter that sends the migration decision to each thread. The distributed load balancer has also four components: (1) the initialization phase; (2) the computation of the prefix sum, which is done by a recursive doubling algorithm, as illustrated in Figure 3.9; (3) a broadcast of the total load in the system, which is done by the last processor in the Hilbert curve, which holds this value as a result of the prefix sum phase; and (4) the computation that each thread does to find its destination (Algorithm 2).

Figure 4.13a presents the time spent in each of the components of the centralized load balancer as the number of threads increases. The time of the initialization phase increases slightly, but stabilizes with 30,000 threads. This behavior is related to the way the *glibc* implements memory allocation; it takes more time to allocate large blocks, but that goes up to a certain limit (LEA; GLOGER, 2000). Nonetheless, this initialization is quite fast and it is done only once for the whole execution. The next component is the gathering of load indexes. This communication is proportional to $log(p)$, which is identical to the complexity of a reduction. However, the sizes of the messages are different. For a gather, the sizes increase as the algorithm executes and that makes the communication more costly than a reduction (KUMAR, 2002). This behaviour is similar to the scatter component, which has also a complexity of $log(p)$ and is similar to a broadcast. These two components are the most expensive phases of this load balancer. Finally, the cut component corresponds to the segmentation of the Hilbert curve. It also scales poorly, because only one processor performs this computation.

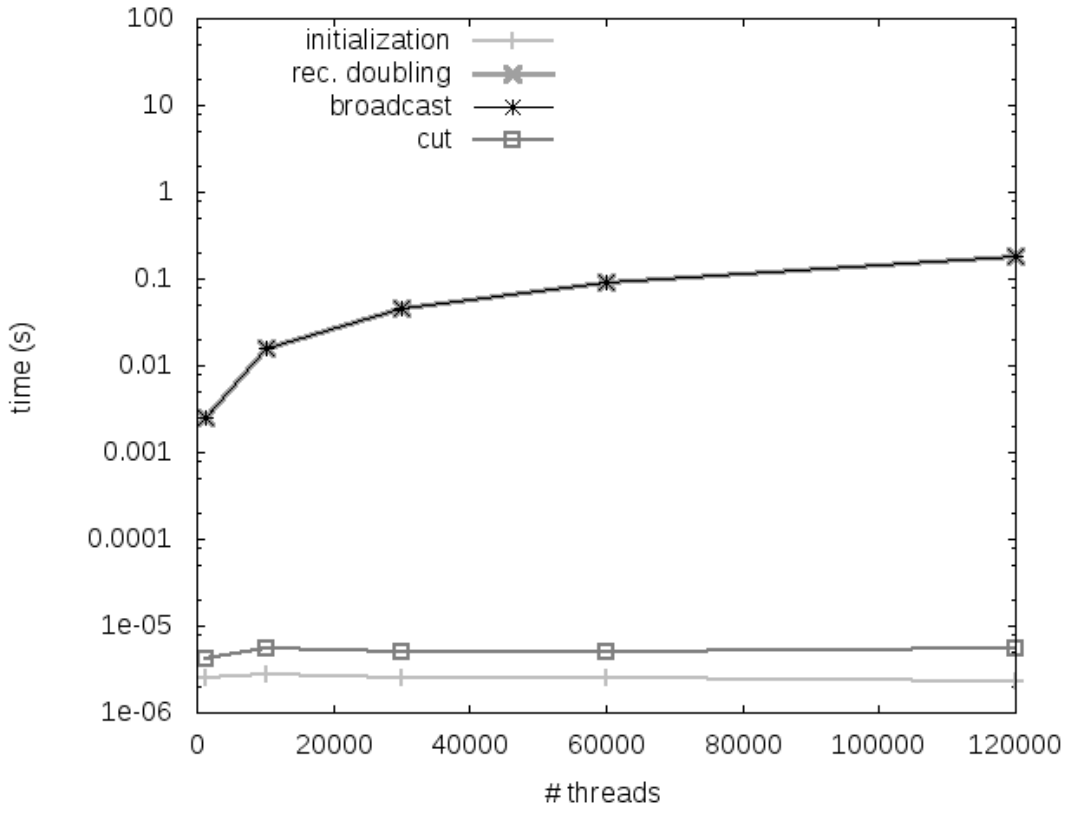
In contrast, the execution time of the components of the distributed strategy is shown in Figure 4.13b. Here, the initialization phase is very fast, because none of the threads needs to store much data. The recursive doubling part and the broadcast scale in an identical way, since both have the same complexity and the message sizes are the same, i.e. only messages of one double precision number. They represent almost all the time spent in this load balancer. Finally, the cut algorithm is also very fast, since it does not require any communication. In conclusion, the distributed strategy is faster than the centralized one because the communication in the latter scales poorly. Nonetheless, the communication in the distributed load balancer still increases when the number of processors goes up. In the next section, we compare this load balancer with another that has an almost constant communication time.

4.6.2 DiffusionLB vs. HilbertLB

All the issues of the diffusion-based load balancer (presented in Section 3.11.2) add to the complexity of this load balancer. However, the execution time of this strategy is short as a result of the reduced amount of communication. Figure 4.14 compares the execution



(a) Centralized approach.



(b) Distributed approach.

Figure 4.13: Execution time of each load balancer component.

time of the diffusion-based and the Hilbert-curve-based strategies. The former presents a constant execution time, because the number of neighbors remains approximately constant even when the total number of threads increases. Meanwhile, as shown before, the communication of the Hilbert-based strategy is proportional to $log(p)$.

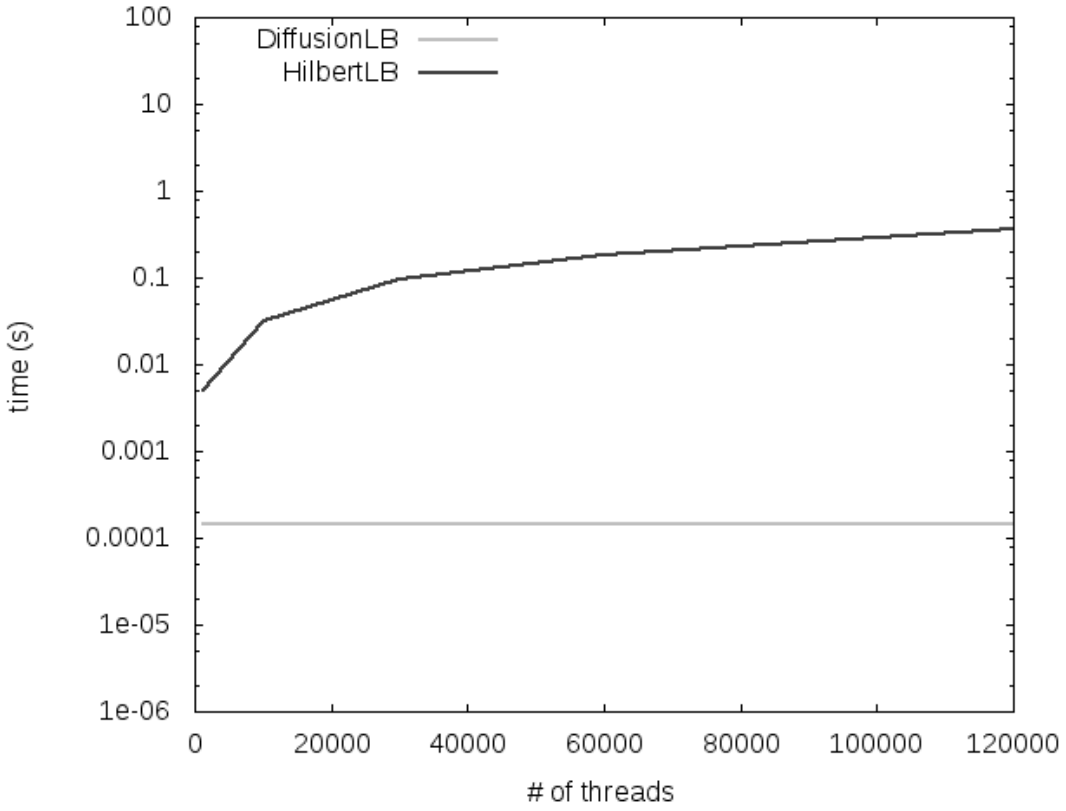


Figure 4.14: Execution-time of two distributed load balancing approaches.

Our implementation of the diffusion strategy uses the depth-first traversal algorithm to keep together the threads of a processor. This approach is simpler to implement and effectively avoids breaking the graph of local threads. However, it may not produce the most cohesive graph. As a consequence, the external communication is not optimal. Nonetheless, this metric is quite similar to that observed with the Hilbert-curve strategy, as it can be seen in Figure 4.15, which shows the processors with the minimum, average and maximum number of neighbor processors after invoking the load balancer 50 times. Naturally, the diffusion-based strategy can produce the same result as the Hilbert-curve, while the opposite is not true. Moreover, the former strategy has potential to optimize the communication further, because it has more options to select threads to migrate. Nonetheless, implementing an efficient strategy is difficult.

Although complexity is an issue for the diffusion-based load balancer, the use of virtualization isolates that from the application, hence, this problem becomes less important. However, a more critical issue is the speed of balancing of this load balancer. For each invocation, the diffusion-based strategy only balances load locally. As the time goes by,

the whole system tends to became balanced. However, that may take many invocations to happen. Conversely, the Hilbert load balancer only needs one invocation to balance load completely, even though that may not be optimal.

Figure 4.16 shows a comparison of the speed of balancing of both the diffusion-based and the Hilbert-curve-based load balancers. For this experiment, we used a fixed load distribution, which represents a localized thunderstorm. We ran both load balancers for a certain number of invocations (shown in the x axis). The load in this figure is the percentage of load at the beginning of the execution for the most loaded processor. The total number of threads is 16,384 and the virtualization ratio is 16.

Since the load is fixed, the Hilbert-based strategy reaches its best load distribution in the first invocation. On the other hand, the diffusion-based load balancer evolves much slower. In fact, the diffusion strategy may not even reach the same degree of balance as the Hilbert strategy. In Figure 4.16, even after 50 invocations, the load of the DiffusionLB is substantially higher than of the HilbertLB. The explanation for this fact relies on the way load moves. In a natural diffusion process, like heat spreading throughout space, the energy or matter moves in a continuous manner. Meanwhile, the diffusion of load in virtualized environment occurs in a discrete way; at least one thread has to migrate so that diffusion happens. As a consequence, the load may became "trapped". One example of this fact can be seen in Figure 4.17, in which $P0$, $P1$ and $P2$ are neighbors in this order in a 1-D domain, and each processor has three threads with different loads (represented by the height of each rectangle). $P0$ cannot give one thread to $P1$, because that would make the load of both much farther from average. In its turn, $P1$ does not move any load because it is already well balanced with respect to its neighbors. Therefore, in this case

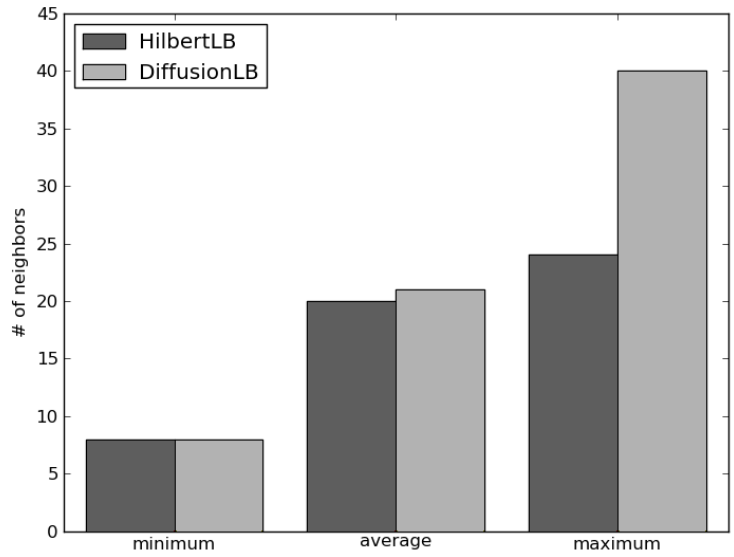


Figure 4.15: Number of neighbor processors after 50 invocations of the load balancer.

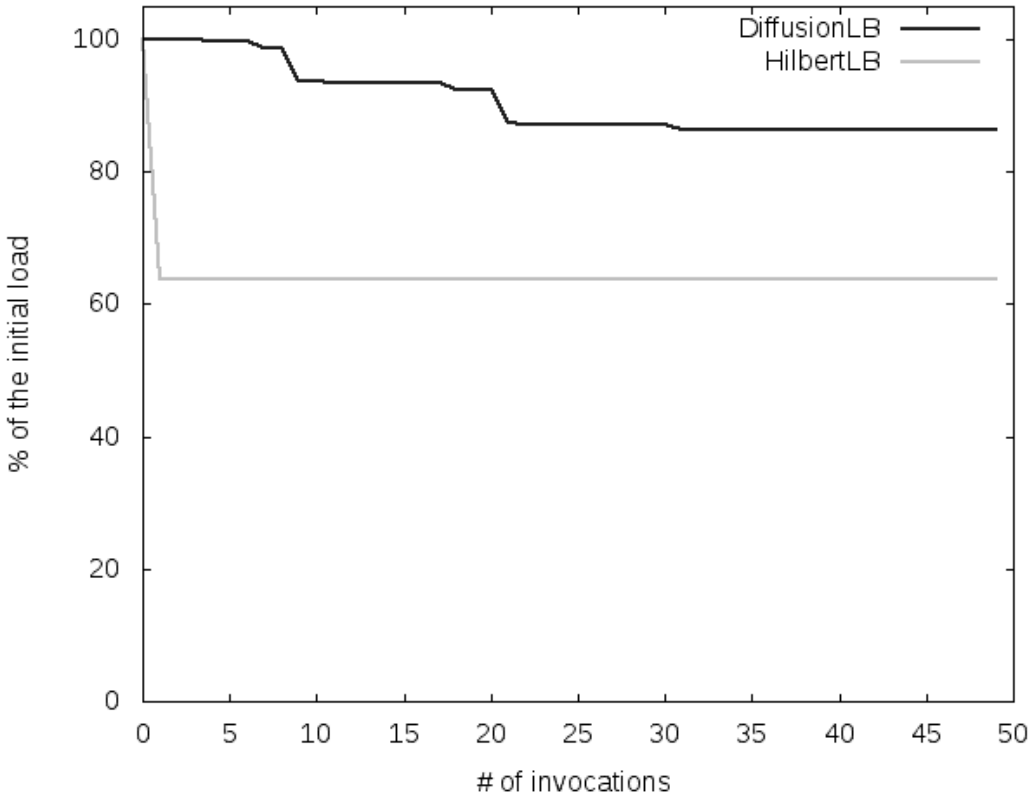


Figure 4.16: Rebalancing speed.

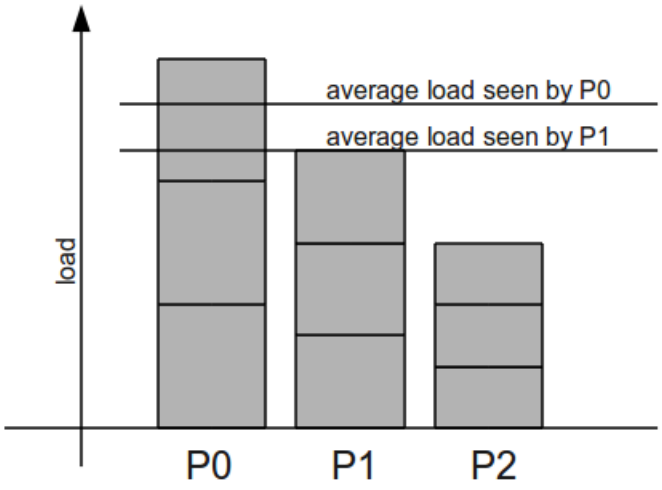


Figure 4.17: Load “trapped”.

no diffusion occurs. Ideally, we should move only the excess of load (a fraction of one thread) from each processor to keep the load moving to homogeneity.

Increasing the virtualization ratio minimizes the possibility of load to become locked. This can be confirmed by an experiment similar to the previous one (whose virtualization ratio was 16) in which 16,384 threads execute with virtualization ratio of 64; in this way,

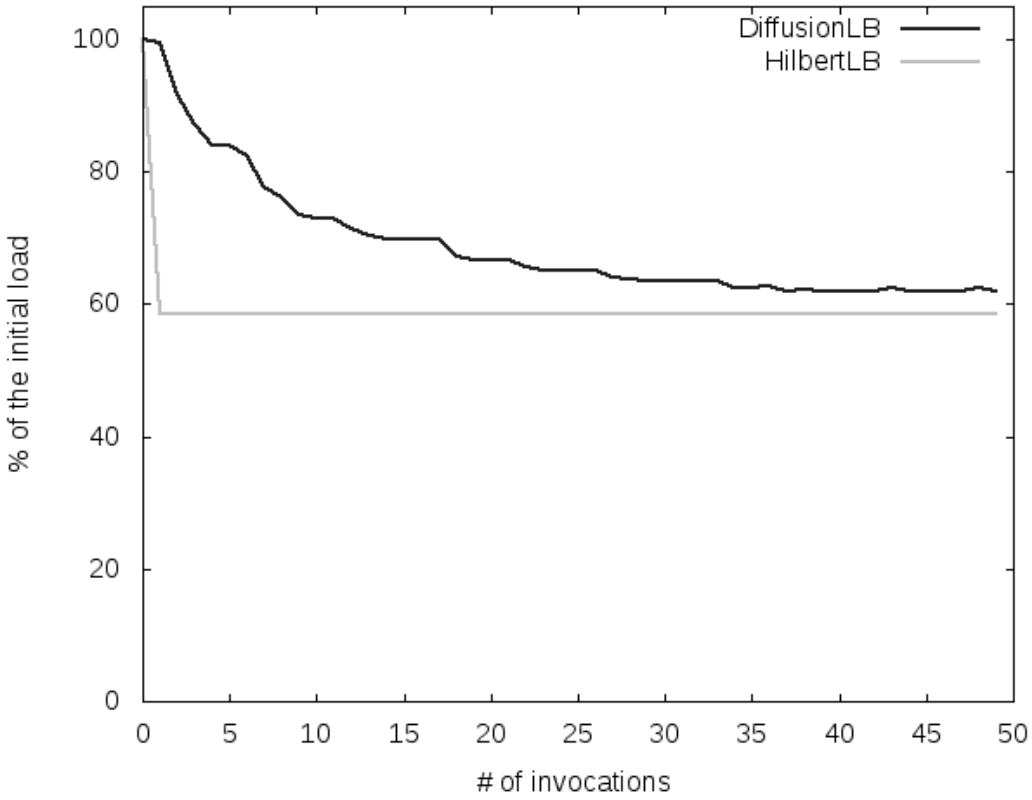


Figure 4.18: Balancing speed with 16K threads and virtualization ratio of 64.

the load becomes more “fluid” than in the previous experiment. Figure 4.19 shows the result. As it can be seen, the load moves more rapidly and the system reaches stability much closer to the HilbertLB result. However, the virtualization ratio cannot be increased indefinitely. As a result, the system may never behave like a real diffusion process, in which energy or matter flows continuously from high to low concentrations.

Although we can increase the virtualization ratio (up to a certain point), the increase in the total number of processors slows down the speed of balance of the diffusion-based strategy. If we run the same experiment as before (virtualization of 64) with more processors (1024), the system takes much longer to become balanced. Figure 4.19 shows the result of this experiment.

In summary, although the diffusion-based load balancer is distributed and has potential to optimize the communication even better than the Hilbert-curve-based load balancer, the complexity and, more important, the speed of balance are two issues that make this strategy inappropriate to the application considered.

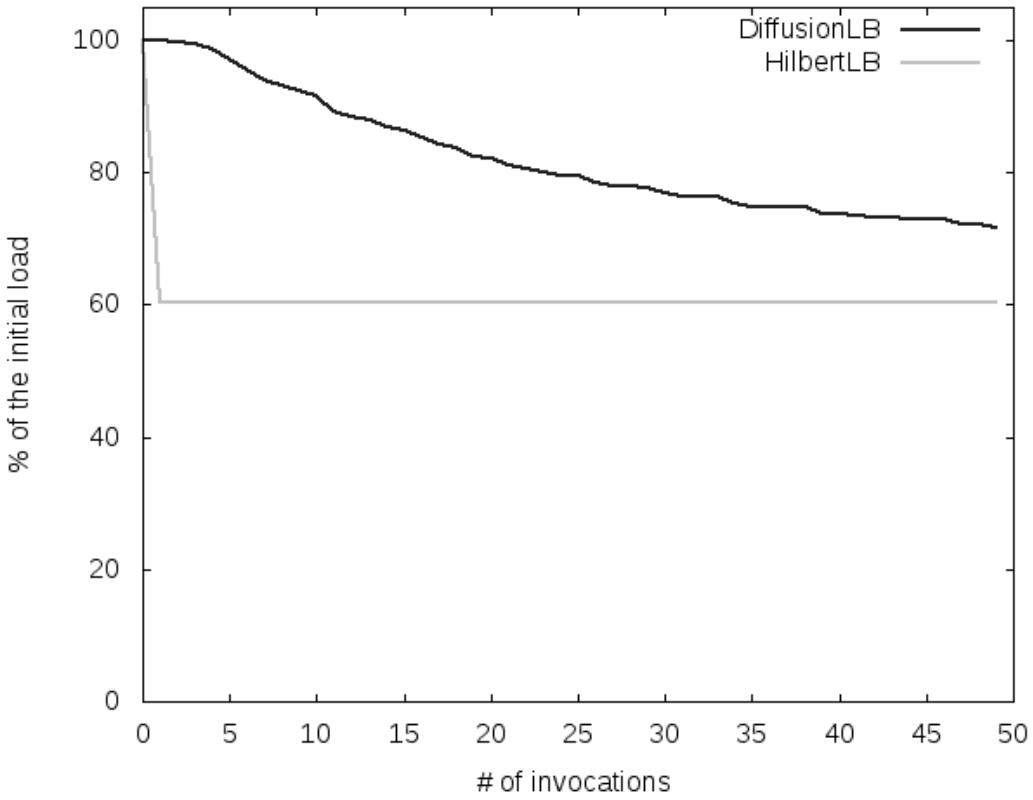


Figure 4.19: Balancing speed with 64K threads and virtualization ratio of 64.

5 FINAL REMARKS AND CONCLUSIONS

Load imbalance is a major impediment to scalability. It refers to the uneven distribution of load across tasks of a parallel application. The most loaded task dominates application completion time. If the application is synchronous, the most loaded task also delays the execution of the other tasks. Ideally, the parallel application should keep load equally distributed among its tasks, so that the efficiency is maximum.

Many load balance strategies have been developed. Nonetheless, many real-world applications lack such feature. This is particularly true for meteorological models. Although these applications typically use a regular three-dimensional grid and, in a parallel execution, each processor executes the same code, meteorological models are subject to load imbalance. Xue *et al.* (XUE; DROEGEMEIER; WEBER, 2007) declare that processors in charge of active thunderstorms may incur up to 30% additional computation. With the increase in complexity, this imbalance may grow even larger. However, most models currently do not perform load balance, due to the complexity involved with this task.

The difficulty of applying load balancing strategies to real weather models stems from the application complexity. For example, the Brams model, used in this work, has 140K lines of code and a few thousand of routines. Similarly, other models, such as WRF, are also large and complex. These applications are developed by many different people during long periods. Moreover, they are typically designed without any concern about load imbalance.

In this text, we propose a new strategy to perform load balancing. This strategy is based on Processor Virtualization. Its main advantage is simplicity; fewer changes to the original application are required than those needed by embedded load balancers. This work is the first one to successfully use processor virtualization with a complete meteorological model. Previous attempts were either too slow or not able to run more than one task per processor, which is allowed by the processor virtualization technique.

In order to use Processor Virtualization, the user has to over-decompose the parallel application in more tasks than processors. Then, a set of tasks, also known as Virtual Processors, is assigned to each processor. If a processor becomes overloaded, then some of its tasks can migrate to underloaded processors. In our experiments, we found that just the over-decomposition is already beneficial for the application: we obtained up to 25.5%

reduction in execution time (RODRIGUES et al., 2010). This reduction can be attributed to the overlap of communication and computation and better use of the cache hierarchy.

In our experiments, we used AMPI, an adaptive MPI implementation, which implements virtual processors as user-threads. Its main advantage is that user-threads are very lightweight. However, this approach changes the semantic that is usually assumed by MPI programs. That is because AMPI tasks running in a same processor do not have a private address space; the static and global variables are shared among those tasks. We developed a new strategy to privatize global and static data so that an existing application can be more easily ported to the AMPI environment. This strategy has a better context switch time than a previous privatization scheme (RODRIGUES et al., 2010).

Using AMPI, however, is not enough to deal with the load balancing issues of meteorological models or any application. A load balancing strategy has to be employed. It is responsible for mapping threads to processors whenever the load balancer is invoked. We tried some existing load balancers that, in principle, would fit the meteorological model requirements. We also developed a Hilbert curve based load balancer. It maps the 2-D domain decomposition to a 1-D space. The curve is then cut into segments so that each segment has approximately the same load. Due to properties of the Hilbert curve, the corresponding sub-domains on a given segment should be close in the 2-D space and, consequently, the cross-processor communication is reduced. This load balancer reduced the execution time by 7% (RODRIGUES et al., 2010) beyond what had already been obtained by virtualization alone.

We investigated an adaptive scheme that invokes the load balancing algorithm more frequently, but only migrates load across processors when the imbalance is beyond a certain threshold. With this adaptive scheme, an additional 5% execution time reduction was obtained. Nevertheless, this scheme is manual, i.e. the frequency and threshold had to be explicitly informed by the user.

Since the user does not know in advance what threshold will produce the best performance, the manual approach is not practical. Thus, we developed a strategy to find the threshold automatically. This strategy is based on the principle of persistence. We assumed that the best threshold for the near past is the best one for the near future. The load balancer keeps load information and periodically uses it to predict the best threshold. With this strategy, we were able to achieve the same performance as the manual approach without any previous information about the execution.

All load balancers discussed up to here in this conclusion were centralized. They have a single processor to gather load balancing information and take load balancing decisions. This approach works well for reasonably large machines. However, for larger machines, fully distributed load balancers are more appropriate. Typically, these algorithms are diffusion-based, i.e. the load “flows” from higher to lower concentrations. In this way, no processor needs to know the load distribution of the entire machine; they need only the load of their neighbors. However, this strategy tends to be very slow to converge.

Our centralized load balancing algorithm can be distributed in a way that each processor still has information about the whole system, but not the individual loads of each processor. This alleviates the communication requirements. In this text, we compared the centralized and distributed versions of the Hilbert-curve-based load balancer. The centralized version does not scale well, as expected, and the reasons for that are the gathering of load indexes and the scatter of load balancing decisions. In its turn, the communication of the distributed version scales better, but it is still $log(p)$. We then compared the distributed approach with a diffusion-based strategy, because the latter has a more efficient communication. However, as it was shown in our results, the diffusion-based load balancer takes much longer to balance load. Furthermore, since the load cannot move continuously but in "packets" (the threads), some imbalance may get locked.

Finally, although our strategy has been developed for weather forecast models, it can also be applied to other applications, as long as the same structure and behavior is present. Our Hilbert-curve load balancer depends on the regularity of the communication pattern; threads communicate with neighbors forming a Cartesian grid. That can be extended provided that a new space-filling curve that fits the new communication pattern is used. In addition, the behavior of the application must be similar to weather forecast models, in the sense that there is no massive imbalance. We expect that applications such as Fluid Dynamics and Ocean Modeling can benefit from our strategy.

As potential future work, one could apply our strategy to other applications. For that, newer space-filling curves may need to be used. We can also explore heterogeneous architectures. In these architectures, the objective may not be only to reduce execution time. Another possible metric can be energy efficiency. Therefore, the load balancer not only has to minimize execution time and communication, but also reduce power consumption. Another direction is to analyze if meteorological information, available in the model, could be used to further improve the balancing decisions. However, this would require an intimate knowledge of the application.

REFERENCES

ALTMAN, E.; AYESTA, U.; PRABHU, B. Optimal load balancing in processor sharing systems. In: INTERNATIONAL WORKSHOP ON GAME THEORY IN COMMUNICATION NETWORKS (GAMECOMM). **Proceedings...** [S.l.: s.n.], 2008.

BANICESCU, I. et al. Design and implementation of a novel dynamic load balancing library for cluster computing. **Parallel Computing**, [S.l.], v.31, n.7, p.736–756, 2005.

BANICESCU, I.; VELUSAMY, V. Load balancing highly irregular computations with the adaptive factoring. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.: s.n.], 2002. p.87–98.

BARAK, A.; LA'ADAN, O.; SHILOH, A. Scalable cluster computing with MOSIX for Linux. In: ANNUAL LINUX EXPO, 5. **Proceedings...** [S.l.: s.n.], 1999. p.95–100.

BARKER, K. et al. A load balancing framework for adaptive and asynchronous applications. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], p.183–192, 2004.

BHATELE, A. **Automating Topology Aware Mapping for Supercomputers**. 2010. Thesis — Department of Computer Science, University of Illinois.

BHATELE, A. et al. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.: s.n.], 2008.

BILLIONNET, A.; COSTA, M.; SUTTER, A. An efficient algorithm for a task allocation problem. **Journal of the ACM (JACM)**, [S.l.], v.39, n.3, p.502–518, 1992.

BLUMOFE, R. et al. Cilk: an efficient multithreaded runtime system. **ACM SigPlan Notices**, [S.l.], v.30, n.8, p.216, 1995.

BOHM, E. et al. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. **IBM Journal of Research and Development: Applications of Massively Parallel Systems**, [S.l.], v.52, n.1/2, p.159–174, 2008.

BOKHARI, S. Dual processor scheduling with dynamic reassignment. **IEEE Transactions on Software Engineering**, [S.l.], p.341–349, 1979.

BOKHARI, S. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. **IEEE Transactions on Software Engineering**, [S.l.], p.583–589, 1981.

BRAUN, T. et al. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. **Journal of Parallel and Distributed computing**, [S.l.], v.61, n.6, p.810–837, 2001.

BROWNE, S. et al. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In: SUPERCOMPUTING'00, Dallas, Texas. **Proceedings...** [S.l.: s.n.], 2000.

CASAVANT, T.; KUHL, J. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Transactions on Software Engineering**, [S.l.], p.141–154, 1988.

CHUNG, K.; HUANG, Y.; LIU, Y. Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query. **Information Sciences**, [S.l.], v.177, n.10, p.2130–2151, 2007.

DEVINE, K. et al. Zoltan data management services for parallel dynamic applications. **Computing in Science and Engineering**, [S.l.], v.4, n.2, p.90–96, 2002.

DEVINE, K. et al. New challenges in dynamic load balancing. **Applied Numerical Mathematics**, [S.l.], v.52, n.2-3, p.133–152, 2005.

DOBBER, M.; KOOLE, G.; MEI, R. van der. Dynamic load balancing experiments in a grid. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID–CCGRID. **Proceedings...** [S.l.: s.n.], 2005. p.1063–1070.

DREPPER, U. **Elf handling for thread-local storage**. [S.l.]: Red Hat, 2003.

ELIANE, S. et al. The SegHidro Experience: using the grid to empower a hydrometeorological. In: FIRST INTERNATIONAL CONVERENCE ON E-SCIENCE AND GRID COMPUTING (E-SCIENCE/05). **Proceedings...** [S.l.: s.n.], 2005. p.64–71.

FAZENDA, A. L. et al. Escalabilidade de aplicação operacional em ambiente massivamente paralelo. In: X SIMPÓSIO EM SISTEMAS COMPUTACIONAIS (WSCAD-SCC). **Anais...** [S.l.: s.n.], 2009. p.27–34.

FOSTER, I. **Designing and building parallel programs: concepts and tools for parallel software engineering**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

FOSTER, I.; TOONEN, B. Load-balancing algorithms for climate models. In: SCALABLE HIGH-PERFORMANCE COMPUTING CONFERENCE. **Proceedings...** [S.l.: s.n.], 1994. p.674–681.

FOX, G.; WILLIAMS, R.; MESSINA, P. **Parallel computing works!** [S.l.]: Morgan Kaufmann, 1994.

FRANCESCHELLI, M.; GIUA, A.; SEATZU, C. Load balancing on networks with gossip-based distributed algorithms. In: IEEE CONFERENCE ON DECISION AND CONTROL, 46. **Proceedings...** [S.l.: s.n.], 2007. p.500–505.

FREITAS, S. et al. The Coupled Aerosol and Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System (CATT-BRAMS). **Atmospheric Chemistry and Physics**, [S.l.], v.9, n.8, p.2843–2861, 2009.

GEVAERD, R.; FREITAS, S. R.; LONGO, K. M. Numerical simulation of biomass burning emission and trasportation during 1998 Roraima fires. In: INTERNATIONAL CONFERENCE ON SOUTHERN HEMISPHERE METEOROLOGY AND OCEANOGRAPHY (ICSHMO) 8. **Proceedings...** [S.l.: s.n.], 2006.

GHAN, S. et al. The thermodynamic influence of subgrid orography in a global climate model. **Climate Dynamics**, [S.l.], v.20, n.1, p.31–44, 2002.

GHAN, S.; SHIPPERT, T. Load balancing and scalability of a subgrid orography scheme in a global climate model. **International Journal of High Performance Computing Applications**, [S.l.], v.19, n.3, p.237, 2005.

GRELL, A. G.; DÉVÉNYI, D. A. A new approach to parameterizing convection using ensemble and data assimilation techniques. **Geophysical Research Letters**, [S.l.], v.29, p.1693, 2002.

GRELL, G.; DEVENYI, D. A generalized approach to parameterizing convection combining ensemble and data assimilation techniques. **Geophysical Research Letters**, [S.l.], v.29, n.14, p.38–1, 2002.

GROSU, D.; CHRONOPOULOS, A. Noncooperative load balancing in distributed systems. **Journal of Parallel and Distributed Computing**, [S.l.], v.65, n.9, p.1022–1034, 2005.

GROSU, D.; LEUNG, A. Load balancing in distributed systems: an approach using cooperative games. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.: s.n.], 2002. p.52–61.

GUIL, N.; ZAPATA, E. Fast Hough transform on multiprocessors: a branch and bound approach. **Journal of Parallel and Distributed Computing**, [S.l.], v.45, n.1, p.82–89, 1997.

HILBERT, D. Über die stetige Abbildung einer Linie auf ein Flächenstück. **Mathematische Annalen**, [S.l.], v.38, p.459–460, 1891.

HOFFMAN, J. **Numerical methods for engineers and scientists**. [S.l.]: CRC Press, 2001.

HUANG, C. et al. Performance Evaluation of Adaptive MPI. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING 2006. **Proceedings...** [S.l.: s.n.], 2006.

HUANG, C.; LAWLOR, O.; KALÉ, L. V. Adaptive MPI. In: INTERNATIONAL WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING (LCPC 2003), 16., College Station, Texas. **Proceedings...** [S.l.: s.n.], 2003. p.306–322.

HUMMEL, S.; SCHONBERG, E.; FLYNN, L. Factoring: a method for scheduling parallel loops. **Communications of the ACM**, [S.l.], v.35, n.8, p.101, 1992.

ICHIKAWA, S.; YAMASHITA, S. Static load balancing of parallel PDE solver for distributed computing environment. In: PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS (PDCS'2000). **Proceedings...** [S.l.: s.n.], 2000. p.399–405.

JÁJÁ, J. **An introduction to parallel algorithms**. [S.l.]: Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1992.

JETLEY, P. et al. Massively Parallel Cosmological Simulations with ChaNGa. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.: s.n.], 2008.

JIAO, X. et al. An Integration Framework for Simulations of Solid Rocket Motors. In: AIAA/ASME/SAE/ASEE JOINT PROPULSION CONFERENCE, 41., Tucson, Arizona. **Proceedings...** [S.l.: s.n.], 2005.

KALÉ, L. V. et al. Scaling Molecular Dynamics to 3000 Processors with Projections: a performance analysis case study. In: TERASCALE PERFORMANCE ANALYSIS WORKSHOP, INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE(ICCS), Melbourne, Australia. **Proceedings...** [S.l.: s.n.], 2003.

KALÉ, L. V. et al. Programming Petascale Applications with Charm++ and AMPI. In: BADER, D. (Ed.). **Petascale Computing**: algorithms and applications. [S.l.]: Chapman & Hall / CRC Press, 2008. p.421–441.

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

KHAN, S.; AHMAD, I. Non-cooperative, semi-cooperative, and cooperative games-based grid resource allocation. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2006. p.101.

KINTER III, J.; WEHNER, M. Computing Issues for WCRP Weather and Climate Modeling. In: WCRP MODELING PANEL, Exeter, UK. **Proceedings...** [S.l.: s.n.], 2005.

KOZIAR, C.; REILEIN, R.; RUNGER, G. Load imbalance aspects in atmosphere simulations. **International Journal of Computational Science and Engineering**, [S.l.], v.1, n.2, p.215–225, 2005.

KUMAR, R. et al. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: COMPUTER ARCHITECTURE, 31. **Proceedings...** [S.l.: s.n.], 2004. p.64.

KUMAR, V. **Introduction to parallel computing**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

LEA, D.; GLOGER, W. **A memory allocator**. 2000.

LEE, S. et al. An Adaptive Load Balancing Approach in Distributed Computing Using Genetic Theory. **Parallel and Distributed Computing: Applications and Technologies**, [S.l.], p.322–325, 2005.

LIU, X.; SCHRACK, G. Encoding and decoding the Hilbert order. **Software, practice & experience**, [S.l.], v.26, n.12, p.1335–1346, 1996.

LOH, E. The ideal HPC programming language. **Communications of the ACM**, [S.l.], v.53, n.7, p.42–47, 2010.

LOTTIAUX, R. et al. **Openmosix, openssi and kerrighed: a comparative study**. France: IRISA, 2004.

MICHALAKES, J. MM90: a scalable parallel implementation of the penn state/ncar mesoscale model (mm5). **Parallel Computing**, [S.l.], v.23, n.14, p.2173–2186, 1997.

MICHALAKES, J. et al. WRF nature run. In: SUPERCOMPUTING, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2007. p.1–6.

MORIN, C. et al. Kerrighed: a single system image cluster operating system for high performance computing. In: EURO-PAR 2003. **Proceedings...** Springer, 2003. p.1291–1294.

OLIVA, A.; ARAÚJO, G. Speeding up thread-local storage access in dynamic libraries. In: GCC DEVELOPER'S SUMMIT. **Proceedings...** [S.l.: s.n.], 2006. p.159–178.

OTTO, S. Processor Virtualization and Migration for PVM. In: WORKSHOP ON ENVIRONMENTS AND TOOLS FOR PARALLEL SCIENTIFIC COMPUTING, 2. **Proceedings...** [S.l.: s.n.], 1994. p.66–75.

PLASTINO, A. et al. **Load balancing in SPMD applications**: concepts and experiments. Norwell, MA, USA: Kluwer Academic Publishers, 2004. p.95–107.

RAPAPORT, D. **The art of molecular dynamics simulation**. [S.l.]: Cambridge Univ Pr, 2004.

RODRIGUES, E. R. et al. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (ISCC). **Proceedings...** [S.l.: s.n.], 2009. p.811–817.

RODRIGUES, E. R. et al. A New Technique for Data Privatization in User-level Threads and its Use in Parallel Applications. In: ACM 25TH SYMPOSIUM ON APPLIED COMPUTING (SAC), SIERRE, SWITZERLAND. **Proceedings...** [S.l.: s.n.], 2010.

RODRIGUES, E. R. et al. Optimizing an MPI Weather Forecasting Model via Processor Virtualization. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING (HIPC 2010), Goa - India. **Proceedings...** [S.l.: s.n.], 2010.

RODRIGUES, E. R. et al. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In: IEEE INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 22., Petrópolis - Brazil. **Proceedings...** [S.l.: s.n.], 2010.

ROTITHOR, H. Taxonomy of dynamic task scheduling schemes in distributed computing systems. In: IEE: COMPUTERS AND DIGITAL TECHNIQUES. **Proceedings...** [S.l.: s.n.], 1994. v.141, n.1, p.1–10.

SHAH, R.; VEERAVALLI, B.; MISRA, M. On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments. **IEEE Transactions on parallel and distributed systems**, [S.l.], v.18, n.12, p.1675–1686, 2007.

SHEN, C.; TSAI, W. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. **IEEE Transactions on Computers**, [S.l.], v.100, n.34, p.197–203, 1985.

SIDDHA, S.; PALLIPADI, V.; MALLICK, A. Chip multi processing aware linux kernel scheduler. In: LINUX SYMPOSIUM. **Proceedings...** [S.l.: s.n.], 2005. p.193.

SOUTO, R. et al. Processing mesoscale climatology in a grid environment. In: SEVENTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID–CCGRID. **Proceedings...** [S.l.: s.n.], 2007.

STONE, H. Critical load factors in two-processor distributed systems. **IEEE transactions on Software Engineering**, [S.l.], p.254–258, 1978.

STREITZ, F. et al. 100+ TFlop solidification simulations on BlueGene/L. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008. **Proceedings...** [S.l.: s.n.], 2008.

TERESCO, J.; FAIK, J.; FLAHERTY, J. Hierarchical partitioning and dynamic load balancing for scientific computation. In: APPLIED PARALLEL COMPUTING. **Proceedings...** [S.l.: s.n.], 2006. p.911–920.

TRIPOLI, G.; COTTON, W. **The Colorado State University three-dimensional cloud/mesoscale model**. [S.l.]: Atmos, 1982. (3).

WALKO, R. et al. Coupled atmosphere–biophysics–hydrology models for environmental modeling. **Journal of Applied Meteorology**, [S.l.], v.39, n.6, 2000.

WILLEBEEK-LEMAIR, M.; REEVES, A. Strategies for Dynamic Load Balancing on Highly Parallel Computers. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.4, n.9, 1993.

XUE, M.; DROEGEMEIER, K.; WEBER, D. Numerical Prediction of High-Impact Local Weather: a driver for petascale computing. **Petascale Computing: Algorithms and Applications**, [S.l.], p.103–124, 2007.

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

ZHENG, G. et al. **Periodic Hierarchical Load Balancing for Large Supercomputers**. [S.l.]: Parallel Programming Laboratory, 2010. (10-20).

ZHENG, G.; LAWLOR, O. S.; KALÉ, L. V. Multiple Flows of Control in Migratable Parallel Programs. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

WORKSHOPS (ICPPW'06), 2006., Columbus, Ohio. **Proceedings...** IEEE Computer Society, 2006. p.435–444.

APPENDIX A PUBLISHED ARTICLES

Main articles:

Rodrigues, E. R.; Madruga, F. L.; Navaux, P. O. A.; Panetta, J. Multi-core Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications. In: IEEE symposium on Computers and Communications - ISCC, Sousse, Tunisia, 2009.

Rodrigues, E. R.; Navaux, P. O. A.; Panetta, J.; Mendes, C. L. A New Technique for Data Privatization in User-level Threads and its Use in Parallel Applications. In: 25th ACM Symposium On Applied Computing - SAC, Sierre, Switzerland, 2010.

Rodrigues, E. R.; Navaux, P. O. A.; Panetta, J.; Mendes, C. L.; Kale, L. V. Optimizing an MPI Weather Forecasting Model via Processor Virtualization. In: International Conference on High Performance Computing - HiPC 2010, Goa, India, 2010.

Rodrigues. E. R.; Navaux, P. O. A.; Panetta, J.; Fazenda, A.; Mendes, C. L.; Kale, L. V. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In: 22nd International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD, Petropolis, Brazil, 2010.

Rodrigues, E. R.; Navaux, P. O. A.; Panetta, J.; Mendes, C. L. Preserving the Original MPI Semantic in a Processor Virtualized Environment. Invited paper to Elsevier Journal Science of Computer Programming. 2010.

Zheng, G.; Negara, S.; Mendes, C.L.; Kale, L.V.; Rodrigues E.R. Automatic Handling of Global Variables for Multi-threaded MPI Programs. Submitted to IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011.

Other articles:

Souto, R.P.; Avila, R.B.; Navaux, P.O.A.; Py, M.X.; Diverio, T.A.; Campos H.F.; Stephany,

S.; Preto, A.J.; Panetta, J.; Rodrigues, E.R.; Almeida, E.S.; Silva P.L.; Gandu, A.W.; Processing Mesoscale Climatology in a Grid Environment. In: Seventh IEEE International Symposium on Cluster Computing and the Grid - CCGrid '07. Rio de Janeiro, Brazil, 2007.

Schneider, J. ; Gehr, J. ; Heiss, H. U. ; Ferreto, T. ; Rose, C. A. F. de ; RighI, R. da R. ; Rodrigues, E. R. ; Maillard, Nicolas ; Navaux, P. O. A. . Design of a Grid workflow for a climate application. In: IEEE Symposium on Computers and Communications - ISCC'09. Susse, Tunisia, 2009.

Diener, M. ; Madruga, F. L. ; Rodrigues E. R. ; Alves, M. A. Z. ; Schneider, J. ; Navaux, P. O. A. ; Heiss, H. U. . Evaluating Thread Placement Based on Memory Access Patterns for Shared Cache Multi-core Processors. In: International Symposium on Advances of High Performance Computing and Networking - AHPCN-2010, Melbourne. 12th IEEE International Conference on High Performance Computing and Communications (HPCC-2010). Los Alamitos : IEEE Computer Society Press, 2010.

Madruga, F.L.; Rodrigues, E.R.; Navaux, P.O.A. Mapeamento de Processos Aplicado a um Modelo Meteorológico. ERAD 2009, Caxias do Sul, Brazil, 2009.

Rodrigues, E. R.; Navaux, P.O.A.; Panetta, J. Load-balancing in large machines applied to weather forecast: a preliminary study. WSPPD 2007, Porto Alegre, Brazil, 2007.

Rodrigues, E. R.; Navaux, P.O.A.; Panetta, J. On Simulation of Massively Parallel Computers. WSPPD 2008, Porto Alegre, Brazil, 2008.

Reis, T. A.; Rodrigues, E. R.; Vizzotto, J. K.; Charao, A. S.; Velho, H. F. C. Visualização de mapas meteorológicos gerados pelo brams no google maps. SICINPE 2007, São José dos Campos, Brazil, 2007.

APPENDIX B *RESUMO EM PORTUGUÊS*

B.1 Introdução

Modelos meteorológicos são aplicações CPU-intensivas e normalmente são executadas em máquinas paralelas. Vários obstáculos, entretanto, impedem a obtenção de máxima eficiência nessas máquinas. Um dos principais obstáculos é desbalanceamento de carga, que corresponde a distribuição desigual de carga entre os processadores da máquina paralela. Desta forma, o tempo de execução total da aplicação é ditado pelo processador mais carregado. Várias técnicas foram desenvolvidas para tratar esse problema, mas elas são embutidas na aplicação, o que as torna inflexíveis e de difícil uso em aplicações legadas.

Essa tese explora o conceito de virtualização de processadores para balancear carga de modelos meteorológicos. Essa abordagem tem a vantagem de desacoplar a aplicação da estratégia de balanceamento de carga. Basicamente, virtualização de processadores corresponde a sobre-decomposição da aplicação em mais tarefas do que processadores reais. Assumindo que várias tarefas podem executar de forma segura em cada processador real, esse conceito permite que tarefas possam migrar de um processador real para outro para se balancear carga.

Nosso objetivo é mostrar que virtualização de processadores pode ser usado em modelos meteorológicos, desde que uma estratégia de balanceamento de carga apropriada seja usada. Nossa estratégia leva em consideração tanto a carga dos processadores como o padrão de comunicação entre as tarefas. Nós também desenvolvemos técnicas para manter a semântica da aplicação mesmo usando-se virtualização de processadores. Ainda, foi desenvolvido uma estratégia automática para seleção do momento em que as migrações devem ocorrer. Finalmente, desenvolvemos uma estratégia distribuída para uso em máquinas com muitos processadores (da ordem de milhares de tarefas).

Como estudo de caso, usamos o modelo meteorológico Brams, um modelo regional de meso-escala baseado em MPI. Escolhemos esse modelo por que ele apresenta desbalanceamento de carga principalmente como resultado de tempestades. Além disso, avaliamos outros benefícios da virtualização de processadores.

B.2 Método

Nesta seção, nós descreveremos o método desenvolvido. Inicialmente apresentaremos o conceito de Virtualização de Processadores, e as ferramentas Charm++ e AMPI. Em seguida, mostraremos como portar uma aplicação para um ambiente virtualizado. Uma questão chave nesse processo é a privatização de variáveis globais e estáticas. Foi desenvolvido uma nova estratégia de privatização que oferece um melhor desempenho que a abordagem previamente existente. Por outro lado, portar uma aplicação para um ambiente virtualizado não é suficiente para resolver o problema de desbalanceamento. Portanto, como parte de nossa estratégia apresentamos um novo balanceador de carga baseado em uma *space filling curve*. Apresentamos aqui também outros balanceadores de carga os quais serão comparados com o nosso na próxima seção.

B.2.1 Virtualização de processadores

Virtualização de processadores (OTTO, 1994) refere-se a ideia de que o programador decompõe a aplicação em um conjunto *VP* de tarefas que serão executadas em *P* processadores. Idealmente, *VP* é muito maior que *P* de forma que, na ocorrência de desbalanceamento de carga, o sistema migra algumas das tarefas de processadores mais carregados para outros menos carregados. Essas tarefas são chamadas de processadores virtuais, pois emula o que um processador iria fazer numa execução convencional, em que há uma tarefa por processador.

A razão entre os números *VP* e *P* é chamado de razão de virtualização. Valores para essa razão depende fortemente da aplicação e da máquina usada. O usuário, tipicamente, não pode aumentar indefinidamente a razão de virtualização, pois a aplicação impõe limites ao número máximo de tarefas. Além disso, um número excessivo de tarefas pode reduzir o desempenho paralelo. Na próxima seção, iremos apresentar alguns exemplos em que isso ocorre.

Além de permitir balanceamento de carga, a virtualização de processadores também produz outros benefícios. Um deles é a sobreposição automática de processamento e comunicação, sem o uso explicito de comunicação não-bloqueante. Afora isso, o uso de virtualização de processadores pode melhorar o uso de cache, por que ao se aumentar o número de tarefas, cada tarefa individualmente recebe menos dados que podem caber mais facilmente em cache. Alguns dos experimentos apresentados na próxima seção mostram resultados que ilustram esses benefícios.

B.2.2 Charm++ and AMPI

Charm++ (KALÉ et al., 2008) é uma linguagem orientada a objetos para processamento de alto desempenho. Nela processadores virtuais são implementados como objetos que podem migrar para balancear carga. A visão que o programador tem é de um conjunto de objetos que interagem entre si. O sistema mapeia esses objetos nos processadores disponíveis.

AMPI é uma implementação de MPI sobre a linguagem Charm++. Cada tarefa MPI é implementada como um objeto Charm que pode migrar. Dessa forma, um programa MPI convencional pode ser executado num ambiente virtualizado. As tarefas MPI são executadas como threads de usuário. Desta forma a troca de contexto é muito mais leve do que processos pesados ou mesmo threads de kernel. Entretanto, essa abordagem quebra a semântica que o programador tipicamente assume com respeito a variáveis globais e estáticas. Em um programa MPI convencional, esses tipos de variáveis são privadas a cada tarefa MPI. Entretanto, threads de usuário num mesmo processador compartilham variáveis globais e estáticas. Na próxima subseção, apresentamos um técnica que desenvolvemos para resolver esse problema.

B.2.3 Adaptações para o AMPI

Para uma aplicação se beneficiar das vantagens do AMPI, é necessário que um novo esquema de decomposição seja usado. A aplicação tem de ser decomposta em mais tarefas (*ranks*) do que processadores. Além disso o usuário tem de informar ao balanceador de carga o momento em que medidas de desbalanceamento serão tomadas. Nesse ponto da execução, o balanceador pode então tomar a decisão de migrar tarefas e o destino das migrações. Essa alteração corresponde tão somente a inclusão da seguinte linha de código:

```
if ( mod(iteration,K) == 0 ) call MPI_Migrate()
```

Quando se usa a sobre-decomposição de domínio, mais de uma tarefa pode executar no mesmo processador. Esse fato pode acarretar problemas com variáveis globais e estáticas. Em plataformas que suportam o formato de execução ELF, o AMPI possui uma forma de resolver esse problema para variáveis globais. Entretanto, essa abordagem não funciona para variáveis estáticas.

Além do fato descrito acima, a abordagem baseada em ELF torna a troca de contexto das tarefas proporcional ao número de variáveis. Esse problema pode ser particularmente oneroso para aplicações científicas e meteorológicas, pois esse tipo de aplicação normalmente apresenta um elevado número de variávies globais e estáticas.

A fim de eliminar os problemas encontrados na forma de privatização tradicional, desenvolvemos uma nova estratégia de privatização de variáveis. Essa estratégia é baseada em *Thread Local Storage* (TLS). Nessa estratégia usamos um mecanismo de *threads* de kernel para privatizar dados de *threads* de usuário. Essa é a primeira vez que essa técnica é usada. Dessa forma, resolvemos tanto o problema da privatização de variáveis estáticas como o do tempo na troca de contexto de tarefas AMPI.

B.2.4 Algoritmos de balanceamento empregados

Nós investigamos vários balanceadores de carga disponíveis, que em princípio seriam apropriados para a aplicação considerada. São eles: GreedyLB, RefineCommLB,

RecBisectBfLB e MetisLB. O GreedyLB tem simplicidade como característica principal; iterativamente ele atribui a tarefa mais pesada para o processador menos carregado. Esse balanceador de carga não considera comunicação.

RefineCommLB é um balanceador que considera ambos computação e comunicação. Ele move tarefas de processadores sobrecarregados para os processadores que estão nas vizinhanças do primeiro. Ainda, ele limita o número de migrações por considerar que essa ação é custosa.

RecBisectBfLB recursivamente particiona o grafo de comunicação das tarefas usando uma enumeração *breath-first*. O número de vezes que o particionamento é feito corresponde ao número de processadores. Embora comunicação seja considerada aqui, não há garantias que a comunicação da aplicação é minimizada.

Finalmente, MetisLB é um balanceador de carga baseado no bem conhecido software de particionamento Metis. Nesse balanceador, tanto comunicação como computação são considerados.

B.2.5 Novo balanceador de carga

Nós desenvolvemos um novo balanceador baseado na curva de Hilbert. Essa curva é uma curva fractal, i.e. a curva é formada por partes que são semelhantes a curva completa. A Figura 3.6 ilustra essa curva. O algoritmo desenvolvido coloca as tarefas MPI ao longo da curva de Hilbert e, interativamente, corta essa curva de forma que cada segmento tenha uma carga aproximadamente igual. Segmentos da curva e as tarefas associadas são atribuídas aos processadores. Essa curva preserva localidade. Desse forma, tarefas que se comunicam frequentemente tendem a ficar no mesmo processador.

O algoritmo de mapeamento é descrito em (LIU; SCHRACK, 1996) e na seção 3.9. O algoritmo de corte também é mostrado nesse seção. Esses algoritmos se aplicam ao caso em que o número de tarefas é um quadrado perfeito de lado que é uma potência de dois. O algoritmo descrito em (CHUNG; HUANG; LIU, 2007) elimina essas restrições para aplicação em processamento de imagens. Aqui empregamos a mesma idéia no contexto de processamento paralelo.

B.2.6 Estratégia adaptativa

Um balanceador de carga centralizado pode ser dividido em quatro etapas:

- Juntar informação de desbalanceamento no processador que toma decisão de balanceamento;
- Executar o algoritmo de balanceamento, que aqui é aquele baseado na curva de Hilbert;
- Enviar as decisões de balanceamento para os processadores;
- Migrar tarefas.

As duas primeiras etapas devem ser executadas frequentemente, pois devem capturar o desbalanceamento assim que ele ocorrer. Entretanto, juntar informações em um processador central é um processo custoso. Para evitar esse custo, agregamos a comunicação regular da aplicação o índice de desbalanceamento. Dessa forma, o custo fica restrito ao envio de alguns poucos bytes extras numa comunicação que existiria de qualquer forma.

Além disso, a migração de tarefas também é um processo caro. Para minimizar seu impacto e evitar que desbalanceamentos pequenos disparem migrações contra-produtivas, estabelecemos um limiar, alem do qual migrações ocorrem. Esse limiar é descoberto automaticamente de forma que o usuário não precisa especifica-lo de antecipadamente.

B.2.7 Balanceador de carga distribuído

Os balanceadores de carga apresentados até agora são centralizados. Sendo assim, eles não são adequados a maquinas grandes (mesmo apenas alguns milhares de processadores). Nosso balanceador de carga pode ser implementado de forma distribuída, fato que o torna melhor escalável.

O primeiro passo dessa versão distribuída do balanceador de carga baseado na curva de Hilbert corresponde a computação da sequência de Hilbert. O algoritmo usado anteriormente pode ser empregado aqui também, pois cada tarefa executa-o de forma independente.

O segundo passo é o calculo da soma prefixa das cargas individuais das tarefas. Ao final desse passo, cada tarefa terá a carga das tarefas que a antecede. O terceiro passo é um *broadcast* da carga total feita pela última tarefa (que terá a carga total como resultado do passo anterior).

Finalmente, cada tarefa calcula a carga ideal (carga total dividida pelo número de tarefas - essa última informação é disponível a todos os processadores no início da computação) e o processador para onde essa tarefa deve migrar (soma prefixa dessa tarefa dividida pela carga ideal).

B.3 Resultados experimentais

Nessa seção descrevemos os experimentos realizados para validação do método apresentado anteriormente. Para tanto, usamos um modelo de previsão de tempo e clima que apresenta um substancial desbalanceamento de carga devido principalmente a fatores dinâmicos. Esse modelo é o Brams, que descrevemos na próxima sub-seção.

B.3.1 Modelo Brams

Brams (*Brazilian developments on the Regional Atmospheric Modeling System, RAMS*) é um modelo regional multi-propósito de previsão de tempo projetado para simular circulação atmosférica em diversas escalas. Ele é usado tanto para produção como pesquisa em várias partes do mundo. Ele se baseia no RAMS, que resolve as equações compressíveis

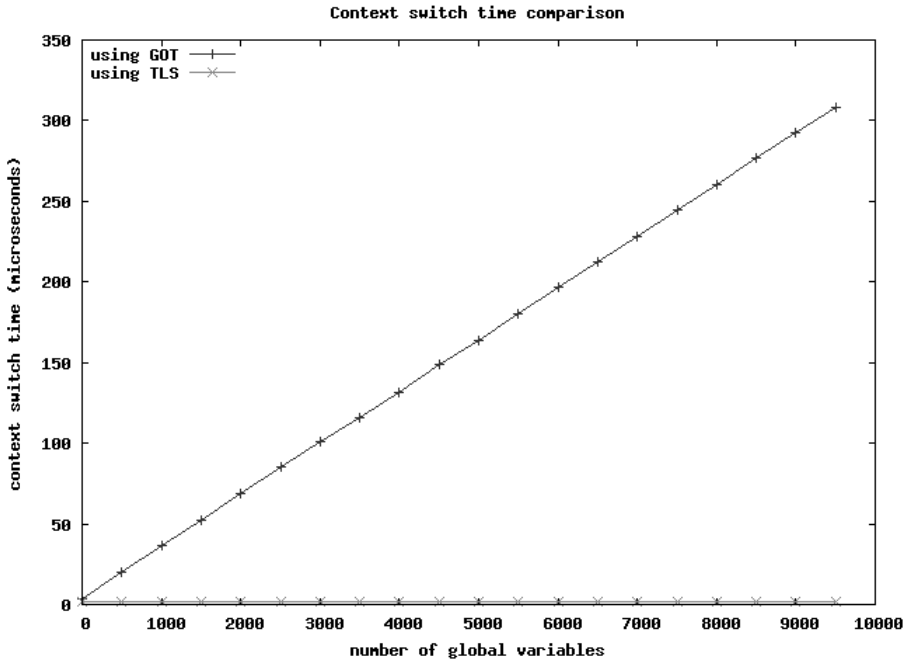


Figura B.1: Comparação do tempo de troca de contexto.

não-hidrostáticas descritas por Tripoli e Cotton (TRIPOLI; COTTON, 1982). O modelo Brams adapta o Rams para condições tropicais.

B.3.2 Primeiro conjunto de experimentos: estratégia de privatização

No primeiro conjunto de experimentos, nós comparamos o tempo de troca de contexto da nossa estratégia de privatização de variáveis. Essa métrica é importante pois toda vez que uma chamada bloqueante do MPI é feita na aplicação, uma troca de contexto é realizada. Portanto, numa aplicação com muita comunicação, esse fator tem um grande impacto na velocidade do processamento.

A Figura B.1 mostra o tempo da troca de contexto em função do número de variáveis globais. O programa nesse experimento é um *microbenchmark* que consiste de duas threads que trocam o contexto entre si continuamente. Para cada número de variáveis globais, um programa diferente é executado com o correspondente número de variáveis desse tipo. Pode-se perceber da figura que o tempo de troca de contexto da estratégia baseada em GOT é proporcional ao número de variáveis globais. Por outro lado, nossa estratégia é proporcional a um.

B.3.3 Segundo conjunto de experimentos: granularidade de rebalanceamento

A granularidade mínima de rebalanceamento da estratégia apresentada aqui é um processador virtual. Isto é, para rebalancear carga, no mínimo um processador virtual precisa ser movido de um processador real para outro. Essa granularidade pode ser muito alta dependendo do *footprint* de memória da aplicação. Para testar o peso desse fator, realizamos um experimento pequeno em que o desbalanceamento de carga é localizado. Além

	tempo de execução (s)	redução	custo da migração (s)
sem migração	850.13	-	-
2 migrações (A e B)	780.65	8%	2.64
3 migrações (A, B e C)	747.55	12%	4.77

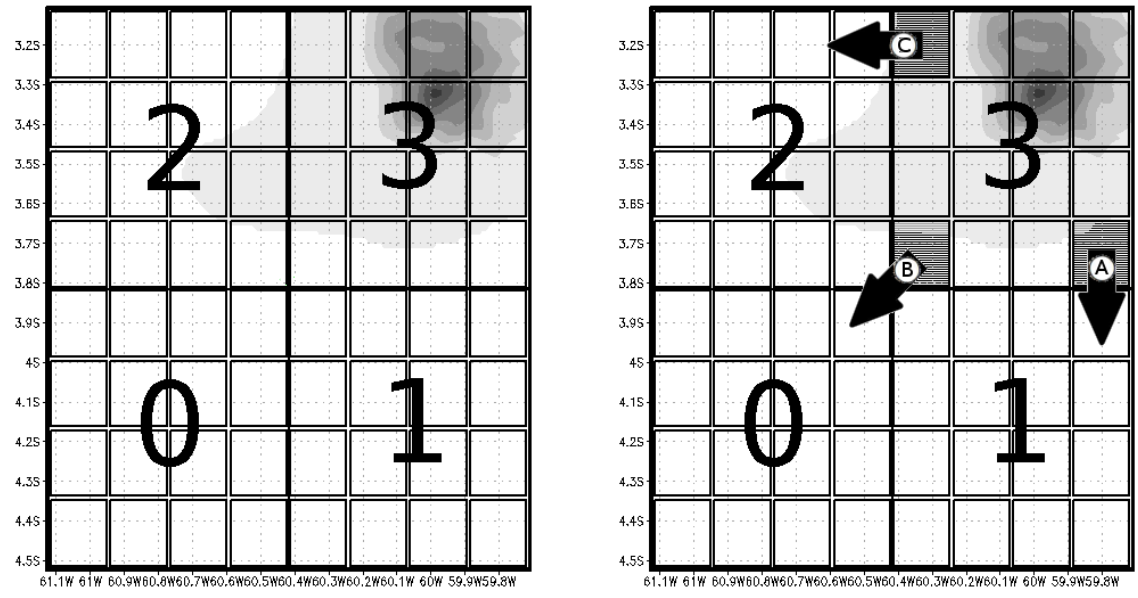
Tabela B.1: Execution time of the artificial thunderstorm case.

disso, movemos carga manualmente (i.e. sem um balanceador) para medir o impacto da migração.

A Figura B.2a mostra o estado do desbalanceamento e a decomposição do domínio. A Figura B.2b mostra os processadores virtuais que migraram. Num primeiro experimento, apenas os processadores virtuais A e B migram, e num segundo experimento o processador virtual C, além dos processadores A e B, migra. Os resultados são apresentados na tabela B.1

B.3.4 Terceiro conjunto de experimentos: balanceadores de carga centralizados

O terceiro conjunto de experimentos corresponde ao uso de balanceadores centralizados. Nesse estudo, uma tempestade na região sudeste foi usada como exemplo. Configuramos o Brams para usar uma grade de 512x512 pontos horizontais e 40 níveis verticais. A resolução foi de 1.6Km e o timestep foi 6 segundos. Realizamos uma previsão de 4 ho-



(a) Localized thunderstorm and domain decomposition

(b) Thread migration

Figura B.2: Artificial thunderstorm

Configuration	Execution Time (s)	Execution Time Reduction
No virtualization	4987.51	-
No load balancer - 1024 VP	3713.37	25.55%
GreedyLB - 1024 VP	3768.31	24.45%
RefineCommLB - 1024 VP	3714.92	25.52%
RecBisectBfLB - 1024 VP	4527.60	9.23%
MetisLB - 1024 VP	3393.12	31.97%
HilbertLB - 1024 VP	3366.99	32.50%

Tabela B.2: Load balancing effects on Brams (all experiments were run on 64 real processors)

ras. Esses experimentos foram feitos num Cray XT5, cujos nós têm dois AMD Opteron de seis cores de 2.6GHz. A rede de conexão é uma SeaStar2+. Usamos 64 processadores reais e até 2048 processadores virtuais.

Os experimentos foram divididos em três partes. Começamos com a avaliação do impacto da virtualização no Brams. Em seguida usamos migração para rebalancear carga. Vários algoritmos de balanceamento foram usados. Finalmente, investigamos a frequência de balanceamento e um limiar além do que migrações ocorrem. Desses resultados, nesse resumo, apresentamos apenas o tempo de execução com diversos algoritmos de balanceamento, que pode ser visto na tabela B.2.

B.3.5 Quarto conjunto de experimentos: limiar automático de desbalanceamento

Na Seção 4.5, descrevemos os resultados da estratégia para procura automática de limiar de rebalanceamento. Além disso, foi combinado a comunicação do índice de desbalanceamento como comunicações pré-existentes da aplicação.

Nossa estratégia para escolha automática de um limiar de balanceamento é baseada no princípio de persistência, i.e. o melhor limiar do passado é usado no futuro. Os experimentos realizados mostram que essa estratégia é melhor que limiares fixos.

B.3.6 Quinto conjunto de experimentos: balanceadores de carga distribuídos

A Seção 4.6 descreve os resultados obtidos como os balanceadores de carga distribuídos. Como apresentado na Seção 3.11, dois balanceadores foram testados: a estratégia baseada na curva de Hilbert e o balanceador baseado em difusão.

O balanceador baseado em difusão tem uma escalabilidade melhor, mas ele requer um número alto de iterações para convergir. Por outro lado, a escalabilidade do balanceador baseado em Hilbert é $log(n)$ (com n igual ao número de threads), mas que requer apenas uma iteração para rebalancear totalmente a carga.

B.4 Conclusões

Essa tese investigou balanceamento de carga em modelos meteorológicos. Esse problema é um dos grandes impedimentos para escalabilidade desse tipo de aplicação. Entretanto, as soluções atuais implicam em modificações do código original da aplicação. Esse requisito impede que aplicações reais usem esses abordagens, pois torna a implementação complexa.

Neste texto propusermos uma nova abordagem baseada em virtualização de processadores. Essa estratégia desacopla o o balanceamento de carga da aplicação. Desenvolvemos mecanismos para facilitar o uso de virtualização em aplicações reais. Além disso investigamos e desenvolvemos estratégias de balanceamento para a aplicação considerada.

Os resultados experimentais apontaram uma melhora na performance de até 30% na aplicação usada. Como trabalho futuro, pretendemos aplicar nossa estratégia em outras aplicações. Possivelmente, usaremos dados da aplicação para dirigir o balanceamento de carga.