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

CLAUDIO SCHEPKE

Exploiting Multiple Levels of Parallelism and Online Refinement of Unstructured Meshes in Atmospheric Model Application

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

Prof. Dr. Nicolas Maillard
Advisor

Porto Alegre, December 2012

CIP – CATALOGING-IN-PUBLICATION

Claudio Schepke,

Exploiting Multiple Levels of Parallelism and Online Refinement of Unstructured Meshes in Atmospheric Model Application /

Claudio Schepke. – Porto Alegre: PPGC da UFRGS, 2012.

116 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2012. Advisor: Nicolas Maillard.

1. Multi-Level Parallelism. 2. Online Refinement of Unstructured Meshes. 3. Ocean-Land-Atmosphere Model. 4. Parallel Tasks. 5. High Performance Computing. I. Maillard, Nicolas. II. Título.

"For everything there is a season,
and a time for every matter under heaven"
— ECCLESIASTES 3:1 ESV

ACKNOWLEDGEMENTS

First and foremost, I give thanks to God for all blessings.

Next, I give thanks to Diana, my wife. She assists me since the beginning of the doctoral. I feel me happy by receive her love, affection and comprehension.

I appreciate the effort of my parents for my education. My parents can barely read and they can not understand the ideas discussed in this thesis, but they always endeavored to ensure quality for my basic education.

Thank you, Nicolas, by the orientation of the work.

I appreciate also the contribution of the members of the GPPD research group, in discussions about implementation of code and result analysis. Thank you João, Stéfano, Rodrigo, Fernando, Alexandre, Cristian, Antonio, Bruno, Vinícius and Silvio, and colleagues of the rooms 305/72 (Valderi, Julio, Felipe, ...), 309/67 (Francieli, Laércio, ...) and 301/67 (Vicente, Mathias, ...).

Thanks also for the colaboration of the members of other institutions: LNCC (Carla, Pedro, Roberto and Pablo), CPTEC (Jairo) and KBS (Heiss, Jörg, Barry, Jan and Jan).

I am grateful for the assistance of all my friends. I am also happy to have made many new friends in the last years, specially in Porto Alegre.

Finally, and not least, thanks for *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq), due to the support to realize the work.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

AMR	*Adaptive Mesh Refinement*
ANSI	*American National Standards Institute*
AMPI	*Adaptative Message-Passing Interface*
API	*Application Programming Interface*
BRAMS	*Brazilian Regional Atmospheric Modeling System*
CAF	*Co-Array FORTRAN*
CMP	*Chip-level Multi-Processing*
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico
CPTEC	Centro de Previsão de Tempo e Estudos Climáticos
CPU	*Central Unity Processing*
CUDA	*Compute Unified Device Architecture*
FPGA	*Field Programmable Gate Array*
GPGPU	*General-Purpose computing on Graphics Processing Units*
GPPD	Grupo de Processamento Paralelo e Distribuído
GPU	*Graphics Processing Unit*
HPC	*High Performance Computing*
IEEE	*Institute of Electrical and Electronics Engineers*
ILP	*Instruction-Level Parallelism*
INPE	Instituto Nacional de de Pesquisas Espaciais
JVM	*Java Virtual Machine*
LNCC	Laboratório Nacional de Computação Científica
MIT	*Massachussets Institute of Technology*
MPI	*Message-Passing Interface*
MPMD	*Multiple Program Multiple Data*
OLAM	*Ocean-Land-Atmosphere Model*
OMR	*Online Mesh Refinement*

OpenMP	*Open Multi-Processing*
PGAS	*Partitioned Global Address Space*
PITAC	*President's Innovation and Technology Advisory Committee*
PUP	*Pack and UnPack*
PVM	*Parallel Virtual Machine*
RAMS	*Regional Atmospheric Modeling System*
RTS	*Run-Time System*
SMP	*Simetric Multi-Processor*
SPMD	*Single Program Multiple Date*
STL	*Standard Template Library*
TBB	*Threading Building Blocks*
TU-Berlin	*Technische Universität Berlin*
UPC	*Unified Parallel C*

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Weather forecasts for long periods of time has emerged as increasingly important. The global concern with the consequences of climate changes has stimulated researches to determine the climate in coming decades. At the same time the steps needed to better defining the modeling and the simulation of climate/weather is far of the desired accuracy. Upscaling the land surface and consequently to increase the number of points used in climate modeling and the precision of the computed solutions is a goal that conflicts with the performance of numerical applications. Applications that include the interaction of long periods of time and involve a large number of operations become the expectation for results infeasible in traditional computers. To overcome this situation, a climatic model can take different levels of refinement of the Earth's surface, using more discretized elements only in regions where more precision are required. This is the case of Ocean-Land-Atmosphere Model, which allows the static refinement of a particular region of the Earth in the early execution of the code. However, a dynamic mesh refinement could allow to better understand specific climatic conditions that appear at execution time of any region of the Earth's surface, without restarting execution. With the introduction of multi-core processors and GPU boards, computers architectures have many parallel layers. Today, there are parallelism inside the processor, among processors and among computers. In order to use the best performance of the computers it is necessary to consider all parallel levels to distribute a concurrent application. However, nothing parallel programming interface abstracts all these different parallel levels. Based in this context, this thesis investigates how to explore different levels of parallelism in climatological models using mixed interfaces of parallel programming and how these models can provide mesh refinement at execution time. The performance results show that is possible to reduce the execution time of atmospheric simulations using different levels of parallelism, through the combined use of parallel programming interfaces. Higher performance for the execution of atmospheric applications that use online mesh refinement was also provided. Therefore, more mesh resolution to describe the Earth's atmosphere can be adopted, and consequently the numerical forecasts are more accurate.

Keywords: Multi-Level Parallelism, Online Refinement of Unstructured Meshes, Ocean-Land-Atmosphere Model, Parallel Tasks, High Performance Computing.

RESUMO

Previsões meteorológicas para longos períodos de tempo estão se tornando cada vez mais importantes. A preocupação mundial com as consequências da mudança do clima tem estimulado pesquisas para determinar o seu comportamento nas próximas décadas. Ao mesmo tempo, os passos necessários para definir uma melhor modelagem e simulação do clima e/ou tempo estão longe da precisão desejada. Aumentar o refinamento da superfície terrestre e, consequentemente, aumentar o número de pontos discretos (utilizados para a representação da atmosfera) na modelagem climática e precisão das soluções computadas é uma meta que está em conflito com o desempenho das aplicações numéricas. Aplicações que envolvem a interação de longos períodos de tempo e incluem um grande número de operações possuem um tempo de execução inviável para as arquiteturas de computadores tradicionais. Para superar esta situação, um modelo climatológico pode adotar diferentes níveis de refinamento da superfície terrestre, utilizando mais pontos discretos somente em regiões onde uma maior precisão é requerida. Este é o caso de Ocean-Land-Atmosphere Model, que permite o refinamento estático de uma determinada região no início da execução do código. No entanto, um refinamento dinâmico possibilitaria uma melhor compreensão das condições climáticas específicas de qualquer região da superfície terrestre que se tivesse interesse, sem a necessidade de reiniciar a execução da aplicação. Com o surgimento das arquiteturas multi-core e a adoção de GPUs para a computação de propósito geral, existem diferentes níveis de paralelismo. Hoje há paralelismo interno ao processador, entre processadores e entre computadores. Com o objetivo de extrair ao máximo a performance dos computadores atuais, é necessário utilizar todos os níveis de paralelismo disponíveis durante o desenvolvimento de aplicações concorrentes. No entanto, nenhuma interface de programação paralela explora simultaneamente bem os diferentes níveis de paralelismo existentes. Baseado neste contexto, esta tese investiga como explorar diferentes níveis de paralelismo em modelos climatológicos usando interfaces clássicas de programação paralela de forma combinada e como é possível prover refinamento de malhas em tempo de execução para estes modelos. Os resultados obtidos a partir de implementações realizadas mostraram que é possível reduzir o tempo de execução de uma simulação atmosférica utilizando diferentes níveis de paralelismo, através do uso combinado de interfaces de programação paralela. Além disso, foi possível prover maior desempenho na execução de aplicações climatológicas que utilizam refinamento de malhas em tempo de execução. Com isso, uma malha de maior resolução para a representação da atmosfera terrestre pode ser adotada e, consequentemente, as previsões numéricas serão mais precisas.

Palavras-chave: Paralelismo Multi-Nível, Refinamento Online de Malhas Não-Estruturadas, Ocean-Land-Atmosphere Model, Tarefas Paralelas, Computação de Alto Desempenho.

1 INTRODUCTION

Numerical models have been extensively used in the last decades to understand and predict weather phenomena and climate, in daily weather forecasts as well as in researches on Global Warming (VASQUEZ, 2006), (WASHINGTON; PARKINSON, 2005). These models calculate the values of the physical conditions of the atmosphere using quantitative methods. To this end, the atmosphere is represented by a discrete space, a mesh of points obtained through the use of a domain decomposition technique, on which interactions are made during discrete time steps.

As the domain refinement increases, more points are used in the mesh representation, and consequently the forecasts become more accurate. Therefore, the impact of various physical factors, that vary in a continuous space, are more visible and taken into account during the simulation.

1.1 Mesh Resolutions of Decomposed Atmospheric Domains

Numerical climatological models represent the Earth surface through a mesh. A mesh is a piecewise approximation from a given geometry defined by a set of simpler elements, such as triangles and quadrilaterals for the two-dimensional case, and tetrahedron, prisms, pyramid and hexahedron for the three-dimensional case. The greater the number of discrete elements used to decompose a domain, the higher the **resolution** of the mesh. Thus, the resolution can be defined as the spacing between two consecutive discrete points of the mesh. The more discrete points are used, the smaller the distance between these points.

The resolution adopted for a domain strongly influences the accuracy of the results. This occurs because physical factors that vary in a continuous space are more visible and taken into account, during the simulation, only from a given resolution level. A mesh representation of regions where the topography is very irregular, for example, will not consider small differences in the Earth surface if low mesh resolution is adopted.

Figure 1.1 presents two different resolution examples of a triangular domain decomposition, where the second domain has 4 times more resolution. The choice of the mesh resolution defines the performance of the simulation and the precision of the results.

The performance of a simulation depends on the number of elements that will be processed. The greater the surface area covered by each discrete mesh element, the lesser the number of elements needed to cover all domain. Therefore, the simulation performs faster with larger elements.

The simulation precision is related to the shape and the domain size covered by each mesh element. Generally, equilateral elements are the preferred shape. On the other hand, the smaller the area covered by a discrete element of the mesh, the more accurate are the results of the forecast.

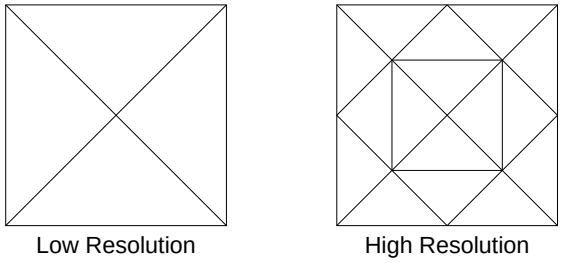

Figure 1.1: Two different mesh resolution for a triangular domain decomposition.

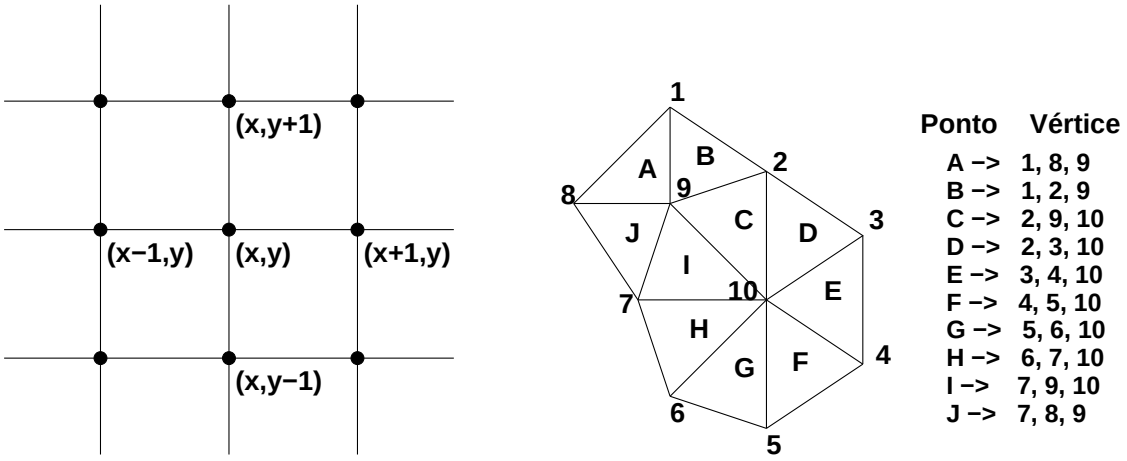

Figure 1.2: Structured and unstructured mesh examples.

Thereby the performance and the precision of the simulations are opposite requirements and it is important to ponder between them.

A further aspect related to the mesh representation is the kind of relation among the mesh points. The discretization process of a domain results in a finite representation through interconnected mesh points. This process can to result in a structured or an unstructured mesh.

In a **structured mesh**, each point has the same number of neighbor points (GALANTE, 2006). Thus, it is possible to access a neighbor point through a index of a coordinate system as, for example, a matrix structure.

An **unstructured mesh** is an irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those neighboring points. There is no communication pattern for this kind of application.

The Figure 1.2 shows examples of a structured and unstructured mesh, respectively. In the left illustration of the figure it is possible to observe the use of a cartesian system to access the neighbor points of a structured mesh. The right part of the figure illustrates the unstructured mesh case. There is a table that identify the vertices ($1, \ldots, 10$) for each element of the mesh ($A, \ldots, J$ triangles).

In this work, the interest is for unstructured meshes of triangles.

1.2 Numerical Models for Climate and Weather Forecast

In general, there are two kinds of models, differing on their domain: global (entire Earth) and regional (country, state, etc).

Global models, like GISS ModelE (SCHMIDT et al., 2006), consider the entire surface of the Earth for modeling and decomposing the domains and are normally used to predict long climatological periods (months, years). The main limitation of this approach is the computing power to execute with higher mesh resolution. Global models have normal spatial resolution of about 0.2 to 1.5 degrees of latitude and therefore cannot represent very well the scale of regional weather phenomena.

Regional models, like BRAMS (FAZENDA et al., 2011), simulate only a specific interesting piece of the Earth atmosphere. They use higher mesh resolution but they are restricted to limited area domains. Therefore, it is necessary to establish the initial entry conditions to the boundary of the domain. These conditions can be determined from previous executions of global models.

Forecasting the atmosphere conditions on limited domains demands the knowledge of future atmospheric conditions at domain borders. Because of this, the integration of initial boundary conditions with the limited area domains are necessary and this coupling is not easily done. On the other hand, local models take into account regional characteristics that are unnoticed by a global model.

A way to use the best characteristics of both approaches is to offer different levels of mesh refinement in global models. An advantage is that it is not necessary to handle boundary conditions, since the transition among different levels of refinement is done by a transparent design.

This is the case of the Ocean-Land-Atmosphere Model (OLAM) (WALKO; AVISSAR, 2008a), (SILVA et al., 2009), which provides a global grid that can be locally refined, forming a single grid. This feature allows simultaneous representation (and forecasting) of both global and local scale phenomena, as well as bi-directional interactions between scales.

Global models with local mesh refinements, like OLAM, define the mesh at the beginning of the execution, before any calculation of the physical properties at the iterative step, in a static approach. For long numerical simulations it is important that mesh refinement can be made while the code is running. Thus, spontaneous atmospheric changes that appear in restricted areas, for a given time during the execution, like storms and hurricanes, can be better investigated by applying more mesh resolution. At the same time, their impact in the whole mesh domain can be better understood.

1.3 Atmosphere Model Problem

The use of environments for High Performance Computing (HPC) has been recurrent for running applications that require a significant capacity for data processing (SCHEPKE; NAVAUX; MAILLARD, 2009), (PANETTA et al., 2007), (SIMS et al., 2000), (DONGARRA et al., 2002). Usually, these solutions are based on the development of parallel architectures (FOSTER; KESSELMAN, 2003), (WILKINSON; ALLEN, 1998), (BUYYA, 1999). The use of vector machines, multiprocessors, and currently multi-core systems have been some of the alternatives (ANDREWS, 2001). Atmospheric simulations have a significant processing load due the high number of operations usually involved. Because of this, climatological software often use programming features that allow the concurrent

execution of operations in both shared and distributed memory systems. Thus, it is possible to obtain satisfactory results in accordance with the available hardware in a reasonable period of time.

High speed execution of atmospheric models is fundamental to operational activities on weather forecast and climate prediction due to execution time constraints – there is a predefined short time window to run a model. The model execution cannot begin before input data arrives, and cannot end after the due time established by user contracts. Experiences in international weather forecast centers point to a two-hour window to predict the behavior of the atmosphere in coming days, simulating in parallel architectures (clusters).

The computational complexity of atmospheric and environmental models is $O(n^4)$, where n is the number of discrete elements resulted from a domain decomposition in relation to the latitude (or longitude) of the geographical domain of the model, if the number of vertical points and number of discrete time iterations also increases with n.[1] Operational models worldwide use the highest possible resolution that allows the model to run during the established time window on the available computer system. New computer systems are frequently selected according to the ability to run the model at even higher resolution during the available time window.

A climatological application needs also to maximize the use of existing computational resources, considering the cost and availability of hardware. In this sense, it is necessary that the application ensures good performance and scalability on parallel architectures, as well as correct results according to the physical properties design of the model and values measured in practice. Therefore, aspects such as execution environment and load distribution must be taken into account during the programming step of the application.

1.4 Objectives of the Thesis

With the introduction of multi-core processors (SHAMEEM; ROBERTS, 2005), (DONGARRA et al., 2007), computers architectures have many parallel layers. Today, there is parallelism inside a processor, among processors and among computers. This new paradigm was not foreseen by parallel applications developed in the past, like OLAM. In order to use the best performance of computers it is necessary to consider all parallel levels to distribute a concurrent application.

Parallel programming interfaces are generally specific to one level of parallelism. Currently, there is not a single programming interface able to explore all levels of parallelism. To perform this, it is necessary to use two or more combined programming interfaces, like Message-Passing methods for Distributed Memory Systems and resources for creating and manipulating threads in Shared Memory Systems. An application developed with a specific programming interface for a determined parallel architecture model is not easily migrated to another architecture or programming interface.

Based on the context described before, this thesis proposes the use of mixed programming interfaces as solution to provide multilevel parallelism for implementations of atmospheric models. For parallel applications, the load distribution in each processing unit is done in order to maximize the parallel performance for a determined architecture. The use of mixed programming interfaces reduces the total execution time of simulations through the maximization of the use of the available processing units. This is achieved when all processors or cores are continuously executing. However, data dependencies arising from

[1] In some models the complexity can be simplified to $O(n^3)$, depending on the method of resolution of the equations that model the atmosphere.

the parallelization of atmospheric models can limited the performance. To overcome this situation, the scheduling of the parallel tasks and the definition of the granularity of each task can contribute for the best load distribution.

Each programming interface abstracts the concurrent execution using different ways to express the parallel task (processes, threads, ...). A parallel task in an atmospheric model is defined by data structures that represent one or more points of the mesh and numerical operations. Each task is composed by data sets of physical properties associated to the points of the mesh, and functions that manipulates the data structures according to the code of the model. These functions are formed by climatological interactions and are iteratively called.

The unstructured meshes employed in the atmospheric model used in this work can be also refined at runtime, in order to increase the precision of the forecasts. We provide and evaluate an Online Mesh Refinement (OMR)[2] implementation to increase the resolution of part of the parallel distributed domain that represents the Earth atmosphere, when special atmosphere conditions are registered during the execution of an atmosphere model. Therefore, more computation is only required for the refined mesh ensuring low performance impact and more precision for the simulations.

The contribution of this thesis includes both, the efficient use of multilevel parallelism and dynamically modification of the domain representation through OMR calls. Multilevel parallelism is explored by the execution of concurrent tasks according to available resources. We provide also arguments that online mesh refinement is better for the application of parallel climatological models. Thus, the development of high-performance applications like climatological applications could be simplified and improved by better exploiting the available hardware resources.

1.5 Text Organization

This thesis is divided in 8 chapters. The reminder of this text is organized as follows:

- **Chapter 2 - Ocean-Land-Atmosphere Model**: Relates all necessary aspects to understand the atmospheric model used as case study, including the description of the domain representation, algorithm, data structures and parallelization of the code.

- **Chapter 3 - High Performance Computing Challenges:** Describes a bibliographic revision about High Performance Computing, pointing challenges to explore multilevel parallelism in computer architectures, and to define a parallel task with currently parallel programming tools.

- **Chapter 4 - Scalability Study of Static OLAM**: Shows experimental results obtained with OLAM original implementation tested in a parallel execution environment. The evaluation of the measured results with static refinement demonstrates the limits of performance of the application.

- **Chapter 5 - Online Local Mesh Refinement**: Presents an implementation of a run time mesh refinement and the performance evaluation results of this implementation in a OLAM prototype. The chapter shows also how it is possible to improve

[2]In this work we consider the word online as a synonymous for dynamic or runtime mesh refinement. Online is the term frequently used in the context of job scheduling in opposition to the static or the offline scheduling approach.

performance when new data elements are added in the computation, after an online mesh refinement call.

- **Chapter 6 - Multi-Level Parallelism**: Describes the implementation of task parallelism, by different parallel programing interfaces, in order to extract parallelism of multi-core, many-core and multiprocessors architectures. Experimental tests using mixed parallel programming interfaces are made, analyzing it impact in atmospheric simulations.
- **Chapter 7 - Scalability Evaluation of OLAM Multi-Level Parallelism**: Provides a performance evaluation (execution time and scalability measurement results) of the multi-level implementation of OLAM, on a high performance environment.
- **Chapter 8 - Conclusions**: Discusses the conclusion of this thesis, relating objectives, implemented solutions and obtained results. Through this relation is possible to point some future works.

2 OCEAN-LAND-ATMOSPHERE MODEL

Ocean-Land-Atmosphere Model (OLAM) was chosen as background to the development of this thesis. This model is used to forecast weather and climate in research and forecast centers. OLAM is also a good example of a large real application of domain decomposition because it uses a significative number of discrete elements to represent the structure of the atmosphere of the Earth and, consequently, requires a large amount of memory and processing.

This chapter describes OLAM: its main features, equations, domain decomposition approach and discrete representation of the domain, and the resource for mesh refinement. The algorithm, data structures to represent the code, and parallel decomposition of the data structures is also presented in order to show all aspects of the model.

The concepts discussed in this chapter are necessary to understand the implementations, tests and results presented and proposed in the remainder of this thesis.

2.1 Main Features

OLAM was developed by Roni Avissar and Robert Walko at Duke University. This model extends features of the Regional Atmospheric Modeling System (RAMS) to cover a global domain (PIELKE; AL., 1992). OLAM uses many functions from RAMS, including physical parametrization, data assimilation, initialization methods, logic and coding structure, and I/O formats (WALKO; AVISSAR, 2008b) (AVISSAR; PIELKE, 1989). A global domain expands widely the range of atmospheric systems and scale interactions that can be represented in the model. This was the primary motivation for developing OLAM.

OLAM introduces a dynamic approach of domain decomposition based horizontally on a global geodesic grid discretization with triangular mesh cells, and vertically through the height levels of the atmosphere, forming vertically-stacked prisms of triangular bases. It also uses a finite volume discretization of the full compressible nonhydrostatic Navier Stokes equations (MARSHALL et al., 1997). These equations formalize conservation laws for mass, momentum, and potential temperature, and numerical operators that include time splitting for acoustic terms.

Local mesh refinement can be applied to cover specific geographic areas with higher resolution. The mesh points that represent these areas are subdivided cyclically while the expected mesh resolution is not achieved. Each cyclical division doubles the resolution. The global grid and its refinements define a single grid, as opposed to the usual nested grids schemes of regional models. The grid points, which represent a more refined area, do not overlap the grid points that represent the global domain - they substitute them.

2.2 Equations

OLAM dynamic equations are:

- Momentum conservation (component i)

$$\frac{\partial V_i}{\partial t} = -\nabla.\left(v_i \vec{V}\right) - \left(\nabla p\right)_i - \left(2\rho\vec{\Omega} \times \vec{v}\right)_i + \rho g_i + F_i \tag{2.1}$$

- Total mass conservation

$$\frac{\partial \rho}{\partial t} = -\nabla.\vec{V} + M = -\frac{\partial U}{\partial x} - \frac{\partial V}{\partial y} - \frac{\partial W}{\partial z} + M \tag{2.2}$$

- Energy conservation

$$\frac{\partial \rho\Theta}{\partial t} = -\nabla.\left(\Theta\vec{V}\right) + H \tag{2.3}$$

- Scalar mass conservation

$$\frac{\partial (\rho s)}{\partial t} = -\nabla.\left(s\vec{V}\right) + Q \tag{2.4}$$

- State equation

$$p = \left[\left(\rho_d R_d + \rho_v R_v\right)\theta\right]^{\frac{C_p}{C_v}} \left(\frac{1}{p_0}\right)^{\frac{R_d}{C_v}} \tag{2.5}$$

- Total density

$$\rho = \rho_d + \rho_v + \rho_c \tag{2.6}$$

- Momentum definition

$$\vec{V} \equiv \rho\vec{v} \tag{2.7}$$

- Potential temperature

$$\theta = \Theta\left[1 + \frac{q_{lat}}{C_p\ max(T, 253)}\right] \tag{2.8}$$

In these equations, $\vec{v}$ and $\vec{V} = \rho\vec{v}$ are velocity and momentum vectors, and $\vec{g}$ and $\vec{\Omega}$ are the Earth's gravity and angular velocity vectors. Subscript i represents a vector component in the x_i direction, t is the time, p is the pressure, and θ is the potential temperature. C_p and C_v are the specific heat of a dry air at a constant pressure and constant volume, R_d and R_v are gas constants for dry air and water vapor, and p_0 is a pressure reference equal to $10^5 Pa$. Total density ρ is given by the sum of the densities of dry air, water vapor, and liquid plus condensate ice.

The scalar variable s represents the specific density or concentration (relative to ρ) of any prognostic scalar quantity, such as various classes of ice and liquid hydrometeors and aerosols. F_i, H, M, and Q are forcing terms for momentum, internal energy, mass, and scalar fields, respectively. These terms represent processes such as radiative transfer, microphysical phase changes, surface fluxes, and/or optional nudging to observational data, as applicable to each equation.

The ice-liquid potential temperature is used in OLAM as the prognostic internal energy variable. It has the desirable property of being nearly constant in a parcel for processes of transport and internal phase change. It is empirically related to potential temperature where q_{lat} is the latent heat required to vaporize any presented liquid and ice water, and T is the air temperature.

Applying Gauss Divergence Theorem and integrate over Finite Volumes:

$$\int \nabla.\vec{\Phi} d\Psi = \oint_{\sigma} \vec{\Phi}.d\vec{\sigma} \tag{2.9}$$

We have the discretized equations:

$$\frac{\partial}{\partial t}\int V_i d\Psi = -\oint (v_i \vec{V}).d\vec{\sigma} - \int \frac{\partial p}{\partial x_i} d\Psi - \int \left(2\rho\vec{\Omega}\times\vec{v}\right)_i d\Psi + \int \rho g_i d\Psi + \int F_i d\Psi \tag{2.10}$$

$$\frac{\partial}{\partial t}\int \rho d\Psi = -\oint \vec{V}.d\vec{\sigma} \tag{2.11}$$

$$\frac{\partial}{\partial t}\int \rho\Theta d\Psi = -\oint \left(\Theta\vec{V}\right).d\vec{\sigma} + \int F_\Theta d\Psi \tag{2.12}$$

$$\frac{\partial}{\partial t}\int \rho s d\Psi = -\oint \left(s\vec{V}\right).d\vec{\sigma} + \int F_s d\Psi \tag{2.13}$$

We can also represent the closed integral as sum over faces:

$$\oint \left(\Phi\vec{V}\right).d\vec{\sigma} = \sum_j \left[\int \left\{\Phi_j \vec{V}_j\right\}.d\vec{\sigma}_j\right] \equiv \sum_j \left[\left(\overline{\Phi}_j \overline{V}_j + SGS\left\{\Phi_j, V_j\right\}\right)\sigma_j\right] \tag{2.14}$$

So, the conservation equations in the discretized finite-volume form are:

$$\frac{\partial \overline{V}_i}{\partial t}\Psi = -\sum_j \left[\left(\overline{v}_{ij}\overline{V}_j + SGS\left\{v_{ij}, V_j\right\}\right)\sigma_j\right] - \frac{\partial \overline{p}}{\partial x_i}\sigma_i \nabla x_i - \left(2\overline{\rho}\vec{\Omega}\times\overline{\vec{v}}\right)_i \Psi + \overline{\rho} g_i \Psi + \overline{F}_i \Psi \tag{2.15}$$

$$\frac{\partial \overline{\rho}}{\partial t}\Psi = -\sum_j \left[\overline{V}_j \sigma_j\right] \tag{2.16}$$

$$\frac{\partial \overline{\rho\Theta}}{\partial t}\Psi = -\sum_j \left[\left(\overline{\Theta}_j \overline{V}_j + SGS\left\{\Theta_j, V_j\right\}\right)\sigma_j\right] + \overline{H}\Psi \tag{2.17}$$

$$\frac{\partial \overline{\rho s}}{\partial t}\Psi = -\sum_j \left[\left(\overline{s}_j \overline{V}_j + SGS\left\{s_j, V_j\right\}\right)\sigma_j\right] + \overline{Q}\Psi \tag{2.18}$$

2.3 Global Grid Structure

OLAM's global computational mesh consists of spherical triangles, a type of geodesic grid that is a network of arcs that follow great circles (like the equator line) on a sphere. The geodesic grid offers important advantages over the commonly used latitude-longitude grid. It allows approximately uniform mesh size over the globe, and avoids singularities and grid cells with very high aspect ratio near the poles.

OLAM's grid construction begins from an icosahedron inscribed in the spherical Earth, as is the case for most others atmospheric models that use geodesic grids. (SILVA

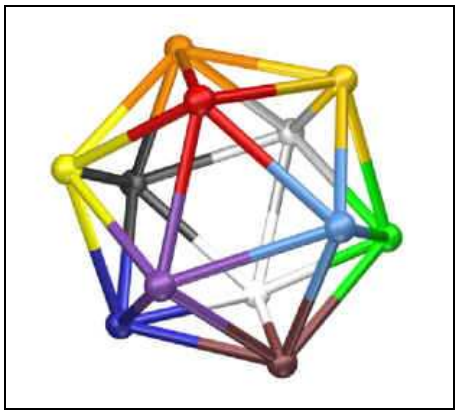

Figure 2.1: Example of icosahedron.

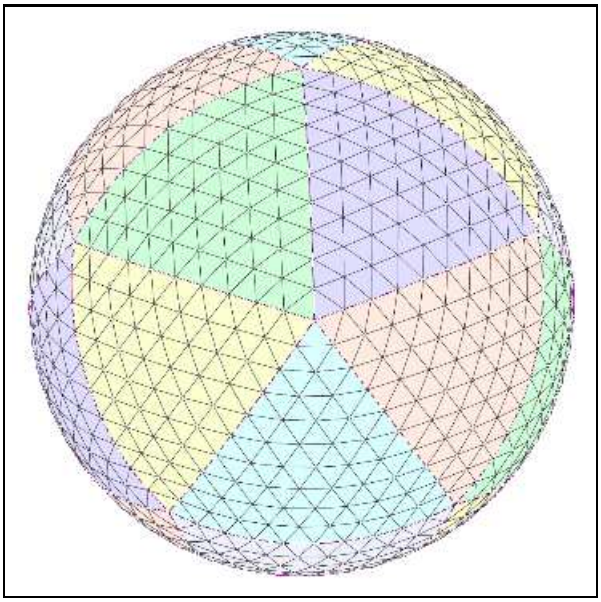

Figure 2.2: OLAM subdivided icosahedral mesh.
(WALKO; AVISSAR, 2008a).

et al., 2009). An icosahedron is a regular polyhedron that consists of 20 equilateral triangle faces, 30 triangle edges and 12 triangle vertices, with 5 edges meeting at each vertex. Figure 2.2 shows an example of icosahedron.

The icosahedron representation used in OLAM is oriented such that one vertex is located at each geographic pole, which places the remaining 10 vertices at latitudes of $\pm tan^{-1}(1/2)$. Uniform subdivision of each icosahedral triangle into $N \times N$ smaller triangles, where N is the number of vertices divisions of each triangle, is performed in order to construct a mesh of higher resolution for any desired degree. The uniform subdivision adds $30(N^2-1)$ new edges to the original 30 and $10(N^2-1)$ new vertices to the original 12, with 6 edges meeting at each new vertex. All newly constructed vertices and edges are then projected radially outward the sphere to form geodesics.

Figure 2.2 shows an example of the OLAM subdivided icosahedral. In the figure the mesh is generated with $N = 10$. The dark lines indicate the edges of the initial icosahedron.

The projection causes the deviation of the majority of the triangles from the equilateral shape, which is impossible to avoid (WALKO; AVISSAR, 2008a). However, the numerical accuracy of the computational mesh can be guaranteed, adjusting the projection of the triangles. This is achieved by relocating the vertices on the sphere. For this, forces are applied to each edge located between endpoints vertices. This force is a linear function and depends on the length of each edge. The length of equilibrium is defined by

$$d_{eq} = \frac{2\pi R\epsilon}{5N} \tag{2.19}$$

Where R is the radius of the Earth and ϵ is a coefficient of adjustment. When ϵ is 1, the length of balance is approximately equal to the average overall length of the edge. Vertices can be moved while the sum of the forces is zero for each vertex. OLAM uses this solution to solve numerically the equations of equilibrium of forces through iterative methods.

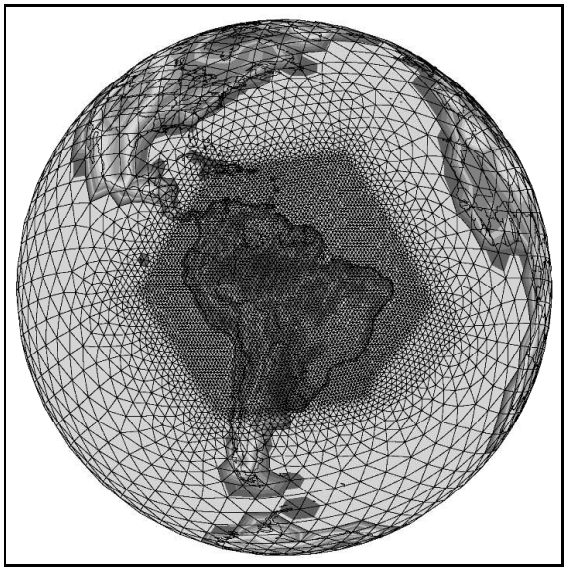

Figure 2.3: Local mesh refinement applied to a selected part of the globe.

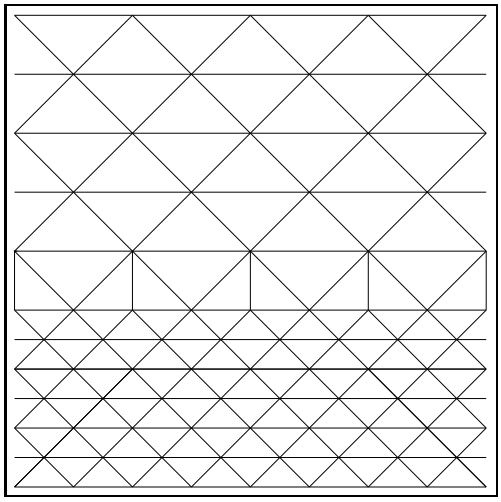

Figure 2.4: Local mesh refinement transition from coarse to fine resolution.

2.4 Mesh Refinement

Building a global mesh, as described above, enables a 2D structured indexing for each element of the grid. However, a logical structure restricts the possibility of mesh topologies. Local mesh refinement is only possible if the mesh type is unstructured[1] Because of this, OLAM uses an unstructured approach and represents each grid cell with a single horizontal index (WALKO; AVISSAR, 2008a). So, required information on local grid cell topology is stored and accessed by linked lists.

If local horizontal mesh refinement is required, it is performed after the step of construction of the global mesh. The refinement follows a three-neighbors rule: each triangle must share the length of each edge with exactly three others triangles. The range of possible topologies that obey this rule is enormous.

An example of local mesh refinement is illustrated in Figure 2.3, where the resolution is exactly twice that of the original resolution. This is achieved by subdividing each previously triangle into 2 smaller triangles. For this purpose, auxiliary edges were inserted at the boundary between the original and refined regions in order to preserve the rule of the three neighboring triangles for each triangle.

A transition from coarse to fine resolution is achieved by use of vertices with more than 6 edges on the coarser side and vertices with fewer than 6 edges on the finer side of the transition, like can be seen in Figure 2.4. In this example each auxiliary line connects a vertex that joins 7 edges with a vertex that joins 5 edges. However, it is not necessary that these vertices are concentrated along a band. A more gradual refinement of the mesh can be obtained by distributing these vertices in a sparse way over a larger area.

A more intensive refinement can be obtained using vertices with more than 7 and less than 5 edges. However, this would force the existence of triangles with acute degrees that may reduce the accuracy of numerical simulations. Moreover, in OLAM a spring adjustment is applied after the step of mesh refinement.

[1] Unstructured grids require a list of the connectivity which specifies the way a given set of vertices make up individual elements.

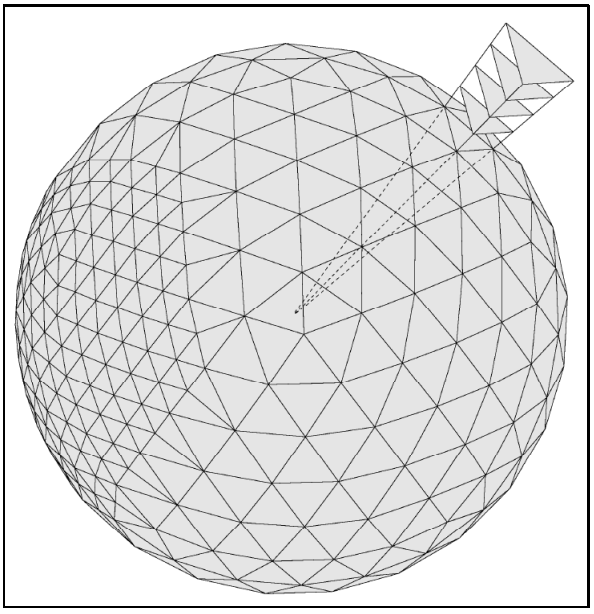

Figure 2.5: Projection of a surface triangle cell to larger concentric spheres in order to generate multiple vertical model levels.

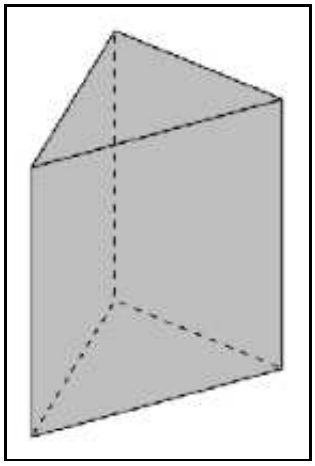

Figure 2.6: Example of a prism-shaped grid cell.

2.5 Vertical Level Definition

The final step of the mesh construction is the definition of the vertical levels. To do this, the lattice of the triangular cells surface is projected radially outward from the center of the Earth to a series of concentric spheres of increasing radius (ADCROFT; HILL; MARSHALL, 1997). The vertices on consecutive spheres are connected with radial line segments as can be see in the left image of Figure 2.5. This creates prism-shaped grid cells having two horizontal faces (perpendicular to gravity) and three vertical faces, like can be see in the Figure 2.6.

The horizontal cross section of each grid cell and column expands gradually according to the height growth. The vertical grid spacing between spherical shells may vary itself and usually is made to expand with the increasing of the height (WALKO; AVISSAR, 2008b). In OLAM, it is possible to define a static input vector with the vertical grid spacing or define an initial value and a rate value to increase at each vertical level of the grid. The first vertical grid created is generally defined $30\ km$ above the Earth surface, where atmospheric pressure is less than $1\ mb$.

OLAM discretization scheme uses a staggered grid for unstructured mesh (WENNEKER; SEGAL; WESSELING, 2002). The scalar properties are defined and considered in the center of the triangles and the normal component of velocity for each edge of the triangle is set in the middle of each edge. The numerical formulation allows the non perpendicularity between the lines that connect the barycenter of two adjacent triangles and the common edge between two triangles. The volume control of movement in horizontal surfaces are similar to those for scalars. This is accomplished by setting the volume control of movement of any triangle edge to the sum of the volume control of the mass of two adjacent triangles. This means that it is not necessary to obtain a spatial average for the mass flow across the dynamic volume control surfaces.

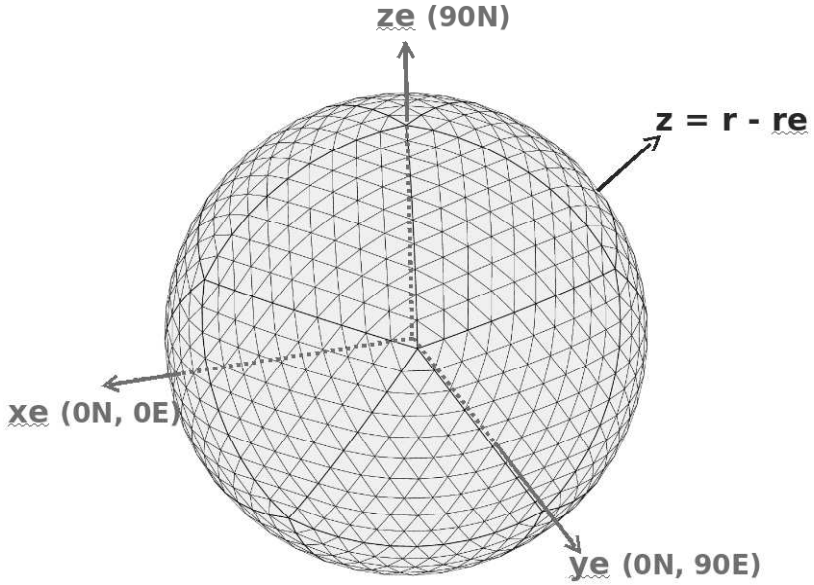

Figure 2.7: Cartesian coordinate system with origin at the center of the Earth.

2.6 Coordinate System

OLAM uses a rotating Cartesian system with origin at the Earth's center, z-axis aligned with the north geographic pole, and x- and y-axes intersecting the equator at $0\ deg$ and $90\ deg$ E. longitude, respectively, as shown in the image of the Figure 2.7.

The three-dimensional geometry of the mesh, particularly relating to terms in the momentum equation and involving relative angles between proximate grid cell surfaces, is worked out in this Cartesian system. The procedure involves computation and storage of the unit vector normal to each surface, and solution of linear systems that contain the unit vector coefficients.

2.7 Algorithm

The implementation of OLAM involves several steps and can by divided in three major parts: the parameter initialization, the atmosphere time state calculation and the output writing results.

The first part of the code involves the pre-processing, where settings are read and applied for memory allocations and the processing of the information of terrain, vegetation, soil and sea.

The remainder of the algorithm consists of an iterative step, involving the physical parametrization. The physical parametrization is similar to the parametrization applied in the RAMS model and includes the radiation transfer, micro-physics, bio-physical schemes, turbulence and convective clouds like cumulus clouds. In this iterative part, further information calculated in the pre-processing step are inserted. At the end of each iteration, the update of the time elapsed is made.

After the iterative step, and before the end of the program, some results are written in specific files, storing values of the physical conditions of the atmosphere to a determined time.

Algorithm 2.1 OLAM algorithm.

```
Initialization;
Input Files (ATM/LAND/SEA) Read;
Grid Configuration/Domain Decomposition;
Variables Memory Allocation;
Pre-processing initial state calculation;
Plot and History Files Initialization;
Initialization Time measure;
Do loop for each time step;
    Atmosphere time state calculation;
    Send frontier variables to neighbors;
    Times step Time measure;
Write atmosphere state on disk;
Barrier; Output Time measure;
```

2.8 Data Structures Used for the Discrete Representation of the Domain

This section describes and illustrates the main data structures used in OLAM code.

OLAM discretization of the horizontal domain of the atmosphere is made by decompose the Earth surface in triangles. A W triangle is formed by $3\,M$ vertices and $3\,U$ edges. Thus, 3 data structures are used to represent the relation of vertices (m), edges (u) and triangles of the domain (W). These 3 data structures are $itab\_m\_vars$, $itab\_u\_vars$ and $itab\_w\_vars$, and are represented in Algorithm 2.2, Algorithm 2.3 and Algorithm 2.4.

Algorithm 2.2 Data structure itab_m_vars.

```
typedef struct {
    int ntpn; //number of U edges and W triangles neighbors of this M vertice M point
    int iw[maxtpn]; //array of W triangles neighbors of this M vertice point
    int iu[maxtpn]; //array of U edges neighbors of this M vertice point
    int imglobe; //global index of this M vertice point (in parallel case)
    double arm; //polygon area bounded by W triangles around this M vertice point
} itab_m_vars;
```

Algorithm 2.3 Data structure itab_u_vars.

```
typedef struct {
    int im1, im2; //neighbor M vertices of this U edge point
    int iu1, iu2, iu3, iu4, iu5, iu6; //neighbor U edge points
    int iu7, iu8, iu9, iu10, iu11, iu12; //neighbor U edge points
    int iw1, iw2, iw3, iw4, iw5, iw6; //neighbor W triangle points
    int irank; //rank of the parallel process at this U edge point
    int iuglobe; //global index of this U edge point (in parallel case)
    int mrlu; //mesh refinement level of this U edge point
} itab_u_vars;
```

All three data structures have information about neighbors vertices (m), edges (u), and triangles (w). Furthermore these data structures keep the parallel process ranking and global index of the respective point (m, u or w)

Algorithm 2.4 Data structure itab_w_vars.

```
typedef struct {
      int im1, im2, im3; //neighbor vertices M of this W triangle point
      int iu1,iu2,iu3,iu4,iu5,iu6,iu7,iu8,iu9; //neighbor U edge points
      int iw1, iw2, iw3; //neighbor W triangle points
      int irank; //rank of the parallel process at this W triangle point
      int iwglobe; //global index of this W triangle point (in parallel case)
      int mrlw, mrlw_orig; //mesh refinement level of this W triangle point
} itab_w_vars;
```

These data structures are important for many segments of the code. They are used to define the global grid of OLAM, in the domain decomposition and local mesh refinement, to relate with other data structures involving physical properties, and to control the data exchanges among the processes in the iterative step.

Figure 2.8, Figure 2.9 and Figure 2.10 illustrate the relation of vertices, edges, and triangles from the data structures $itab\_m\_vars$, $itab\_u\_vars$ and $itab\_w\_vars$ presented in Algorithm 2.2, Algorithm 2.3 and Algorithm 2.4), respectively.

Figure 2.8 presents all vertices iu and triangles iw neighbors of a edge im.

In the Figure 2.9, the arrow indicates the positive direction of a vertice U_i. The area formed by the triangles $iw1$ and $iw2$ is the control volume. Another numbered vertices iu indicate the localization of the 12 neighbors of the vertice U_i, where the values of u_i and/or U_i are necessary to estimate U_i in a iu point. The iw numbered triangles indicate the localization of the 6 w neighbors, where the value of ρ and p are necessary.

In Figure 2.10, iw is the control volume. Numerated iu indicate the localization of flux transport U_i of the control volume.

Another way to view these structures is given in Figure 2.11. In this figure:

- **A** is the control volume for scalar quantities, a prism-shaped single grid cell. The normal momentum component is defined and prognosticate at each of the five faces.
- **B** is the control volume for horizontal momentum, comprised of two prism cells. The prognosticate momentum in the control volume is also the flux across the darkened face between the two prisms.
- **C** is the control volume (light gray) for the vertical momentum component. The vertical momentum in the control volume is also the flux across the darkened face between the upper and between the lower prism.

2.9 Parallelization of the Model

OLAM was developed in FORTRAN 90 and parallelized with Message-Passing Interface (MPI) (GROPP et al., 1996) to Single Program Multiple Data (SPMD) model.

All MPI processes have initially the original representation of the grid domain and its data structures created, as described in Section 2.3. Next, if the execution is set to parallel run, each process defines its sub-domain. Data are reallocated after the definition of the sub-domain in each process, so that only the sub-domain is kept in memory.

The steps described previously are realized by the functions $para\_decomp()$ and $para\_init()$.

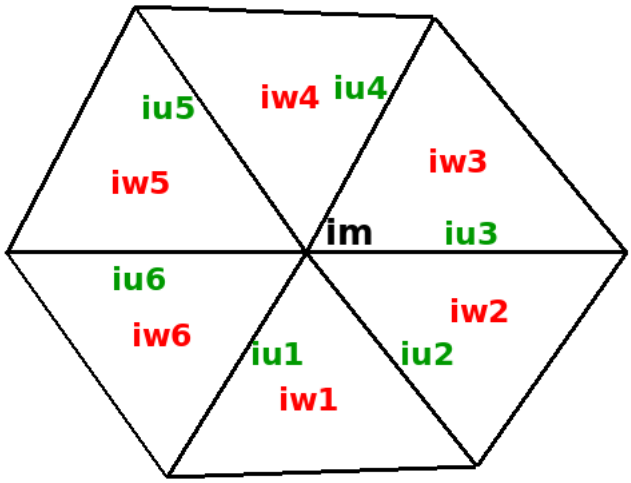

Figure 2.8: Polygon formed by w boundary points around a m vertex.

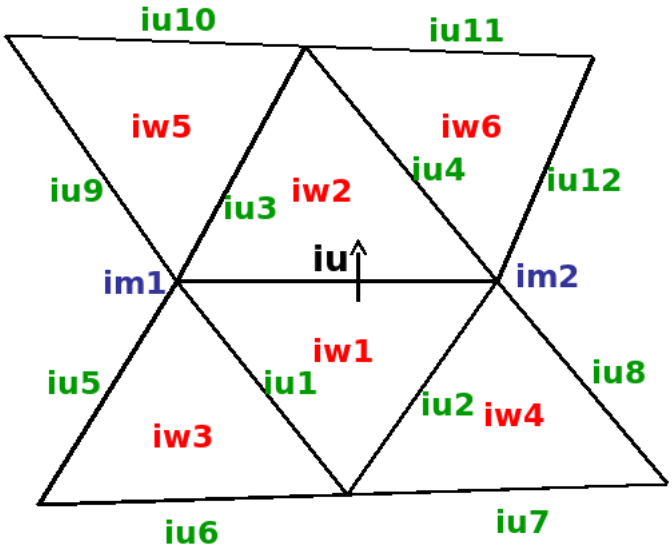

Figure 2.9: Computational horizontal stencil for a value of U_i at a iu localization.

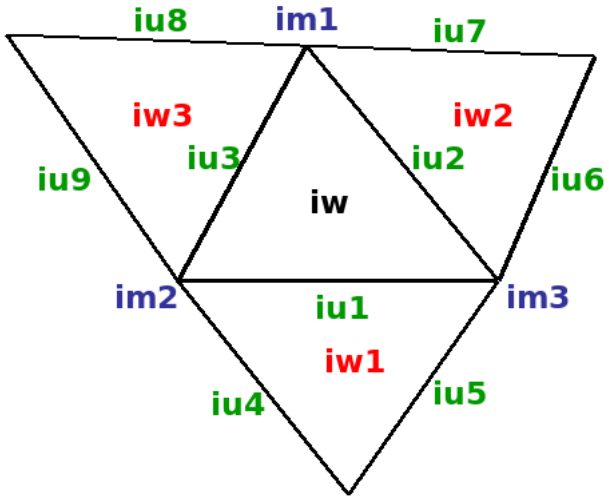

Figure 2.10: Horizontal computational stencil to a ρ value in iw localization.

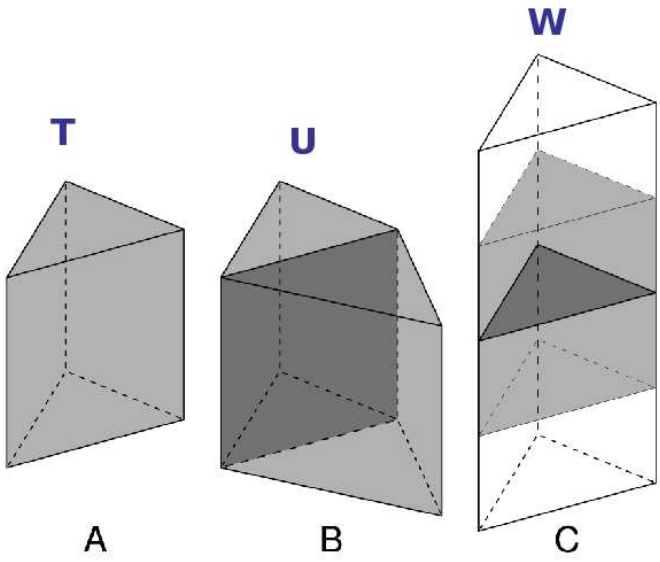

Figure 2.11: Control volume.

The first function defines a data structure in each process, indicating the sub-domain that will be set to each process.

The second function is responsible in each process to:

- Deallocate the global domain and allocate the new sub-domain;
- Fill all data structures of the sub-domain that belong to the indicated process rank;
- Prepare the data structures used by the communication functions in the iterative step of the execution of the code. These structures indicate the necessary elements from other data structures that need to be updated. Each process maintains a list of processes to send and to receive the updated elements.

Figure 2.12 presents a graphic visualization of the global domain decomposed in 18 and 180 processes, respectively. Each distinct tone represents the domain of a process. We can see that the covered global domain is smaller for processes computing on a refined region (Amazon region), although the data structure points are balanced distributed among the processes.

The iterative step will process after the parallel grid domain decomposition and data structures redefinition. In this step, there are data exchange among neighbor processes through asynchronous messages to update physical properties of the submeshes' border.

The communication among processes in the iterative step of the atmosphere simulation occurs basically using 3 encapsulated send and its 3 respectively receive functions. The first and the second group of communication functions, $mpi\_send\_u()$ - $mpi\_recv\_u()$ and $mpi\_send\_uf()$ - $mpi\_recv\_uf()$, are responsible to the exchange of data of physical variables associated to edge elements of the mesh. The third group, $mpi\_send\_w()$ - $mpi\_recv\_w()$, are related to the communication of the physical properties associated to triangle elements of the mesh. In all communication group of functions, and specially in the last, input parameters specify the kind of data that need to be send or receive.

Data exchange in the iterative step of OLAM occurs according to tables previously defined. Before the iterative step of the model, buffers are allocated to store data to

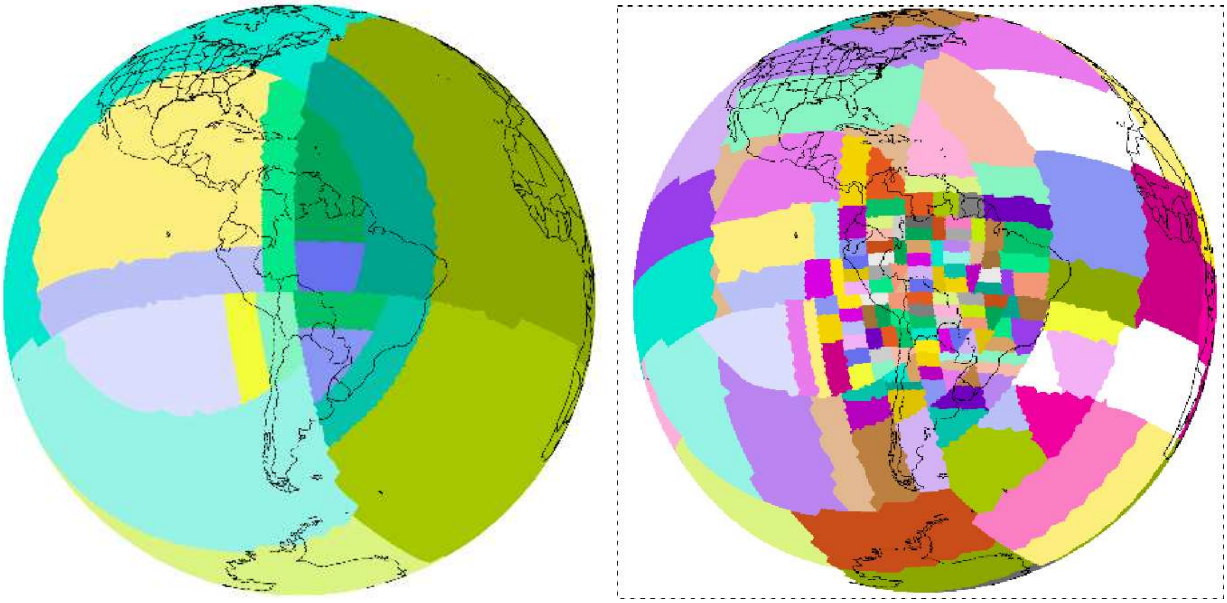

Figure 2.12: Global domain divided in 18 and 180 processes.

the send and receive messages. At this part of the algorithm is also defined which data structures need to be updated and how this process will be made. These information are stored in special data structures and searched when the encapsulated communications functions are called.

2.10 Final Considerations

This chapter described details of the OLAM, including aspects of domain decomposition, algorithm and parallelization.

OLAM represents the atmospheric domain through a unstructured mesh. Consequently, each operation on a discrete element of the mesh needs to use auxiliary data structures to identify it neighbor elements. Because of this, the codification of the model and its parallelization demand more programming efforts.

OLAM seems a good example of high performance application to evaluate multi-level parallel architectures and to be parallelized through different approaches. It demands many computational resources for processing a simulation. Moreover, the use of different mesh refinement can be well explored in architectures with different levels of parallelism.

The model is currently parallel implemented with MPI. In order to explore new parallel architectures other parallel programming interfaces could be used in parallel implementations. Multiple programming interfaces could be also combined to provide the concurrently execution of the algorithm. Thus, the elements of the data structures associated to the mesh could be parallelized in two levels.

Next chapter presents aspects related to parallel architectures and programming interfaces that can be used to compute and parallelize high performance applications. We discuss the notion of parallel task and how is possible to explore multiple levels of parallelism.

3 HIGH PERFORMANCE COMPUTING CHALLENGES

For many years multicomputers have been the prevalent architecture adopted to develop high performance applications through parallel programs. This was the obvious solution to match processing capacity, using single and/or multiple processors. At the same time many tools were produced to abstract the programming process of multicomputer systems.

New computer architectures were produced in recent years improving intra-chip parallelism. This form of concurrency proposes a new kind of parallelism, which is already adopted in the development of applications. However many pre-existing applications are not prepared to use these architectures. Because of this, new challenges appear in the High Performance Computing context (DONGARRA, 2004). Some of these challenges will be investigated in this chapter.

3.1 Parallel Applications

Computer Science has introduced a revolution in scientific research. It is considered as the "third pillar", along with theory and experimentation, that supports scientific research (PITAC Report to the President, 2005). Computer simulation has been one of the alternatives to find the numerical solution of scientific or industrial applications, modeling complex systems (LUCQUIN; PIRONNEAU, 1998).

Simulation is a viable alternative, once to build a prototype or to create a real situation is not always possible due the costs involved, the risks that the experiment could result or physical inviability to reproduce the tests. Examples of simulations can be found in several areas, such as hydrodynamics, with the flow in aqueous media and the modeling of climate and weather, health, through the representation of human organs and tissues, aerodynamics of vehicles, to model cars, trucks and aircraft, and virtual reality environments, like games or situations of human risk (SCHEPKE; MAILLARD, 2007; SOUTO et al., 2007; XAVIER et al., 2007; FANG et al., 2002; EXA CORPORATION, 2008; LOCKARD; LUO; SINGER, 2000).

The previously cited applications demand very high processing power. For example, a operational forecast of a typical hurricane requires both ultra-high-resolution of gradients across the eye-wall boundaries (at $1\ km$ or less), and correctly representation of the turbulent mixing process (at $10\ m$ or less) (BERGMAN et al., 2008).

For this problem, considering an atmospheric domain of:

- 100 square kilometer of horizontal area,
- 10 meter of horizontal grid spacing resolution,

- 150 vertical levels,
- 60 milliseconds of time step model.

This results a mesh with 15 billion of finer decomposed elements.

At a sustained petaflop/second on $100,000$ processors, such a computation consumes about 18 machine hours per simulated day and takes up about 100 MB per task of data not counting buffers, executable size, operating system calls, etc (10 TB of main memory for the whole application in aggregate). The computation generates a data set of 241.8 Terabytes (TB), or 43.2 TB per simulation day, if hourly 30 three-dimensional fields are calculated. At an integration rate of 18 machine hours per simulated day at a sustained petaflop, the average sustained output bandwidth required is 700 MB/second.

Thus, it is important to choose programming techniques and parallel software resources that extract the maximum performance of the computer infra-structure.

3.1.1 Initiatives for Improving the Development of Applications

Some initiatives were proposed in order to provide the infra-structure for the development of applications in the next years (BERNHOLDT, 2007).

Brazilian Computer Society promotes the **Grand Challenges in Computer Science** since 2006 (MEDEIROS, 2008), (CARVALHO, 2010). The objective of this proposal is to generate 5 grand research challenges in Computer Science for Brazil to be reached in the next 10 years. One of these challenges is to model complex systems like artificial, natural, socio-cultural, and human-nature interactions. To achieve this goal, specific applications need to be developed.

In the United States document **Computational Science: Ensuring America's Competitiveness**, proposed by President's Innovation and Technology Advisory Committee (PITAC), some research and development challenges are presented for algorithms and applications in scientific and social sciences (PITAC Report to the President, 2005).

The Landscape of Parallel Computing relates challenges in some classes of application (ASANOVIC et al., 2006), (ASANOVIC et al., 2009). Particularly the challenges of the 6th class of applications, unstructured grids class of application, is important to us because it is related to the global representation of Earth for climatological applications.

A key output of The Landscape of Parallel Computing report was the identification of 13 benchmark dwarves that together can delineate application requirements in a way that allows insight into hardware requirements. In addition, more attention must be given to both dependability and performance. The document also discusses power monitoring and the use of autotuners, which are software systems that automatically adapt to performance characteristics of hardware, often by searching over a large space of optimized versions.

In terms of computer architectures, the text concludes that multi-core systems are unlikely to be the ideal answer to achieving enhanced performance. Consequently, a new solution for parallel hardware and software is necessary (ASANOVIC et al., 2006). Increasing explicit parallelism will be the primary method to improve processor performance. New models of programming will also be needed for such systems.

3.1.2 Changes to Improve Exascale Computing

Multi-core appears to outline the limits of performance of traditional processors (limits of the increase of the clock frequency of the processors). Multi-core is a way to provide Exascale Performance Computing, that is, upscaling the performance of today applications in $1000\times$ faster (DONGARRA et al., 2011). For a Exascale Performance

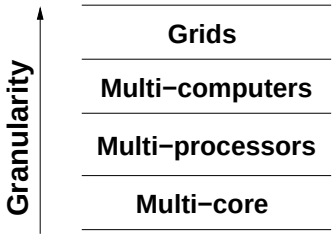

Figure 3.1: Multi-level parallelism.

Computing system, thousands of multi-core processors operating simultaneously are necessary.

There are two major reasons to invest in a new computing system: for solving problems not previously solvable, either because of the execution time to solve it or because the size of the data set of the problem, or to compute the same kind of problems previously solved on a prior system, but faster or more frequently (BERGMAN et al., 2008).

New applications: where the desired properties of computation are different of what is supportable today. This includes the kind of operations that dominates the computational rate requirements, the amount and type of memory, and bandwidth. These new applications might as well use algorithms that are unknown today, along with new software and architecture models.

Upscaling of current peta applications: where the overall application is similar to a currently peta scale system, but the data set size representing the problems needs to grow considerably. If the computation is linear time in the data size, then this corresponds to a $1000\times$ increase in memory capacity along with computation and bandwidth.

In this scenario, if the basic speed of the computational units does not increase significantly (as is likely), then **new hardware levels of parallelism must be discovered in the underlying algorithms**, and if that parallelism takes a different form than the current coarse-grained parallelism used on current high end systems, then software models need to be developed to support this form of parallelism.

3.2 Multi-Level Parallelism

Today, the composition of a parallel computing environment is increasingly heterogeneous. On one side there are clusters and grids architectures. On the other side, multi-core architectures began to appear with different numbers of processing cores. Consequently, these environments end up also providing a multilevel parallelism.

In a multilevel parallelism there are several levels of abstraction of parallelism. The different levels of abstraction of parallelism may be in the processor itself (multi-core), internal to a computer (multiprocessor) or between multiple computers (cluster and grids), creating a hierarchy as shown in Figure 3.1. The granularity of processes or tasks that can be run on each level is also highlighted in this figure, increasing as the level of parallelism increase.

The management of each of the parallel levels of abstraction is done through specific mechanisms:

- **At processor level** - The instruction stream is defined by the core or the implementation of registers required in hardware. Thus, the control is done by instructions in assembly.

- **At level of the operating system kernel** - The instruction stream is defined by processes or threads. The control of the flow of instructions is done through calls to the operating system.

- **At level of middleware management** - The set of instructions is grouped, forming a communicating process. The control is done through inter-process communication libraries.

Therefore, it is usually the responsibility of the programmer to use different tools for implementing a program that explores the various levels of parallelism.

We can suppose, for example, a program implemented in parallel using the divide and conquer approach. This implementation creates processes in the first recursive divisions, and decomposes each process in threads after. Consequently, it is possible to efficiently use multi-core processors in a cluster environment.

The programming development and execution environment must be considered to increase the performance of a concurrent application. Ensuring architecture portability of applications and efficient use of hardware resources is a great difficulty existing in parallel execution environments, since the programming tools currently available are designed specifically to only one level of parallelism. This limits the potential performance of a parallel application if it is executed on a different level of hardware parallelism of that it was originally projected. Furthermore, it is difficult to control how a parallel implementation will be executed once that different ways of mapping the flow of instructions can occur regardless of the level of abstraction. Who decides this can be a parallel programming library, a code compiler, the operating system, the user-level thread scheduler or the CPU.

3.3 Parallel Architectures

New computer architectures have been produced recently in order to improve performance for individual processors. This occurred because physical properties and technological resources used in hardware conception do not allow the increase of clock frequency of an individual processor (clock speed).

In the early 2000s, the limitations to provide head dissipation to chips and the reduction in the ability to include more transistors for higher Instruction-Level Parallelism (ILP) led to a stagnation of single-core single-thread performance. The solution was to switch from microprocessors of general purpose to Chip-level Multi-Processing (CMP). Thus, many processing units were implemented in a same chip, yielding, the multi-core architecture (GEPNER; KOWALIK, 2006).

From the viewpoint of computer architecture, multi-core processors are prevalent nowadays in systems ranging from embedded devices to large-scale high performance computing systems (RAUBER; RÜNGER, 2010). This can be clearly seen if we compare the composition of the 500 machines with the largest processing power of the world. These machines are used to process different kind of applications and their infrastructure are composed of several processing units (processors) interconnected in most of the cases through special network technology (TOP 500, 2011).

There are several options of multi-core processors available on the market. For example, Intel produces Quad-, Six- and Eight-Core processors (INTEL, 2011a). AMD presents Eight- and Twelve-Core processors (AMD, 2011).

Table 3.1: Examples of different multi-core architectures.

Vendor	Processor Model	Cores	Clock Rate	L3 Cache	Manufacture
Intel	Xeon Nehalen W5590	4	3.33 GHz	8 MB	$45\ nm$
Intel	Xeon Nehalen X7560	8	2.26 GHz	24 MB	$45\ nm$
Intel	Xeon Westmere X5677	4	3.46 GHz	12 MB	$32\ nm$
Intel	Xeon Westmere X5690	6	3.46 GHz	12 MB	$32\ nm$
AMD	Opteron 6136	8	2.3 GHz	12 MB	$45\ nm$
AMD	Opteron 6176SE	12	2.4 GHz	12 MB	$45\ nm$

Table 3.1 presents some examples of multi-core architectures. Information about the number of cores, clock rate, level 3 cache, and manufacturing technology are compared for different processors models.

In terms of computer architecture research, Intel presented a 80-core processors as part of the Teraflops Research Chip project (INTEL, 2011b). Features of the processor include dual floating point engines, sleeping-core technology, self correction, fixed-function cores, and three dimensional memory stacking. The degree of on-chip parallelism will increase significantly over the next decade and processors commercially used will contain tens and even hundreds of cores, increasing the impact of multiple levels of parallelism on clusters.

At the same time, the performance of a contemporary Graphics Processing Unit (GPU) has increased much faster than conventional processors, in part because these processors can easily exploit parallelism. (NICKOLLS; DALLY, 2010). Figure 3.2 shows an example of GPU Tesla architecture. In this figure it is possible to see the organization of GPU devices, composed by many processing units and different kind of memory spaces.

Modern GPUs incorporate an array of programmable processors to support the programmable shaders (a set of software instructions) found in graphics APIs (NVIDIA, 2012). For example, the Nvidia GForce 9800 includes a double array of 128 processors. Each processor can execute only one single-precision floating-point operation in each cycle. This is a significative power processing because until 128 concurrently executing instructions can be run at each clock cycle. Another hardware, the Nvidia Tesla GPU M2090, developed for High Performance Computing, has 512 cores and can process 665 Gflops.

The programmability, high performance, and efficiency of modern GPUs have made them an attractive target for scientific and other non-graphics applications (KIRK; W. HWU, 2010). Programming libraries such as Nvidia's CUDA have evolved to support general purpose applications on these platforms (NICKOLLS et al., 2008). Emerging hardware such as AMD's Fusion processor is expected to integrate GPUs with conventional processors. These initiatives simplify the programming of a code.

Computers can also be composed by heterogeneous processing units. A machine can be formed by many multi-core processors for general purpose, graphic cards (GPU) and reconfigurable hardware (Field Programmable Gate Array - FPGA) (BROWN et al., 1997). Computers can also be combined forming a cluster of network interconnected machines or cluster of clusters, as can be seen in Figure 3.3. Possible components of each individual node of a cluster are shown in the circle positioned at the first part of this figure. In fact, clusters are the solution to give high power processing to large applications.

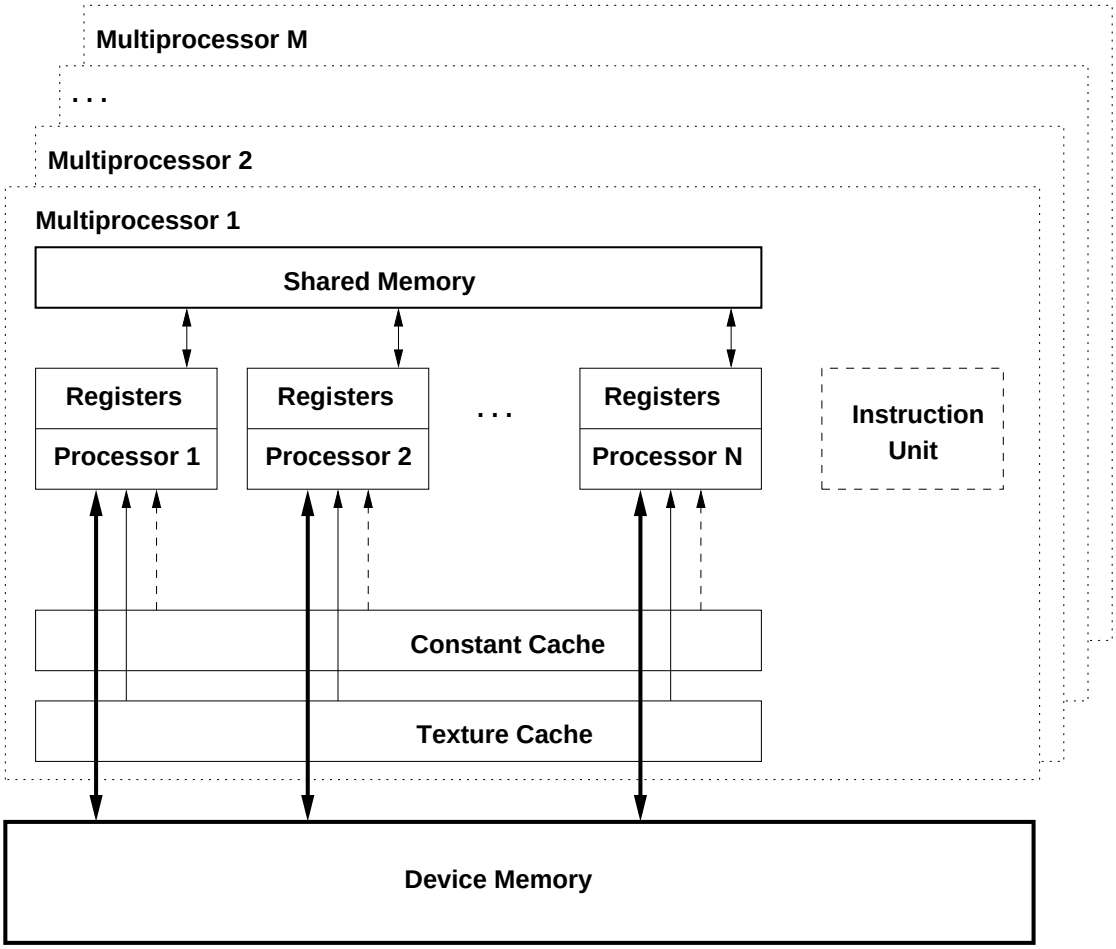

Figure 3.2: GPU Tesla architecture.

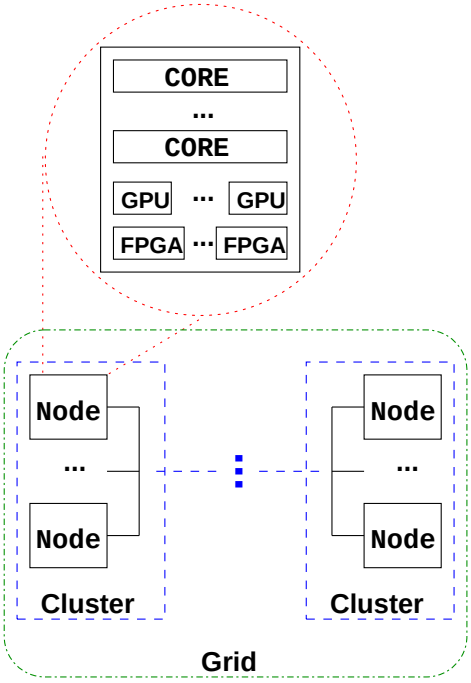

Figure 3.3: Different levels to explore parallelism in current architectures.

3.4 State of the Art in Parallel Programming Tools

The process of implementation of applications is simplified by the existence of tools for parallel programming. These tools abstract both shared and/or distributed memory architectures and provide standard development approaches for several parallel programming paradigms.

Message-passing libraries were developed to abstract the network layer (sockets) and to offer a clean interface for communication functions. These libraries were used to develop several high performance applications in the last two decades. The Message-Passing Interface (MPI) communication library is one of the mechanisms widely used to simplify parallel programming. MPI has a large number of functions to be used both in parallel and in distributed implementations. This resources are necessary to obtain parallel performance and are recurrently used in many kinds of applications.

However, MPI is not the only choice to write a concurrent code. Classical tools like Pthreads and OpenMP are adopted also in applications running in multiprocessor machines. Moreover, in recent years other tools were developed like Intel *Threading Building Blocks* (TBB) and *Compute Unified Device Architecture* (CUDA). These tools are employed in the development of multi-core and GPUs codes, respectively. Programming interfaces were also created to abstract different types of parallel hardware, simplifying the development process of concurrent code. They are discussed in the following subsections.

3.4.1 Message Passing

In the message passing model each processor has its own memory. The exchange of information occurs through communication between processors using normally a high-speed network. This model introduces a new problem: how to distribute the computational task into multiple tasks to multiple processors accessing different units of memory and to organize the results into a single solution. To solve this problem, some approaches of scheduling were proposed.

The main advantage of this model is scalability, since there is no limit on the number of processes that can be created, nor the number of processors that can be used. There is also a possibility (although demote the overall performance) of using heterogeneous machines. In the model of message passing the tasks usually are performed in a distributed way into distinct processors and the end result is grouped in one process or shared among all processes.

Message-Passing Interface (MPI) is the main representative message passing programming interface. MPI is a standard for exchange data (message-passing) in parallel computation (GROPP et al., 1996). MPI supports the portability of code and provides efficient parallel performance for many types of parallel machines.

MPI can be considered an evolution of *Parallel Virtual Machine* (PVM) (GEIST et al., 1994) and allows to write parallel programs in FORTRAN, C or C++ languages. MPI is a norm, supported by several implementations (LAM/MPI, OpenMPI, MPI-CH) with specific optimizations (LAM/MPI PARALLEL COMPUTING, 2012), (OPEN MPI: Open Source High Performance Computing, 2012), (MPICH HOME PAGE, 2012). The norm standardizes the name, parameters and return codes of each routine.

A MPI application is composed by one or more processes that can be executed on processors of distinct machines. The processes may communicate with other processes by sending and receiving messages. This resources are applied in the *Single Program Multi-*

ple Data (SPMD) and *Multiple Program Multiple Data* (MPMD) programming models.

MPI provides different communications primitives. The most simple communication mechanism that can be used is the point to point communication, where operations of message exchange occur between two processes. More structured communication functions are obtained calling collective communication operations (*collective*) for a group of processes. This operations may involve all processes in execution.

Moreover, MPI supports asynchronous communication and modular programming by mechanisms of communicators (*communicator*). The communicators allow the MPI user to define functions that encapsulate internal communication structures (*group communications*).

Advanced programming resources like **cartesian communication** mechanisms offer services that allow addressing messages to the processes according to identifiers assigned to them, using a Cartesian communication structure, through functions for mapping and accessing the processes.

The basic operation of an MPI program consists of all processes to execute the same code normally as a sequential program. Each process has an identification number assigned. The identification number can be used to restrict the execution of part of the code for a specific process or a group of processes. This identifier is also necessary to address a message for a process in function calls to exchange data.

An example of pseudocode using MPI is shown in Algorithm 3.1. In this example, the process with rank 0 send a set of data to process with rank 1. The process with rank 1, receive the data set, computes the data and send it to process with rank 0. Process with rank 0 receives the data processed by rank 1, and shows the results.

Algorithm 3.1 MPI example of parallelization of the code.

```
if (rank == 0)
        send(data, 1)
        receive(data, 1)
        show_results( )
else  // rank == 1
        receive(data, 0)
        calculation_execution( )
        send(data, 0)
```

The functions found in MPI are very important. It provides parallel implementations with efficient communication mechanisms and a greater independence among the execution of the processes. Evaluating this resources we conclude that MPI offers conditions for parallel programming applications to run on multicomputers machines, abstracting the granularity of a parallel task by (a MPI) process.

3.4.2 Parallel Programming Interfaces for Shared Memory

There are also libraries developed for programming shared memory architectures. For shared-memory programming, the standard tools provide constructs to allocate and access data in the global address space, common to all the running threads. Examples of shared memory programming tools are described below.

3.4.2.1 Cilk

Cilk is a general purpose programming language for any operating system platform proposed by the Technology of Supercomputation Group of MIT (FRIGO, 2007), (BLUMOFE et al., 1995), (FRIGO; LEISERSON; RANDALL, 1998). Cilk is based on ANSI C standard and offers a multi-thread parallel programming environment. It extends C language through keywords that enable to express the parallelism of the application. A Cilk program without keywords is called as C elision and results in a syntacticly and semantically valid C program.

The execution of Cilk is responsible to make load balancing and to schedule the created threads to execute concurrently over the processors.

Cilk tasks can be scheduled by shared tasks or by work stealing. In the first case, a thread is scheduled to execute concurrently in each parallel function call. Such a concurrently execution maximizes the computer processing but it is penalized by the high cost to create a new thread. In the second case, a processor can search more tasks to process when it end its current works (adaptive scheduling). The advantage of this method is to provide better parallelism conditions, minimizing the amount of thread and maximizing efficiency. Scheduling decisions in Cilk are defined by information obtained in compile and execution time.

The parallelism and synchronization primitives of Cilk are: cilk, spawn, sync and return. The cilk primitive identifies a parallel function to the environment, defining it as a Cilk procedure. The parallelism begins in the spawn primitive that launch a new task for the specified function. The semantic of spawn differs of a C method because spawn does not wait for the end of the called function in oposition to a C method call. The sync primitive offers a local barrier as a way to wait for the end of the tasks created by a father task. The Cilk environment inserts a sync before the implicit return of a task in order to guarantee that all child tasks end before the return of the currently running task.

3.4.2.2 OpenMP

OpenMP (Open Multi-Processing) provides directives that allow the expression of data parallelism in parts of the code and loops, and parallelism of tasks, introduced in its version 3.0 (CHANDRA, 2001), (CURTIS-MAURY et al., 2008). An example of loop parallelization is shown in Algorithm 3.2, where all operations of a step of the loop can be concurrently executed, according to the number set in the `omp_set_num_threads()` function (4 threads, in this case).

Algorithm 3.2 OpenMP loop parallelization.

```
omp_set_num_threads(4);
#pragma omp parallel for
for (i = 0; i < MAX; i++)
        A[i]= c*A[i] + B[i];
```

The API of OpenMP consists of compilation directives, library of methods/functions and environment variables that describe how the workload can be shared among different threads running on different processors or cores. The programmer can choose the number of threads to execute by calling library methods or by setting environment variables. Moreover, the granularity of tasks, using the approach of data parallelism, can be determined by the programmer or by the compiler.

The OpenMP standard does not specify a scheduling algorithm. This is attributed to the implementation of the API, in order to define the best choice in terms of load balancing.

3.4.2.3 Threading Building Blocks

Threading Building Blocks (TBB) is a C++ library developed by Intel to program software that run on multi-core processors (PHEATT, 2008), (VOSS, 2009), (WILLHALM; POPOVICI, 2008). The first version of the library was announced in 2006 for the first x86 dual-core, Pentium D processors.

In order to reach the best way to use processor resources, TBB provides the division of the workload into threads and gives a scheduling solution for the threads. TBB parallel tasks are called work units. The granularity of the loop parallelism is defined by the library and the granularity of the threads using tasks parallelism is defined by the programmer. A set of threads executes the available tasks in user mode according to the work stealing scheduling inspired by the Cilk environment. This makes easier the programming because it is not necessary to understand how the threads were implemented.

In general, the focus of this library is the high level parallelization through, for example, the distribution of data among threads. This means that the programmer can concentrate his efforts on solving problems, and not in small details, by using threads.

TBB library has performance, scalability, and is similar to the OpenMP library, supporting loops and tasks parallelism. However the library provides it with different approaches. Another significant difference of TBB is that its offers the utilization of generic programming in parallel loops, in order to avoid limiting the parallel data structures to basic types of the language. This resource is similar as Standard Template Library (STL) containers programming tool (MUSSER; SAINI, 2004).

3.4.2.4 Charm++

Charm++ is an object-oriented paradigm for parallel programming and asynchronous message exchange that adds several features and structures to the C++ language (KALE; KRISHNAN, 1993). It is based on the manipulation of special objects called *chares*. *Chares* have their own data (local). They communicate with other *chares* and have special methods called input methods, responsible for receiving and processing messages destined to its objects. Input methods are different from traditional methods because they return immediately after their invocation, but not necessarily after the asynchronous execution of the called method, ensuring that this method will be executed eventually.

A Charm++ program is a set of *chares* referred as a global space of objects, and has their execution initiated through a specific chare called *main chare*. The messages exchanged between *chares* are also known as Remote Method Invocation, because the *sender* of a message does nothing more than invoking entry methods of the *receiver*. Migration of *chares* between processing nodes can be made using the framework Pack and UnPack (PUP) in order to facilitate the packaging of data (classes).

A Charm Run-Time System (RTS) is provided to remove the responsibility of the programmer to identify and manage quantity and type of processors, type of communication (network) between them, and the amount of resources available. RTS is responsible for:

- Mapping chare objects in physical processors.
- Load balancing of objects through dynamic migration.

- Routing of messages: it is important due the migration of objects between processes.
- Checkpoint: by enabling objects' "state" migration.
- Fault tolerance: recreation of objects in distinct processors when processors crash.
- Dynamic reallocation of physical resources: the possibility to allocate more or less space according to the extra load present in the cluster.

RTS provides support for other languages or programming models like Adaptive MPI (AMPI) (HUANG et al., 2006), where MPI programs can take advantage of virtualization, load balancing, fault tolerance, among other characteristics already listed.

3.4.2.5 CUDA

Compute Unified Device Architecture (CUDA) is a parallel programming computing architecture developed by nVidia (NICKOLLS et al., 2008). It enables the use of Graphics Processing Units (GPU), integrated in the video boards. This technology was available initially for the GeForce (series 8 and after) and Quadro editions, and more specifically for the Tesla edition (developed for HPC) and Ion (for mobile computers).

The use of video boards to execute an application normally performed by a CPU is called General-Purpose computing on Graphics Processing Units (GPGPU) (GARLAND et al., 2008). The first advantage of using CUDA is the use of shared memory for quick access to arbitrary addresses in memory. Since the version 3.1, CUDA has support for recursion, double-precision floating point data type, implemented according to the IEEE 754 standard, and rendering of textures.

The programming model of CUDA consists of extensions to C and C++ in a sequential program that can boot a kernel (NICKOLLS et al., 2008). The kernel is similar to a C function and runs concurrently through several CUDA threads. The threads are mapped to the execution core of the GPU by the GPU. The programmer is responsible for transferring data from CPU to the GPU and GPU to CPU.

The programming model of CUDA is ideal for applications with high data parallelism level and for applications that have not dependencies among tasks. However, CUDA limitations include no control of coherence of the data used and the lack of support for the execution of multiple kernels. Thus, significant performance gains in CUDA depend on good knowledge about the architecture and the programming model.

3.4.3 Distributed Shared Memory

Programming interfaces for distributed systems use explicit two-sided communications. On the other hand, the interfaces used for programming shared memory systems provide simple statements of concurrency, but not applied for intercomputer architectures. To overcome this situation, some tools were proposed to join the best features of these two paradigms (BERNHOLDT, 2007). These solutions are based on *Partitioned Global Address Space* (PGAS).

In the PGAS model the languages are developed over a memory model in that a global address space is logically partitioned in order to give each part to a local processing unit. This kind of language is typically implemented on distribute memory machines and use communication libraries to address the virtual space.

PGAS languages provide abstract resources to develop distributed data structures and communication of the cooperative instances of the code. Although the objective of these

languages is to improve the capability to write codes, it is still limited in terms to provide a global vision of parallel computing.

Some programming languages, that allow the programmer consider a large scale computational environment as a unified system like a shared memory environment, are presented below.

- **Unified Parallel C (UPC) -** It is a extension of the programming language C, developed in Berkeley, for HPC on large scale parallel machines (YELLICK; BONACHEA; WALLACE, 2004). The language provides an uniform programming model for both shared and distributed memory systems. UPC abstracts the SPMD programming model. The parallelism is defined before the execution of the code. Each execution stream is destined to a processor. Thus, the execution environment can be viewed as a single shared memory system in that the processor can read and write variables, although these variables are physically associated to one distinct processor.

- **Co-Array FORTRAN (CAF)** - This language is an extension of FORTRAN to support the SPMD programming model, developed in Berkeley (NUMRICH; REID, 1998). It has similar properties as the UPC implementation and includes resources expected for the next version of FORTRAN. The name of the language arises from the implementation of a new kind of *array* called *co-array*. This resource is used to reference multiple cooperative instances (*images*) of a SPMD program. Each *image* can access remote instances from a variable through the index of a dimension of *co-array*. A variable declared in a *co-array* dimension allocates a copy of the variable in each image. The way that a *co-array* is created is similar to the creation of a normal *array* in FORTRAN. The language offers also synchronization routines to coordinate the cooperative images.

- **Titanium -** It is a language developed in Berkeley to implement the SPMD paradigm for Java (YELICK et al., 1998). Titanium increases several features of Java, including support to multi-dimensional vectors and sub-vectors iterations, copy operations, class with unchanged values and *regions*, an alternative garbage collector that supports memory management oriented to performance. The language offers support among instances of the program developed in SPMD through synchronization and communication primitives, methods and variables that allow an alternative synchronization way, and a notion of private and shared references.

- **Chapel** - It is a programming language developed by Cray (CHAMBERLAIN; CALLAHAN; ZIMA, 2007). Chapel is part of a larger project knew by *Cascade*. It provides a higher abstract level to express parallel programs. It offers a separation between the development of the algorithm and the details of data structure implementation. Chapel supports multithreaded programming model, offering data and tasks abstract parallelism.

- **Fortress** - It is a high performance programming tool projected by SUN (WEILAND, 2007). The language is based on FORTRAN and provides efficiency and security. Fortress has an innovative syntax: it was developed to provide a mathematical notation style. With that, the development of a code can be more easily done for scientists. The fundamental components of Fortress code are the *objects* that define the variables and methods, and the *traits*, where the conjunct of abstract and

Table 3.2: Different levels of parallelism covered by programming interfaces.

Parallel Level	Cilk	OpenMP	TBB	PGAS	MPI	CUDA/OpenCL
Distributed Memory				x	x	
Interprocessors	x	x	x	x	x	
Intraprocessor	x	x	x		x	
GPGPU						x

concrete methods are declared. Fortress is an interpreted language. The interpreter runs over a Java Virtual Machine (JVM) and interprets the class of the specification of the language.

- **X10 -** It is an experimental programming language developed by IBM in association with academic institutions (CHARLES et al., 2005). The objective of the language is to offer new programming techniques for scalable parallelism. It is optimized for a environment management at execution time. X10 offers all classical resources of the Java programming language for Symmetric Multi-Processors (SMP) and clusters environments.

All these tools provide an abstract layer, making the implementation mechanism homogeneous. At the same time is not possible to extract parallelism from all hardware layers because they are not abstract so well, for example, multi-core architectures. Thus, these interfaces can be considered solutions for abstract two-level and do not for multi-levels of parallelism.

3.4.4 Evaluation of the Presented Tools

To explore the performance of the parallel architectures today, a programming tool needs to provide mechanisms to access all the parallel levels of a machine. However, no programming interface provides this. Some languages presented have the intention to provide more code abstraction but they do not include some parallel levels in its conception. Moreover, this resources are still in development.

Table 3.2 presents a summary of the different parallel architectures levels covered by the different programming interfaces previously described.

The notion of parallel task is presented in different ways in the parallel programming interfaces discussed in this chapter. CILK and TBB support natively this notion. This notion is not so well defined in MPI. In this programming tool, each MPI process is the task itself.

The solution to explore multi-parallelism level is to combine more programming tools. Therefore a parallel program that utilizes the MPI library could be combined with the OpenMP or the TBB programming interfaces. An MPI program can also incorporates CUDA functions, in order to explore GPU hardware. In the Chapter 6 we will to discuss how the combination of these interfaces can be made for applications of atmospheric models.

3.5 Final Considerations

There are many challenges to create and execute high performance applications today. The scale of the simulations for solving real problems are very large. New applications

demand ever more memory and processing since the volume of data and operations to compute in these applications is increasing.

Applications need to use the entire architectures available for the executions in order to maximize it performance. Currently, the architectures provide many level of hardware concurrency. But there is not a programming interface able to abstract all these levels. Because of this, each of these levels can be only explored using a specific parallel programming interface. Thus, it is important to understand the application functionality and how parallel tasks can be defined.

Next chapter presents a performance evaluation of OLAM. The evaluation of the application, simulating some case studies on a multicore cluster environment, is necessary to understand the limits of performance of the model using only a parallel programming interface.

4 SCALABILITY STUDY OF STATIC OLAM

The previous knowledge of the execution behavior of OLAM is important to better understand how is possible to improve the model. Because of this, we present in this chapter the simulation environment and the performance measurements of the original code of OLAM in order to evaluate the impact (execution time and speed up) of multi-core architectures in simulations of atmospheric models using low mesh resolution. This study highlights many performance aspects of the original version of the model, implemented in FORTRAN 90.

4.1 Simulation Environment

We evaluate the 3.0 version of OLAM. In our case study, each side of the initial icosahedral triangle was divided in 25 parts ($N = 25$). So, the distance between the discretized points on the horizontal globe surface was near 200 Km. The atmosphere was divided vertically (z dimension) in 28 levels.

The objective of the simulation was to evaluate the execution costs of the model. Therefore we measure the impact of fluid dynamics methods during 24 hours of simulation of an atmosphere, without any computation of physical methods, in the iterative step of the model. Each timestep of integration simulates 60 seconds of the real time.

All measurements have been made on the two clusters ICE and SUNHPC.

The ICE platform at the Institute of Informatics of the Federal University of Rio Grande do Sul is composed by 14 dual nodes with Intel Xeon E5310 Quad-Core of 1.6 GHz and 4 MB of cache, with 16 GB of RAM memory in each node. The cluster is interconnected by a *Gigabit Ethernet* network.

The SUNHPC platform at the National Laboratory of Scientific Computing (LNCC) is composed by 23 dual node Intel Xeon E5440 Quad-Cores with 4 MB of cache and 16 GB of RAM memory in each node. The cluster is interconnected by an *InfiniBand* network.

MPICH 2-1.0.7 and 2-1.2.p1 versions were respectively used in the implementations evaluated at ICE and SUNHPC platforms.

4.2 Scalability Intra-Node

The first test realized evaluates the scalability of the code in a multi-core machine using all 8 cores of two processors in one node of the cluster ICE. This **Intra-Node** test simulates the execution of the model using 1 to 8 number of processes. The processes are balanced distributed between the processors.

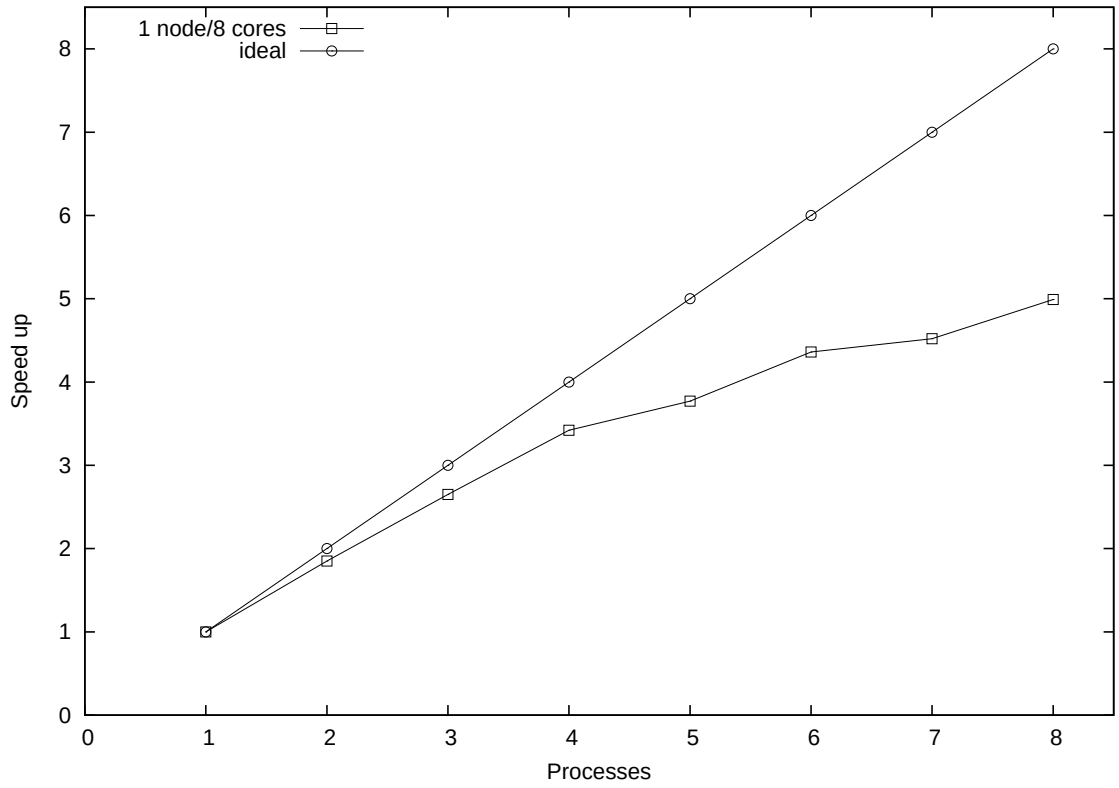

Figure 4.1: Speed up using 1 cluster node with 8 cores.

The results presented in the Figure 4.1 show an increase of performance when more processes are used. In fact, the speed up using all 8 cores of one node is only around 5 instead an ideal of 8.

4.3 Scalability Inter-Nodes

The scalability of OLAM code was evaluated in a second test, using multiple nodes of the cluster. Figure 4.2 presents the speed up obtained using only one core of each node of the ICE cluster. The number of processes used to simulate the model was 1 to 14. The speed up using 8 processes was around 6.4 and using 14 processes was up to 11 in this **Inter-Node** test.

The performance results of the parallel execution of OLAM using only 1 core of each node of the cluster provide more scalability than the test using all 8 cores of one node, as present at the Section 4.2. In fact, the maximum efficiency in the **Intra-Node** test was **62%**, using **8 cores**, and in the **Inter-Node** case the efficiency was **77%**, using **14 cores**. This occurs because the impact of cache misses in the **Intra-Node** test is very high, as will be shown later.

4.4 Execution Time - Multi-Core Impact

A third test was made using a different number of cores in each node of the cluster. P processes are distributed to C cores processors to better understand how multi-core architectures influence the execution of OLAM. Thus P/C nodes will be used to execute the model.

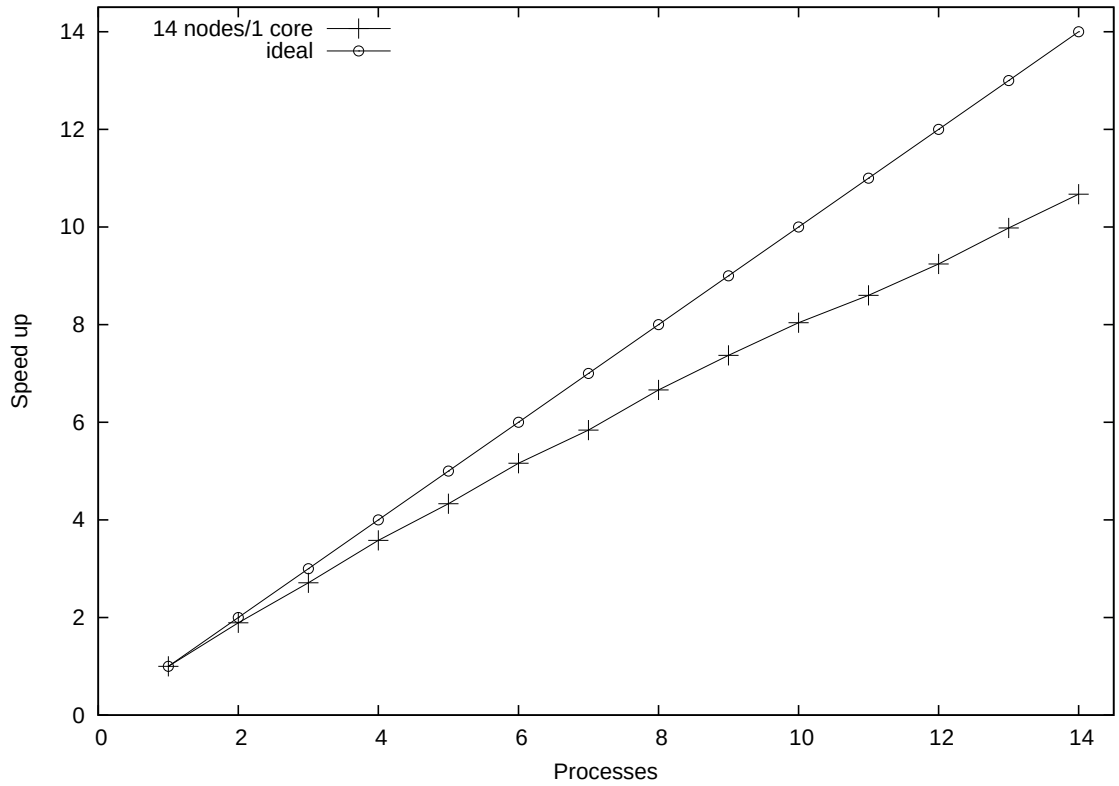

Figure 4.2: Speed up using 1 core from each of the 14 nodes of the cluster.

Figure 4.3 shows a comparison of the parallel execution time of OLAM, distributing 14 processes among the ICE cluster nodes in five different configurations. These configurations consist of processes distributed respectively for one, two, four, six, and eight cores per node ($C = 1, 2, 4, 6, 8$).

The results show that using only a core per node is better than using more cores per node. The results also demonstrate reduction of performance when the number of cores used in each node increases. This is more visible when more than 4 cores per node are used. In fact, quad-core processors share the access to the bus. Because of this, the performance on simultaneous accesses of memory is not so good, due to the large volume of data manipulated in OLAM.

Data access latency has been a problem even on single-core systems, as processors are much faster than memory. With the emergence of multi-core processors, a more severe problem arises with data access due to the limited bandwidth to access shared resources in the memory hierarchy. When multiple cores are processing different sets of data, the shared resource becomes a performance bottleneck, if the bandwidth is not high enough to support the multiple cores. This has been already experienced in currently available processors (BYNA; SUN; HOLMGREN, 2009), (SHALF, 2007).

4.5 Execution Time - OLAM Routines

Seven timestamps barriers ($TS1, ..., TS7$) are inserted on selected points of the original source (a few module boundaries) in order to correctly assign partial execution times to OLAM main modules. OLAM pseudo code is presented in Algorithm 4.1 indicating where selected timestamps barriers were placed.

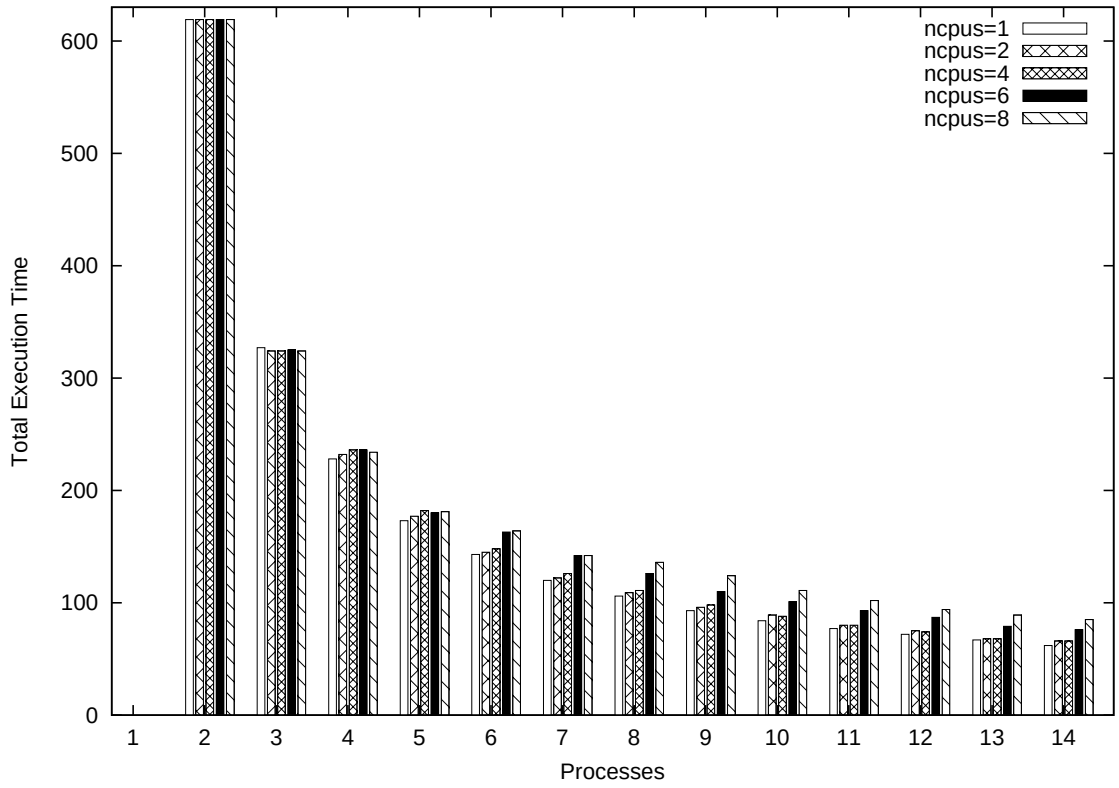

Figure 4.3: Execution time using 1, 2, 4, 6 and 8 processes/cores per node.

Algorithm 4.1 OLAM pseudo code and the localization of the timestamps.

```
Initialization;
Input Files (ATM/LAND/SEA) Read; (TS1)
Grid Configuration/Domain Decomposition; (TS2)
Variables Memory Allocation; (TS3)
Pre-processing initial state calculation; (TS4)
Plot and History Files Initialization; (TS5)
Initialization Time measure;
Do loop for each time step;
    Atmosphere time state calculation;
    Send frontier variables to neighbors;
    Times step Time measure;
    If time equal END then; (TS6)
        End Do Loop;
    Write atmosphere state on disk; (TS7)
    Barrier; Output Time measure;
```

In Table 4.1 the best (T. Min) and worst (T. Max) execution times are shown in milliseconds (ms) of each of the 7 parts of the instrumented code. The results compare the use of 8 processes in only one node and 8 processes executed in distinct nodes of the ICE cluster, respectively.

Each timestamp (TS) stage $TS1, ..., TS7$ is followed by a synchronization stage. The synchronization stage measures the time elapsed between the end of each execution time stage and a barrier, where all processes must arrive and wait before to continue the exe-

Table 4.1: Execution time using 8 processes in 1 node and in 8 nodes.

Time-Stamp	T. Min 1 node	T. Max 1 node	T. Min 8 nodes	T. Max 8 nodes
TS1	598	618	180	182
Syn	0	31	0	6
TS2	186	225	58	59
Syn	0	39	0	1
TS3	20	23	15	17
Syn	0	39	0	3
TS4	565	601	554	555
Syn	0	1	0	0
TS5	109	280	103	266
Syn	0	170	0	163
TS6	111268	111924	81641	82194
Syn	83	837	129	704
TS7	173	487	132	138
Syn	109	585	130	302
Total	**114497**	**114510**	**83830**	**83837**

cution.

In this table we observe that $TS6$ is the most impacting step of the execution. This timestamp monitors the iterative step of OLAM and, in this case, represents around 97% of the total execution time in both cases evaluated. However, the difference between the cases $C1$ and $C8$ is very significant. The Inter-Node approach demands only 73% of the total time in relation to the Intra-Node approach.

A similar evaluation was made using 14 processes.

In Table 4.2 respectively the best and worst execution time of the instrumented code are reported in milliseconds (ms), using 1 core ($C = 1$) and 8 cores ($C = 8$) per node of the cluster.

$TS6$ dominates the overhead for both $C = 1$ and $C = 8$ synchronization time. The synchronization time increases in all timestamps for $C = 8$, indicating that the use of all cores of a node impacts in reduction of performance.

Synchronization time from $TS6$ is related to a small load imbalance from the atmosphere time state calculation part. The increase of synchronization time on $TS1$ to $TS7$ on $C = 8$ test are related to the multi-core memory contention (OSTHOFF et al., 2010).

Applications running on multi-core systems using many cores and a large amount of RAM memory in each process have low reuse of cached data. The most external cache levels are generally shared among the cores of a multi-core system. In applications of domain decomposition, this implies low cache reuse, since each core handles different data from the others. In addition, large volumes of data manipulation involve often rewriting of data in the most internal cache levels.

Table 4.2: Execution time using 14 processes with $C = 1$ and $C = 8$.

Time-Stamp	T. Min $C = 1$	T. Max $C = 1$	T. Min $C = 8$	T. Max $C = 8$
TS1	178	183	177	1095
Syn	0	149	0	959
TS2	57	60	55	201
Syn	1	4	0	147
TS3	9	13	15	18
Syn	1	5	0	38
TS4	323	367	339	374
Syn	0	1	0	8
TS5	88	195	93	191
Syn	0	107	0	90
TS6	47323	49249	70063	72481
Syn	244	2224	463	2862
TS7	43	134	121	175
Syn	254	510	513	964
Total	**49884**	**50786**	**75496**	**75552**

4.6 Performance Analysis with Vtune Analyzer

Timestamp 6 ($TS6$) was the part of the code that demanded more execution time. Because of this, the iterative step was executed using Intel Vtune Performance Analyzer 9.1 (INTEL, 2010) in order to investigate the execution behavior of the processes.

OLAM was parallel executed with 8 processes, each one on distinct nodes of the cluster SUNHPC, obtaining a cache miss (L2) of **19.48%**. The execution of 8 processes in a same node results in a measured cache miss of **99.94%** and the rate of data transfer from the memory bus increases to **99.83%** of the bus capacity/time.

Hardware prefetching was another aspect investigated (ZHURAVLEV; BLAGODUROV; FEDOROVA, 2010). The tests allow to conclude that OLAM execution not increase hardware prefetching when more cores are used.

These results show that there is an increased cache miss rate and data transfer rate in the memory bus when more cores are used in a multi-core machine, that is, there is memory contention. The tests realized with Vtune prove that memory and cache access affect directly in the execution time of the model.

4.7 Summary of the Results

This chapter evaluates the performance of the parallelized version of the Ocean-Land-Atmosphere Model (OLAM) on a multi-core cluster environment. In order to evaluate the scalability of the model we present the speed up obtained using all 8 cores of two processors of a cluster node and the speed up resulted using only one core of each node of the cluster.

We insert timestamp barriers on parts of the OLAM code to find the routines of the algorithm that increase the execution time as more processes are used. The execution time

are more impacted by the routines called in the iterative step and by the output operations of OLAM. The results indicate also that the barrier synchronization time increases in the same order as the increment of cores number used per node.

In order to evaluate OLAM multi-core contention of resources we instrumented OLAM with Intel Vtune Performance Analyzer. The results indicate that the L2 cache miss and the memory bus traffic increases as the number of cores per node increases.

4.8 Final Considerations

This chapter contributes with an evaluation of performance on a multi-processor/multi-core cluster of a real scientific application characterized by high processing load. We simulate a parallelized version of the OLAM atmospheric application using MPI processes. The tests show that the scalability of OLAM application get worst as we increase the number of cores used in each node of the cluster.

We observe in the previous results that using only MPI processes in a today parallel architecture does not explore well all levels of parallelism. Some of the features observed in this chapter could be outlined by the use of techniques that better exploit the different levels of parallelism (multiprocessors/multicore) and by the expression of parallelism of the application. The Chapter 6 will discuss some ideas to improve programming techniques in order to improve multilevel parallelism in the code.

Another challenge for climatological models is to increase the resolution of the meshes, without impacting in the application performance. This can be done through the use of multiple layers of mesh refinement. These multiple levels can eventually also explore the parallelism levels of the architectures. Next chapter discusses how mesh refinement at run time can be provided in an atmospheric model.

5 ONLINE LOCAL MESH REFINEMENT

The mesh resolution impacts directly in the performance of a climatological model. In Chapter 2 and Chapter 4 we discussed concepts about an atmospheric model, and evaluated its parallel performance on a specific cluster architecture, respectively. We saw that the atmospheric model implementation defines a mesh resolution, to cover the global domain, statically at the beginning of the simulation, and provides also resources to set parts of the domain with more (local) mesh resolution.

In this chapter we propose an Online Mesh Refinement (OMR) approach for unstructured meshes distributed in distinct processes. The OMR implementation allows local mesh refinement at execution time, increasing the resolution of a discrete representation of part of a domain. This solution provides higher mesh resolution for atmospheric models with low impact in the execution time, providing also better numerical results.

5.1 Motivation to Improve Online Mesh Refinement

The forecast accuracy of climatological models are limited by computing power and time available for the executions. As the number and speed of processors increases, the resolution of the mesh adopted to represent the Earth's atmosphere may also be increased (without changing a maximum execution time limit reserved for the simulation in a high performance system), and, consequently, the numerical forecast will be more accurate. However, a finer global mesh resolution, able to include local phenomena in an atmosphere simulation, is still not possible because of the large number of mesh elements to be included in the model, and consequently a large increase of execution time.

To overcome this issue, different mesh refinement levels can be set statically to cover distinct parts of the global domain simultaneously. This is a good approach if we previously know what parts of the global domain need to have a high mesh resolution, due the impact of these parts in the precision of the final solution of a simulation. However, if the regions that impact in the precision are know only at execution time, the best solution is to improve, at run time, mesh refinement for parts of the domain.

A mesh refinement mechanism, like presented in the OLAM, can be only applied in a static way, that is, before the iterative step of the model and before of each physical properties calculation. In this context, this chapter evaluates how mesh refinement at run time can improve performance for climatological models. In order to contribute with this analysis, an Online Mesh Refinement (OMR) mechanism was provided. The implementation of the OMR increases mesh resolution in parts of a parallel distributed model, when special atmosphere conditions are registered during the execution of an atmospheric model. Thus, it is not necessary to reboot the application to increase local mesh resolution.

5.2 Related Work

The Adaptive Mesh Refinement (AMR) technique is frequently cited in the literature as a way to represent complex geometry and to increase locally the resolution for a thin part of a domain (PLEWA; LINDE; WEIRS, 2003). This technique is used in computational fluid dynamics to add fine grid patches to regions of the flow where more resolution is needed, such as near shocks and detonations. The AMR can significantly speed up a computation and/or enable simulations with a much higher effective resolution as compared to the uniformly refining of the grid approach. Efficient numerical schemes can be written for overlapping grids since they are composed of structured grids and Cartesian grids (ZUMBUSCH, 2003).

There are many applications and interfaces developed using this technique (ZUMBUSCH, 2003). In (HORNUNG; TRANGENSTEIN, 1997) is presented a solution using AMR in a porous media fluid flow application. (DEBREU; VOULAND; BLAYO, 2008) provide a set of function to apply AMR in FORTRAN codes. The Parallel Hierarchical Adaptive MultiLevel (PHAML) library develops new methods and software for the efficient solution of 2D elliptic partial differential equations (PDEs) on distributed memory parallel computers and multicore computers using adaptive mesh refinement and multigrid solution techniques (MITCHELL, 2006).

An another example of programming interface for AMR applications is provided by the PARAMESH toolkit, a software designed to offer parallel support with adaptive mesh capability for a large and important class of computational models, those using structured logically Cartesian meshes (MACNEICE et al., 2000). The PARAMESH package of subroutines is designed to provide an application developer an easy way to extend an existing serial code into a parallel code with adaptive mesh refinement.

However, mesh refinement solutions, like PARAMESH, are restricted to structured grids and differ from the unstructured mesh adopted in many climatological models.

In relation to climatological applications, an execution time mesh refinement was suggested for one of the first sequential versions of Brazilian Regional Atmospheric Modeling System (BRAMS), during the design phase of the model (FAZENDA et al., 2011) (PIELKE; AL., 1992). However, this option has not been included in the parallel implementation of the model and is consequently not used in real climate simulations.

5.3 Static Mesh Refinement in the Ocean-Land-Atmosphere Model

Local horizontal mesh refinement can be specified to cover delimited geographic areas with higher resolution after the setup of the global mesh, at the initialization step of the model. The definition of the region to be refined begins with the choice of a specific geographic coordinate point of the Earth surface. After this, all points included in the area formed by a radius surrounding that point will be marked to be refined.

Summarized, we can say that the refinement procedure depends on a Cartesian coordinate point (Earth latitude and longitude), a radius of latitude, a radius of longitude and an angle of inclination of the ellipse formed by using the combination of these two radii. Figure 5.1 illustrates the distribution impact of each of these variables to define a region of refinement. The figure shows also how the choice of the angle allows the rotating of the ellipse in order to better cover a region of the physical atmosphere.

After the choice of the region market to be refined, each discretized element contained in this area is subdivided horizontally into four new elements. Figure 5.2 illustrates the

Figure 5.1: Mesh refinement area definition of a specific region of the Earth. In this example, parameters were determined for an ellipse area to cover Argentine.

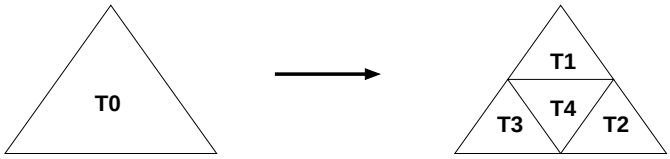

Figure 5.2: Example of one level mesh refinement applied to a point.

decomposition of a mesh element marked to be refine ($T0$), view horizontally as a triangle, in 4 new triangular elements ($T1, T2, T3, and T4$).

The level of resolution of the new refined region, in relation to the previous horizontally representation of the region, is always doubled. In order to achieve specific local mesh resolution values, the mesh points that represent these areas are subdivided cyclically while the expected mesh resolution is not overcome.

OLAM allows various levels of horizontal mesh refinement that can be applied in different parts of the domain, that is, a given domain can be refined several times. Since the resolution of the final level of refinement applied is always double in relation to the previous one, when x multilevel refinements are adopted to an Initial Level (IL) of resolution, the Final Level (FL) of resolution will always be:

$$FL = IL/2^x$$

The global grid and its refinements define a single grid, as opposed to the usual nested grids of regional models. Grid refined cells do not overlap with the global grid cells - they substitute them. The parallel data distribution takes into account the number of triangles (the horizontal mesh points of the domain) after the static mesh refinement to ensure a good load balancing. Once defined the distribution of subdomains among the processes,

Table 5.1: Number of vertices, edges and triangles mesh elements for different mesh resolutions.

Resolution	Vertices	Edges	Triangles
100 Km	25,002	75,000	50,000
50 Km	102,012	306,030	204,020
10 Km	2,550,252	7,650,750	5,100,500
5 Km	10,201,002	30,603,000	20,402,000

each process discards the global mesh, and keeps in memory only its respective sub-meshes.

Each OLAM MPI process is responsible for operating the functions of the iterative step on a given subdomain. There is no master process responsible for determining the division of the load and then assigning it to slave processes, as occurs, for example, in BRAMS. The distribution of data among the processes is set in each one. Each process determines its operating subdomain from the global grid according to its MPI *rank*.

5.4 Finer Mesh Resolution Execution

The OLAM horizontal mesh representation is made by decomposing the Earth surface in triangle points, according to the requested Resolution (R) given in Kilometers. The Number of Triangles (NT) for a specific resolution depends on the Earth's circumference and is given by:

$$NT = 20 \times (5050/R)$$

Table 5.1 presents the number of edges, vertices and triangles for 4 specific horizontal mesh resolutions. In this table, we can see that the number of points increases by a factor 100 if the adopted resolution doubles.

There is also a specific number of vertical layers (atmosphere column) associate to each horizontal decomposed triangle point. This number is chosen according to the physical characteristics of the atmospheric layers. There is also relationship between horizontal resolution and the size of atmospheric level to ensure realistic physical proprieties during the simulation. For example, for a 200 Km of horizontal mesh resolution, around 20 vertical levels can be adopted. For higher horizontal mesh resolution, this number needs to be increased.

Data values of the physical properties of the model are associated to edge and triangle elements of the horizontal mesh, and its specific vertical levels. Ignoring auxiliary data structures, and considering only physical proprieties, at least 30 different data structures are required for a simple simulation.

Many atmosphere simulation steps, called in the iterative part of the model, each one representing a small real elapsed time, increase substantially the execution time. This computational time could be not acceptable, even using high performance architectures.

For a parallel execution using 40 core or processors, and execution parameters of 20 Km of mesh resolution, 28 vertical levels, simulating only one day of atmospheric integrations, where each step represents 60 s of the real elapsed time, around 24 hours of execution time are required for the simulation. That is, for very simple simulation setups, the simulation execution time would be almost equal to the real time of the atmospheric transition.

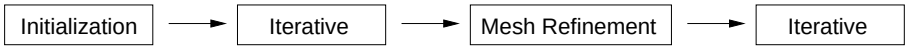

Figure 5.3: Execution steps of an atmospheric model improved by an OMR.

The total execution time elapsed in the simulation is correlated to the number of elements that represent the domain. However, finer mesh refinement needs to be adopted only to cover local weather phenomena. In this context, to reduce the execution time, without loss of precision simulation, different mesh refinement levels can be used. The best solution to cover local phenomena is to adopt higher mesh resolution in a global Earth model when it is really necessary, using a runtime mesh refinement.

5.5 Online Mesh Refinement Implementation

The refinement of meshes at runtime needs to take into consideration that the discretized points of the domain to be refined can be distributed into different processes. Thus, the implementation of this feature considers that each process must be able to identify whether its respective subdomain has a region to be refined if the refinement is called at runtime.

Just as each process is responsible for setting its sub-mesh at the beginning of the code execution, according to its MPI rank, now each process realizes locally the identification of the domain area to be refined, since each point of the sub-mesh maintains a reference to a global geographical coordinate system. Thus, it is only necessary to check if the global localization of a mesh point is circumscribed in the region of the domain that will to be online refined. If the point is circumscribed, it will be subdivided into four new triangles, as illustrated and discussed previously in Figure 5.2.

All distributed processes know if the mesh refinement must be made in its specific sub-mesh. Thus, each process knows which points must be refined. After the refinement, data structures of the new created points will be completed.

The mesh refinement at execution time stops the execution and refines the distributed sub-meshes points on specific Earth regions, according to a climatological condition. Next, data exchange are made among neighbor processes, in order to update the data structures of the boundary points of each sub-mesh. These data structures are used by communication functions called in the iterative step of the code.

After the conclusion of an online mesh refinement call, the iterative execution proceeds normally. Figure 5.3 illustrates all steps considered for an atmospheric simulation using an OMR implementation. In the figure only one OMR call is shown, but in the simulations the iterative step can be stopped more than one time by the OMR.

We compare the numerical results of all physical data structures used in the simulation of the ONR version with the results of a static mesh refinement version of the code. The computed results of both versions are similar considering the use of identical initial parameters.

5.6 Performance Evaluation

In order to measure the performance impact of the OMR we made several experiments. This section presents the simulation environment, execution parameters, and execution time measurements.

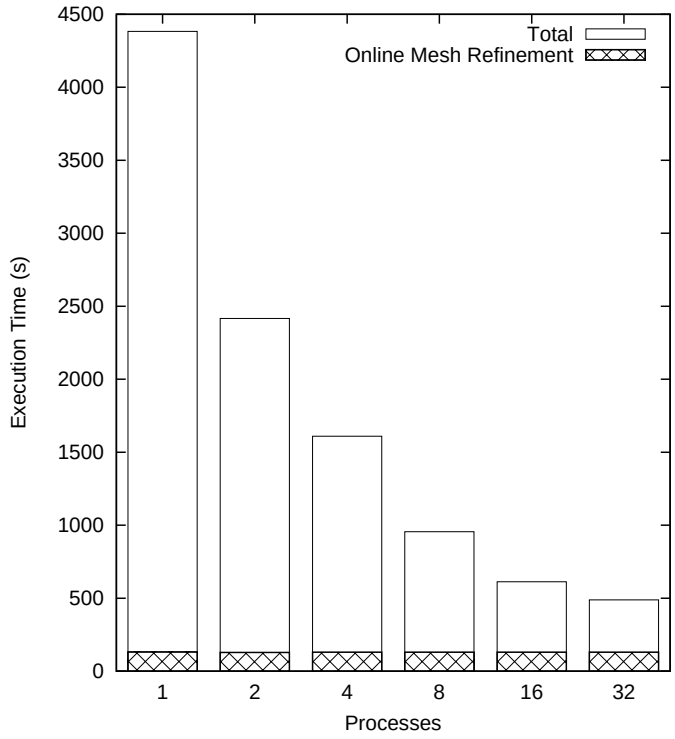

Figure 5.4: Execution time using 1 to 32 processes for a 100 Km mesh resolution with Online Mesh Refinement call.

5.6.1 Execution Environment

All experimental measurements were obtained using a cluster composed by 28 Sun Fire X2200+ workstations, each one with 2 Quad-Core AMD Opteron 2.2 GHz processors and 16 GB of RAM, interconnected by an InfiniBand network technology. We could use a maximum of 16 nodes of this cluster.

In all executions we simulated the atmosphere for 24 hours ahead. Each timestep simulated 60 seconds of the real elapsed time of the weather condition. The vertical atmosphere layer was divided in 28 layers. We use two horizontal resolution cases, 100 Km and 50 Km. The number and the size of each one of this layers is chose according to the parameters adopted in large climatological simulation centers for its daily weather forecasts.

An OMR occurs for specific tests in the half time of the execution of the simulation. The refinement of the mesh is realized 8700 Km around a specific point of the Earth after 12 hours of atmosphere integration.

The standard deviation for all obtained results was less than 2% in relation to the median time measured.

5.6.2 Online Mesh Refinement Execution Time Impact

A first test was made in order to analyze the impact of the OMR call on the total execution time. Figure 5.4 and Figure 5.5 present the execution time results of an atmospheric integration, using a mesh with 100 Km and 50 Km of horizontal mesh resolution, where an OMR occurs during the execution of the code. The graphics of these figures show the total execution time and the total time spent to call the OMR, using 1 to 32 processes.

Each column of the graphic represents the total execution time for a determined num-

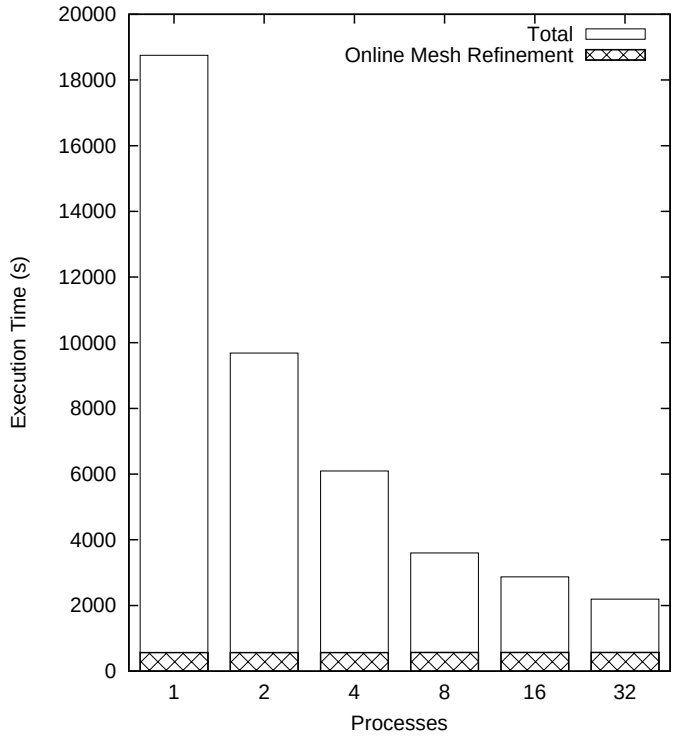

Figure 5.5: Execution time using 1 to 32 processes for a 50 Km mesh resolution with Online Mesh Refinement call.

ber of processes. We can see that this time decreases when more processes are used. Consequently, there are performance gains.

The second measurement (scratched area) of each group of processes presents the time spent for the OMR call. The duration time of this step is approximately 130 s and 570 s, respectively, for the 100 Km and 50 Km of mesh resolution cases. This time is a little more than the time spent with the initialization of the model, that is 115 s and 415 s, respectively, for the two analyzed cases. The time spend with the OMR includes all necessary procedures to interrupt the iterative step, to refine the mesh in each process and to reallocate variables.

The OMR has low impact on the total execution time. The time spent for this refinement is constant independently of the number of processes used in the atmospheric simulations. The relation between the execution time of the OMR call and the total execution time decreases if more high mesh resolutions are used.

5.6.3 Comparison between Static and Dynamic Local Mesh Refinement

The second test evaluates the execution time impact of a simulation using a runtime mesh refinement in relation to finer and larger global mesh refinements simulation cases. Figure 5.6 shows a comparison of the parallel execution time (in seconds) of 3 different configurations using 1 to 32 processes. The first and third columns show the total execution time using a global mesh resolution of 100 Km and 50 Km, respectively, without any OMR call. The second column represents the total execution time for a 100 Km grid resolution, where an OMR occurs during the execution of the code.

The results of Figure 5.6 show that all configurations have a decrease of the execution time, as a larger number of processes are used. The results demonstrate also that if we use a double resolution (50 Km) instead of a large resolution (100 Km), without a run time

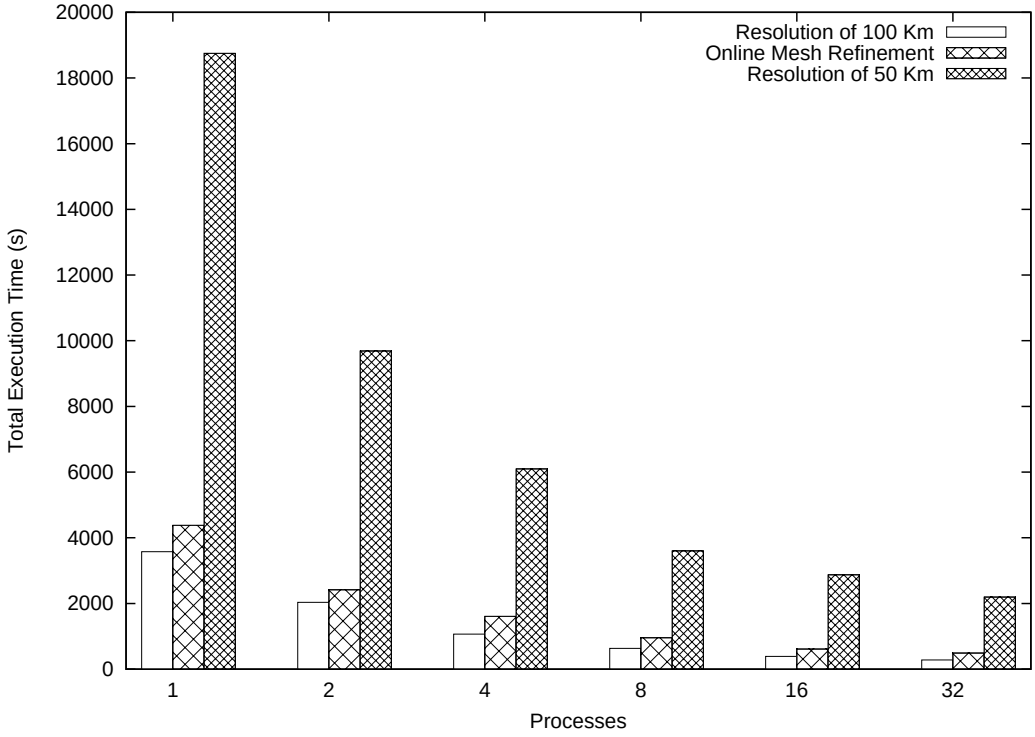

Figure 5.6: Execution time using 1 to 32 processes for 100 Km, 100 Km with Online Mesh Refinement and 50 km of mesh resolution.

mesh refinement call, we spend 5 to 8 times more execution time.

The execution time using OMR was always between the results using 100 Km and the 50 Km resolution cases configuration. Thus, the evaluation of the implementation shows that it is efficient, since not all the surface of the Earth needs to be refined all the time. In fact, the total execution time increases a little in relation to the 100 Km resolution case, because the costs of the OMR call.

The OMR improve numerical quality for the simulation results if it is necessary to the model. That is, a region of the Earth need to be higher refined only when special atmosphere conditions occur.

5.6.4 Speed up Evaluation of the Iterative Step of the Model

OLAM execution time measurements are also made, evaluating the partial execution of the iterative step. Through these measurements it was possible to establish the speed up of the iterative step of the model in parallel simulations.

In Figure 5.7 and Figure 5.8 are presented the speed up of the iterative step before and after an OMR call for a mesh with global resolution of 100 Km and 50 Km, respectively. The number of MPI processes used was 1 to 32.

In both cases, the continuous lines present the speed up before an OMR call and the non continuous lines are the speed up after an OMR call. The base to calculate the speed up was the execution time using a single process for each mesh resolution.

The use of more processes provides more performance in the iterative step of the model for both mesh resolution cases. However, the iterative step executed after the OMR call increases less speed up than the iterative step executed before the OMR call. This occurs because the unbalance load among the processes. The reason why this occurs, and solutions to resolve this issue are discussed in the next section.

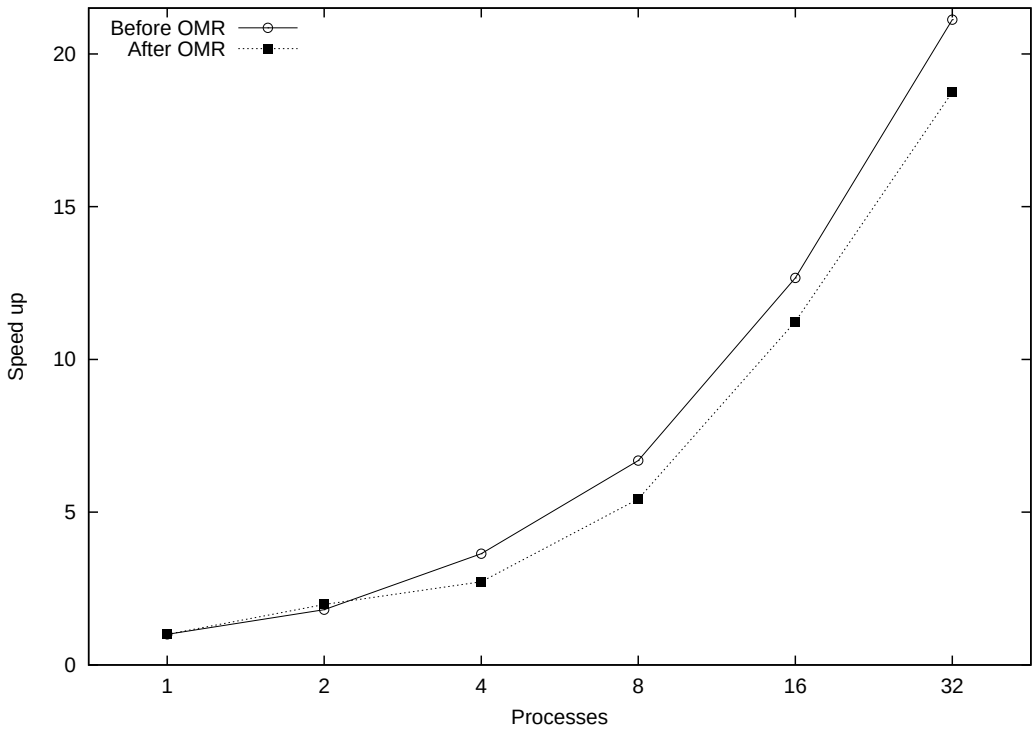

Figure 5.7: Speed up comparison of the iterative step of the model **before** and **after** the OMR call for a global mesh resolution of 100 Km.

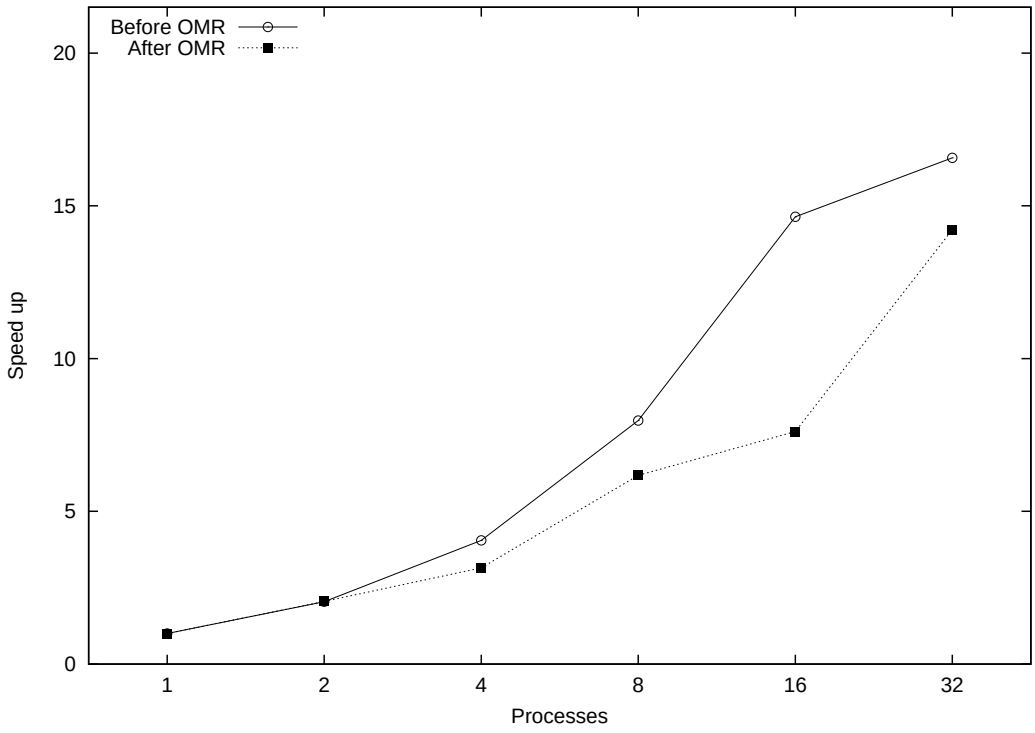

Figure 5.8: Speed up comparison of the iterative step of the model before and after the OMR call for a global mesh resolution of 50 Km.

Table 5.2: Unbalancing Load after an Online Mesh Refinement using 8 processes.

	Before Online Refinement			After Online Refinement		
Proc.	Vertices	Edges	Triangles	Vertices	Edges	Triangles
0	3408	9873	6468	3408	9873	9873
1	3394	9925	6534	3394	9925	6534
2	3412	9952	6543	5667	16549	10857
3	3439	10041	6605	5933	17362	11404
4	3421	9959	6541	3421	9959	6541
5	3432	10042	6613	3432	10042	6613
6	3451	10070	6622	6427	18491	12067
7	3452	10131	6682	5952	17556	11607

5.7 Improvement of Load Balance Distribution

The runtime mesh refinement approach, described before, brings unbalanced distribution of load after it is called, since some processes may to have new data elements to compute and others not. These new data elements are not redistributed among all processes. Because of this, the number of data elements to compute is higher in some processes, where the sub-mesh is refined, than others.

5.7.1 Unbalanced Load Problem

Table 5.2 presents the number of decomposed elements for a domain with 100 Km of mesh resolution divided in 8 processes before and after the OMR call.

In this table it is possible to see that the number of Vertices, Edges, and Triangles for the processes 2, 3, 6 and 7 increase after the OMR execution. The localization of the increased points depends on the place of the Earth atmosphere where the mesh refinement occurs.

Load balancing can be provided by redistributing the load into all processes. But this involves many data exchanges. On the other hand, the creation of new processes or threads is a simple solution and can be applied for all MPI processes.

5.7.2 OpenMP Solution

In order to better distribute the load among the processes, we have added an OpenMP layer to the MPI program.

OpenMP is a parallel programming interface used to abstract multi-processors architectures. The interface is also a good solution to explore parallelism in multi-core systems (CURTIS-MAURY et al., 2008). This approach enables the execution of parallel loops in order to increase the performance of the code. OpenMP is also a good solution for climatological applications (OSTHOFF et al., 2011a,b).

In this work, OpenMP enables to benefit from thread-based concurrency, added to the MPI parallelism. We choose OpenMP programming interface because it abstract very well loop parallelization. Thus, each MPI process divides the load among a specific number of OpenMP threads.

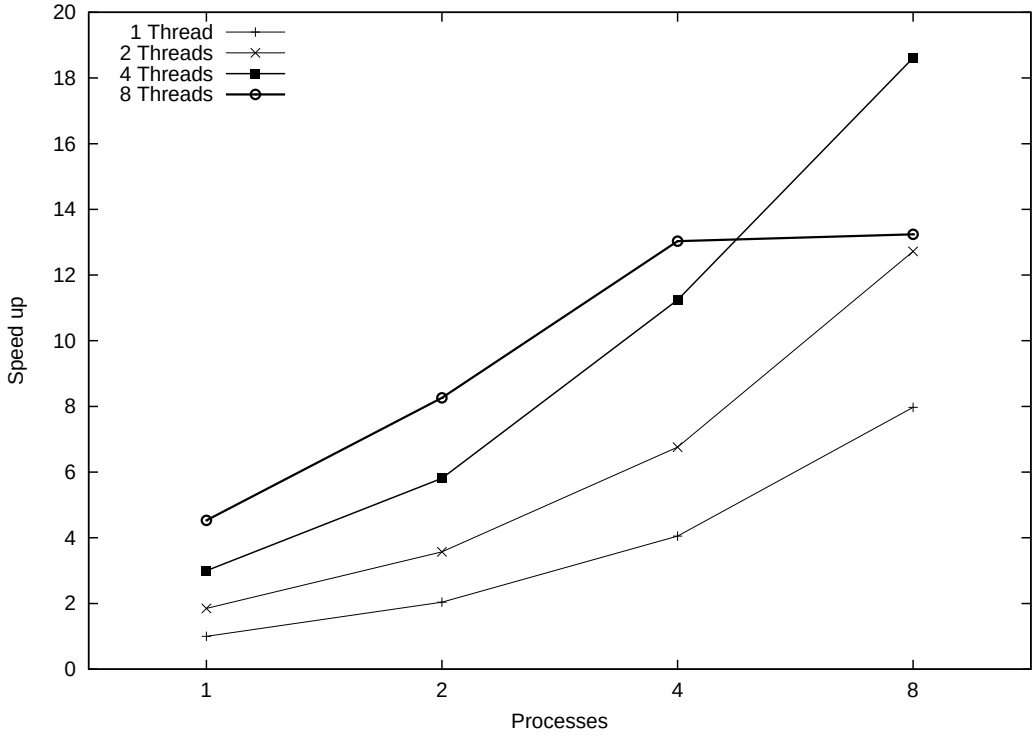

Figure 5.9: Speed up of the iterative step executed **before** the OMR call using different number of OpenMP threads in a simulation with MPI processes.

5.7.3 Performance Impact of OpenMP Threads

The use of OpenMP threads was evaluated in some atmospheric simulations considering meshes with initial horizontal resolution of 100 Km. Figure 5.9 and Figure 5.10 present the speed up of the iterative step of the model before and after an OMR call, respectively. In the tests we compare 1 to 8 MPI processes, each process running in one node, for an atmospheric simulation using an initial horizontal mesh resolution of 100 Km. We run 1 to 8 OpenMP threads in each MPI process.

The results show that the use of threads OpenMP increases performance in the partial iterative steps before and after the OMR execution for all numbers of MPI processes evaluated. The combined use of more than 32 threads/processes does not add a significantly performance because the most part of the execution time is spend with the initialization step.

Table 5.3 presents comparatively the speed up shown in Figure 5.9 and Figure 5.10. The first column indicates the partial kind of the iterative execution: $before$ indicates the simulation before the OMR, and $after$ points the simulation after the OMR call. The second column shows the number of MPI processes used in each partial kind of the iterative execution. The third to the seventh column presents the speed up obtained using 1, 2, 4, and 8 OpenMP threads. The initial speed up is based on the sequential execution of each part of the iterative step.

For the simulation results presented in the table, the first lines considers only OpenMP threads in the simulation, the third column uses only MPI processes, and the other measurements combine MPI processes with OpenMP threads. The results presented in the table show that the second part of the iterative step has a speed up close to the first part in most of the cases. The use of more threads improves load balancing for the last part of

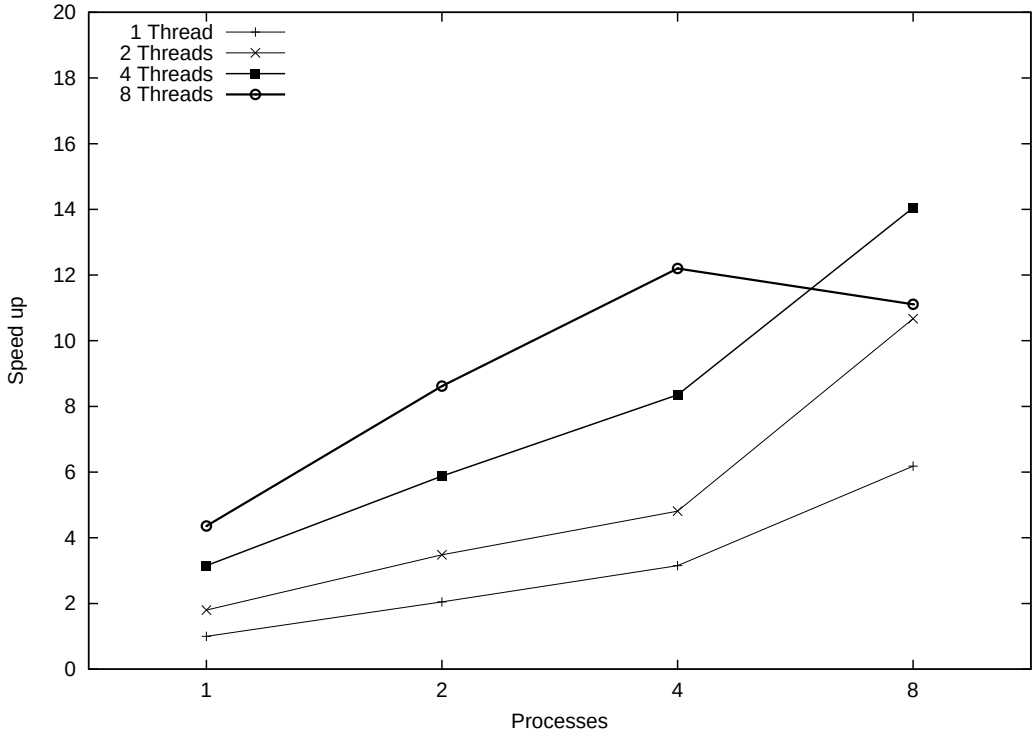

Figure 5.10: Speed up of the iterative step executed **after** the OMR call using different number of OpenMP threads in a simulation with MPI processes.

Table 5.3: Speed up for the iterative execution steps before and after an OMR.

Step	Processors	1	2	4	8
before	1	1.00	1.85	3.00	4.53
after	1	1.00	1.80	3.15	4.36
before	2	2.04	3.57	5.81	8.26
after	2	2.05	3.48	5.88	8.62
before	4	4.05	6.76	11.24	9.70
after	4	3.15	4.81	8.35	8.86
before	8	7.97	8.16	10.69	13.24
after	8	6.18	8.68	10.55	11.11

the iterative step and, consequently, less total execution time.

5.8 Conclusions of this Chapter

In this chapter we have presented online mesh refinement as a way to improve the mesh resolution for climatological models without a significant increase in the execution time. This refinement scheme enables to refine the global mesh of a model during the execution of the code without rebooting the application. Mesh refinement at execution time is critical for climatological models that will cover the impact of local phenomena, inputing more resolution only when it is necessary.

We presented partial and comparative execution time in order to evaluate the dynamic mesh refinement. The partial measurement results show that there is a time spent with the

refinement step. However, it pays because we do not need to run the code considering all Earth surface, with more resolution, all the time. Thus, high resolution is only adopted when special climatological conditions occur.

We also evaluated a mixed MPI/OpenMP parallel implementation. The mixed implementation improves the parallel performance for simulations of the model, where mesh refinements occurs at execution time. In this sense, next chapter discusses more about the use of different parallel programming interfaces in order to perform atmospheric applications on hardware of multi-levels of parallelism.

6 MULTI-LEVEL PARALLELISM

In Chapter 4, OLAM was executed over a multilevel parallel architecture, obtaining restricted speed up in the performance results. In the end of Chapter 5, the combined use of MPI and OpenMP was utilized to provide better performance to executions using Online Mesh Refinement. In this chapter we continue to discuss how applications for simulating atmospheric models can well exploring different levels of parallelism.

6.1 Motivation to Explore Multilevel Parallelism

Theoretically it is possible to determine an optimal distribution of load if we know the processing capacity of each level of parallelism and the stream execution of the application. However, some factors like irregular execution of the code, large waiting time for synchronization and load redistribution at execution time, do not ensure a good performance for the application.

Different execution times can be obtained in a multilevel parallelism environment, depending how the parallelism of the application is express and distributed in each parallel level. Because of this, it is important to define the number of tasks and what kind they are (threads, processes, ...) for each type of application or class of applications to efficiently explore a parallel architecture. This depends basically of the granularity of the tasks to be performed in each parallel level.

6.2 Related Work: Multi-Level Parallelism in Atmospheric Models

There are several works describing the use of multi-core processors and GPUs to compute applications of domain decomposition, fluid dynamics and, also, weather forecast (COHEN; GARLAND, 2009).

Hybrid programs that combine multiple parallelization paradigms, such as message passing and/or multi-threading with an accelerator library, are still relatively rare (HACKENBERG; JUCKELAND; BRUNST, 2012). Their importance, however, has increased as hybrid HPC systems such IBM Cell and NVidia GPU clusters.

In (LINFORD; SANDU, 2011), methods for improving the performance of two-dimensional and three-dimensional atmospheric simulations of constituent transport are examined. A offloading function approach is used in a 2D transport module, and a vector stream processing approach is used in a 3D transport module. Two methods for transferring noncontiguous data between main memory and accelerator local storage are compared (LINFORD, 2010). The results of the study demonstrates the potential use of heterogeneous multicore chipsets to speed up geophysical simulations, through the use of an

IBM BlueGene/P with eight Intel Xeon cores on a single PowerXCell 8i chip.

In (MICHALAKES; VACHHARAJANI, 2008) is shown the speed up for a computationally intensive portion of the Weather Research and Forecast (WRF) model, increasing $8\times$ the performance on a variety of NVIDIA GPU. This change alone in the model speeds up the whole weather model by $1.23\times$.

In (SHIMOKAWABE et al., 2010) is presented a full CUDA porting of the high resolution weather prediction model ASUCA. ASUCA is a next-generation of a production weather code, developed by the Japan Meteorological Agency. ASUCA is similar to WRF. Benchmark on the 528 (NVIDIA GT200 Tesla) GPU TSUBAME Supercomputer at the Tokyo Institute of Technology demonstrated over 80-fold speedup and good weak scaling, achieving 15.0 TFlops in single precision for a mesh with $6956 \times 6052 \times 48$ elements.

WRF and ASUCA are examples of local atmospheric models. In our work we provide parallel implementations for a global atmospheric model in order to run the experimental executions over multi-core and GPUs cluster.

6.3 OLAM Parallel Task

A Parallel Task is an abstraction that defines the granularity of a concurrent execution, that is, a set of instructions that necessarily operate on a sequential execution flow. The observation of where the parallelism can be found in an application, considering the input data that are processed and the dependencies that exist among these data, helps in the determination of the parallel task.

A parallel task is defined in OLAM code by data structures that store the physical atmospheric state, and functions (methods) that manipulate these data structures, simulating the atmospheric conditions during the elapse of time. A task can be, for example, a process, a thread or other execution kind abstracted by a programming interface.

6.3.1 Data Structures for Atmospheric States

Each element of the discretized atmospheric domain (mesh point) has values of data structures associated with itself. Thus, the number of elements contained in a determined data structure is equal to the number of discretized points of the atmosphere.

The data structures store the values of different physical proprieties that are simulated, like pressure, temperature or wind velocity. Some data structures are associated with the discretized points of the Earth surface (triangles), while others are associated with the discrete edges. In both cases, each element of the data structures has also values associated for each level of the vertical atmosphere column, according to the number of levels of this dimension.

Examples of data structures that represent the atmospheric proprieties are shown in Algorithm 6.1.

The number of elements allocated for the first pointer of these data structures includes all horizontal discretized points of the Earth. Vertical level points are associated to each horizontal point (second pointer) in order to storage the numerical value of each physical propriety from the discrete atmosphere.

6.3.2 Procedures or Methods to be Executed

The simulation of the model involves the invocation of several functions iteratively. These functions are controlled by the discrete time of the simulation. Each step of the

Algorithm 6.1 Data structures for atmospheric proprieties variables.

```
double **ump; //past horizontal momentum [kg/(m^2 s)]
double **umc; //current horizontal momentum [kg/(m^2 s)]
double **wmc; //current vertical momentum [kg/(m^2 s)]
double **uc; //current horizontal velocity [m/s]
double **wc; //current horizontal velocity [m/s]
double **sh_w; //total water specific density [kg_wat/kg_air]
double **sh_v; //specific humidity [kg_vap/kg_air]
double **thil; //ice-liquid potential temperature [K]
double **theta;//potential temperature [K]
double **press;//air pressure [Pa]
double **rho; //total air density [kg/m^3]
```

iteration is equivalent to a determined transition of the real time.

The functions that manipulate the data structures simulate the transition of the atmospheric state. Each function operates on some data structures. Many computational operations involve the interaction of elements of a data structure or among data structures of neighbor discretized points of the mesh. These operations are made by looping through each element of the mesh. Usually, nested loops are used in order to follow the loop of horizontal and vertical elements of the mesh.

6.3.3 Data Dependencies and Communication Between Tasks

For parallel executions, the data structures are divided among the parallel tasks. That is, each task invokes the same functions over part of the elements of the data structures. Therefore, each task keeps only the necessary part of the data structures in relation to a sequential process.

The computational operations for a specific element of a data structure depend on the element itself and some neighbor elements. These neighbor elements could be in other processes. Because of this, there is a data dependency among the partitioned elements of the data structures that are parallel distributed.

Consequently, data exchange is necessary in each iterative step of the model if MPI processes are used. Consequently, each process has boundary elements associated for all data structures (subdomain). These boundary elements are updated in each iteration in order to provide the correct values for the computation.

Data exchanges are made in each step of the main loop execution of the code according to Section 2.9. Each process have auxiliary data structures. In these data structures, all discretized elements of the global mesh of OLAM have a local index value and the respective process ranking for this element. The auxiliary date structures for vertices, edges and triangles elements are presented in Algorithm 6.2.

An example of utilization of these data structures for indexing triangles is shown in Figure 6.1. The use of these data structures in association with the data structures of the domain representation, as presented in Section 2.8, enables to map the elements of the physical data structures. The physical data structures need to be interchange in each step of the iterative execution part of the model, after they are updated.

For MPI processes, data elements are grouped and encapsulated according to the destination of the message. One or more elements of distinct data structures can be encapsulated in the same message. The number of distinct elements of data structures encap-

Algorithm 6.2 Auxiliary date structures for indexing processes and local vertices, edges and triangles.

```
typedef struct { //data structure for vertices pts (global)
        int im_myrank; //local subdomain index of this vertex pt
        int irank; //parallel process rank at this vertex pt
} itabg_m_vars;

typedef struct { //data structure for edges pts (global)
        int iu_myrank; //local subdomain index of this edge pt
        int irank; //rank of parallel process at this edge pt
} itabg_u_vars;

typedef struct { //data structure for triangles pts (global)
        int iw_myrank; //local subdomain index of this triangle pt
        int irank; //parallel process rank at this triangle pt
} itabg_w_vars;
```

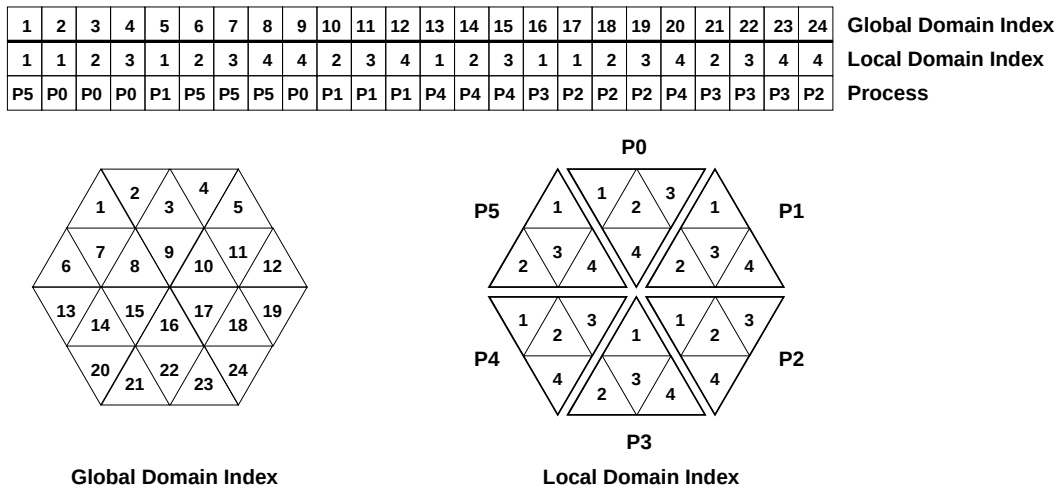

Figure 6.1: Example of indexing triangle elements of the mesh.

sulated can change, depending on the settings of the simulation. A encapsulated message includes also the global index value of the edge or triangle point, in which the elements of physical data structure belong. The receiving process decapsulates the message and finds the local index value, using the global index value received in the message, to save the received physical data.

6.3.4 Computation and Communication Costs

The number of triangles and edges for a specific resolution depends on the Earth circumference as presented in Section 5.4. Considering $N = 5050/R$, where R is the resolution of the mesh, the global number of triangles and edges for a specific R resolution is given by $20(N^2)$ and $30(N^2)$, respectively.

If we divide the total number of points by P processes, each process computes iterative loops over $20(N^2)/P$ and $30(N^2)/P$ triangle and edge mesh elements, respectively. The number of physical data structures associated with mesh elements of triangles or edges differs depending on the parameters of the simulation. At least 20 data structures are used

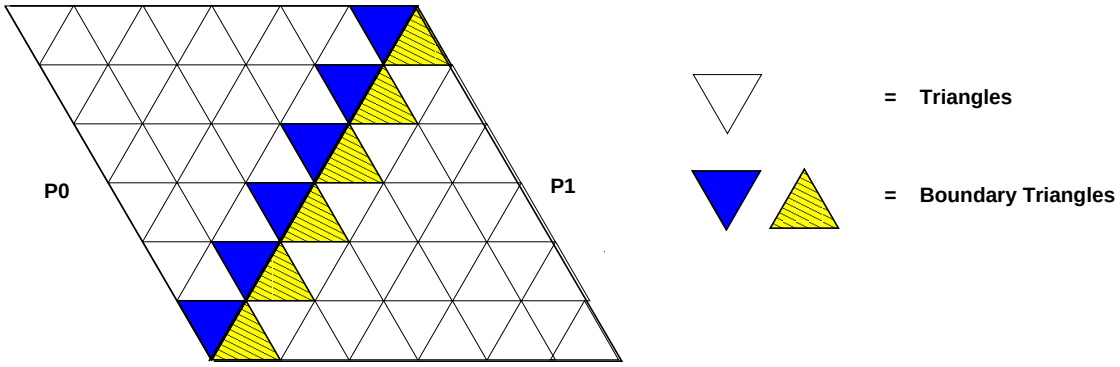

Figure 6.2: Boundary elements to update between Process 0 and Process 1.

in both cases. The number of loop operations using these data structures can to change too.

In a mesh without local refinement, each parallel process has at last 3 neighbor processes. The communication costs to update each one of the 3 neighbor processes is given by $20N/P$, for both triangle and edge elements. Usually, the number of physical data structures encapsulated in each message is between 3 and 5.

Three calls of data exchange are made in each iterative step of the model. In the first message, encapsulated boundary elements of physical data structures associated to the triangle kind of the mesh are send/receive. The two last messages encapsulate boundary elements of physical data structures associated to the edge kind of the mesh.

Figure 6.2 illustrates 2 sub-meshes and the boundary triangles and edges between $P0$ and $P1$ processes, that need to be send/receive to/from the neighbor process. In this example, each process operates data structures associated to 36 hypothetical triangles. Data structures associated to the 6 boundary triangles (filled and shaded triangles) need to be exchanged. In this figure, data exchange was illustrated only for one boundary side of the mesh.

In both, processing and communication of data elements, the number of vertical levels were disregarded in the previous calculations. This dimension of the data structures is not parallelized because it has low granularity. The operations over all vertical levels of a element of a data structure can be concurrently executed by vectorial operations or by PARALLEL FOR executions.

6.4 OLAM Parallel Implementation

Programming a climatological application demands contextual knowledge since there is a strong dependency between code and model. The source code of this kind of application is also very large and difficult to understand and to be parallelized.

In this context, it is important to provide an abstraction of the parallelism, allowing the expression of different levels of granularity. Thus, it is possible to adapt the granularity according to the multiple levels of parallelism provided by an architecture.

6.4.1 OLAM Prototype

In order to reach the objectives of this thesis, a simplified version of OLAM was implemented in the C language. This prototype includes the main characteristics of the model, including domain decomposition, mesh refinement, parallel data distribution, en-

capsulated MPI send and receive resources, and all necessary data structures and functions to execute it. We opt to rewrite the code because C was the first language that allowed to couple with a CUDA code.

The objective of this prototype implementation of OLAM is to provide different levels of abstraction of parallel code for climatological applications. Thus, is possible to adapt different parallel programming interfaces for the simulations, like message-passing resources for distributed memory systems and shared memory interfaces for multiprocessors and multi-core systems.

We use some parallel programming interfaces such as MPI, OpenMP, and CUDA on the developed prototype. Next, we discuss how these interfaces were coupled to the implementation of the OLAM. In the sequence, we will to evaluate the performance impact of these interfaces for the simulations on a hardware platform.

6.4.2 Programming Interfaces Used

Although the OLAM parallel implementation was developed using MPI already, another natural possibility for the parallelization is to use OpenMP pragmas (CHANDRA, 2001). OpenMP is a programming interface that exploits parallelism of shared memory systems. In general, the parallelization of a implementation with OpenMP is very simple, using concurrency on the loops.

OpenMP parallelism was combined with the MPI implementation. Thus, it is possible to exploit parallelism of multiple levels of hardware, at both shared memory (multiprocessors and multi-cores), and distributed memory systems (multicomputer).

Other parallel programming interface to provide multilevel parallelism adopted was CUDA (KIRK; W. HWU, 2010). CUDA was used to access the parallelism provided by GPUs. We choose CUDA because it was the first interface developed to program graphic cards.

OLAM implementation of the prototype using CUDA involves to rewriting some functions of the C code, converting it to a CUDA kernel code. The implementation of the functions that encapsulate the allocation, deallocation, and memory copies between CPUs and GPUs is also necessary. Thus, all temporary array variables used in CUDA kernel functions need to be allocated before the call of the iterative step of the model and released only after the execution of all steps of iterative part. This is make to reduce the number of memory allocations in the CUDA kernel and, consequently, the execution time.

Moreover, in each iterative step of the model, before each call of a CUDA kernel function, it is necessary to move data from CPU to GPU and, after the execution of a iteration, data exchange from GPU to CPU.

CUDA kernel functions were also embedded in a MPI implementation. In this way, three levels of parallelism (GPUs, Cores/Processors, Inter-processors) can be employed for an atmospheric simulation.

6.5 Exploration of Multi-Level Parallelism

Multi-level parallelism for climatological models was explored by the combination of MPI with OpenMP or CUDA. MPI processes are created at the beginning of the simulation. New threads OpenMP or CUDA are launched for the iterative part of the simulation.

6.5.1 Implementation

The iterative step of the OLAM prototype is composed by three functions intercalated by MPI encapsulated functions. A pseudo-code of the iterative step is presented in Algorithm 6.3.

Algorithm 6.3 Iterative step of the OLAM prototype.

```
prog_wrtu() {
        hflux();
        send_recv_uf();
        prog_wrt();
        send_recv_w();
        prog_u();
        send_recv_u();
}
```

OpenMP threads are created through the addition of the instruction:

```
#pragma omp parallel for
```

These pragmas are added to the loops of `hflux()`, `prog_wrt()` and `prog_u()` functions.

For the implementation of the CUDA version, each of the three functions of the iterative step are converted to kernel CUDA functions. These functions are called by C functions of the iterative step of the code.

Each GPU core run concurrently data structure elements associated to a discrete mesh point. Data structures are exchanged before and after each function call, between kernel CUDA and CPU memory. To due this we use the function `cudaMemcpy()`.

Both OpenMP and CUDA versions can be combined with a MPI implementation. In this case, if an execution uses more than 1 process, it is necessary to made data exchanges among the MPI processes. In order to evaluate the performance using different configurations of these interfaces, some experiments were made, as presented later.

We compare the numerical results computed for the different parallel implementations of the prototype. The results are similar among all cases, considering the use of identical initial parameters for the simulations.

6.5.2 Execution Environment

The experimental measurements for the tests of the multi-level parallelism were made using two machines. Each machine is composed by a Intel Core i7 930 model, a quad-core processor with 2.80 GHz, Hyper-Threading technology and 8 MB of cache size. These machines have also, each one, 12 GB of RAM memory and a Nvidia GTX 480 graphic card, used for the tests with the CUDA version of the implementation.

GTX 480 is a Nvidia Fermi architecture. It contains 15 multiprocessors (Scalar Multiprocessors - SM) with a processor clock equal to 1401 MHz. Each multiprocessor is composed by 32 CUDA cores (warps). Thus, the total number of CUDA cores is 480 in each GPU. Each core has one floating point and one integer processing unit.

In each simulation, we integrate an atmosphere for 12 hours, considering 60 s for the real time transition of the atmosphere at each iteration. The number of vertical atmospheric levels was 28. The horizontal mesh resolution was set in 60 Km.

The standard deviation for all obtained results was less than 2% in relation to the median time measured.

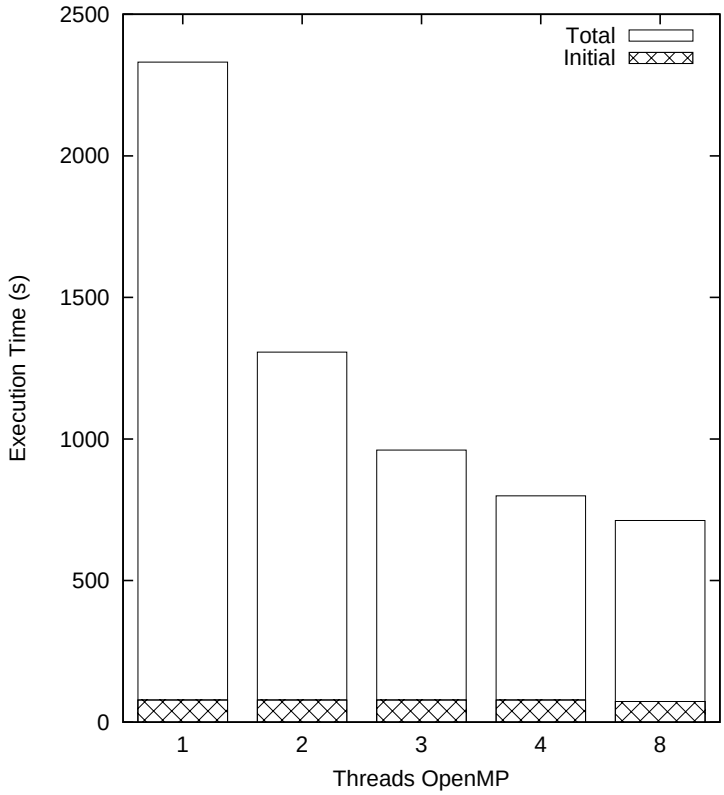

Figure 6.3: Execution time using OpenMP threads running in one node.

For the tests, we evaluate first the use of OpenMP in shared systems. Next, we combine OpenMP pragmas with MPI processes. In the sequence, the use of CUDA was tested using one GPU. At last, CUDA functions were used into MPI processes.

6.5.3 OpenMP Parallelism in Shared Memory Systems

A first test was made evaluating the performance of the model using OpenMP threads.

In Figure 6.3 is presented the initialization step and total execution time of the prototype paralleled only with OpenMP, running on a quad-core machine. As only one process is used, the initialization time is constant, independently of the number of threads used in the iterative step execution.

The results show also that the use of more threads improve scalability for the implementation. The speed up obtained, using 8 threads, was 3.27. Therefore, OpenMP can be an alternative to the use of MPI processes in shared memory systems like multi-core processors.

6.5.4 OpenMP and MPI Multi-Level Parallelism

A second experiment was made, combining OpenMP threads and MPI processes, in order to explore multiple levels of parallelism of the prototype.

In Figure 6.4, the initialization and the total execution time of the model are presented, using different number of OpenMP threads, in simulations on two quad-core machines. X-axis presents the name of the configurations evaluated, where the number before the letter P indicates the quantities of processes used and the number between P and T represents the sum of threads used in the simulations.

The first 4 left columns of the graphic present the results, for all possible number

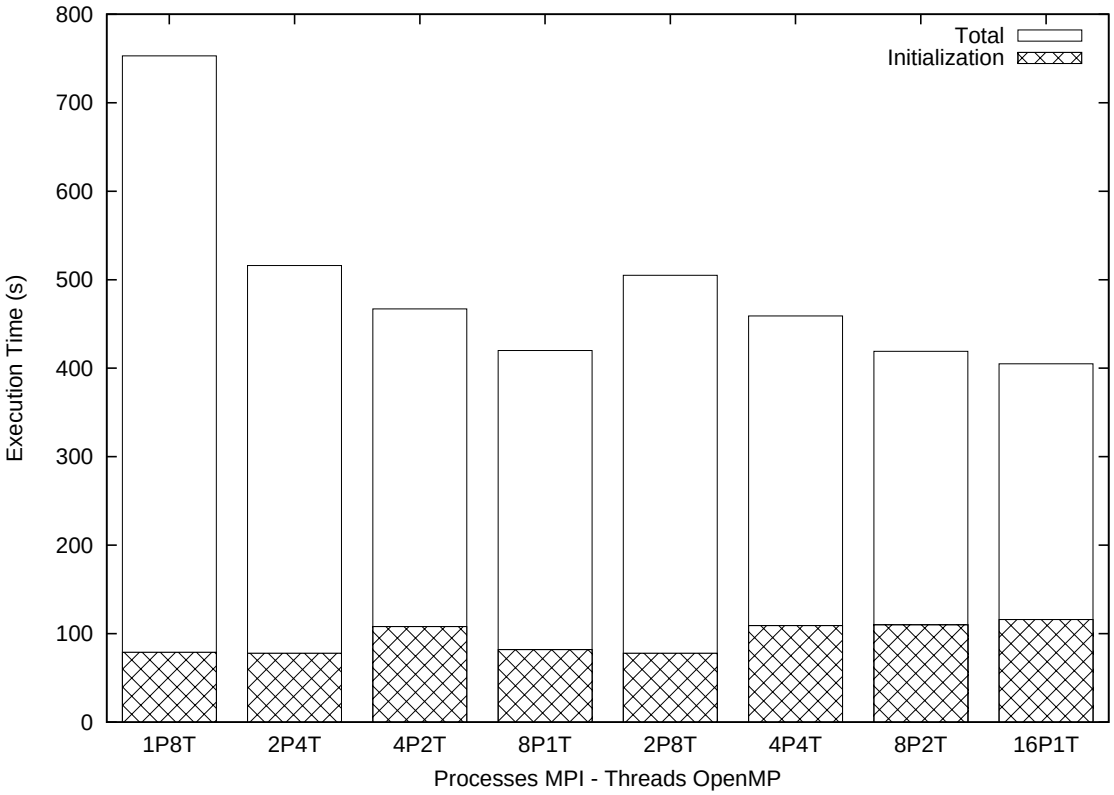

Figure 6.4: Execution time using threads OpenMP and processes MPI in two nodes.

combinations between processes and threads, to run the model with 8 processes/threads (tasks). The last 4 columns show the results for the combination of processes and threads to run the model with 16 parallel tasks.

The use of more processes and less threads presents better execution time for a same number of parallel tasks. Although the increase of the number of OpenMP threads called in a simulation (considering a fixed number of MPI processes) reduces the total execution time, this not occur if the same total number of tasks are compared.

OpenMP threads run concurrently only some functions of the iterative step of the model, as presented in Subsection 6.5.1, whereas MPI processes run all functions in parallel. Moreover, data exchanges are made only between MPI processes running in distinct nodes. Processes running on a shared memory system not need to use the network interface if a function for data exchange is called. Instead, only copies of memory are made.

In these initial results, the use of restricted MPI processes provides a somewhat better performance than the use of OpenMP threads, if the same number of threads and processes are compared. On the other hand, the association of 4 or 8 OpenMP threads to 2 MPI processes running in two nodes decreases the total execution time in relation to the best performance result obtained running the prototype in only one node.

6.5.5 Performance Impact of CUDA for Different Mesh Resolutions

Another alternative to improve performance for the prototype is adopt GPU parallelism. Different execution time were measured for the parallel implementation using CUDA, running on a GPU.

A first test was made to evaluate the performance impact of CUDA, for different mesh resolution sizes. Figure 6.5 shows the execution time measurements, for a sequential

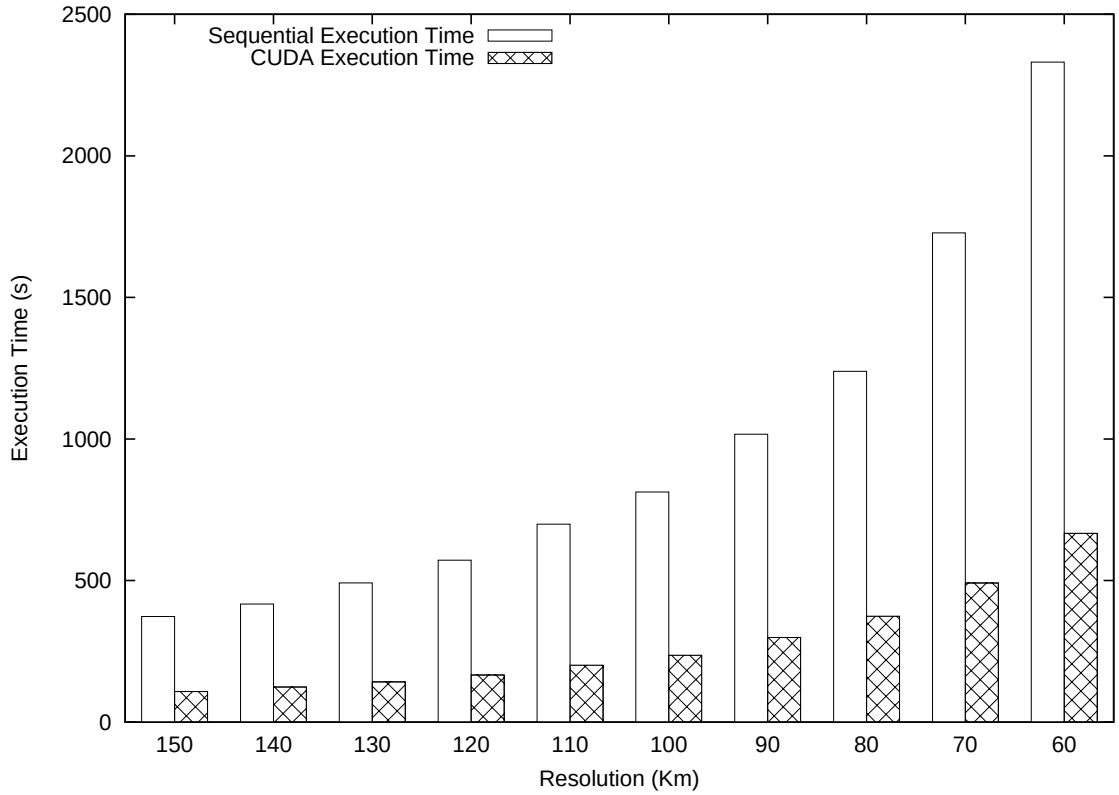

Figure 6.5: Execution time measurement for sequential and CUDA implementation using different mesh resolutions. CUDA threads = 512.

implementation and a CUDA parallel version of the code. The mesh resolution adopted in the tests was from 150 to 60 Km, varying in 10 Km between each case. For the simulations using a CUDA kernel, 512 threads were utilized.

The results show that, the more the mesh resolution set, the more the execution time for both sequential and CUDA implementation. For all mesh resolution cases, CUDA implementation has a smaller execution time than the sequential version. The performance gain of the simulations using CUDA in relation to the version running on only one core/processor is almost similar for all mesh resolutions. The average of performance gains for the 10 cases evaluated was 3.43.

6.5.6 Execution Time Impact for Different CUDA Threads Number

In a second test, a simulation using high mesh resolution was evaluated, comparing the execution time obtained for the use of different numbers of CUDA threads.

Figure 6.6 presents the initialization and the iterative step of the execution time using a number of 128 to 2048 CUDA threads. Only one CPU process was utilized. In the figure is possible to observe that the initialization time is similar for all cases. The tests using 512 threads or less demand more execution time for the iterative step of the model.

The use of 1024 CUDA threads presents the lowest execution time of the iterative step. This configuration provides the best granularity for a resolution of 60 Km of the decomposed domain of the Earth. The number of 1024 CUDA threads will be adopted for the next measurements of execution time.

The total number of concurrent instructions and the number of threads used for a CUDA simulation impacts in the size of the block. The block size is the number of con-

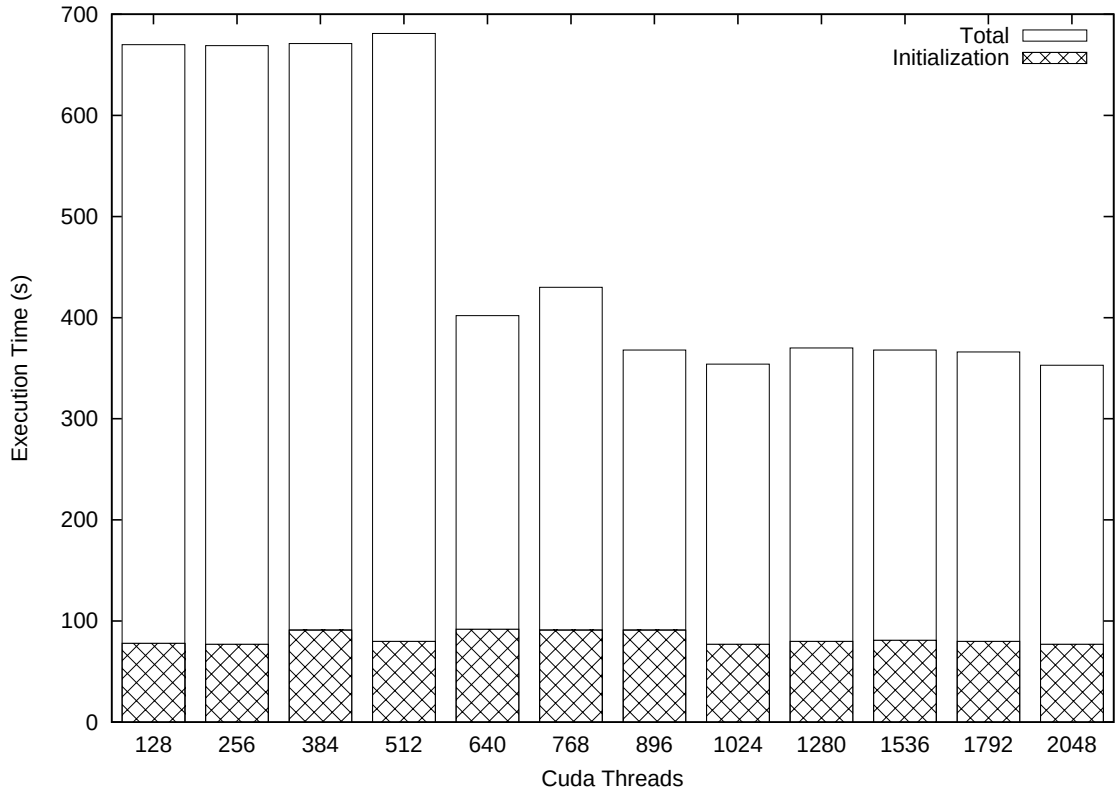

Figure 6.6: CUDA execution time ranging the number of threads used for a simulation of 40 Km of mesh resolution.

current tasks that each thread needs to compute. In the prototype of OLAM, these tasks are composed by instructions computing data structures associated to edges or triangles elements of the mesh.

CUDA threads and its composition in blocks were indexed (only in one dimension) for the simulations. Table 6.1 presents the relation of the number of threads and the block size (number of partial points of the Earth surface) for the execution of each function called in the iterative step. Functions `hflux()` and `prog_u()` operate over edges and `prog_wrt()` over triangles mesh elements.

Although the number of blocks decreases as more threads are used, there are other issues that impact in the efficiency of the execution. These issues are related to the number of registers used in each core and the memory size demanded to store data and instructions.

6.5.7 Using CUDA with MPI processors

The parallelization of the prototype using CUDA was also combined with MPI processes. This mixed implementation allows to compute the model over GPU and cores of a CPU.

In Figure 6.7, the initialization and total execution time of the simulations using only one machine are shown. The number of CUDA threads used in the tests was 1024. We evaluate the use of 1 to 4 MPI processes number.

The results of the figure show that the execution time decreases as more MPI processes are used. Although there is only a graphic card in the computer node, the use of more processes improves better utilization of the CPU and GPU, because the total size of the

Table 6.1: Number of CUDA threads and the respective block size of elements used in each function called in the iterative step.

Threads	Block hflux	Block prog_w	Block prog_u
128	1694	1129	1694
256	847	565	847
384	565	377	565
512	424	283	424
640	339	226	339
768	283	189	283
896	242	162	242
1024	212	142	212
1280	170	113	170
1536	142	95	142
1792	121	81	121
2048	106	71	106

data structures is sub-divided among the processes.

A reduction of the execution time also occurs if more processes are created. Figure 6.8 presents the measured results of initialization and total execution time using 2 machines for the simulations of 1, 2, 4, 8 and 16 MPI processes.

In this case, there are 2 CPUs and 2 GPUs to compute the code of the atmospheric model prototype. Each CPU runs the half of the total number of MPI processes created. MPI processes need to exchange data through the network, if the processes are running in distinct machines.

In the results of the Figure 6.8 is possible to observe that the use of 2 processes not improve performance gain, because the costs of communication is higher than the gain in the parallel computation.

The test using 16 processes has higher initialization time than the other cases evaluated because there are more processes running than cores available. This test was made only for analyze the performance of this condition.

The use of 4 processes in the simulations decreases the execution time in relation to the result obtained, using only one node. The use of 8 processes has the best performance. These results demonstrate that the mixed implementation of CUDA and MPI is a good alternative to explore multiple CPU and GPUs architectures.

6.5.8 Execution Time Comparison Between MPI and CUDA/MPI Implementation

In order to understand the real impact of the use of GPUs in the performance of the atmospheric application, comparisons between the execution time using CPU and CPU/GPU are made.

Figure 6.9 presents the execution time of a version of the prototype running only with MPI processes (white columns) and a version implemented with MPI processes and CUDA threads (scratched columns). Two machines, as the description in Subsection 6.5.2 are used for the tests.

In the figure it is possible to see a reduction of the execution time for both cases, as more processes are used. This is more impacting for the simulation using only MPI processes. However, CUDA threads combined with MPI processes has better performance

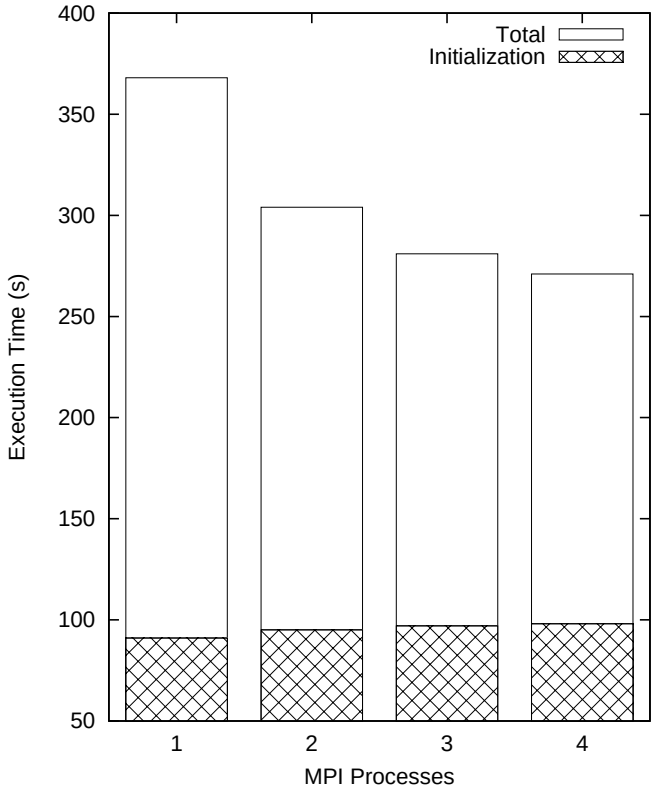

Figure 6.7: Execution time of a combined implementation of CUDA and MPI running on 1 node.

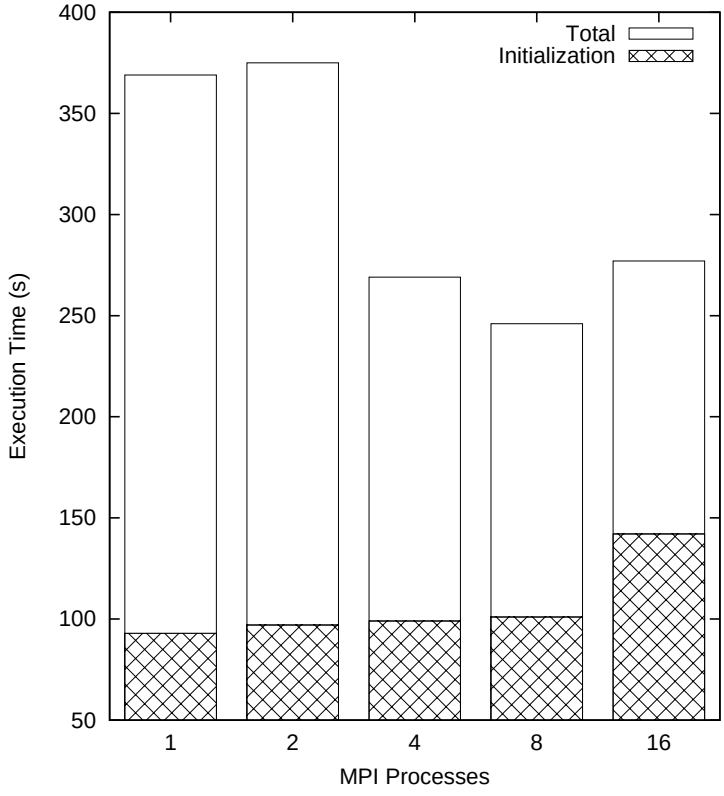

Figure 6.8: Execution time of a combined implementation of CUDA and MPI running on 2 nodes.

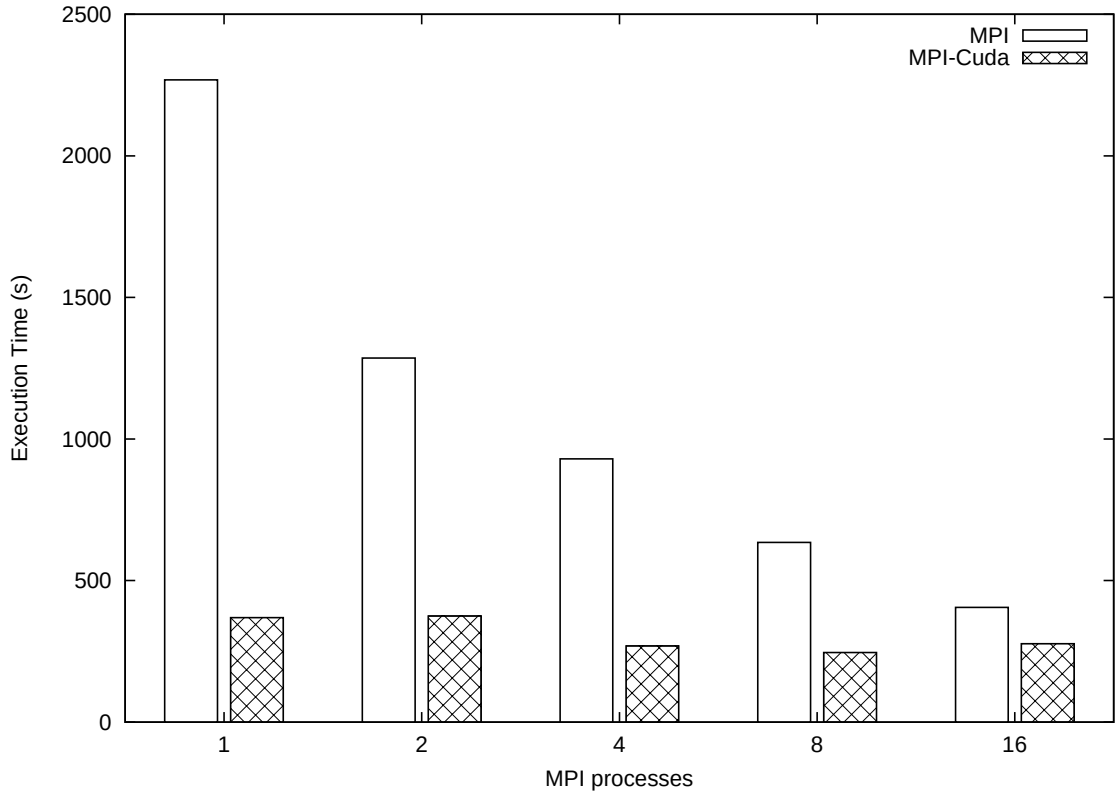

Figure 6.9: Comparison of the execution time between MPI and CUDA/MPI implementation.

than the implementation using only MPI processes.

Considering the use of only one process, the simulation using CUDA combined with MPI has a performance up to $6\times$ over the sequential version. This relation decreases if more processes are used, but the performance of the mixed version is more than $2\times$ faster than the restricted MPI version if 8 processes are used.

The maximization of the use of the hardware resources provides the better performance for the prototype.

6.5.9 CUDA Atmospheric Simulation of the Online Mesh Refinement

The Online Mesh Refinement (OMR) presented in Chapter 5 was parallel implemented and evaluated with MPI and OpenMP combined with MPI. Now, performance evaluations of the prototype implemented using CUDA are made, considering a simulation that realizes an OMR.

In Figure 6.10 is presented a comparison of the execution time of the OLAM prototype, for parallel implementations using MPI, OpenMP and CUDA combined with MPI. The number of MPI processes or OpenMP threads used is 1 to 4. The simulations were made in only one machine.

An OMR occurs after 12 hours of atmospheric integrations. After the OMR call, the execution continues until 24 hours of the atmospheric simulation.

The results show that all configuration decrease the total execution time as more processes/threads are used. Comparatively MPI has a small better performance than OpenMP. However, CUDA combined with MPI has the best execution time for the three cases evaluated. For all cases, the initialization and OMR step has low impact in the total

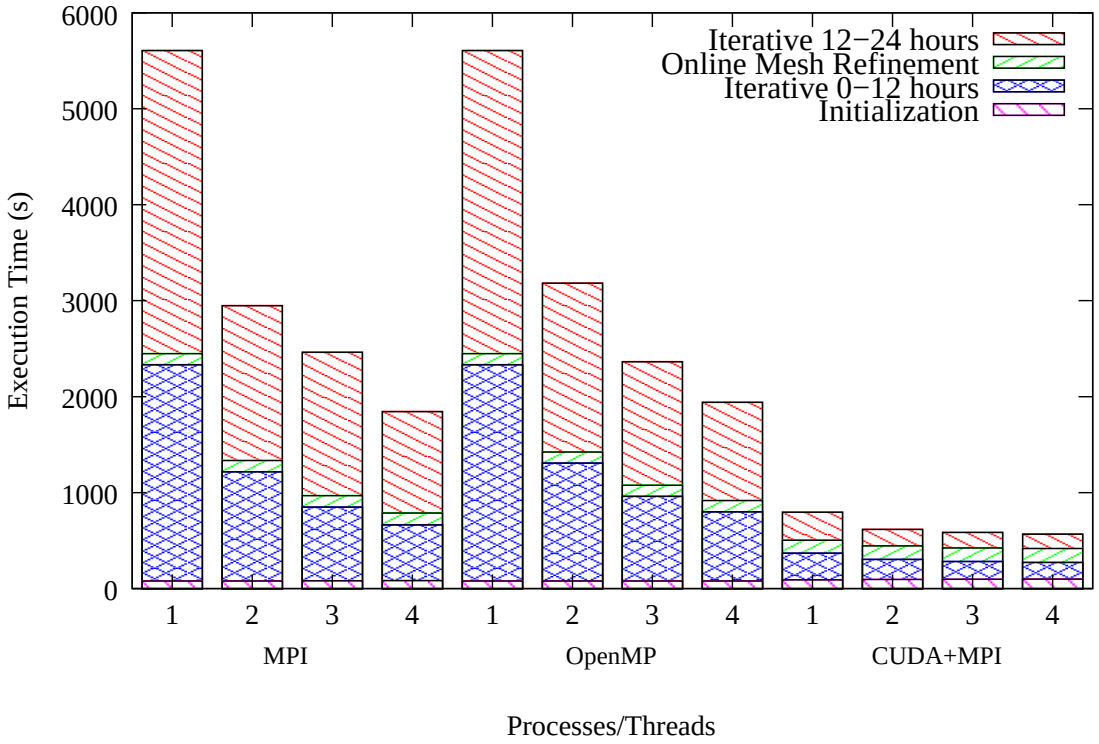

Figure 6.10: Execution time comparison among MPI, OpenMP and CUDA with MPI implementations.

execution time.

Table 6.2 presents the partial execution time of each step evaluated in the tests. In this table is possible to observe that the OMR step of the execution demands more execution time than the initialization, because a large size of memory needs to be reallocated.

Table 6.2: Partial execution time for MPI, OpenMP, and CUDA with MPI implementations.

Interface	Processes/ Threads	Initialization	Partial 0-12 hours	OMR	Partial 12-24 hours
MPI	1	78.48	2253.32	116.91	3158.60
MPI	2	79.92	1137.36	118.69	1612.50
MPI	3	81.56	766.52	121.69	1491.79
MPI	4	83.06	579.93	124.37	1057.58
OpenMP	1	78.48	2253.32	116.91	3158.60
OpenMP	2	78.56	1228.98	116.18	1759.07
OpenMP	3	78.61	882.44	116.46	1285.49
OpenMP	4	78.61	720.43	119.67	1022.87
CUDA/MPI	1	91.98	276.40	136.36	290.88
CUDA/MPI	2	95.70	209.27	139.87	170.22
CUDA/MPI	3	97.76	183.53	143.67	160.04
CUDA/MPI	4	98.86	172.97	145.19	149.96

The execution time of the iterative step executed after the OMR call is larger than the

execution time of the iterative step executed before the OMR call because more elements need to be computed, for the MPI and OpenMP implementations. However, this not occur for some cases of the mixed CUDA/MPI implementation.

The use of CUDA threads combined with more than 1 MPI processes improves reduction of the execution time for the iterative step executed after the OMR call. The increase of the number of atmospheric points to be simulate not impact in the performance in this case, because the GPU explores more parallelism after the increment of data to compute.

The mixed implementation of CUDA and MPI provides also good performance for simulations using OMR.

6.6 Conclusions

The use and adoption of different parallel programming interfaces is a way to extract parallelism from many levels of currently architectures. Each interface evaluated in this work provide resources to represent parallel tasks.

We evaluated, in this chapter, mixed OpenMP/MPI and CUDA/MPI parallel implementations of a OLAM prototype using meshes with 60 Km of horizontal resolution. These mixed implementations were made by the inclusion of OpenMP threads or CUDA kernel functions in a code paralleled with MPI.

The experimental results using 2 nodes, each one composed by a quad-core processor and a GPU board, shown that the use of OpenMP or CUDA threads associated to MPI processes reduces the total execution time of the model. The execution of 4 OpenMP threads in each node increases the performance of the application to $3\times$ (using one node) and $2.5\times$ (using two nodes) in relation to the performance results of simulations running only a MPI process in each node. Tests of the combined implementation of CUDA with MPI speed ups in $6\times$ in relation to a sequential execution and speed ups in more than $2\times$ in relation to a restricted MPI parallel implementation.

We also evaluated the mixed CUDA/MPI implementation in atmospheric simulations with OMR. In some cases, the mixed implementation can also provide load balance for the CUDA tasks after an OMR call. This occurs if the occupancy rate of the GPU multiprocessors is not high for the iterative step simulation executed before the OMR. Thus, the new elements of data structures, arising from the OMR call, can be concurrently executed with the original elements, without increase the total execution time.

CUDA and combined solutions like CUDA/MPI or OpenMP/MPI increase performance for the OLAM prototype in parallel executions. The results prove that it is possible to accelerate the execution time if all available concurrency of the machines is utilized.

In the next chapter, more performance measurements will be made in order to evaluate the use of mixed parallel programming interfaces in atmospheric simulations on large systems.

7 SCALABILITY EVALUATION OF OLAM MULTI-LEVEL PARALLELISM

In Chapter 6 we present ways to explore multiple levels of parallelism. The adoption of different parallel programming interfaces was the solution for increasing the performance of atmospheric models. We evaluate the implementations using limited hardware resources, presenting some partial results.

In this chapter we run the parallel implementations of the prototype version of OLAM using a cluster environment. Thus, is possible to evaluate the scalability of the implementations for some simulations.

7.1 Simulation Environment

All experimental measurements were obtained using the *Newton* cluster of the *Centro Nacional de Supercomputação*. This cluster is interconnected by an InfiniBand network technology, and has currently 28 Sun Fire X2200+ workstations (each one with 2 Quad-Core AMD Opteron 2.2 GHz processors and 16 GB RAM) and a coupled performance of 1.97 TFlops; and 8 GPUs nVidia Tesla distributed in two S1070 units, with coupled performance of 8.28 TFlops. We could use a maximal of 16 nodes of this cluster.

For parallel executions, the processes are distributed among the CPUs and/or GPUs of the nodes of the cluster. In the simulations using GPUs, the number of CUDA threads was fixed in 128. This number was chosen according to previous experiments.

All execution time presented below are measured in seconds. Each execution simulates 12 hours of an atmosphere integration. The vertical level of the atmosphere was divided in 28.

We simulate the OLAM prototype in the cluster using MPI processes, mixed OpenMP threads and MPI processes, and mixed CUDA threads and MPI processes.

7.2 MPI Implementation

A first test was made in order to analyze the impact of the use of MPI processes in the total execution time. Figure 7.1 and Figure 7.2 present the execution time results of an atmospheric integration, using a mesh with 100 Km and 50 Km of horizontal mesh resolution, respectively. The graphics of these figures show the total execution time and the time spent to call the initialization step, using 1 to 32 MPI processes.

Each column of the graphic represents the total execution time for a determined number of processes. We can see that this time decreases as more processes are used. Consequently, there are performance gains.

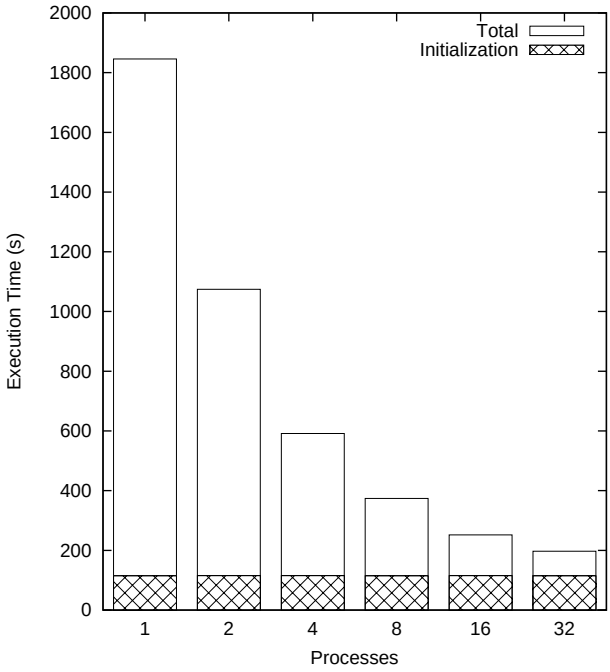

Figure 7.1: Execution time using 1 to 32 MPI processes for a simulation of 100 Km of mesh resolution.

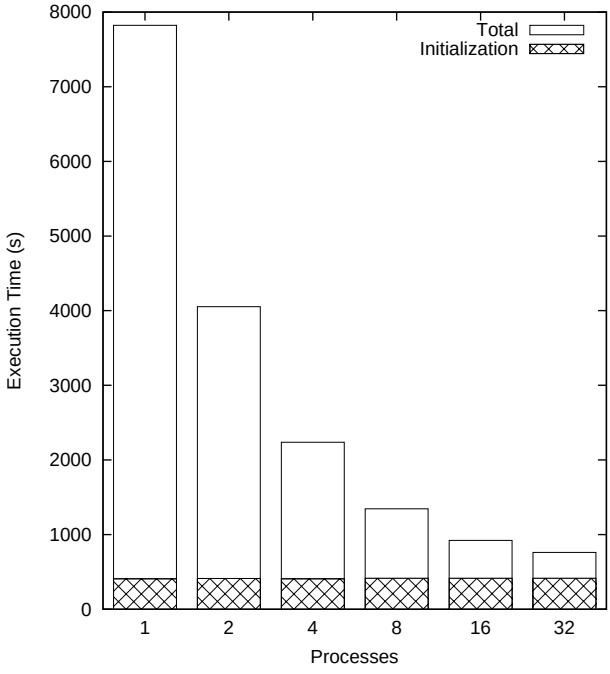

Figure 7.2: Execution time using 1 to 32 MPI processes for a simulation of 50 Km of mesh resolution.

Table 7.1: Speed up for the iterative execution step using MPI processes.

Processes	100 Km	50 Km
1	1.00	1.00
2	1.81	2.04
4	3.64	4.05
8	6.69	7.97
16	12.67	12.81
32	21.12	21.49

The second measurement (scratched area) of each group of processes presents the initialization execution time. The duration time of this step is approximately 115 s for the 100 Km of mesh resolution case and 415 s for the 50 Km of mesh resolution case. The time spent for the initialization step is constant independently of the number of processes used in the atmospheric simulations. The relation between the initialization time and the total execution time decreases if more high mesh resolutions are used.

Table 7.1 presents comparatively the speed up of the iterative step of the simulation results shown in Figure 7.1 and Figure 7.2. The first column shows the number of MPI processes used to each mesh resolution adopted. The second and third columns present the speed up for simulation using 100 and 50 Km of horizontal mesh resolution, respectively. The initial speed up are based on the sequential execution of each part of the iterative step.

The results show that the speed up increases for both mesh resolution cases evaluated. The speed up achieved using 32 processes was around 21. A similar speed up can be obtained for other mesh resolutions.

7.3 MPI and OpenMP Implementation

The use of OpenMP threads was evaluated in some atmospheric simulations considering meshes with horizontal resolution of 100 Km.

Figure 7.3 presents the total execution time of an atmospheric simulation, using 1 to 8 MPI processes. In the tests we compare the performance running 1 to 8 OpenMP threads for different number of MPI processes. Each white filled column of the graphic represents the simulations using only MPI processes. The other columns show the execution time of the MPI processes with the inclusion of OpenMP threads.

The parallelism using OpenMP threads provides the reduction of the total execution time of the model independently of the number of threads used. However, there is a limitation in the performance gain when more than 32 threads/processes are used because the execution time of the initialization step predominates in the total execution time and the iterative step is not more scalable.

Moreover, the OpenMP parallelism is restricted to determined functions of the iterative step of the model, whether the MPI parallelism includes all iterative step. Because of this, the comparison among the execution time of the simulations that use only OpenMP threads (1 MPI process) and the simulations that use only MPI processes (1 OpenMP thread) shows better results for the first case.

Figure 7.4 presents the speed up of the iterative step of the model. In the tests we compare the 1 to 8 MPI processes for an atmospheric simulation. We run 1 to 8 threads in each MPI process.

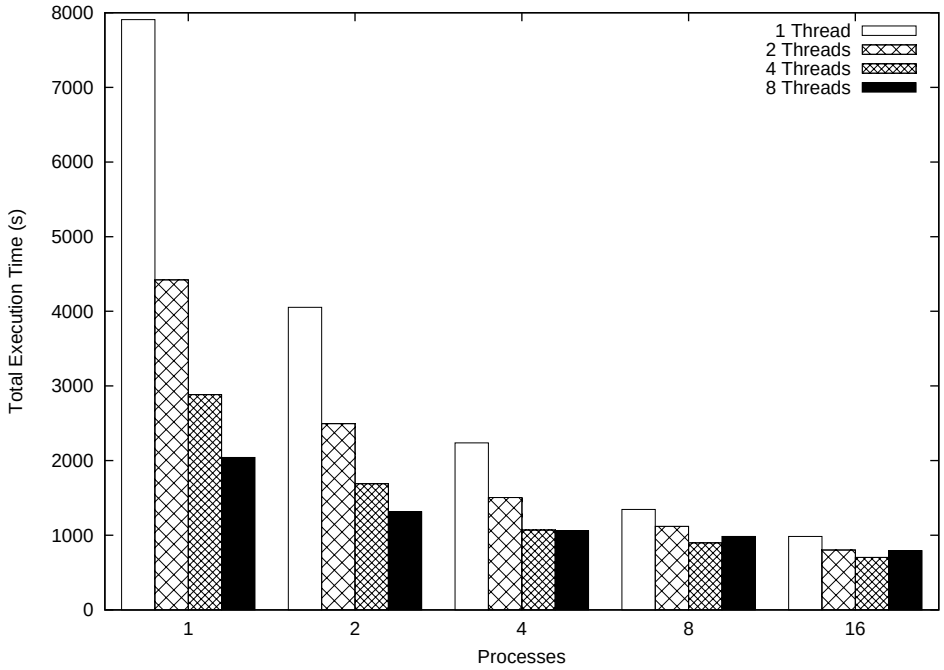

Figure 7.3: Total execution time using different number of OpenMP threads in a simulation with MPI processes. Horizontal mesh resolution of 100 Km.

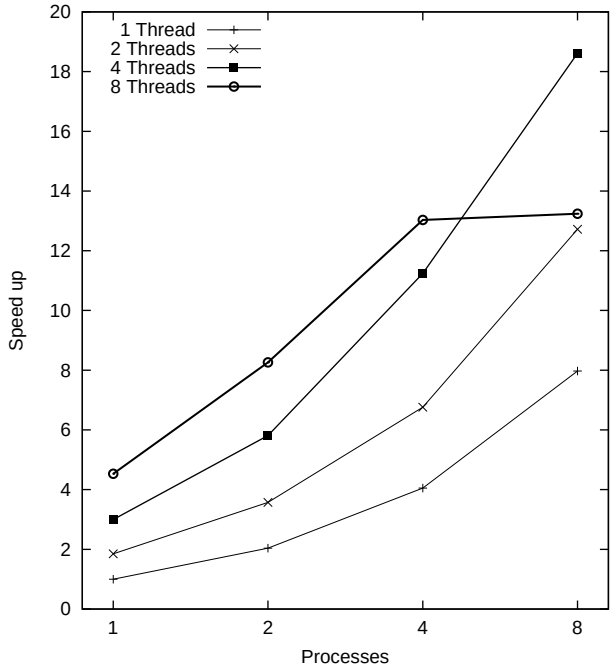

Figure 7.4: Speed up of the iterative step of the model using different number of OpenMP threads in a simulation with MPI processes. Horizontal mesh resolution of 100 Km.

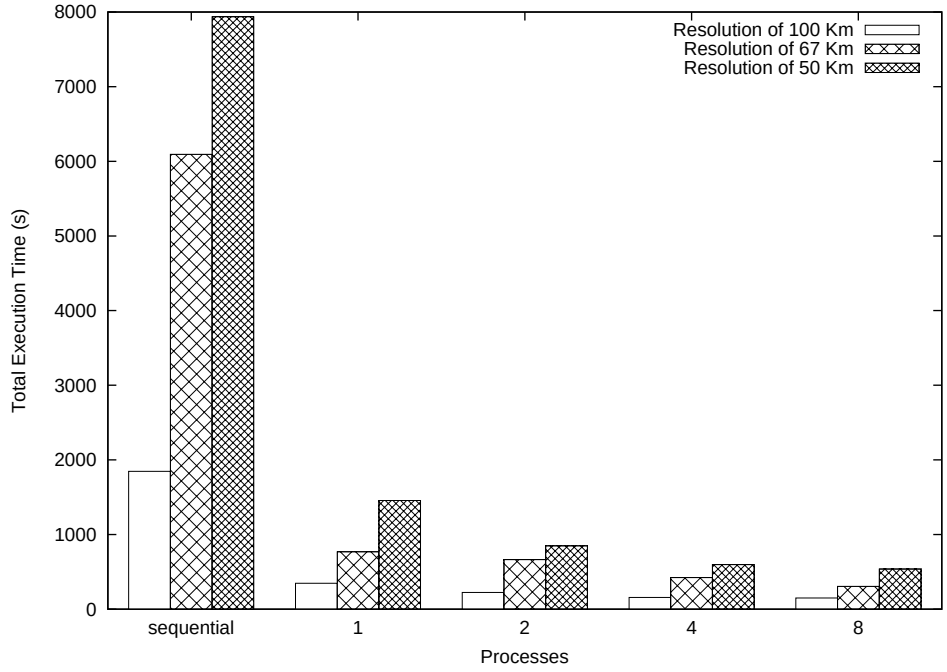

Figure 7.5: Execution time evaluation using different number of GPUs for simulations of 100 Km, 67 Km, and 50 Km of mesh resolution.

The results show that the use of OpenMP threads increases performance in the iterative step for all numbers of MPI processes evaluated. The coupling of 2 and 4 threads in each MPI process increase the speed up in more than 50% and 100%, respectively, in relation to the restricted MPI version for all number of MPI processes used.

In the graphic is also possible to see that the use of 4 threads and 8 processes provides a limited performance gain. This result is obtained because each processor/thread computes a task with low granularity. Thus, the parallel performance not overcome the communication and thread creation costs.

7.4 MPI and CUDA Implementation

Some simulation were also made, exploring GPU parallelism.

Figure 7.5 presents the total execution time for the three resolutions considered in this work (see Section 6.5.2). In this figure, the first three columns show the sequential execution time for each mesh resolution. The sequential execution time not include the use of GPUs. The other columns of the graphic present the execution time using 1, 2, 4, and 8 GPUs.

The results of the Figure 7.5 show that the use of one GPU reduces the total execution time more than $5\times$ in relation to the execution using only CPU processing. This reduction is more expressive as more GPUs are used in the simulations.

Figure 7.6, and Figure 7.7 present the execution time of the initialization and iterative step of the model for a mesh resolution of 100 Km, and 50 Km, respectively. These results demonstrate that the execution time for the initialization step is constant independently of the number of GPUs used. On the other hand, the execution time of the iterative step decreases as more GPUs are included in the computation in all cases evaluated.

Figure 7.8 shows the speed up of the iterative step of the model using 1 to 8 GPUs in simulations with mesh resolution of 100 Km, and 50 Km. For all mesh resolution cases,

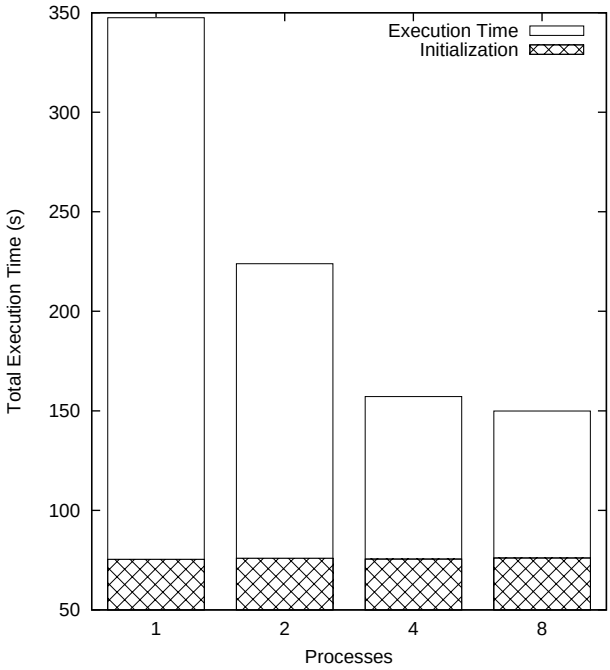

Figure 7.6: Initialization and iterative step execution time for simulations using 100 Km of mesh resolution in a CUDA/MPI mixed implementation.

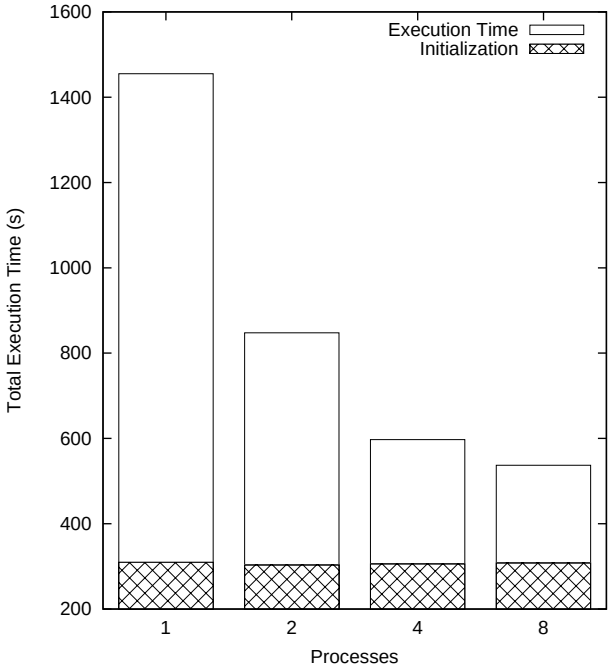

Figure 7.7: Initialization and iterative step execution time for simulations using 50 Km of mesh resolution in a CUDA/MPI mixed implementation.

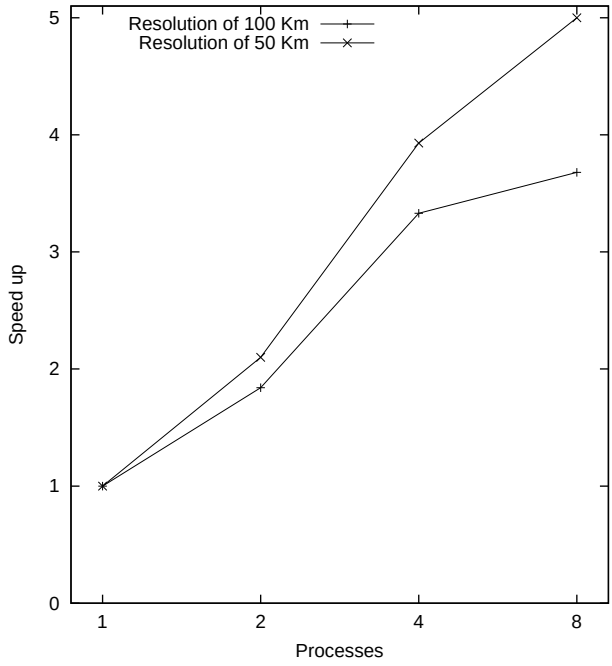

Figure 7.8: Speed up evaluation using different number of GPUs for simulations of 100 Km and 50 Km of mesh resolution.

the speed up increases as more GPUs are used. The graphic shows also that a mesh with high resolution (50 Km) has more speed up than a mesh with low resolution (100 Km) because the difference granularity among the processes. High mesh resolutions have more data structures to compute.

7.5 Conclusion

This chapter presented performance results of parallel implementations of an atmospheric model using MPI, OpenMP and CUDA programming interfaces. We measured the execution time and speed up of a prototype version of OLAM in a cluster system composed by multi-core, multiprocessor and GPU architecture, that is, a multi-level parallelism environment. In order to evaluate the different parallel implementations of the model, we present partial and comparative execution time and speed up using 1 to 32 quad-core processors and 1 to 8 GPUs units of a cluster.

The partial measurement results shown that MPI and MPI combined with OpenMP implementations can increase the parallel performance of the atmospheric model as more processes and/or threads are used. The use of 32 MPI processes achieved an speed up of 21. This speed up could to be larger if more functions of the iterative step of the model were wrote for the prototype. Thus, more data structures will to be computed, increasing the granularity of the processes, and reducing the impact of the initialization step in the total execution time of the model. This possibility could also increase the performance of implementations using mixed OpenMP threads and MPI processes. However, as a prototype, we not implement all functionalities of currently atmospheric models in this thesis.

The restrict use of MPI parallelism in the implementation of the model improves a better execution time in relation to the combined use of MPI and OpenMP parallelism, if we compare the same number of MPI processes again the sum of MPI processes and

OpenMP threads. A similar result was also obtained in Chapter 6. This occurs because the MPI parallel implementation includes all iterative step of the model whether the OpenMP parallelism is restricted to some functions of the iterative step of the model. Although OpenMP threads provides less performance than restricted MPI processes, its use in simulations with OMR can supply better execution time results for the iterative step after an OMR call, as presented in Chapter 5.

CUDA combined with MPI version of the prototype was also evaluated. In the tests, the execution of the model using only one GPU increases $5\times$ the performance in relation to a sequential execution restricted to one CPU. We also evaluate the performance of the prototype computing in more than one GPU. In Chapter 6, this experiment was restricted to 2 GPUs. In this chapter we use 8 GPUs. The results shown that there are an increase of speed up, as more GPUs are used, for all mesh resolutions selected in the tests.

All implementations using mixed programming interfaces presented scalable solutions for the prototype of OLAM. In this context, other atmospheric models could also to be improved by the addition of other kind of parallel tasks in order to explore multiple levels of parallelism. The same could be expected for other applications kind.

8 CONCLUSION AND FUTURE WORKS

Recent performance improvements in both general-purpose and special-purpose processors have come primarily from increased on-chip parallelism. On-chip parallelism with multi-core processors and GPU accelerators can now commonly to be used for running concurrently applications developed using appropriated programming libraries. This new manner to support parallelism has received significant attention in the past of a few years because the large number of cores that can be used for concurrently executions. There are also a tendency to increase the number of cores in GPU and multi-core processors in the next years, contributing to provide exa-escale systems. Thus, the shift to an increasing on-chip parallelism will place new burdens on software application.

On-chip parallelism is of considerable interest to a broader group of parallel applications for high-end supercomputers. These applications have a large processing load and each new developed architecture brings the possibility for increasing the performance of the executions. A significative set of these applications are related to data simulations of domain decomposition problems, like weather and climatological forecasts.

Multi-core and GPU provide a limited parallelism approach for the applications. Furthermore, in currently architectures, there are also parallelism levels among processors and among computers. Each parallelism level was designed for a specific processing granularity. In order to use the best performance of the computers it is necessary to consider all parallel levels to distribute a concurrent application. However, nothing parallel programming interface abstracts all these different parallel levels.

In this context, this thesis investigated how different levels of parallelism can be explored in atmospheric models, including models that provide mesh refinement at execution time, using classical parallel programming interfaces. We used the notion of parallel tasks as a way to abstract the parallel granularity (processes, threads) for a concurrent application. A parallel task for an atmospheric model implementation was defined by data structures that store the physical atmospheric state, and functions (methods) that manipulate these data structures, simulating the atmospheric conditions during the elapse of time.

Multi-level parallelism for a prototype version of OLAM was provided by the combination of MPI with OpenMP or CUDA programming interfaces.

MPI processes were created at the beginning of the simulation. New threads OpenMP or CUDA were launched for the iterative part of the simulation of a prototype version of OLAM code. Thus, it was possible to exploit parallelism at multiple levels of hardware, at both shared memory systems (multiprocessors and multi-cores), and distributed memory systems (multicomputer).

We also propose an Online Mesh Refinement (OMR) approach for parallel distributed unstructured meshes. Nothing atmospheric model provides mesh refinement at execution

time. The objective of the OMR implementation in the context of this work was to show how dynamic high performance applications can benefit if its run on parallel multi-level architectures. The OMR implementation allows local mesh refinement at execution time, increasing the resolution for a discrete representation of a part of the domain. This solution offers higher mesh resolution for atmospheric models with low performance impact, providing also better numerical results.

Experimental measurements for simulations of the multi-level parallelism implementation were made. We obtained execution time and speed up results for the simulation of the prototype, using different mesh resolution sizes. The tests evaluate the implementations using MPI, and mixed versions of MPI and OpenMP, and MPI and CUDA.

The adoption of MPI processes improved a significative speed up. In the tests, the use of 32 processes achieved a speed up of 21. The speed up could to be larger if all functions of the iterative step of a typical atmospheric model were included in the simulations. Thus, more data structures could to be computed, increasing the granularity of the processes.

The mixed OpenMP/MPI implementation provided thread and process parallelism. The experimental results shown that the use of OpenMP combined with MPI reduced the execution time of the simulations. The use of 4 threads in each MPI process number increased the performance in more than $2\times$ in relation to the simulation using only one MPI process in each quad-core processor. Although OpenMP threads provides less performance than restricted MPI processes, OpenMP parallelism is useful for load balance in simulations with OMR.

The results of the mixed CUDA/MPI parallelization version shown that the use of one GPU reduces the total execution time more than $5\times$ in relation to the execution using only CPU processing. This reduction is more expressive as more GPUs are used in the simulations.

All these performance results indicate that is possible to reduce the execution time of atmospheric simulations using different levels of parallelism, through the combined use of parallel programming interfaces. Therefore, more mesh resolution to describe the Earth's atmosphere can be adopted, and consequently the numerical forecasts are more accurate.

The contribution of this thesis is both online mesh refinement and exploration of multiple level of parallelism in atmospheric models.

This work improves the refinement of unstructured meshes at execution time. Unstructured meshes are less considered in domain decomposition works due the difficulty to describe the relation among the discrete elements. The solution provided in this work could to be considered in other kind of unstructured meshes.

The use of multiple representation forms of a parallel task is a solution to compute on different levels of hardware parallelism. This approach is necessary, specially for large applications, to maximize the performance of the executions. The combined use of CPU and GPU is now a tendency for atmospheric models. Research and forecast centers are expending efforts to rewrite piece of meteorological code to better perform in multi-core and many-core architectures.

Future Work

In this work we use MPI, OpenMP and CUDA to improve multiple levels of parallelism for climatological models. However, a combined test, using the three interfaces was not made. Although the simulations using MPI and CUDA interfaces present good execution time results, experiments considering the three interfaces can to emerge also excellent performance.

The OpenMP implementation could to be rewrote to change the place of the loop parallelism. The currently implementation considers the concurrency of each function inner the iterative step. However, all iterative step could to be computed by OpenMP threads, as occurs in the parallelization using MPI processes. In this case, some variables need to be set as private, in order to maintain the accuracy of the results.

Another parallel programming interface, like Intel Threading Building Blocks (TBB) and/or Message-Passing Interface 2, that offer run time creation of processes, could be also evaluated in order to maximize the use of the hardware resources by atmospheric simulations.

We are planning to evaluate the behavior of the atmospheric model prototype in GPU architectures changing the number of CUDA threads. Although the number of threads was also evaluated in some simulations, it was restricted to only one mesh resolution size. The addition of more threads can increase the performance of the model in some specific mesh resolutions or number of GPUs used in the simulations.

Cooperation

This thesis was developed under cooperation projects.

The work conducted in this thesis is part of the *Atmosfera Massiva* project, a cooperation among GPPD (*Instituto de Informática - UFRGS*) with another brazilian research groups, like LNCC, INPE and CPTEC. The project was supported by CNPq (*edital Grandes Desafios*).

The general purpose of this research project was to study the impact of new multi-core architectures and the multiple levels of parallelism in meteorological and environmental models. These cooperation produced some works, that were published as articles in conference proceedings.

This thesis was also developed as part of an international cooperation between *Instituto de Informática - UFRGS* and *Technische Universität Berlin*, Germany. A sandwich doctoral was made in the *Fachgebiet Kommunikations- und Betriebssysteme (KBS) - Institut für Telekommunikationssysteme* in the period of October, 2010 and June, 2011, under supervision of prof. Hans-Ulrich Heiss. The interchange was supported by CNPq/DAAD.

In this period it was made the implementation of the online mesh refinement to the atmospheric prototype. Some tests was also conducted in a cluster of the KBS group. The results of this part of the work were published in the CLCAR and SBAC-PAD conferences. See: (SCHEPKE et al., 2011a) and (SCHEPKE et al., 2011b).

Publications

During the doctoral studies some papers are submitted and approved in workshops and conferences, as listed below:

- Performance Evaluation of an Atmospheric Simulation Model on Multi-Core Environments - Proceedings of Conferencia Latino Americana de Computación de Alto Rendimiento (CLCAR 2010) (SCHEPKE et al., 2010).

- Improving Core Selection on a Multicore Cluster to Increase the Scalability of an Atmospheric Model Simulation - Proceedings of XXIX Iberian-Latin-American Congress on Computational Methods in Engineering, 2010, Buenos Aires. Mecanica Computacional Vol. XXIX. Buenos Aires : Asociación Argentina de Mecanica Computacional (CILAMCE 2010) (OSTHOFF et al., 2010).

- I/O Performance Evaluation on Multicore Clusters with Atmospheric Model Environment - 1st Workshop on Applications for Multi and Many Core Architectures

(WAMMCA 2010) - 22nd International Symposium on Computer Architecture and High Performance computing (SBAC-PAD 2010) (OSTHOFF et al., 2010).

- Online Mesh Refinement in Parallel Meteorological Applications - Proceedings of Conferencia Latino Americana de Computación de Alto Rendimiento (SCHEPKE et al., 2011a).
- I/O Performance of a Large Atmospheric Model using PVFS - Actes des 20éme Rencontres francophones du parallélisme (RENPAR'11) (BOITO et al., 2011).
- GPU for Accelerators Performance Evaluation on Atmosphere Model's Application System - Proceedings of XXX Iberian-Latin-American Congress on Computational Methods in Engineering, 2011, Ouro Preto. Mecanica Computacional Vol. XXX (CILAMCE 2011) (OSTHOFF et al., 2011b).
- Improving Performance on Atmospheric Models through a Hybrid OpenMP/MPI Implementation - The 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2011) (OSTHOFF et al., 2011a).
- Why Online Dynamic Mesh Refinement is Better for Parallel Climatological Models - 23th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2011) (SCHEPKE et al., 2011b).
- Trace-based Visualization as a Tool to Understand Applications I/O Performance - 2st Workshop on Applications for Multi and Many Core Architectures (WAMMCA 2011) - 23nd International Symposium on Computer Architecture and High Performance computing (SBAC-PAD 2011) (KASSICK et al., 2011).
- Evaluation of Programming Models for Atmospheric Application - IADIS International Conference Applied Computing 2011 (OSTHOFF et al., 2011).
- Exploring Multi-Level Parallelism in Atmospheric Applications - XIII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC 2012) (SCHEPKE; MAILLARD, 2012).

Some of the previous papers where the basis to write and to publish two journal articles, as presented below:

- Atmospheric Models Hybrid OpenMP/MPI Implementation Multicore Cluster Evaluation - International Journal of Information Technology, Communications and Convergence (IJITCC) (OSTHOFF et al., 2012).
- Online Mesh Refinement for Parallel Atmospheric Models - International Journal of Parallel Programming (IJPP) - approved and waiting for publication (SCHEPKE et al., 2012).

A book chapter was also produced together with other researches:

- Improving Atmospheric Model Performance on a Multi-Core Cluster System - Atmospheric Model Applications (OSTHOFF et al., 2011).

This chapter presents in the Atmospheric Model Applications book some aspects evaluated during the doctor degree work.

All these publications are important in the design of the thesis and to evidence the proposed solutions adopted in the work.

REFERENCES

ADCROFT, A.; HILL, C.; MARSHALL, J. Representation of topography by shaved cells in a height coordinate ocean model. **Monthly Weather Review**, Boston, MA, v.125, p.2293–2315, 1997.

AMD. **Model Number Methodology for AMD Opteron 6000 Series Processors**. Available at: <http://www.amd.com/us/products/server/processors/6000-series-platform/pages/6000-series-model-number-methodology.aspx>. Last access: June, 2011.

ANDREWS, G. R. **Foundations of Multithreaded, Parallel, and Distributed Programming**. Reading, Massachusetts: Addison-Wesley, 2001. 664p.

ASANOVIC, K. et al. **The Landscape of Parallel Computing Research**: A View from Berkeley. Berkeley, CA: EECS Department, University of California, Berkeley, 2006. (UCB/EECS-2006-183).

ASANOVIC, K. et al. A view of the parallel computing landscape. **Communications of the ACM**, New York, NY, USA, v.10, n.52, p.56–67, 2009.

AVISSAR, R.; PIELKE, R. A parameterization of heterogeneous land surfaces for atmospheric numerical-models and its impact on regional meteorology. **Monthly Weather Review**, Boston, MA, v.117, p.2113–2136, 1989.

BERGMAN, K. et al. **ExaScale Computing Study**: Technology Challenges in Achieving Exascale Systems. Peter Kogge, Editor & Study Lead. 2008.

BERNHOLDT, D. E. Component architectures in the next generation of ultrascale scientific computing: challenges and opportunities. In: COMPFRAME '07: PROCEEDINGS OF THE 2007 SYMPOSIUM ON COMPONENT AND FRAMEWORK TECHNOLOGY IN HIGH-PERFORMANCE AND SCIENTIFIC COMPUTING, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.1–10.

BLUMOFE, R. D. et al. Cilk: An Efficient Multithreaded Runtime System. In: JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING, 1995, San Francisco, CA, USA. **Anais...** Elsevier, 1995. p.207–216.

BOITO, F. Z. et al. I/O Performance of a Large Atmospheric Model using PVFS. In: ACTES DES 20ÉME RENCONTRES FRANCOPHONES DU PARALLÉLISME (RENPAR'11), 2011, Saint-Malo, France. **Anais...** INRIA, 2011.

BROWN, S. D. et al. **Field Programmable Gate Arrays**. Berlin, Germany: Springer, 1997.

BUYYA, R. **High-Performance Cluster Computing**: Architectures and Systems. USA: Prentice Hall, 1999.

BYNA, S.; SUN, X.-H.; HOLMGREN, D. Modeling Data Access Contention in Multicore Architectures. In: ICPADS '09: PROCEEDINGS OF THE 2009 15TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, 2009, Washington, DC, USA. **Anais...** IEEE Computer Society, 2009. p.213–219.

CARVALHO, A. de. **Grand Challenges for Computer Science Research in Brazil 2006 - 2016**. Workshop Report, 2006; Available at: <http://sistemas.sbc.org.br/Arquivos ComunicacaoDesafios_ingles.pdf>. Last access: Sep. 2010.

CHAMBERLAIN, B. L.; CALLAHAN, D.; ZIMA, H. P. Parallel Programmability and the Chapel Language. **International Journal of High Performance Computing Applications**, London, v.21, n.3, p.291–312, August 2007.

CHANDRA, R. **Parallel Programming in OpenMP**. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2001.

CHARLES, P. et al. X10: an object-oriented approach to non-uniform cluster computing. **SIGPLAN Not.**, New York, NY, USA, v.40, n.10, p.519–538, 2005.

COHEN, J.; GARLAND, M. Solving Computational Problems with GPU Computing. **Computing in Science and Engineering**, Los Alamitos, CA, USA, v.11, p.58–63, 2009.

CURTIS-MAURY, M. et al. An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In: OPENMP SHARED MEMORY PARALLEL PROGRAMMING, 2005., 2008, Heidelberg, Berlin. **Proceedings...** Springer-Verlag, 2008. p.133–144. (IWOMP'05/IWOMP'06).

DEBREU, L.; VOULAND, C.; BLAYO, E. AGRIF: Adaptive Grid Refinement In Fortran. **Comput. Geosci.**, Tarrytown, NY, USA, v.34, n.1, p.8–13, 2008.

DONGARRA, J. Trends in High Performance Computing. **The Computer Journal**, Oxford, UK, v.47, n.4, p.399–403, 2004.

DONGARRA, J. et al. The Impact of Multicore on Computational Science Software. **CTWatch Quarterly**, Urbana, IL, v.3, n.1, February 2007.

DONGARRA, J. et al. The International Exascale Software Project Roadmap. **International Journal of High Performance Computing Applications**, London, v.25, n.1, p.3–60, 2011.

DONGARRA, J. et al. (Ed.). **The Sourcebook of Parallel Computing**. San Francisco, CA, USA: Elsevier, 2002.

EXA CORPORATION. **PowerFLOW for CFD - Driving Fluid Flow Simulation Technology Into the next Century**. Available at: <http://www.exa.com/newsite/frames/powerflowmaster.html>. Last access: Aug. 2010.

FANG, H. et al. Lattice Boltzmann method for simulating the viscous flow in large distensible blood vessels. **Physical Review E**, New York, USA, v.65, n.5, p.1–11, May 2002.

FAZENDA, A. L. et al. **First Time User Guide (BRAMS Version 4.2)**. 2011.

FOSTER, I.; KESSELMAN, C. **The Grid**: Blueprint for a New Computing Infrastructure. 2.ed. San Francisco, CA, USA: Morgan Kaufmann, 2003. 800p.

FRIGO, M. Multithreaded Programming in Cilk. In: PASCO '07: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.13–14.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. In: IN PROCEEDINGS OF THE SIGPLAN '98 CONFERENCE ON PROGRAM LANGUAGE DESIGN AND IMPLEMENTATION, 1998, New York, NY, USA. **Anais...** ACM, 1998. p.212–223.

GALANTE, G. **Métodos Multigrid Paralelos em Malhas Não Estruturadas Aplicados à Simulação de Problemas de Dinâmica de Fluidos Computacional e Transferencia de Calor**. 2006. 130p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

GARLAND, M. et al. Parallel Computing Experiences with CUDA. **IEEE Micro**, Los Alamitos, CA, USA, v.28, n.4, p.13–27, 2008.

GEIST, A. et al. **PVM**: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. Cambridge, MA, USA: MIT Press, 1994.

GEPNER, P.; KOWALIK, M. F. Multi-Core Processors: New Way to Achieve High System Performance. In: INTERNATIONAL SYMPOSIUM ON PARALLEL COMPUTING IN ELECTRICAL ENGINEERING (PARELEC'06), 2006, Washington, DC, USA. **Anais...** Institute of Electrical and Electronics Engineers (IEEE), 2006. p.9–13.

GROPP, W. et al. High-performance, portable implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, Cambridge, MA, USA, v.22, n.6, p.789–828, 1996.

HACKENBERG, D.; JUCKELAND, G.; BRUNST, H. Performance analysis of multilevel parallelism: inter-node, intra-node and hardware accelerators. **Concurrency and Computation: Practice and Experience**, [S.l.], v.24, n.1, p.62–72, 2012.

HORNUNG, R.; TRANGENSTEIN, J. Adaptive Mesh Refinement and Multilevel Iteration for Flow in Porous Media. **Journal of Computational Physics**, [S.l.], v.136, p.522–545, 1997. Available at: <http://www.math.duke.edu/∼johnt/amr.html>. Last access: Mar. 2006.

HUANG, C. et al. Performance evaluation of adaptive MPI. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.12–21. (PPoPP'06).

INTEL. **Intel VTune - Intel Software Network**. Available at: <http://www.intel.com/software/products/vtune>. Last access: Oct. 2011.

INTEL. **Intel Xeon Processor 6000 Sequence**. Available at: <http://www.intel.com/products/server/processor/xeon6000/index.htm>. Last access: June, 2011.

INTEL. **Teraflops Research Chip**. Available at: <http://techresearch.intel.com/ProjectDetails.aspx?Id=151>. Last access: June, 2011.

KALE, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on c++. **SIGPLAN Not.**, New York, NY, USA, v.28, p.91–108, October 1993.

KASSICK, R. V. et al. Trace-based Visualization as a Tool to Understand Applications I/O Performance. In: WORKSHOP ON APPLICATIONS FOR MULTI AND MANY CORE ARCHITECTURES (WAMMCA) - 23ND INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD 2011), 2., 2011, Vitória, ES. **Anais...** IEEE, 2011.

KIRK, D. B.; W. HWU, W. m. **Programming Massively Parallel Processors**: A Hands-on Approach. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann, 2010.

LAM/MPI Parallel Computing. Available at: <http://www.lam-mpi.org>. Last access: Jul. 2012.

LINFORD, J. C. **Accelerating Atmospheric Modeling Through Emerging Multi-core Technologies**. 2010. PhD Dissertation — Department of Computer Science, VirginiaTech, Blacksburg, Virginia, USA.

LINFORD, J. C.; SANDU, A. Scalable heterogeneous parallelism for atmospheric modeling and simulation. **J. Supercomputing**, Hingham, MA, USA, v.56, n.3, p.300–327, Jun. 2011.

LOCKARD, D. P.; LUO, L.-S.; SINGER, B. A. **Evaluation of the Lattice-Boltzmann Equation Solver PowerFLOW for Aerodynamic Applications**. Availabel in: <http://www.engr.uky.edu/vac/public_html/CTEMPpowerflow.pdf>. Accessed in: Oct. 2011.

LUCQUIN, B.; PIRONNEAU, O. **Introduction to Scientific Computing**. New York, USA: J. Wiley & Sons, 1998. 380p.

MACNEICE, P. et al. PARAMESH: A parallel adaptive mesh refinement community toolkit. **Computer Physics Communications**, Amsterdam, The Netherlands, The Netherlands, v.126, n.3, p.330–354, 2000.

MARSHALL, J. et al. A Finite-Volume Incompressible Navier-Stokes Model for Studies of Ocean on Parallel Computers. **Journal of Geophysical Research**, Washington, DC, USA, v.102, n.C3, p.5753–5766, 1997.

MEDEIROS, C. B. Grand Research Challenges in Computer Science in Brazil. **Computer**, Los Alamitos, CA, USA, v.41, p.59–65, 2008.

MICHALAKES, J.; VACHHARAJANI, M. GPU Acceleration of Numerical Weather Prediction. **Parallel Processing Letters**, [S.l.], v.18, n.4, p.531–548, 2008.

MITCHELL, W. F. **PHAML user's guide**. Technical Report NISTIR 7374, National Institute of Standards and Technology, Gaithersburg, MD, 2006. Software available at: http://math.nist.gov/phaml.

MPICH home page. Available at: <http://www-unix.mcs.anl.gov/mpi/mpich1>. Last access: Jul. 2012.

MUSSER, D. R.; SAINI, A. **STL Tutorial and Reference Guide - C++ Programming with the Standard Template Library**. Reading, Massachusetts: Addison-Wesley, 2004.

NICKOLLS, J.; DALLY, W. The GPU Computing Era. **IEEE Micro**, Washington, DC, v.30, n.2, p.56–69, Mar. 2010.

NICKOLLS, J. et al. Scalable Parallel Programming with CUDA. **Queue**, New York, NY, USA, v.6, n.2, p.40–53, 2008.

NUMRICH, R. W.; REID, J. Co-array Fortran for parallel programming. **SIGPLAN Fortran Forum**, New York, NY, USA, v.17, n.2, p.1–31, 1998.

NVIDIA. **High Performance Computing - Supercomputing with Tesla GPUs**. Available at: <http://www.nvidia.com/object/tesla_computing_solutions. html>. Last access: June, 2012.

OPEN MPI: Open Source High Performance Computing. Available at: <http://www.open-mpi.org>. Last access: Jul. 2012.

OSTHOFF, C. et al. Improving Core Selection on a Multicore Cluster to Increase the Scalability of an Atmospheric Model Simulation. In: XXIX IBERIAN-LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING, 2010, BUENOS AIRES. MECANICA COMPUTACIONAL VOL. XXIX. BUENOS AIRES : ASOCIACIÓN ARGENTINA DE MECANICA COMPUTACIONAL, 2010, Buenos Aires, Argentina. **Proceedings...** Asociación Argentina de Mecánica Computacional (AMCA), 2010. p.3143–3153.

OSTHOFF, C. et al. I/O Performance Evaluation on Multicore Clusters with Atmospheric Model Environment. In: WORKSHOP ON APPLICATIONS FOR MULTI AND MANY CORE ARCHITECTURES (WAMMCA) - 22ND INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD 2010), 1., 2010, Petrópolis. **Anais...** IEEE, 2010. p.49–54.

OSTHOFF, C. et al. Evaluation of Programming Models for Atmospheric Application. In: IADIS INTERNATIONAL CONFERENCE APPLIED COMPUTING 2011, 2011, Rio de Janeiro, RJ, Brazil. **Anais...** IADIS Press, 2011.

OSTHOFF, C. et al. Improving Atmospheric Model Performance on a Multi-Core Cluster System. In: YUCEL, I. (Ed.). **Atmospheric Model Applications**. Rijeka, Croatia: InTech, 2011. p.1–24.

OSTHOFF, C. et al. Improving Performance on Atmospheric Models through a Hybrid OpenMP/MPI Implementation. In: THE 9TH IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WITH APPLICATIONS (ISPA 2011), 2011a, Busan, Korea. **Anais...** IEEE Technical Committee on Scalable Computing, 2011a. p.69–74.

OSTHOFF, C. et al. GPU for Accelerators Performance Evaluation on Atmosphere Model's Application System. In: XXX IBERIAN-LATIN-AMERICAN CONGRESS

ON COMPUTATIONAL METHODS IN ENGINEERING, 2011, OURO PRETO. MECANICA COMPUTACIONAL VOL. XXX, 2011b, Ouro Preto, Brazil. **Proceedings...** Asociación Argentina de Mecánica Computacional (AMCA), 2011b. p.–.

OSTHOFF, C. et al. Atmospheric Models Hybrid OpenMP/MPI Implementation Multicore Cluster Evaluation. **International Journal of Information Technology, Communications and Convergence**, Olney, Bucks, UK, v.2, n.3, p.212–233, 2012.

PANETTA, J. et al. Computational Characteristics of Production Seismic Migration an its Performance on Novel Processor Architectures. In: SBAC-PAD 2007, 2007, Gramado. **Anais...** IEEE, 2007. p.11–18.

PHEATT, C. Intel Threading Building Blocks. **Journal of Computing Sciences in Colleges**, USA, v.23, n.4, p.298–298, 2008.

PIELKE, R. A.; AL. et. A comprehensive meteorological modeling system-RAMS. **Meteor. Atmos. Phys.**, Berlin, v.49, p.69–91, 1992.

PITAC Report to the President. **Computational Science**: Ensuring America's Competitiveness. EUA, June, 2005, Available at: <http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf>. Last access: Sep. 2010.

PLEWA, T.; LINDE, T.; WEIRS, V. G. **Adaptive Mesh Refinement - Theory and Applications**. Berlin: Springer, 2003.

RAUBER, T.; RÜNGER, G. **Parallel Programming**: for Multicore and Cluster Systems. Berlin: Springer Publishing Company, Incorporated, 2010.

SCHEPKE, C. et al. Performance Evaluation of an Atmospheric Simulation Model on Multi-Core Environments. In: CONFERENCIA LATINO AMERICANA DE COMPUTACIÓN DE ALTO RENDIMIENTO, 2010, Gramado, RS, Brazil. **Proceedings...** Instituto de Informática/UFRGS, 2010. p.330–332.

SCHEPKE, C. et al. Online Mesh Refinement in Parallel Meteorological Applications. In: CONFERENCIA LATINO AMERICANA DE COMPUTACIóN DE ALTO RENDIMIENTO, 2011, Colima, Mexico. **Proceedings...** -, 2011. p.–.

SCHEPKE, C. et al. Why Online Dynamic Mesh Refinement is Better for Parallel Climatological Models. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD 2011), 23., 2011, Vitória, Espírito Santo. **Anais...** IEEE, 2011.

SCHEPKE, C. et al. Online Mesh Refinement for Parallel Atmospheric Models. **International Journal of Parallel Programming**, Berlin, p.–, 2012.

SCHEPKE, C.; MAILLARD, N. Performance Improvement of the Parallel Lattice Boltzmann Method. In: SBAC-PAD 2007 / 19TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2007, Gramado. **Anais...** IEEE Computer Society, 2007. p.71–78.

SCHEPKE, C.; MAILLARD, N. Exploring Multi-Level Parallelism in Atmospheric Applications. In: XIII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (WSCAD-SSC 2012), 2012, Petrópolis, RS. **Anais...** IEEE, 2012.

SCHEPKE, C.; NAVAUX, P. O. A.; MAILLARD, N. Parallel Lattice Boltzmann Method with Blocked Partitioning. **International Journal of Parallel Programming**, Berlin, v.37, n.6, p.593–611, 2009.

SCHMIDT, G. A. et al. Present-Day Atmospheric Simulations Using GISS ModelE: Comparison to In Situ, Satellite, and Reanalysis Data. **Journal of Climate**, Boston, MA, v.19, n.2, p.153, 2006.

SHALF, J. **Memory Subsystem Performance and QuadCore Predictions**. In Presentation at NERSC User Group Meeting, September 17. 2007. Available at: <http://www.nersc.gov/about/NUG/meeting_info/Sep07/charts/Shalf-NUG2006_QuadCore.pdf>. Last access: Oct. 2010.

SHAMEEM, A.; ROBERTS, J. **Multi-Core Programming - Increasing Performance through Software Multithreading**. Hillsboro, OR: Intel, 2005.

SHIMOKAWABE, T. et al. An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In: ACM/IEEE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2010., 2010, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.1–11. (SC '10).

SILVA, R. R. da et al. Modelo OLAM (Ocean-Land-Atmosphere-Model): descrição, aplicações, e perspectivas. **Revista Brasileira de Meteorologia [online]**, São José dos Campos, SP, v.24, n.2, p.144–157, 2009.

SIMS, J. S. et al. Accelerating Scientific Discovery Through Computation and Visualization. **Journal of Research of the National Institute of Standards and Technology**, Gaithersburg, MD, v.105, n.6, p.875–894, Nov.-Dec. 2000.

SNIR, M. et al. **MPI-The Complete Reference, Volume 1**: the mpi core. 2nd. (Revised).ed. Cambridge, MA, USA: MIT Press, 1998.

SOUTO, R. P. et al. Processing Mesoscale Climatology in a Grid Environment. In: CCGRID'07, 2007, Rio de Janeiro. **Proceedings...** Hoes Lane: IEEE Computer Society, 2007. p.363–370.

TOP 500. **Top 500 Supercomputing Site**. Available at: <http://www.top500.org>. Last access: June, 2011.

VASQUEZ, T. **Weather Forecasting Red Book**. Garland TX, USA: Weather Graphics Technologies, 2006. 304p.

VOSS, M. Intel; Threading Building Blocks: Programming for Current and Future Multicore Platforms. **IEEE/ACM International Symposium on Code Generation and Optimization**, Los Alamitos, CA, USA, v.0, p.XX, 2009.

WALKO, R. L.; AVISSAR, R. The Ocean-Land-Atmosphere Model (OLAM). Part I: Shallow-Water Tests. **Monthly Weather Review**, Boston, MA, v.136, n.11, p.4033–4044, 2008.

WALKO, R. L.; AVISSAR, R. **OLAM**: Ocean-Land-Atmosphere Model - Model Input Parameters - Version 3.0. Durham, NC, USA: Duke University, 2008.

WASHINGTON, W. M.; PARKINSON, C. L. **An Introduction to Three Dimensional Climate Modeling**. 2.ed. Herndon, VA, USA: University Science Books, 2005.

WEILAND, M. **Chapel, Fortress and X10**: novel languages for HPC. Edinburgh-UK: University of Edinburgh, 2007. Technical Report.

WENNEKER, I.; SEGAL, A.; WESSELING, P. A Mach-uniform unstructured staggered grid method. **International Journal of Numerical Methods in Fluids**, New York, USA, v.40, n.9, p.1209–1235, 2002.

WILKINSON, B.; ALLEN, M. **Parallel Programming**: Using Networked Workstations and Parallel Computers. New Jersey: Prentice Hall, 1998.

WILLHALM, T.; POPOVICI, N. Putting Intel Threading Building Blocks to Work. In: IWMSE '08: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON MULTICORE SOFTWARE ENGINEERING, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.3–4.

XAVIER, C. et al. Multi-level Parallelism in the Computational Modeling of the Heart. In: SBAC-PAD 2007, 2007, Gramado. **Anais...** IEEE, 2007. p.3–10.

YELICK, K. et al. Titanium: A high-performance Java dialect. **Concurrency: Practice and Experience**, Trier, Germany, v.10, n.11-13, p.825–836, 1998.

YELLICK, K.; BONACHEA, D.; WALLACE, C. **A Proposal for a UPC Memory Consistency Model, v1.0**. Berkeley, CA: Lawrence Berkeley National Lab, 2004. Technical report. (LBNL-54983).

ZHURAVLEV, S.; BLAGODUROV, S.; FEDOROVA, A. Addressing Shared Resource Contention in Multicore Processors via Scheduling. **SIGPLAN Not.**, New York, NY, USA, v.45, n.3, p.129–142, 2010.

ZUMBUSCH, G. **Parallel Multilevel Methods**: adaptive mesh refinement and loadbalancing. [S.l.]: Teubner, 2003.

APPENDIX A RESUMO EM PORTUGUÊS

A qualidade das soluções obtidas em aplicações climatológicas é limitada pela capacidade computacional e o tempo disponível para a execução das simulações. Quanto maior for a capacidade dos computadores utilizados no processamento, maior será a resolução da malha que pode ser adotada para representar a atmosfera terrestre e, consequentemente, mais acurada será a precisão numérica das soluções.

Com o surgimento das arquiteturas multi-core e a adoção de GPUs para a computação de propósito geral, existem atualmente diferentes níveis de paralelismo. Hoje há paralelismo interno ao processador, entre processadores e entre computadores.

Com o objetivo de extrair ao máximo a performance dos computadores atuais, é necessário utilizar todos os níveis de paralelismo disponíveis durante a execução de aplicações concorrentes. No entanto, nenhuma interface de programação paralela explora simultaneamente bem os diferentes níveis de paralelismo existentes.

Neste contexto, esta tese propõe o uso combinado de diferentes interfaces de programação paralela com o objetivo de prover performance para aplicações climatológicas. A execução das simulações mostra que o uso de CPUs multi-core e GPUs, em sistemas paralelos, pode reduzir consideravelmente o tempo de execução das aplicações.

A.1 Introdução

Atualmente há diversas classes de aplicações, com o objetivo de prover soluções para problema científicos e de engenharia, que demandam uma considerável capacidade de computação. Ao mesmo tempo, há um constante incremento na capacidade de processamento dos sistemas de alta performance disponíveis para simulações. Este incremento é alcançado através da replicação dos recursos de hardware, tornando possível a execução concorrente de software sobre hardware paralelo.

Hoje há diferentes níveis de paralelismo oferecidos pelas arquiteturas computacionais. O paralelismo pode ser expresso internamente em um processador, através das arquiteturas multicores; interno a um computador, usando multiprocessadores, *Graphics Processing Units* (GPUs) (GARLAND et al., 2008) e *Field-Programmable Gate Arrays* (FPGAs) (BROWN et al., 1997); e entre computadores, formando sistemas paralelos e distribuídos como *clusters* ou *grids*.

Uma vez que existem diferentes níveis de paralelismo, há também diferentes interfaces de programação paralela adotadas para gerar códigos concorrentes. Entretanto, cada interface de programação geralmente atua sobre um nível específico de paralelismo. Não há uma interface de programação unificada que abstrai todos os níveis de hardware paralelo disponíveis.

A Tabela A.1 apresenta uma comparação entre diferentes interfaces de programação

Table A.1: Diferentes níveis de paralelismo cobertos por interfaces de programação.

Nível Paralelo	**Cilk**	**OpenMP**	**TBB**	**PGAS**	**MPI**	**CUDA/OpenCL**
Memória Distribuída				x	x	
Interprocessadores	x	x	x	x	x	
Intraprocessadores	x	x	x		x	
GPGPU						x

paralela, que podem ser adotadas no desenvolvimento de programas, para a utilização dos diferentes níveis paralelos disponibilizados pelas arquiteturas.

A noção de tarefa paralela é representada de diferentes formas pelas interfaces de programação relacionadas nesta tabela. CILK e TBB suportam nativamente esta noção. A definição de tarefa não é tão bem definida em MPI. Nesta interface de programação paralela, cada processo é a própria tarefa paralela.

Para explorar todos os recursos de hardware disponíveis de um determinado ambiente de execução é necessário combinar diferentes interfaces de programação paralela no código concorrente.

Neste contexto, esta tese discute como é possível explorar diferentes níveis de paralelismo em simulações de modelos atmosféricos. Modelos atmosféricos demandam uma quantidade significativa de processamento. Além disso, há uma relação entre a precisão das soluções numéricas e a capacidade computacional. Quanto maior a capacidade de processamento dos recursos usados, melhor é a precisão que pode ser considerada nas simulações.

Esta tese propõe o uso combinado de diferentes interfaces de programação paralela para aumentar a performance de aplicações climatológicas. Para avaliar a viabilidade das soluções propostas, foram desenvolvidas versões paralelas de aplicações que usam as interfaces de programação *Message-Passing Interface* (MPI) (SNIR et al., 1998), *Open Multi-Processing* (OpenMP) (CHANDRA, 2001) e *Compute Unified Device Architecture* (CUDA) (KIRK; W. HWU, 2010). Deste modo é possível a execução das implementações em sistemas de memória compartilhada (Multi-core, multi-processadores e GPU) e distribuída (multi-computadores).

A.2 Trabalhos Relacionados

Programas híbridos que combinam múltiplos paradigmas de paralelização, tais como troca de mensagens e/ou *multi-threading*, com bibliotecas de aceleração de hardware, são relativamente raros até o momento (HACKENBERG; JUCKELAND; BRUNST, 2012). Porém, este tipo de programação tem se tornado cada vez mais comum e importante, devido à existência de diferentes sistemas híbridos de alto desempenho, como é o caso de *clusters* formados por processadores *multi-core* da INTEL ou AMD, Cell da IBM e GPUs da NVidia.

Diversos trabalhos descrevem o uso de *multi-core* e GPUs para o processamento de aplicações das áreas de decomposição de domínios, dinâmica dos fluidos e, especificamente também, previsões atmosféricas (COHEN; GARLAND, 2009).

(LINFORD; SANDU, 2011) examina métodos para prover performance em simulações de transporte de componentes da atmosfera em duas e três dimensões. Uma função com abordagem *offload* é usada em um módulo de transporte bidimensional e uma abor-

dagem de processamento de *stream* vetorial é usada no módulo de transporte tridimensional. Dois métodos para o transporte não contínuo de dados entre a memória principal e o local de armazenamento no acelerador de hardware são comparados (LINFORD, 2010). Os resultados do estudo demonstram que processadores *multi-core* heterogêneos tem potencial para prover *speed up* para simulações geofísicas.

(MICHALAKES; VACHHARAJANI, 2008) discute os resultados de *speed up* para um trecho de código executado intensivamente pelo modelo atmosférico *Weather Research and Forecast* (WRF). Testes mostram que a performance pode ser incrementada $8\times$ em execuções usando uma variedade de GPUs NVIDIA. Esta pequena alteração de código no modelo aumenta o *speed up* global do modelo atmosférico em $1.23\times$.

Em outro trabalho, (SHIMOKAWABE et al., 2010), é apresentado uma implementação completa em CUDA de um modelo atmosférico japonês de alta resolução, similar ao WRF. A execução experimental desse modelo em um *cluster* com 528 GPUs NVIDIA alcançou um incremento de *speed up* de $80\times$ e possibilitou um escalonamento de 15 TFlops, usando precisão simples, para um domínio dividido em $6956 \times 6052 \times 48$ elementos de malha.

WRF e ASUCA são exemplos de modelos de simulação atmosférica local, ou seja, modelos que atuam somente sobre uma determinada parte da atmosfera terrestre, necessitando obter informações a respeito das condições de contorno de modelos globais. Neste trabalho, implementações paralelas para um modelo atmosférico global são propostas com o objetivo de executar simulações experimentais em *cluster* compostos de processadores *multi-core* e GPUs.

A.3 Paralelismo Multi-Nível

Atualmente a composição dos ambientes computacionais paralelos é bastante heterogênea. De um lado, existem arquiteturas formadas por *clusters* e *grids*. Por outro lado, as arquiteturas *multi-core* oferecem diferentes unidades de processamento no próprio *chip*. Consequentemente, o uso combinado de diferentes arquiteturas acaba provendo ambientes com múltiplos níveis de paralelismo.

Em um paralelismo multi-nível há diferentes níveis de abstração paralela. Estes níveis podem ser internos ao processador (*multi-core*) interno ao computador (multiprocessadores) ou entre múltiplos computadores (*clusters* e *grids*), criando, por fim, uma hierarquia de abstração paralela, conforme mostrado na Figura A.1. A granularidade das tarefas que podem rodar em cada nível aumenta a medida que aumenta o nível de paralelismo, ou seja, *threads* podem ser utilizados para explorar o paralelismo interno ao processador enquanto processos comunicantes podem ser adotados em *clusters*.

O gerenciamento de cada nível de abstração paralela é feito através de mecanismos específicos:

- **A nível de processador** - O fluxo de instruções é definido pelo núcleo do sistema operacional ou pela implementação dos registradores requerido em hardware. Assim, o contro é feito por instruções em *assembler*.

- **A nível de núclo do sistema operacional** - O fluxo de instruções é definido por processos ou *threads*. O controle do fluxo de instruções é feito através de chamadas ao sistema operacional.

- **A nível de gerenciamento de *middleware*** - O conjunto de instruções é agrupado,

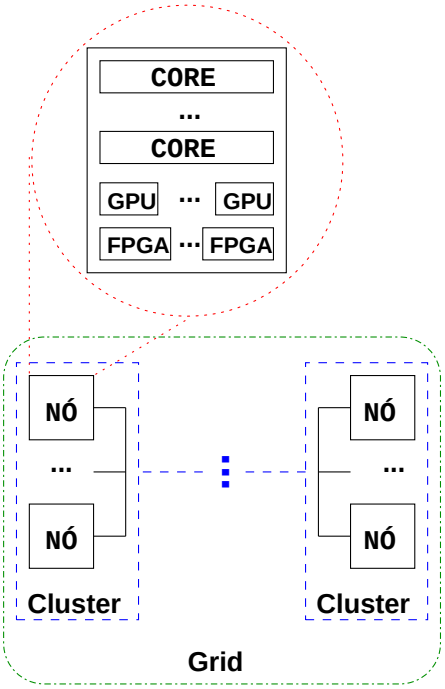

Figure A.1: Diferentes níveis de concorrência em arquiteturas paralelas.

formando um processo comunicante. O controle é feito através de bibliotecas de comunicação inter-processos.

Por isso, geralmente é de responsabilidade do programador usar diferentes ferramentas para implementar um programa que explore os vários níveis de paralelismo.

A.4 Interfaces de Programação Paralela

O processo de implementação de aplicações de alto desempenho é simplificado pela existência de diversas ferramentas de programação paralela. Estas ferramentas abstraem tanto sistemas compartilhados como distribuídos e provêm uma abordagem de desenvolvimento padrão para diversos paradigmas de programação paralela.

A.4.1 Message-Passing Interface

Bibliotecas de troca de mensagens foram desenvolvidas para abstrair a camada de rede (*sockets*) e para oferecer uma interface clara de funções de comunicação de dados. Estas bibliotecas foram utilizas para o desenvolvimento de diversas aplicações de alta performance na década de 90 e início da última década.

A biblioteca de comunicação Message-Passing Interface (MPI) é um dos mecanismos amplamente utilizado para simplificar a programação paralela (GROPP et al., 1996). MPI possui um amplo conjunto de funções que podem ser utilizadas em implementações paralelas e distribuídas. Estes recursos são necessários para se obter performance paralela e são usadas frequentemente em muitos tipos de aplicação.

A.4.2 OpenMP

OpenMP (Open Multi-Processing) é uma API de programação para arquiteturas de memória compartilhada (CHANDRA, 2001), (CURTIS-MAURY et al., 2008). A API

provê diretivas que permitem a expressão de paralelismo de dados em partes de código e laços, e o paralelismo de tarefas.

A API de OpenMP consiste de diretivas de compilação, métodos da biblioteca e variáveis de ambiente que descrevem como a carga de trabalho pode ser compartilhada entre diferentes *threads* executando em diferentes processos ou, atualmente, *cores*. O programador pode definir o número de *threads*, que serão executados, através da chamada de métodos da biblioteca ou através da configuração de variáveis de ambiente.

Além disso, o grafo de tarefas no paralelismo de dados pode ser determinado pelo programador ou pelo compilador. O padrão OpenMP não especifica um algoritmo de escalonamento. Isto é atribuído à implementação da API, a fim de que o balanceamento de carga seja feito da melhor forma possível.

A.4.3 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) é uma arqutetura de computação e programação paralela desenvolvido pela nVidia (NICKOLLS et al., 2008). CUDA posibilita o uso de Graphics Processing Units (GPU), como arquiteturas programáveis de alto desempenho. Isto simplifica a programação de software, tornando possível a execução do mesmo em placas de vídeo.

O uso de placas de vídeo para executar uma aplicação normalmente feita para a execução em CPU é chamado de General-Purpose computing on Graphics Processing Units (GPGPU) (GARLAND et al., 2008). A primeira vantagem do uso de CUDA é o uso de memória compartilhada para um rápido acesso de endereços arbitrários de memória. Desde a versão 3.1, CUDA tem suporte a recursão, tipo ponto-flutuante de dados de precisão duplae rendenização de texturas.

O modelo de programação consiste de extensões da linguagem C e C++, para programas sequenciais, que pode ser executado em um *kernel* CUDA (NICKOLLS et al., 2008). O *kernel* é uma função similar a um código C e roda paralelamente em diversas *threads*, sendo este mapeado pela própria GPU (KIRK; W. HWU, 2010). No entanto, o programador é responsável pela transferência de dados entre CPU e GPU.

O modelo de programação CUDA é ideal para aplicações com alto nível de paralelismo e para aplicações que não possuem dependências entre as tarefas. Entretanto, há limitações em CUDA que incluem o controle de coerência dos dados usados e uma ausência de suporte para a execução de múltiplos *kernels*. Assim, um ganho significativo de performance depende do conhecimento sobre a arquitetura da GPU e sobre o modelo de programação CUDA.

A.5 Ocean-Land-Atmosphere Model

Ocean-Land-Atmosphere Model (OLAM) foi o modelo atmosférico escolhido com a finalidade de avaliar as interfaces de programação paralela previamente descritas (WALKO; AVISSAR, 2008a).

OLAM é um exemplo típico de problema de decomposição de domínio, uma classe de aplicações que frequentemente ocorre em muitas áreas da ciência. Além disso, esta aplicação real tem uma quantidade significativa de carga computacional, sendo um bom candidato para a avaliação de performance.

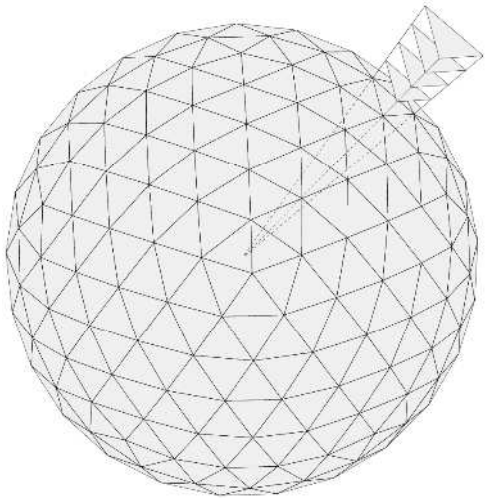

Figure A.2: Projeção dos elementos triangulares da superfície do modelo em esferas concêntricas para gerar múltiplos níveis verticais.

A.5.1 Implementação do Modelo

Ocean-Land-Atmosphere Model (OLAM) é um modelo atmosférico para a representação e simulação de toda a superfície terrestre. O modelo consiste essencialmente em discretizar através da técnica de volumes finitos as equações de Navier-Stokes, aplicadas sobre uma atmosfera planetária, com a formulação de equações que respeitem as leis de conservação de massa, momento e temperatura potencial, e de operações numéricas que incluem a divisão do tempo (MARSHALL et al., 1997). Os volumes finitos são definidos horizontalmente por um conjunto de elementos triangulares formando uma malha global e sub-divididos verticalmente pelas camadas atmosféricas, formando uma espécie de prisma orientado verticalmente com uma base triangular.

A Figura A.2 mostra um exemplo de malhas decompostas. OLAM usa uma abordagem não estruturada de malha e representa cada ponto horizontal discreto através de um único índice linear (WALKO; AVISSAR, 2008a). Informações requeridas de um ponto local da topologia da malha podem ser armazenadas e acessadas através de estruturas de dados que definem a relação entre os pontos.

OLAM foi inicialmente desenvolvido e paralelizado com MPI. Cada processo MPI é responsável por operar funções sobre um determinado sub-domínio durante a etapa **iterativa**. Cada processo determina seu sub-domínio da malha global de acordo com seu *rank* MPI. Uma vez definido a distribuição dos sub-domínios entre os processos (**inicialização**), cada processo descarta a malha global e mantém em memória apenas sua respectiva parte da malha, a fim de que a mesma seja processada. Troca de mensagens são feitas entre os processos cujos domínios sejam vizinhos, em cada iteração, a fim de atualizar as estruturas de dados localizadas nas bordas da malha.

A.5.2 Protótipo do Modelo

Uma versão simplificada de OLAM foi implementada em C, com o objetivo de alcançar as metas propostas neste artigo. Esta versão prototipada inclui as principais características do modelo, incluindo decomposição de domínios, refinamento de malhas, distribuição paralela de dados, encapsulamento de chamadas MPI para o envio e recebimento de dados e todas as estruturas de dados e funções necessárias, a fim de que o

modelo possa ser executado. Este protótipo provê um modelo abstrato de paralelismo de tarefas para aplicações climatológicas.

Embora OLAM tenha sido inicialmente implementado com MPI, outra escolha natural para a paralelização do código e o uso de *pragmas* OpenMP nos principais laços do código. Para tanto, foram feitas modificações no código prototipado original, adicionando instruções que permitam a decomposição de laços em diferentes execuções concorrentes, através da criação de *threads* OpenMP. O paralelismo OpenMP foi combinado com a implementação feita com MPI. Assim, é possível ter uma outra forma de execução concorrente em sistemas de memória compartilhada.

Outra interface de programação paralela para prover paralelismo multi-nível que foi adotado é CUDA. Para tanto, algumas funções da implementação do protótipo OLAM foram reescritas, convertendo o código C destas em código de *kernel* CUDA. Também foram necessárias a implementação de funções que encapsulem a alocação e desalocação de memória, além da cópia de dados entre CPUs e GPUs.

Para reduzir o número de alocações de memória no *kernel* CUDA, todas os vetores de variáveis temporárias a serem utilizados pelas funções foram alocados antes da chamada da parte iterativa do modelo e desalocadas depois do término dessa etapa. Além disso, em cada passo da etapa iterativa, antes da chamada de qualquer função do *kernel* CUDA, foi necessário a cópia de dados da CPU para a GPU e, após a execução do passo iterativo a cópia de dados da GPU para a CPU.

As funções do *kernel* CUDA também foram embutidas na implementação MPI. Com isso, três diferentes níveis de paralelismo (GPUs, *cores* e processadores podem ser utilizados em simulações atmosféricas.

A.6 Avaliação de Performance

Esta seção apresenta o ambiente de simulação, parâmetros de execução e as medições de tempo de execução e de *speed up* efetuadas.

A.6.1 Ambiente de Simulação

Todas as medições experimentais foram obtidas utilizando o *cluster Newton* do *Centro Nacional de Supercomputação*. Este *cluster* é interconectado através da tecnologia de rede *InfiniBand* e tem atualmente 28 Sun Fire X2200+ nós (cada um com 2 processadores Quad-Core AMD Opteron de 2,2 GHz e 16 GB RAM) e 8 GPUs nVidia Tesla S1070.

Em todas as execuções foram simuladas 12 horas de interação da atmosfera. Cada etapa da interação simula 60 segundos do tempo real da condição atmosférica. O eixo vertical da atmosfera foi dividido em 28 camadas, conforme padrões utilizados em centros de climatologia.

A distância média entre cada ponto discreto da superfície global foi em torno de100 Km, 67 Km d 50 Km. Em simulações utilizando GPUs, o número de *threads* CUDA foi fixado em 128.

A.6.2 Implementação com MPI

Um primeiro teste foi feito com o objetivo de analisar o impacto do use de processos MPI no tempo total de execução.

A Figura A.3 e a Figura A.4 apresentam o tempo de execução (em segundos) resultante de uma simulação atmosférica, usando uma malha de 100 Km e 50 Km de resolução horizontal. Os gráficos dessas figuras mostram o tempo total de execução e o tempo gasto

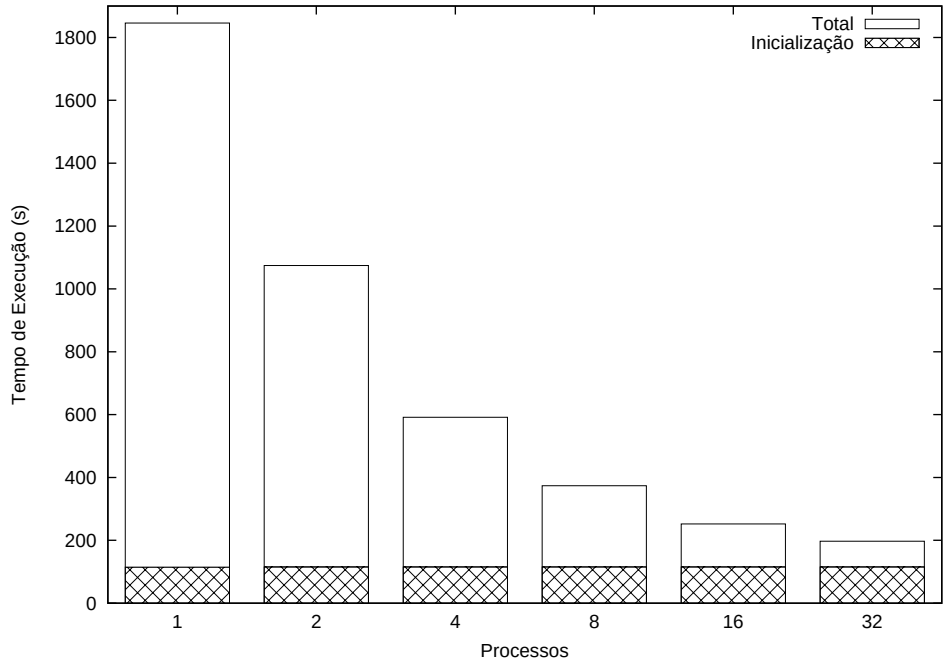

Figure A.3: Tempo de execução usando de 1 a 32 processos para uma simulação de uma malha de 100 Km de resolução.

na execução da inicialização, usando de 1 a 32 processos MPI.

Cada coluna do gráfico representa o tempo total de execução para um determinado número de processos. Pode-se observar que este tempo decresce a medida que mais processos são utilizados. Consequentemente há ganho de performance.

A segunda medição (área quadriculada) de cada grupo de processos representa o tempo de execução da etapa de inicialização do modelo. O tempo de duração dessa etapa é de aproximadamente 115 s para uma resolução de malha de 100 Km e 415 s para uma malha com $50Km$ de resolução. O tempo gasto com a etapa de inicialização é constante, independente do número de processos utilizado nas simulações. A relação entre o tempo de inicialização e o tempo total de execução decresce a medida que malhas com maior nível de resolução são utilizadas.

A.6.3 Implementação com MPI e OpenMP

O uso de *threads* OpenMP foi avaliado em algumas simulações atmosféricas, considerando malhas com resolução horizontal de 100 Km.

A Figura A.5 apresenta o tempo total de execução (em segundos) de uma simulação atmosférica utilizando de 1 a 8 processos MPI. Nos testes, a performance de execução utilizando de 1 a 8 *threads* OpenMP foi comparada para cada número de processos MPI. Cada coluna branca preenchida no gráfico representa o tempo de simulação usando somente processos MPI. As demais colunas mostram o tempo de execução dos processos MPI com a inclusão de *threads* OpenMP.

O uso de *threads* OpenMP provê redução no tempo total de execução do modelo, independente do número de *threads* utilizado. Entretanto, há uma limitação no ganho de performance quando mais do que 32 *threads*/processos são usados, uma vez que o tempo de execução da etapa iterativa predomina em relação a otempo total de execução. Com isso há uma limitação de escalabilidade.

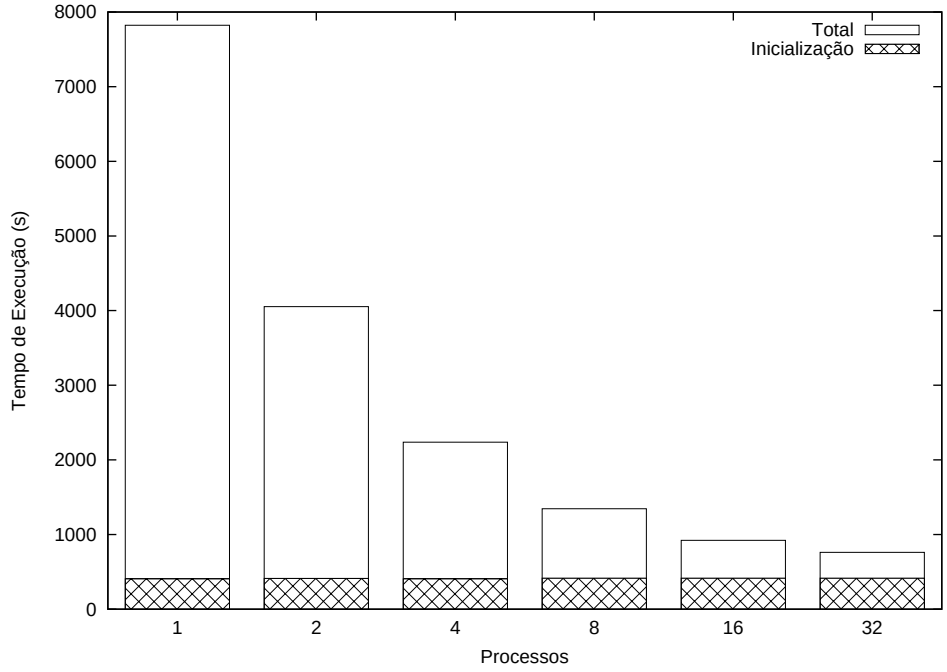

Figure A.4: Tempo de execução usando de 1 a 32 processos para uma simulação de uma malha de 50 Km de resolução.

Além disso, o paralelismo OpenMP é restrito a determinadas partes do código, enquanto que o paralelismo MPI inclui toda a parte iterativa do código. Por causa disso, a comparação entre o tempo de execução de simulações que usam somente *threads* OpenMP (1 processo MPI) em relação ao uso de somente processos MPI (1 *thread* OpenMP) mostra um melhor resultado para o primeiro caso.

A Figura A.6 apresenta o *speed up* da parte iterativa do modelo. Nos testes são comparados de 1 a 8 processos MPI. De 1 a 8 *threads* são criadas e executadas em cada processo MPI.

Os resultados mostram que o uso de *threads* OpenMP aumenta a performance da etapa iterativa para qualquer número de processos MPI adotado. O uso combinado de 2 ou 4 threads em cada processo MPI aumenta o speed up em mais de 50% e 100%, respectivamente, em relação a versão paralelizada restritamente com MPI, para todos os números de processo MPI utilizados nas simulações. No gráfico é possível ver que o uso de 4 threads e 8 processos prove um ganho de performance limitado. Isto occore porque cada processo/*thread* computa uma tarefa com baixa granularidade. Assim, a performance paralela não se sobrepõem em relação aos custos de comunicação e de criação das *threads*.

A.6.4 Implementação com MPI e CUDA

Algumas simulações também foram feitas explorando o paralelismo em GPUs.

A Figura 7.5 apresenta o tempo total de execução (em segundos) para as três resoluções de malha consideradas neste trabalho (veja Subseção A.6.1). Nesta figura, as primeiras três colunas mostram o tempo de execução sequencial de cada resolução. A execução sequencial não inclui o uso de GPUs. As outras colunas do gráfico mostram o tempo de execução usando, 1, 2, 4 e 8 GPUs.

Os resultados da Figura A.7 mostram que o uso de 1 GPU reduz o tempo total the execução em mais do que $5\times$ em relação à execução usando somente de processamento

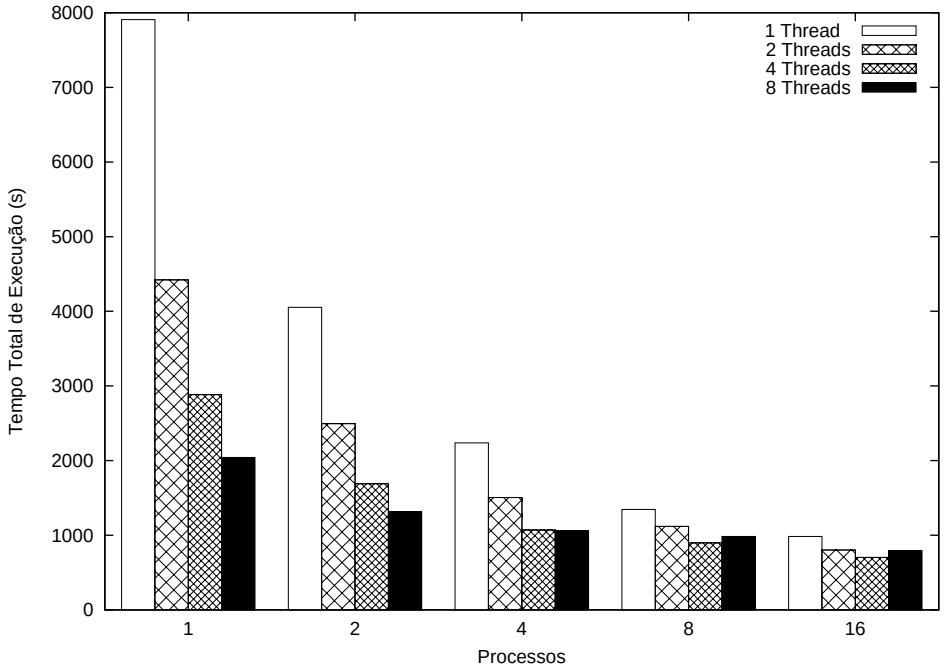

Figure A.5: Tempo total de execução usando diferentes números de *threads* OpenMP em uma simulação com processos MPI.

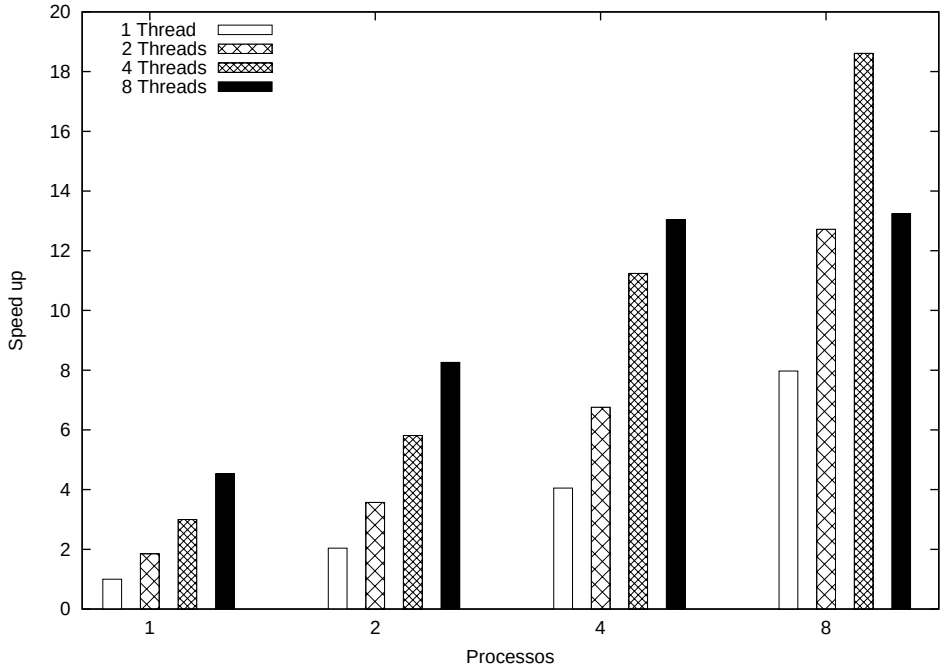

Figure A.6: *Speed up* da etapa iterativa usando diferentes números de *threads* OpenMP em uma simulação com processos MPI.

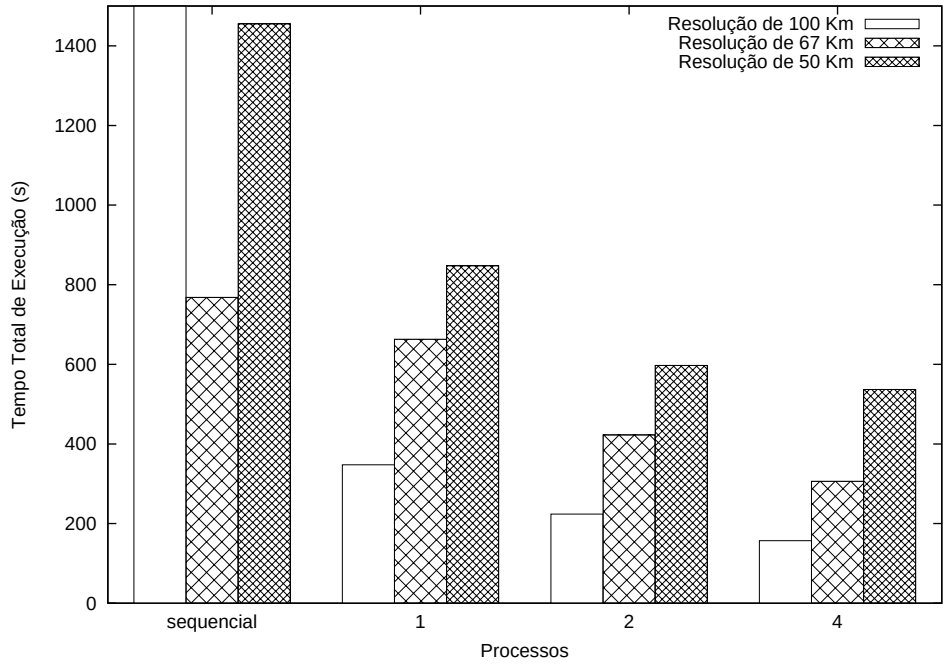

Figure A.7: Avaliação do tempo de execução utilizando diferentes números de GPUs para simulações de resolução de malhas de 100 Km, 67 Km e 50 Km.

CPU. Esta redução é mais expressiva a medida que mais GPUs são usadas na simulação.

A Figura A.8 e Figura A.9 apresenta o tempo de execução (em segundos) das etapas de inicialização e iterativa do modelo para uma resolução de malhas de 100 Km e 50 Km, respectivamente. Nestes resultados é possível ver que o tempo de execução da etapa de inicialização é constante, independente do número de GPUs usado. Por outro lado, o tempo de execução da etapa iterativa decresce para todos os casos avaliados quando mais GPUs são incluidas na computação.

A Figura A.7 exibe o *speed up* da etapa iterativa do modelo, quando são usados de 1 a 8 GPUs nas simulações com resolução de malha de 100 Km e 50 Km. Em todas as resoluções de malha utilizadas o *speed up* aumenta a medida que mais GPUs são usados. O gráfico mostra também que uma malha com alta resolução (50 Km) tem mais *speed up* do que uma malha com baixa resolução (100 Km), devido a diferença de granularidade entre os processos. Malhas com alta resolução tem mais estruturas de dados para computar. Consequentemente, a granularidade dos processos é maior nesse caso.

A.7 Conclusão e Trabalhos Futuros

Esta tese apresentou uma implementação paralela de um modelo atmosférico utilizando as interfaces de programação MPI, OpenMP e CUDA. Foi feita uma avaliação da performance de uma versão prototipada de OLAM em uma arquitetura *cluster* composta por *multi-core*, multi-processadores e GPUs, isto é um ambiente com paralelismo multi-nível.

Com o objetivo de avaliar as diferentes implementações paralelas do modelo, foram apresentados resultados parciais e comparativos de tempo de execução e de *speed up* utilizando de 1 a 32 processadores quad-core e de 1 a 8 GPUs de um *cluster*. Os resultados parciais mensurados mostram que as implementações com MPI e MPI combinado

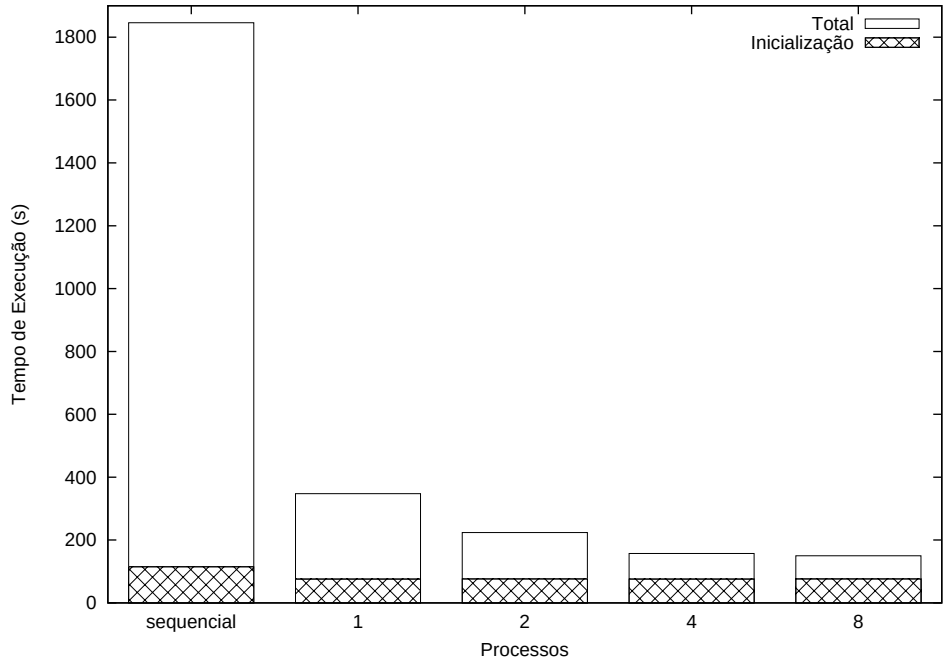

Figure A.8: Tempo de execução das etapas de inicialização e iterativa para simulações usando 100 Km de resolução de malha.

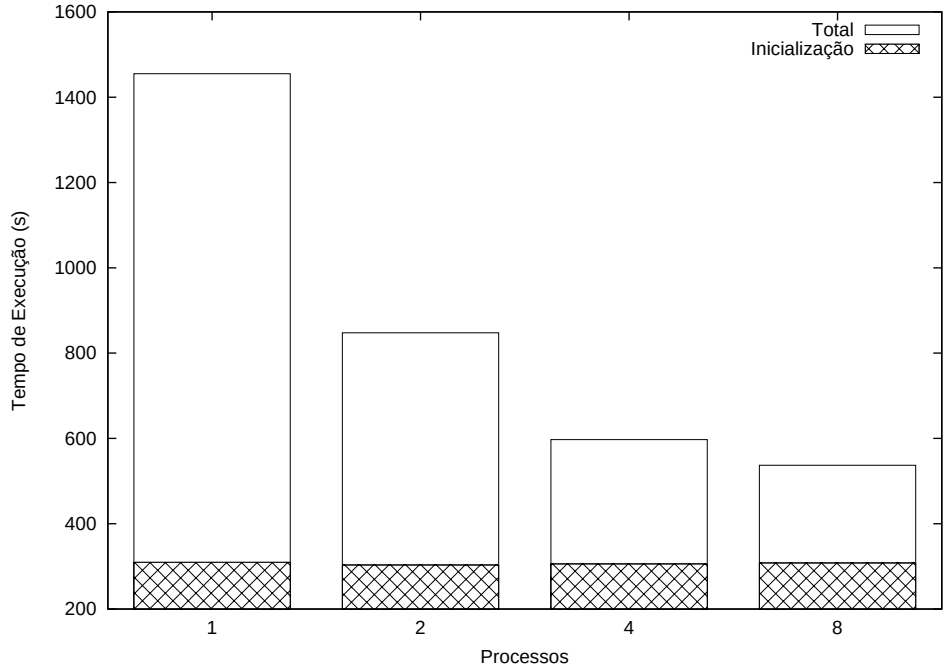

Figure A.9: Tempo de execução das etapas de inicialização e iterativa para simulações usando 50 Km de resolução de malha.

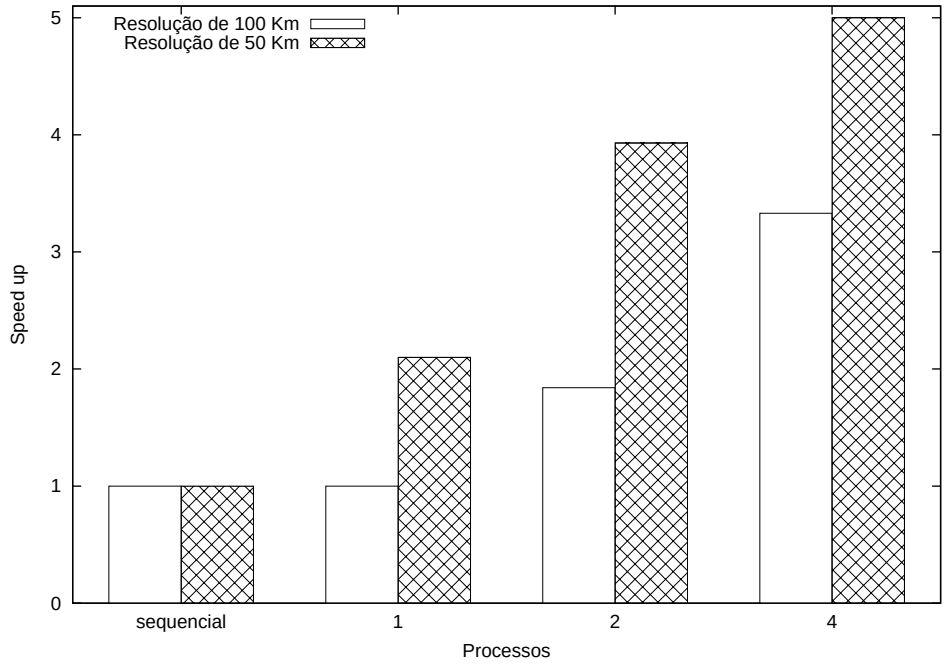

Figure A.10: Avaliação do *speed up* usando diferentes números de GPUs para simulações de resolução de malhas de 100 Km e 50 Km.

com OpenMP aumentam a performance paralela do modelo atmosférico a medida que mais processos e/ou *threads* são utilizados. O uso de OpenMP maximiza o *speed up* se o número de *threads* em execução for o mesmo que o número de cores existente no processador (4 *threads* para um processador quad-core).

O uso restrito de paralelismo MPI na implementação do modelo provê um melhor tempo de execução em relação ao uso combinado de paralelismo MPI e OpenMP, se forem comparados o mesmo número de processos MPI contra a soma do número de processos MPI e *threads* OpenMP. A implementação paralela com MPI envolve toda a etapa iterativa do modelo enquanto que o paralelismo com OpenMP é restrito a algumas funções da etapa iterativa.

Por outro lado, a versão implementada com CUDA incrementou a performance em 5 vezes em relação à versão executada sequencialmente em uma CPU. A performance do protótipo também foi avaliada em execuções em mais uma GPU. Os resultados mostram que há um aumento de *speed up*, para todas as resoluções de malhas adotadas nas simulações, a medida que mais GPUs são usadas.

Como trabalhos futuros pretende-se avaliar o comportamento do protótipo do modelo atmosférico em arquiteturas GPU, variando o número de *threads* CUDA usado. Atualmente este número está fixo em 128 *threads*. A adição de mais *threads* pode incrementar a performance do modelo em algumas resoluções de malha específicas.

Outras interfaces de programação paralela, como *Threading Building Blocks* da *Intel* e *Message-Passing Interface 2*, que oferecem a criação de processos em tempo de execução, podem ser avaliadas com o objetivo de maximizar o uso dos recursos de hardware em simulações atmosféricas.