UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME ANTONIO CAMELO

Performance Characterization of the Alya Fluid Dynamics Simulator

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
Dezembro 2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Sérgio Luis Cechin
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

This research presents results of a performance characterization and a tracing methodology used for Alya running with a fluid dynamics model. Alya is a robust physics simulator that runs in parallel and is capable of solving different physics in a coupled way. One problem faced by Alya is the irregular load between resources and time. Experiments are conducted in parallel using the MPI specification implemented by OpenMPI, and the application is traced using the tracing tools Extrae and ScoreP. The analysis of the trace provides information about the different performance patterns such as the communications among ranks, and the effective application load imbalance. To evaluate the load balance, percent imbalance metric is used, along with an analysis of the execution/-communication ratio per timestep. The goal of the characterization of Alya is to provide information about aspects of the application that affect the performance providing a possible path of improvement to developers of the application. The tracing methodology comprises the usage of different tracing tools to provide a way to reassure the results and create complementary experiments.

Keywords: HPC, Alya, ScoreP, Trace, MPI, OpenMPI.

LIST OF FIGURES

Figure 3.1 Structure of Alya input files. 16
Figure 3.2 Scheduling strategy. Left shows an ideal communication schedule in three steps, on the right a bad communication schedule in five steps is shown. Image based on (VÁZQUEZ et al., 2014) 17
Figure 3.3 Organization of data and types of communication. This image was extracted from (VÁZQUEZ et al., 2014) 18

Figure 5.1 General methodology....................22
Figure 5.2 Methodology used for the execution and analysis of the experiment.23

Figure 6.1 Schematics of draco nodes architecture....................27
Figure 6.2 Aggregated time of effective computation (Y axis) per rank (X). 28
Figure 6.3 Timeline (X axis) showing computation states per process (Y axis). .29
Figure 6.4 Time spent in each state by process. 30
Figure 6.5 Gantt Chart with Space/Time visualization of the MPI events..........32
Figure 6.6 Gantt Chart with Space/Time visualization of the MPI events during iteration border....................32
Figure 6.7 Aggregated time of effective computation (Y axis) per rank (X). 33
Figure 6.8 Plot of duration of each timestep. 34
Figure 6.9 Space/Time plot of `timste` call event with division per rank. X axis represents time, Y axis represents rank....................35
Figure 6.10 (a) Communication Ratio without rank 0 and without iteration 1 and 10. (b) Computation Ratio without rank 0 and without iteration 1 and 10....................36
Figure 6.11 Local percent imbalance metric by timestep. X axis represents the percent imbalance, Y axis represents the timestep. In (a) the whole plot is presented, and in (b) a zoom is shown. 37

LIST OF ABBREVIATIONS AND ACRONYMS

MPI	Message Passing Interface
CSV	Comma Separated Values
BSC	Barcelona Supercomputing Center
HPC	High Performance Computing
NFS	Network File System
GNU	"GNU's Not Unix, Linux OS"
SSH	Secure Shell
CFD	Computational Fluid Dynamics
OTF	Open Trace Format

CONTENTS

1 INTRODUCTION **7**
2 BASIC CONCEPTS **9**
2.1 The Message Passing Interface (MPI) **9**
2.2 Tracing HPC Applications Behavior **10**
2.3 Metrics for Identifying Load Imbalance **11**
2.4 Summary **12**
3 ALYA **13**
4 RELATED WORK **19**
5 INVESTIGATION WORKFLOW AND EVALUATION METHODOLOGY **21**
5.1 General Overview **21**
5.2 Per-tracing tool Implementation **22**
5.2.1 Extrae 22
5.2.2 Score-P 24
5.3 Analysis Tools **25**
5.4 Summary **25**
6 RESULTS **26**
6.1 Distributed Memory Experiment with Extrae - three timesteps **26**
6.1.1 Overview of Load Imbalance 27
6.1.2 Space/Time Execution 28
6.1.3 Process Behavior by State 29
6.1.4 Percent Imbalance 30
6.2 Shared Memory Experiment with ScoreP - 10 timesteps **31**
6.3 Comparison between the tracing tools **37**
7 CONCLUSION AND FUTURE WORK **39**
REFERENCES **41**

1 INTRODUCTION

Numerical simulation is used to understand and model natural behaviors. Very often these simulations are carried out by techniques based on fluid dynamics, where a model of a real world scenario is implemented and solved with iterative numerical methods using time-steps. High performance computers are employed as execution platforms to run these models, as any sequential approach would make certain simulations executions impractical for the level of details required by the scientific community.

Distributed and parallel programming techniques provide high computational power. Interfaces such as OpenMPI enable portable implementations for the execution of complex programs in less time and are ideal for dealing with physical problems for real world simulations. Alya (VÁZQUEZ et al., 2014) is an example of MPI-based framework simulator of various physical problems. It is an open source project, part of the PRACE Benchmark (PRACE, 2013), from the Barcelona Supercomputing Center (BSC) to numerically solve physical problems.

Very often numerical simulation applications have irregular loads during execution. Such irregularity appears for many reasons, such as irregular control and data structures, adaptive mesh refinement (AMR) (BERGER; COLELLA, 1989), or even irregular iteration patterns among processes. These reasons ultimately lead to a load imbalance among both resources and time as the simulation advances. Finding out the actual application behavior on a particular platform is key to apply optimization techniques, such as better balancing algorithms or more appropriate communication patterns. The most efficient way to obtain such information, when you do not know the application code, is to apply tracing techniques. They use files to keep track of information regarding the application, which is saved in the form of events and is used for instance to track MPI operations.

In the literature, it is possible to find many works that investigate the behavior of Alya. For an application that simulates such physical problems with such massive amount of data and processing, any hampering in the performance can have a significant influence in the scalability and execution time. Load imbalance has been found in Alya in the experiments performed in the works (CAJAS; HOUZEAUX; EGUZKITZA, 2015) and (RODRíGUEZ, 2014). Thus, we seek to

evaluate the load balance, and if a load imbalance is found in our experiments, we intend to figure out why it is happening.

In this work, the Alya fluid dynamic simulation tool is executed on a set of different environments settings with a different number of cores and nodes. The goals are threefold. First, to investigate whether Alya has an irregular execution behavior regarding the resources and the time for a representative input. Second, to develop a methodology to trace this application. Third, to understand the behavior on a smaller scale platform. A similar study has already been conducted on a larger scale platform (RODRíGUEZ, 2014).

As part of the investigation process, different tools were used and evaluated, highlighting the positive and negative points. To reach an appropriate methodology, the tracing tools Extrae and ScoreP were employed. The tracing tools record all MPI events. Extrae records all events, not only MPI events, which make the trace files too big. ScoreP is able to record only MPI events and a set of wanted functions reducing the size of the trace. ScoreP assumes that the system has Network File System (NFS) to run with more than one node although it is possible to run on a system without NFS with a specific setting. The environment in which the experiments were executed did not have NFS so the experiment was conducted in a single node.

The work is structured as follows. Chapter 2 presents the basic concepts of the technologies used. Chapter 3 shows the basic concepts regarding Alya as well as its structure and its input and output data format. Chapter 4 shows the most relevant works related to this research. The workflow and methodology used in the performance characterization and analysis are detailed in Chapter 5. The results of the experiments are discussed in Chapter 6. The main contributions and the future work are detailed in Chapter 7.

2 BASIC CONCEPTS

This chapter presents the basic concepts used in this work. Fundamentals of MPI are explained, the importance of a balanced parallel application is highlighted and overview of the metrics used to evaluate the load imbalance is shown.

2.1 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a specification used by developers of message passing libraries, and contains the definition that the library must have. MPI works on top of the message-passing parallel programming model and when there is communication between the processes it moves data from the address space of one process to the other using cooperative operations (BARNEY; LIVERMORE, 2016). The implementation used in this work is the OpenMPI library that is an open source implementation of MPI and is a powerful resource for high performance computing being compatible with C and Fortran.

Initially, MPI was developed to run on distributed memory architectures with one CPU and one memory per computer and has been adapted to more sophisticated architectures as they were created. Today it can be used with shared memory, where multiple CPU share the same memory in one computer, or distributed memory that runs on more than one computer and each process only operates on local data, having to send a communication message to access and send data to other computers. The usage of both shared and distributed memory approach is called hybrid. In this work experiments with shared memory and distributed memory are performed.

MPI defines rank as a unique number for each process, so it is possible to say that the communication happens between ranks. The operations can be point-to-point or collective, and blocking or non-blocking. Point-to-point operations involve only two ranks, while collective operations can communicate with multiple ranks. A blocking operation such as MPI_Send forces ranks to wait until communication is completed before the execution continues, in contrast, non-blocking communication such as MPI_ISend returns immediately even if the communication is not completed. For instance, MPI_Send sends a message from

one rank to another (the sending rank will only be unblocked after the communication is completed), while MPI_Bcast send the same message to all ranks of a certain group.

2.2 Tracing HPC Applications Behavior

Parallel and distributed applications have been providing intense performance gains since its initial use. However, the improvement in performance comes with an increasing complexity. In order to acquire relevant information, it is required precision and robustness from the performance analysis tools. It is expected that they can monitor the whole system, recording information from hardware as well as high-level performance abstractions (SHENDE; MALONY, 2006). This makes it possible to create traces that describe the behavior of an application.

Despite the usual gain in time that parallel applications have, the complexity of the application usually raises. Thus, understanding how the code works internally, and what functions are being executed at each given time can be a challenge. Finding out what is affecting a parallel code is not a simple task, and is usually facilitated by the use of tracing tools.

A tracing tool is able to precisely inform when a function started and stopped running, which core executed it, which function called it and which function was called by it. A variety of other information can be obtained from the trace, like the communications amongst the processes, MPI calls, and communications operations as well as their target. One common way of doing that is by instrumenting the code.

Instrumentation is typically done by probes and instruction added to the code. The probes capture relevant information about the execution as the program runs and it can be done in several different ways explained in details in (SHENDE; MALONY, 2006). The experiments performed in this work use source-base instrumentation, selective instrumentation, and binary instrumentation. Binary instrumentation can be done at runtime without the need of recompiling the program. Selective instrumentation allows us to include or exclude functions and blocks of code in the trace, this can significantly reduce the size of trace files. When using source-base instrumentation the program has to be recompiled with

the tracing tool.

There are many tools available to create and analyze traces such as Score-P, TAU, EZTrace, Akypuera, SimGrid, Extrae (SCHNORR, 2014). The choice of which tool to use is dependent on the environment, on the application, and on the goals that the user aims to achieve with the analysis of the trace. In order to have a variety of tracing tools, due to some system requirements, and to experiment with the instrumentation methods cited above, Score-P and Extrae were chosen for the experiments.

2.3 Metrics for Identifying Load Imbalance

The load balance is the evaluation of the distribution of workload across multiple computing resources. In the case of parallel computing, it commonly refers to the amount of processing load that each core have compared to the others. Having a good load balance means reduced idle time and improvement in the performance. The trace of an application provides us with the required resources to evaluate the load balance of the execution. In cases of bad load balancing, the analysis of the trace can provide information about when and between which resources the bad balance is coming from. It is then possible to create a fundamented hypothesis about why it happens and how to fix it.

One interesting metric is the percent imbalance metric that takes into account the average load of each processor and the load of the busiest processor and provides information about the load. According to (PEARCE et al., 2012), and then validated by (RODRIGUES, 2016), the percent imbalance formula is used to formally calculate the load balance. It is described by the Equation 2.1 where λ is the load imbalance value to be calculated, L_{max} is the value of the process with the highest load, and $\overline{L}$ is the average computational load among processes. The metric may be calculated either for the entire execution or for different phases e.g. one measure for each time interval or timestep. Higher values mean higher load imbalance, 0 is the ideal value to have a totally balanced application.

$$\lambda = (\frac{L_{max}}{\overline{L}} - 1) * 100 \tag{2.1}$$

2.4 Summary

The aggregated use of the tracing tools cited in this chapter made it possible to achieve the results presented in this work. MPI was the specification that developers used to create Alya. The tracing tool made it possible to extract information about the application. With the analysis of the trace, the percent imbalance metric was extracted generating relevant information about the load balance and the behavior of the application.

3 ALYA

Alya is a computation mechanics code capable of solving different physics simulations. It was developed by BSC and can solve partial differential equations in non-structured meshes using finite element method (RODRíGUEZ, 2014). Alya can solve a variety of different physical problems, among them are turbulence, bi-phasic flows, free surface, convection-diffusion reaction, incompressible flows, compressible flows, excitable media, acoustics, thermal flow, quantum mechanics and solid mechanics. Each physics has a different modelization characteristic, so the solvers are divided into modules.

In order to solve a problem that involves more than one physics module Alya uses coupling. Coupling combines the modules and tries to solve the problem from all modules involved. To do that, it shares the domain and domain discretization between modules, therefore all problems are solved in the same mesh. Depending on the physics, coupling can generate a high complexity, but as long as the modules are synchronized, sub-timesteping can be applied. In other words, two modules can have different time iteration configuration. For example, one module can use a timestep five times faster than the other module. If one of the modules face scalability issues, the overall scalability will be affected (VÁZQUEZ et al., 2014).

The general pseudoalgorithm of Alya is presented in the Algorithm 1. It was done based on the documentation and the source code of Alya (VÁZQUEZ; HOUZEAUX, 2015). In the pseudoalgorithm it is possible to see the different stages of the application. After reading the input, and defining the mesh the loop starts. Each iteration represents one timestep. At each iteration, the boundary conditions, as well as any other relevant variables, are updated. In the `DoBlocks` stage of Algorithm 1, Alya divide modules into blocks, this is done to simplify cases where some modules or equations are dependent but others are not. For instance, if three modules are being coupled, Nastin, Turbul and Chemic, but only Nastin and Turbul are dependent, one block solves Nastin and Turbul, and the other solves Chemic, avoiding unnecessary dependency. With this method, Chemic can be solved separately without any perturbation from the other modules. After that, the coupling is done and the modules are solved. After all couplings, blocks, and timesteps are processed, the final result is processed, checked

and outputted.

Algorithm 1 Alya Pseudocode.

```
Read Files
Define Mesh Dependencies
Output and Post-process
loop
    Compute time step                          ▷ Calculate Timesteps
    Begin time step                            ▷ Update boundary conditions
    Modules Are Grouped Into Blocks
    DoBlocks
        DoCoupling
            Solve module1 (i.e. Nastin)
            Solve module2 (i.e. Turbul)
            ...
            Check coupling convergence
        EndCoupling
    EndBlocks
    End time step
    Output and Post-process
Output and Post-process
End Execution
```

There are many modules, such as Incompressible Flows (Nastin), Turbulence models (Turbul), Compressible Flows (Nastal), Non-linear Solid Mechanics (Solidz), Species transport and chemical reactions (Chemic), Excitable Media (Exmedi), Thermal Flows (Temper), N-body collisions and transport (Immbou).

In this work, the focus has been given to a test case that only uses one module. The test case represents incompressible flows and does not use coupling. Therefore the complications that coupling modules create will not be covered in this work.

Alya has been developed in Fortran90/95 and parallelized using MPI and OpenMPI. And according to the developers, it is not an initially sequential code that solved individual modules that was later parallelized, it was initially designed to run in parallel and solve multi-physics problems. This is a positive feature of the simulator since many problems caused by parallelization of sequential code have been taken care of in the development of the application. One of Alya's main feature is that it has been designed to run in the most powerful and efficient large scale supercomputing facilities such as Blue Waters, MareNostrum III, JUGENE/JUQUEEN, FERMO(BlueGene/Q) and CURIE(Bull Cluster) (VÁZQUEZ et al., 2014) (RODRíGUEZ, 2014).

Alya is structured in a modular way, being divided into kernel, services, and modules. The modules can be compiled separately and then linked. Each module is dedicated to a specific problem, solving a different set of partial differential equations. A sequential execution could be performed using only the kernel and the modules. The kernel will control the execution and the modules will solve the problem. The services have the function of parallelizing the code, by communicating with the modules through well-defined interfaces and connection points. Thus, the parallelization in Alya is a service.

Alya uses METIS (KARYPIS; KUMAR, 2009) library to partition the mesh, and is executed by the root node sequentially. If the mesh is too big to fit in the memory, it can take a long time to finish. For that reason, it is possible to perform the partition in parallel with parMETIS, a library similar to METIS that runs in parallel, significantly reducing the partition time as shown in (ARTIGUES; HOUZEAUX, 2015).

METIS does an automatic mesh partition, and create a set of subdomains that will interact using MPI in a Master-Worker fashion. The root (Master) process will be responsible for the partition and input of the mesh. The other processes will solve individually each sub-mesh in parallel. When gathering the results no communication is needed between them, and the load balance severely influences the scalability. The size of the interfaces and the communication schedule highly influences the scalability. The iterative solvers use global communicators (MPI_AllReduce) and Point-to-Point communicators (MPI_SendRecv) (RODRíGUEZ, 2014).

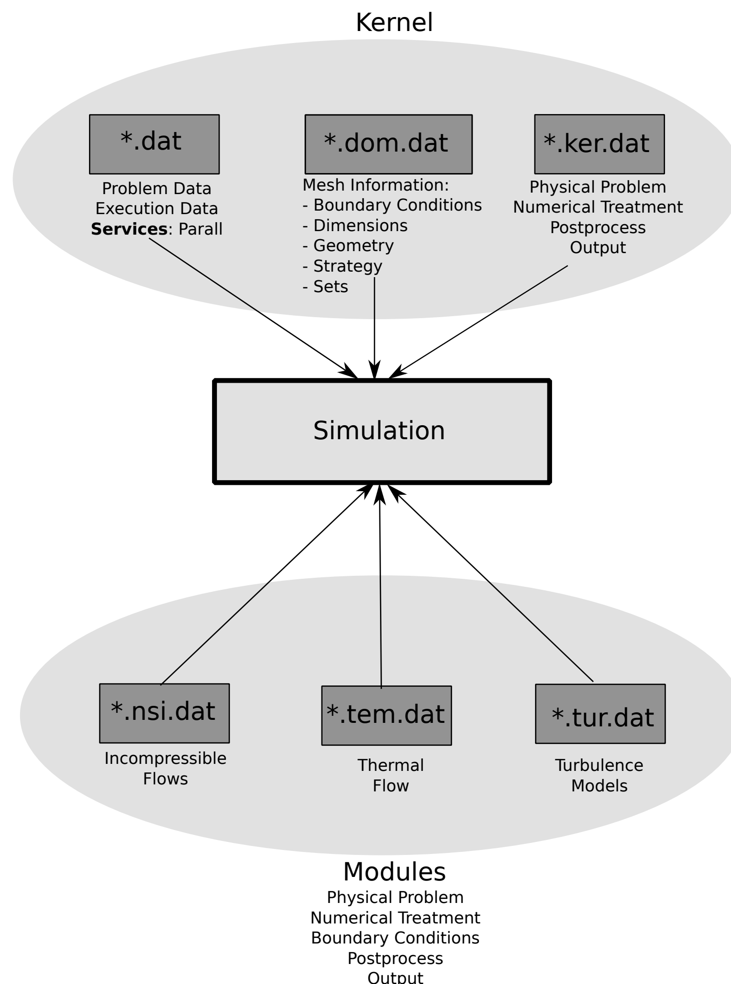
In Figure 3.1 it is possible to see the structure of Alya input files. It was created based on a similar scheme presented in the documentation about Alya (VÁZQUEZ; HOUZEAUX, 2015). To run a simulation the kernel, the modules, and the services must be defined. The figure shows that the kernel is minimally composed of three files, `.dat`, `.dom.dat`, `.ker.dat`.

The information about what modules and services will be used as well as the general data used in the execution is kept in the `.dat` file. Parall is the service responsible for the parallelization and is also kept in the `.dat` file. The information about domain input data that contains the mesh description, and all the dependent variables such as mesh dimensions, element connectivity, boundaries, among others information is kept in the `.dom.dat` file. The `.ker.dat`

keeps the variables shared between modules, the information about the physical problem and the numerical treatment.

The input files for the modules have different format depending on what will be used. For instance, `nsi.dat` represents Nastin input data file (incompressible flows), and it keeps information about the physical problem, numerical treatment, output, post-process, and boundary conditions. The service files are not always needed.

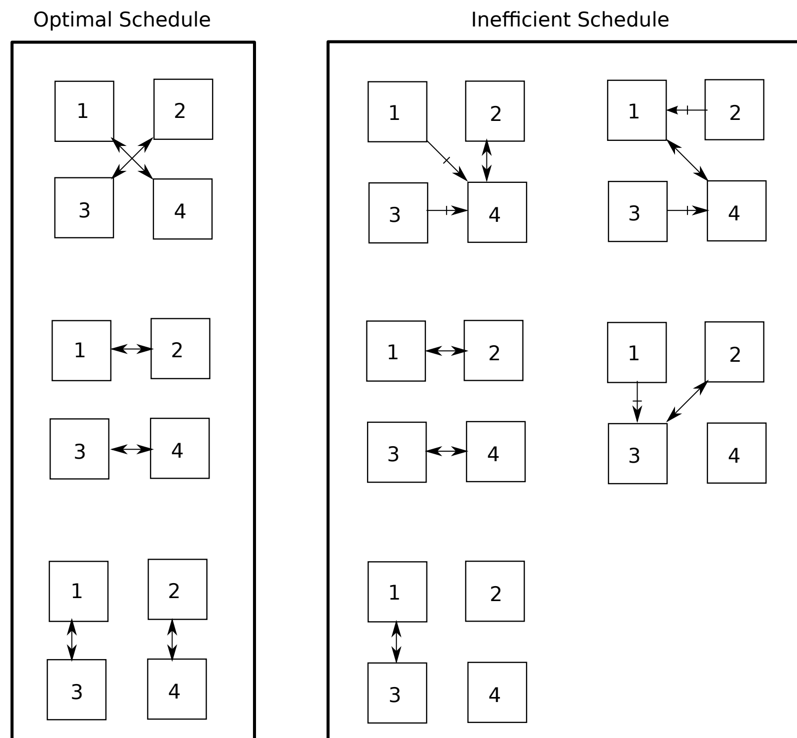
Figure 3.1: Structure of Alya input files.



The Master-Worker strategy is explained in (VÁZQUEZ et al., 2014). It also presents the communication types, scheduling, data structure, division of the mesh and how the boundaries are dealt with. To divide the mesh, Alya uses a coloring scheme, build upon an adjacency graph scheme of each subdomain. The adjacent color groups that have no communication can be scheduled in non-overlapping stages. The Figure 3.2 is based on (VÁZQUEZ et al., 2014). It shows the kind of problem that can be faced with a bad data transfer schedule. The im-

age represents four subdomains that have to exchange data between all others. On the left of the figure an ideal schedule is shown where only three steps are required. The right part of the figure shows a bad communication schedule that tries to communicate in a bad order to exchange boundary data. It needs five steps to finish communication. For instance, subdomain 3 has to wait for step three to be able to transfer data with subdomain 4, staying idle for two steps. There are four steps where only two subdomains are able to communicate properly.

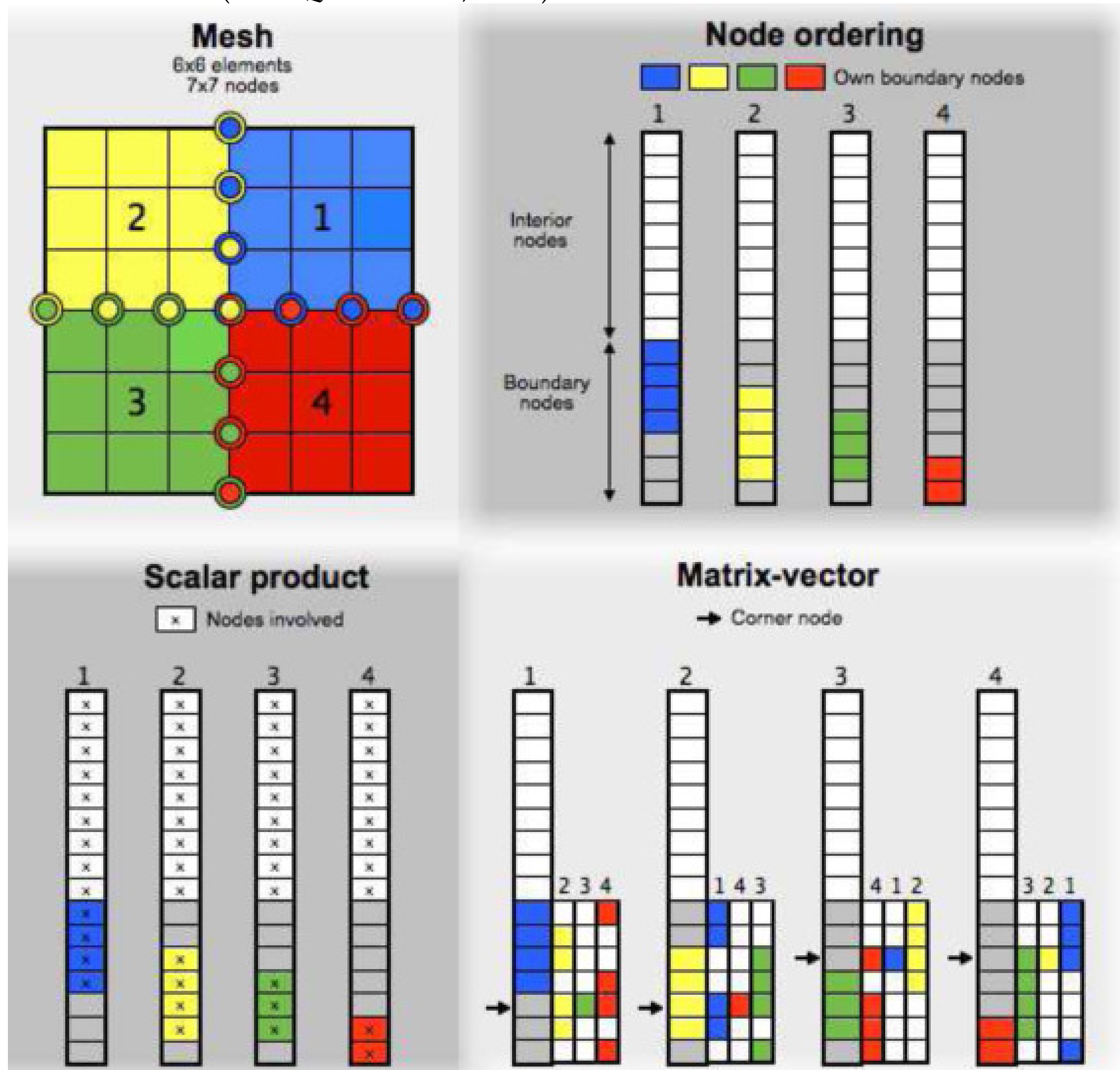
Figure 3.2: Scheduling strategy. Left shows an ideal communication schedule in three steps, on the right a bad communication schedule in five steps is shown. Image based on (VÁZQUEZ et al., 2014)



The data structure used is illustrated in Figure 3.3 with a mesh partition of four subdomains that was done using METIS (top left) was extracted from (VÁZQUEZ et al., 2014). In the top left it shows that each worker orders the interior nodes, and then boundary nodes are divided into `own boundary nodes` and {others boundary nodes}, presented on the top right. The division is needed to avoid repeating the contribution of a boundary nodal value. In the bottom left, the `x` sign represents the nodes involved in scalar products. And the nodes involved in MPI_ISend and MPI_IRecv are shown in the bottom right.

Alya uses Sparse Matrix-Vector (SMV) product to help calculate and com-

Figure 3.3: Organization of data and types of communication. This image was extracted from (VÁZQUEZ et al., 2014)



municate results between neighbors nodes. First, SMV product is performed on boundary nodes. Then this information is exchanged between neighbors in a non-blocking manner with commands MPI_ISend and MPI_IRecv, then the local SMV is performed on the interior nodes, and then the solutions are updated and synchronized with MPI_Wait_All.

As our main goal is to analyze the performance, the final output data will not be the focus of this work. In the end of Alya execution, there is a checker that evaluates the final answer and indicates if the results are correct or not. Alya has been tested in several situations such as in (AVILA et al., 2013), (VÁZQUEZ et al., 2014), (RODRíGUEZ, 2014). Thus, proving that the final outputted data is correct is not our goal.

4 RELATED WORK

Alya is a part of the PRACE Benchmark and is present in (PRACE, 2013) and (ALYA, 2013). It has been used in several projects related to physical simulation and parallel application. One of them is (VÁZQUEZ et al., 2014). It describes the simulation strategy of Alya and the equations of the cases studied. Through experiments, it concludes that Alya is scalable since the executions use up to 100.000 cores simulating different physical problems. One of the key problems pointed by the paper is the post-processing since the file generated are of several gigabytes. This problem is also faced in this dissertation. The same work also describes what is an ideal scheduling strategy with good communication, and a bad communication scheduling for ALYA. It also explains how the mesh data is divided between the nodes and shows how the boundary problems are solved.

A parallel Computational Fluid Dynamics (CFD) model for Alya is presented in (AVILA et al., 2013). It describes a wind farm model showing in details the model strategy and the numeric algorithm using finite element discretization. The result of an Alya simulation is visualized containing information about the final simulation, such as temperature, wind speed, viscosity, pressure, among others.

A similar experiment done in this dissertation has been presented in (RODRíGUEZ, 2014) where Alya performance has been analyzed with Extrae tracing tool. Relevant differences are the physical problems solved, the size of the problem, and the execution environment. The paper focus on massive size problems, only trace a part of the execution, and run on Tier-0 machines, with up to 20,000 cores such as JUGENE/JUQUEEN, FERMI, CURIE. The authors state that the Extrae tracing tool generates a large amount of data since it monitors information about hardware counters, states burst, and events generating traces of several gigabytes. For this reason the paper concludes that it is not possible to manage such amount of data on an average computer, and to escape that problem, it does not trace the whole application. In our approach, we tackle a smaller problem but get information about the entire execution.

The creation of traces for big parallel applications is a challenge due to the big amount of data present in resulting files. This problem is reported in

(RODRíGUEZ, 2014). One approach is to only trace part of the execution, however, depending on the number of processors being traced, even a small share of timesteps can generate massive files of several gigabytes. Analyzing big amount of data has a lot of particularities and difficulties that can only be managed by using certain data manipulation techniques combined with the R plotting tool as can be seen in (SCHNORR; LEGRAND, 2013).

The paper (PEARCE et al., 2012) presents metrics for load balance quantification and characterization, such as percent imbalance, taking into account disparities in the data distribution. Those metrics are used and validated in (RODRIGUES, 2016) that formally calculates the load balance metrics for a set of different applications.

Alya presents load imbalance in some cases as reported in the works (CAJAS; HOUZEAUX; EGUZKITZA, 2015) and (RODRíGUEZ, 2014). This information was extracted using Extrae and tracing only part of the execution. In those works Alya is executed on high end computers like MareNostrum III using up to 2048 cores and solving massive problems. In this project, we will verify if a load imbalance can be found in a smaller problem running on computers with lower processing power. To do that we trace the entire application using three different tracing tools for different situations and experiment specification. The next chapter presents the workflow employed to investigate Alya performance.

5 INVESTIGATION WORKFLOW AND EVALUATION METHODOLOGY

This chapter contains a general overview of the workflow, as well as detailed information about the tool, techniques and approaches used in this project. Different experimental methodologies had to be used in the experiments performed in order to better evaluate the particularities of each tracing tool but they present several similarities. The general methodology for the creation of this work is also presented in this chapter.

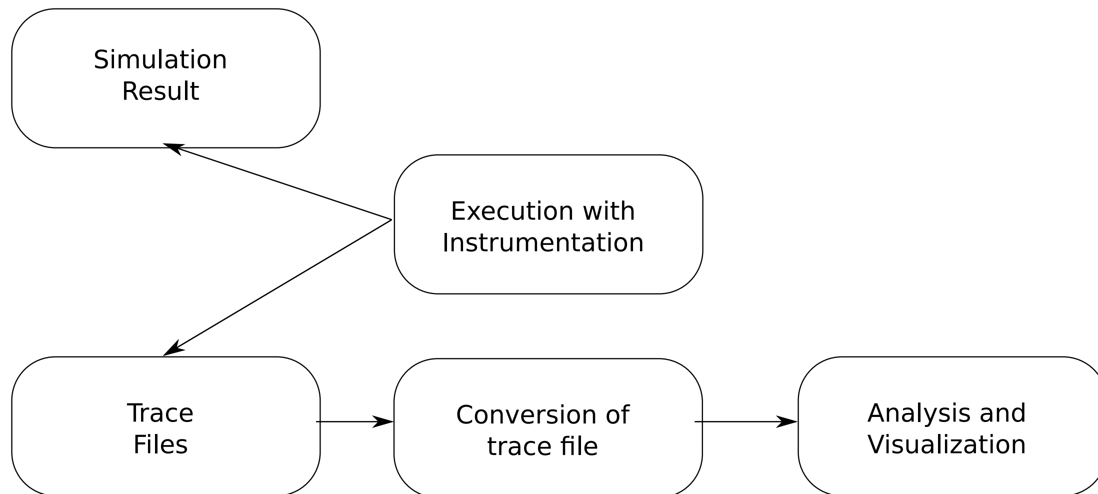
5.1 General Overview

Figure 5.1 presents an overview of the investigation methodology. The methodology used in all tracing tools follows the same pattern. Despite that, each tool has its own particularities presented in Section 5.2. As shown in the figure, first the application is instrumented and executed, then outputs the simulation result and the trace files. Those trace files are converted to a format that can be easily read, and then the data is plotted, visualized and analyzed.

Applications can be instrumented in many different ways as previously stated in Section 2.2 and the tracing tool receives the instructions of what should be traced in the application. As the application runs, the trace files are created, and one trace file is created per process per node. If the execution is multi-nodal after the execution all trace files have to be copied to one node and then merged. Each tracing tool outputs a trace that has to be converted to a format that makes it possible to read and make a post-mortem analysis in R. With the files in a proper format, the analysis in R is performed and plots containing behavioral information about Alya are created.

The amount of timesteps calculated in each execution can be easily defined and has been changed according to the goals of the experiment. The specific process of compiling Alya is described in detail in the labbook present in (CAMELO; SCHNORR, 2016). The general steps are to modify the `config.in` file to comprise the wanted settings, then run the `configure` file to create the `makefile`, run `make` of metis4 that is responsible for the automatic mesh partition (KARYPIS; KUMAR, 2009), then run `make`. After this, the binary file of Alya can be executed sequentially simply running the `Alya.x` file after compilation.

Figure 5.1: General methodology



In order to run in parallel OpenMPI commands must be used.

Alya can be executed in one node (shared memory) or multiple nodes (distributed memory). In order to run in multiple nodes, some communication tags and a machinefile must be used. The important communication flags are `-mca btl_tcp_if_include em1` and `-mca btl tcp,self`, that set the appropriate communication protocols for the environment used. The parameter `-machinefile` receives the machinefile file that defines the number of ranks and in which host the application will be executed. Listing 5.1 shows an example of a machinefile that uses 64 cores divided between two nodes, draco1, and draco8.

Listing 5.1 – Example of machinefile for an execution with two nodes.

```
draco1 slots=32 max_slots=32
draco8 slots=32 max_slots=32
```

5.2 Per-tracing tool Implementation

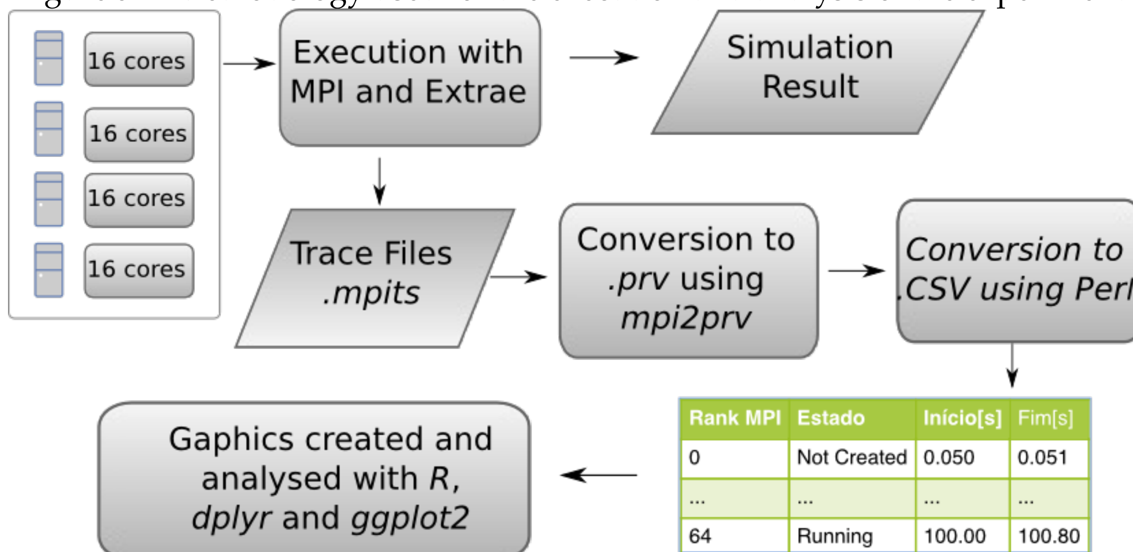
5.2.1 Extrae

The experiment involving Extrae used only four nodes of the Draco cluster with a total of 64 cores. The instructions to execute in a multi-node fashion are described at the end of Section 5.1. The simulation ran only until the end of

the third timestep due to the large size of the trace files. The trace files for this execution have a size of 2,7 gigabytes. The tool Extrae creates individual traces for each processor, keeping information regarding the communication and execution. Multiple files are created in `.mpits` format containing information about what was executed by a given process. It is then necessary to convert and merge them using the tool `mpi2prv` that outputs the Paraver format `.prv`. After that, a Perl script is used to filter the relevant data creating a Comma-Separated Values (CSV) file used for the analysis scripts. (RODRíGUEZ, 2014) also uses Extrae to trace part of the application, but uses a tool to visualize it.

The resulting CSV output has four columns of data: the MPI rank (Rank), the time in which the process entered the state (Start), the time in which the process finished the state (End), and the MPI state name (State). With this information, it is possible to calculate the actual workload (running time in seconds) of each process, as well as create space/time graphics that tells us the order and what was the time that each state occurred in each process. All calculations are made from scripts written in the R language, making use of ggplot2 and dplyr libraries. Figure 5.2 gives a summary of the steps to trace Alya and to conduct the performance analysis of the experiment when using Extrae.

Figure 5.2: Methodology used for the execution and analysis of the experiment.



The schematics on Figure 5.2 represent a possible execution setting with four nodes each using 16 cores representing an arbitrary setting. The first and second blocks represent the execution with MPI and trace with Extrae. The next two blocks in the flow are the output of the simulation results and the trace files in

.mpits format. The following two blocks represent the conversion of the trace to .prv and then to CSV using mpi2prv and a Perl script, respectively. Then a block representing the CSV format is presented followed by the last block that represents the creation of graphics and analysis of the experiment using language R.

5.2.2 Score-P

As previously stated, the trace generates files of massive size, so in order to filter functions ScoreP receives as input a configuration file to filter events. That way only functions that are relevant for the experiment will be present in the final trace. In the filter file initially all functions are excluded with the selector * and then MPI and communication functions are added. After that, the Alya functions endste_ and timeste_ are included. Those functions mark the beginning and end of each timestep. The filter instructions used can be seen in the Listing 5.2.

Listing 5.2 – Filter file that excludes all events then includes MPI events and timestep functions.

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE *
INCLUDE MPI COM
INCLUDE endste_ timste_
SCOREP_REGION_NAMES_END
```

One particularity of ScoreP is that for executions with more than one node, it assumes that there is a NFS system. The available machines did not have NFS so multi-nodal experiments were not performed. Chapter 7 presents some possibilities of future approaches that tackle this problem.

Some environment variables had to be set to configure the execution, such as: SCOREP_EXPERIMENT_DIRECTORY it defines where the traces will be stored; SCOREP_OVERWRITE_EXPERIMENT_DIRECTORY=true to overwrite old traces; SCOREP_ENABLE_TRACING=true so that the tracing of the application is enabled; SCOREP_FILTERING_FILE=<dir_to_filter_file> to use the filter file wanted; SCOREP_TOTAL_MEMORY=<memory_per_core> that defines the memory used by each core. With these flags, the execution can be performed

in a single node.

The trace files are outputted in OTF2 format and converted to paje format with `otf22paje`. Command `pj_dump` is used to create raw data that can be loaded into R. Then pre-processing is performed removing trailing spaces and unwanted information. Finally, the data is ready to be plotted in R.

5.3 Analysis Tools

The technology used for this project was the combination of Emacs 24.5 (and newer version) and Org mode 8.3.5. Org mode is an Emacs mode for note keeping, project planning, TODO lists and authoring. With Org mode it is possible to describe the work being done while doing it. Using tags managed by Lisp it is possible to run blocks of code inside the labbook. All commands that a user type inside a terminal can be executed from the labbook and the results are outputted inside the labbook. With this approach, all the work done will be logged into the labbook. Org mode is also used to write this work, once written the document is exported to Latex and results in a PDF file.

Several tools were used in this work. R was used to analyze the results and organize some experiments. Screen is a full-screen window manager that multiplexes a physical terminal between several processes screen, was used to run experiments in remote machines without having to keep a open terminal on the local computer. Both ssh and scp were used to remotely control the high performance computers, and copy files between computers. Vim editor was used on remote computers. Latex was used to create the text about this work. Extrae and ScoreP are the tracing tools used.

5.4 Summary

This chapter presented the general process needed to trace Alya followed by an explanation that details the particularities that each tracing tool require. The technical configurations in order to run the application in one node and in multiple nodes were also explained. Lastly, the tools and programs used throughout the entire process were listed along with their contribution to this work.

6 RESULTS

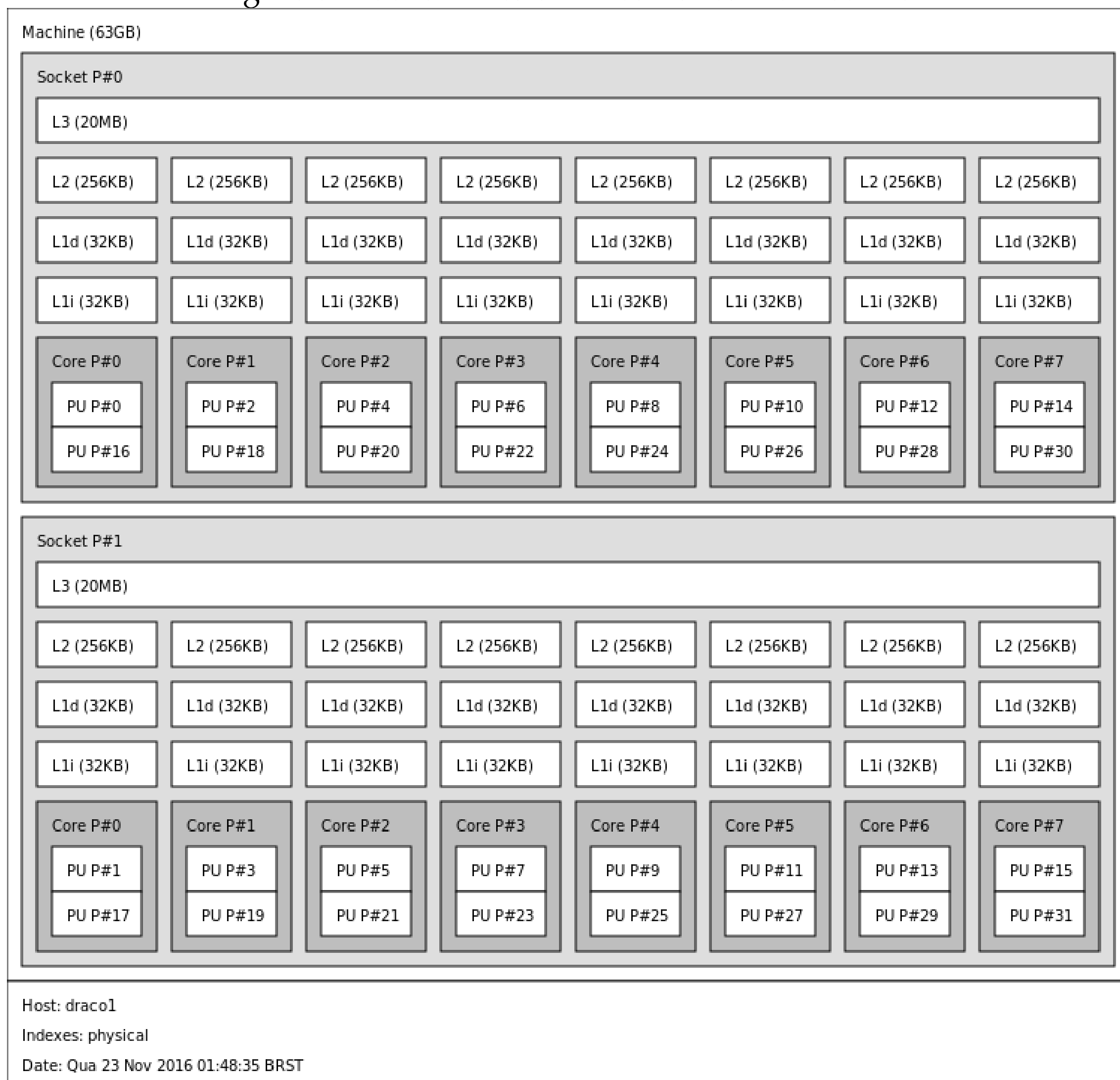
In this chapter results of three experiments are reported. The first experiment was conducted using the BSC tool Extrae in a four node distributed memory environment, running only three timesteps, with a general load balance evaluation of the whole execution. In order to improve the precision of the load balance metrics the second experiment was performed with ScoreP tracing tool in only one node in shared memory environment (due to ScoreP environment restrictions as explained in Section 5.2.2), running for 10 timesteps. The load balance metrics evaluation were measured in each timestep giving an insight of the local load balance.

Experiments were conducted on computers that are part of the Draco cluster at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS). The cluster is formed by eight nodes with identical architectures, each one equipped with two Intel (R) Xeon (R) E5-2640 v2 CPU @ 2.00GHz processors with eight physical cores (16 with hyper-threading) resulting in 32 cores in each node, 64 gigabytes of RAM, running Ubuntu 14.04.4 LTS. Different amount of cores and nodes were used in the experiments and are described in this chapter along with the experiments description and results. The schematics in Figure 6.1 shows the detailed architecture of a single Draco that is part of the Draco cluster. It is possible to see both logical and physical cores, as well as the cache sizes, L1i (instructions) with 32 KB, L1d (data) with 32 KB, L2 with 256 KB, and L3 with 20 MB. The schematic was extracted with the tool lstopo.

6.1 Distributed Memory Experiment with Extrae - three timesteps

The goal of our performance analysis is to identify relevant characteristics of the application and evaluate whether it presents load imbalance among resources and along the time. We also intended to refine our methodology to conduct larger scale experiments. It is possible to see some of these refinements in the experiments with more timesteps presented in other sections of this chapter. We provide an overview about the load balance, a detailed analysis using a traditional space/time view, and an attempt to explain the identified behavior using per-rank state statistics. We end the analysis by globally applying the

Figure 6.1: Schematics of draco nodes architecture.



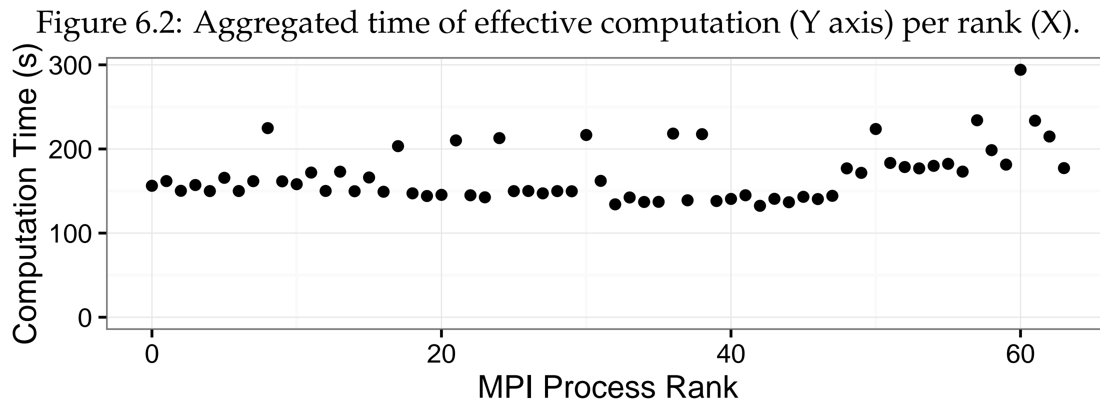
percent imbalance metric to evaluate the load imbalance. All data presented in the graphics from Section 6.1 are acquired from one execution using four nodes and 64 cores, running until the end of the third timestep in a distributed memory environment.

6.1.1 Overview of Load Imbalance

Figure 6.2 shows the aggregate time of effective computation for each process in an experiment that executed three timesteps. Computation is calculated by summing all time periods in which the process performs data processing. All periods of time on MPI communication, point-to-point synchronization, and col-

lective synchronization are not included in these measures. It is observed that most processes have a total computing time of roughly 150 seconds. In some cases, however, the time reaches up to 300 seconds, for instance, the process 60. This is an indicative that a load imbalance occurs considering this specific case study.

Figure 6.2: Aggregated time of effective computation (Y axis) per rank (X).

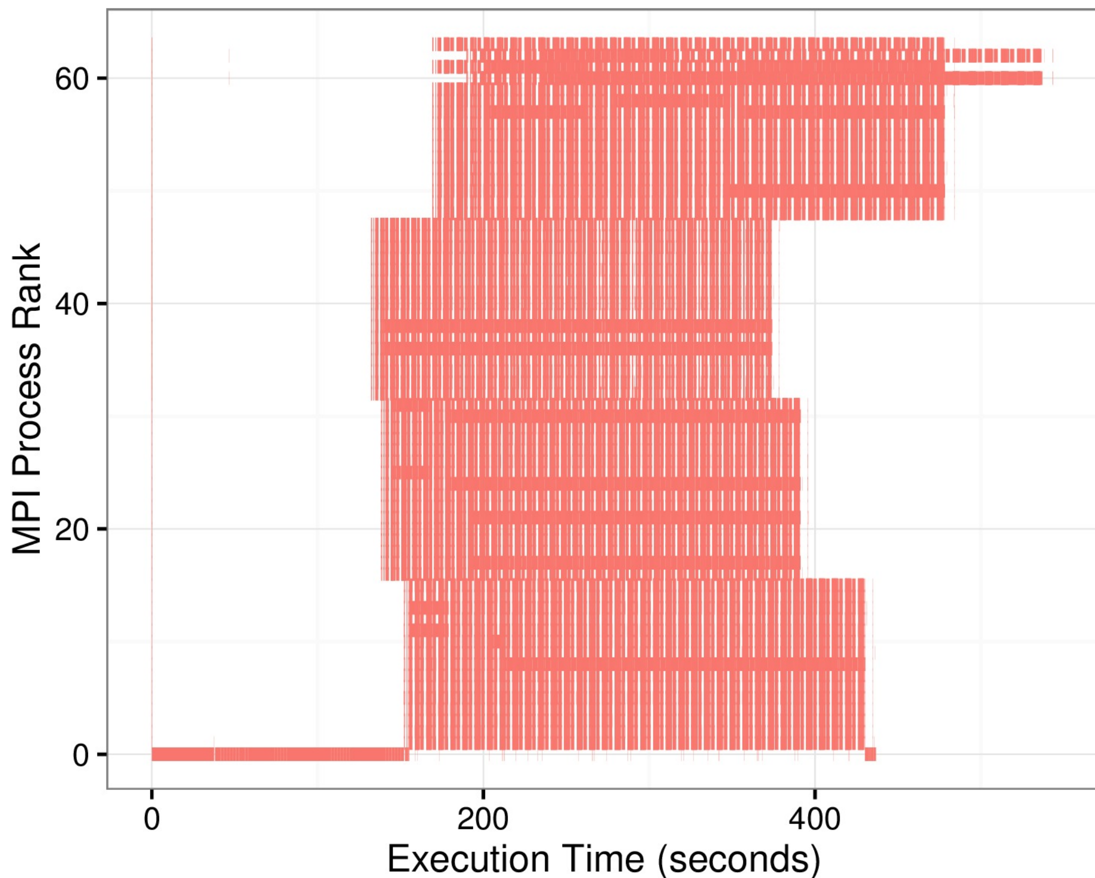


6.1.2 Space/Time Execution

Figure 6.3 presents detailed information about the load balancing, showing the time that each process was in a state of effective computation for an execution of three timesteps. At the beginning of the execution, process zero engages in dividing the mesh sequentially in order to distribute the work among processes. This activity continues until the 150 seconds mark. After this time mark, the remaining processes begin to work and the root process (zero) becomes idle while waiting for the response of other processes.

An interesting feature that can also be observed in Figure 6.3 is the existence of four groups of processes: the 0-15, 16-31, 32-47, and 48-63. This behavior correlates with the number of nodes used in the experiment, indicating that internally in a node all processes begin their computation roughly at the same time. This means that the initial load distribution from process zero is not scalable because visually the computation in each one of the four machines start sequentially: first the group between the processes 32-47, after the group of processes 16-31, then the group containing the zero process (0-15) and finally the group of processes 48-63. In the latter group, we also observe an anomaly in the processes

Figure 6.3: Timeline (X axis) showing computation states per process (Y axis).



60 and 62. They start and finish their computation after the other processes from their group. Such anomaly is observed only within this group.

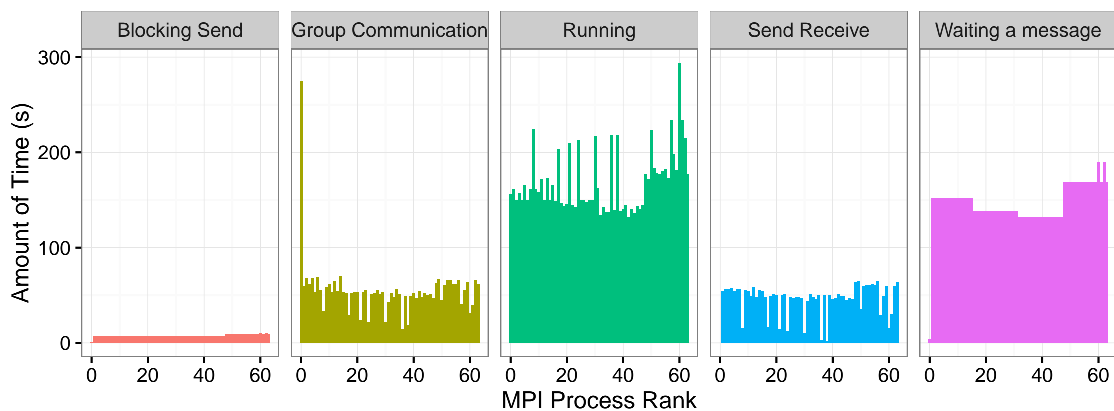
The process 60 has a peculiar behavior when compared to the other processes. Figure 6.2 features the time of effective computation: it runs for roughly 300 seconds. Figure 6.3 shows that process 60 has an anomalous behavior, beginning and ending its execution last. Moreover, the execution state (Running) of this rank, where the computation is actually performed, does not seem to contain as many spaces as other ranks. This probably indicates that the time spent on communication functions for this particular case is much lower than in other processes.

6.1.3 Process Behavior by State

Figure 6.4 displays a summary of dedicated time (Y axis) by process (X axis) to each state (different facets). Only the five most important states are pre-

sented (*Blocking Send*, *Group Communication*, *Running*, *Send Receive* and *Waiting a message*). The other states present very little or nonexistent time. The *Blocking Send* has less influence in load balance than the other states since they are somewhat similar in all processes (except for the 48-63 group, slightly higher). The *Group Communication* has quite different times among processes and a very long time to process zero, as detailed in previous sections. Load imbalance is shown in the facet *Running*, similar to the data presented in Figure 6.2. The sending and receiving times are relatively homogeneous between processes, except in some cases in which they are smaller. Therefore, it is possible to see in the rightmost facet of the graphic that the possible reason for the anomaly of the processes 60 and 62 is due to additional time spent in the state *Waiting a message*. Such fact contradicts our previous hypothesis drafted in the previous section, where the timeline visually indicated less communication time for process 60. This is probably due to drawing much more events than the screen space available to draw them (SCHNORR; LEGRAND, 2013).

Figure 6.4: Time spent in each state by process.



6.1.4 Percent Imbalance

In this experiment the calculation of the metric is performed considering the entire execution, thus being a global indicator of load imbalance. The metric characterizes the uneven distribution of work formalized by the Formula 2.1, and when applied to our measurements gives the value $\lambda = 74.25161$. That represents a workload imbalance of 75%. This indicates that if the load is more evenly distributed between the computational resources there would be room for a po-

tential performance improvement.

6.2 Shared Memory Experiment with ScoreP - 10 timesteps

In order to evaluate the load balance, an advised approach is to analyze the execution time of each core. With this method, it is possible to see the load balance state for a given period of time. This has been done in the experiment with Extrae in Section 6.1. The issue with that result is that only a global value is given about the load balance, it is not possible to know in which part of the execution the load balance is better or worst nor what might be affecting the execution performance. With that in mind, an experiment that evaluates the local load balance rather than only the global load balance has been designed.

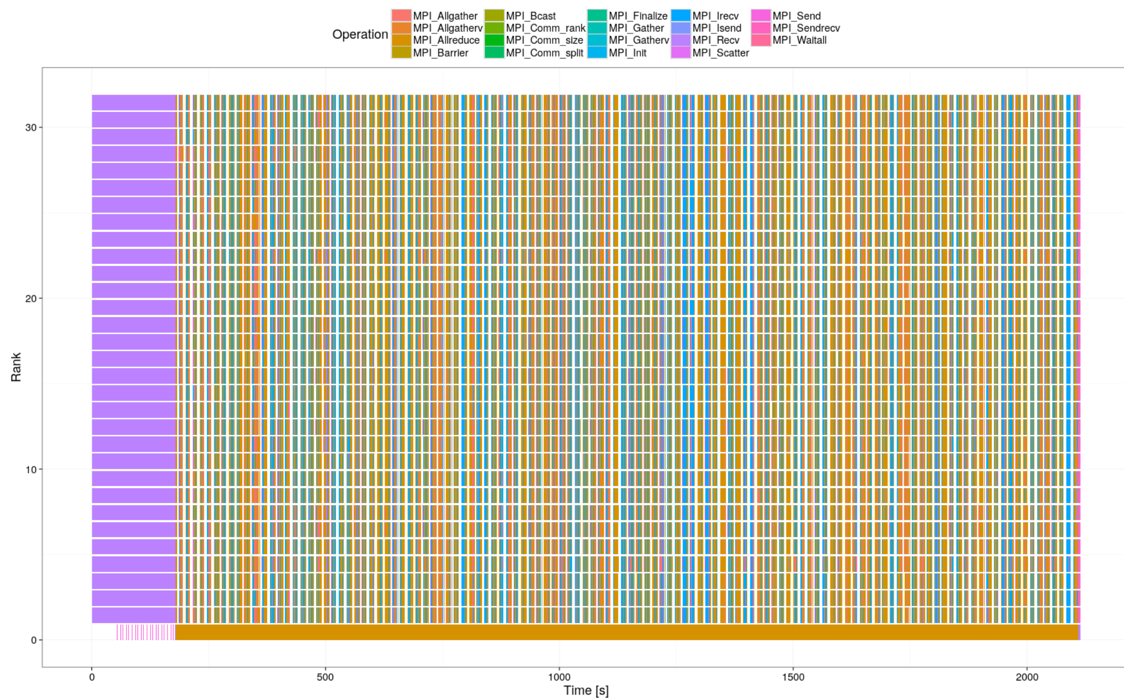
This experiment was executed on only one node using 32 cores in shared memory environment and tracing with ScoreP. The environment used was the computer draco3 that is part of the Draco cluster described in Section 5.1.

It was possible to identify the timesteps by tracing the functions `timste` and `endste` called by the main loop in `Alya.f90`, those functions mark the beginning and end of each timestep. By doing that, in the trace files, there will be an event for each core that can be used to create one data frame per core per timestep so that analysis can be made in a more significant way.

In Figure 6.5 it is possible to see all MPI events that occur but there is too much information. It is possible to identify the problems mentioned in Chapter 4 about the abundance of data and the difficulty of showing all information on a single plot. Despite that, it is possible to see some patterns. For instance, in the first part, all worker ranks were in a state of MPI_Recv, while the master rank was dividing the mesh. After that, the workers start to execute, solve the problems and communicate, and the master rank goes into a state of MPI_All_Reduce and waits for the ranks to finish solving the problem and report to the master rank. During 8,5 % of the total execution time, the root process is the only one doing effective computation. Until the beginning of the first timestep at the 181 seconds mark all other processes are idle.

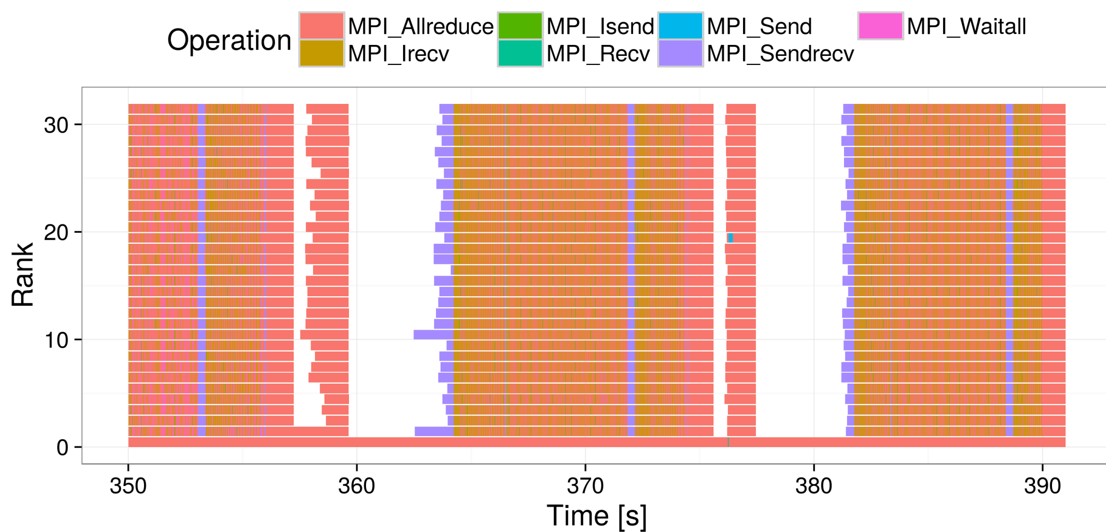
Zooming in specific parts of the execution can provide more significant information. In Figure 6.6 a zoom has been applied in the iteration border between the first and the second iteration. The mark of 376 seconds divides the

Figure 6.5: Gantt Chart with Space/Time visualization of the MPI events.



iterations, and at that time it is possible to see that all ranks synchronize with an MPI_All_Reduce, then a small execution period, then another synchronized MPI_All_Reduce period, then a large portion of effective computation. Based on Chapter 3, one possible hypothesis is that the first execution period was working on the boundaries of the subdomains, then communicated the dependencies, and on the second, large execution period, the heavy work started.

Figure 6.6: Gantt Chart with Space/Time visualization of the MPI events during iteration border.

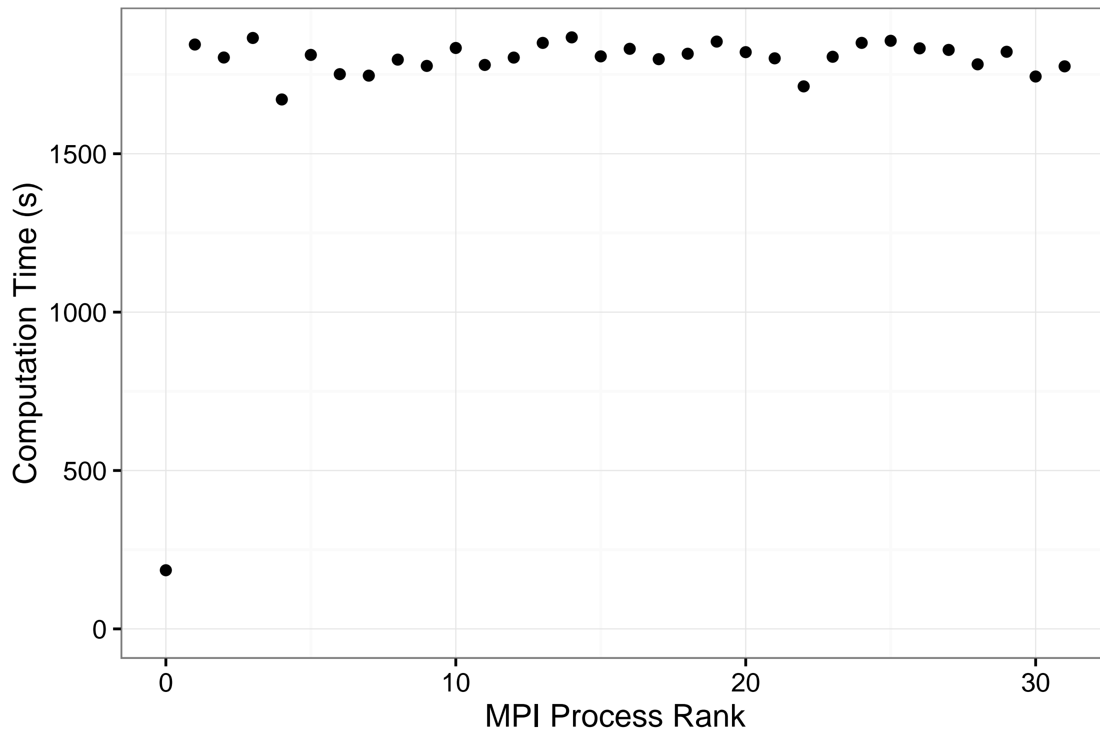


In order to reach the aggregated time of effective execution by process, the data was grouped by rank, and all the time intervals that MPI operations occurred were summed and then subtracted from the total execution time, which was 2113.686 seconds. By doing that the information of how much time each rank spends in effective computation becomes available.

An image with the aggregated time of effective computation by core is presented in Figure 6.7. It is possible to see that most processes have execution time ranging from 1671 seconds to 1867 seconds with one exception, the root process. The root process uses some time in the beginning of the execution to divide the mesh, organize and communicate with all other processes, and after that stays idle waiting for the other processes. Thus in total it runs for less than 186 seconds.

Apart from the root process, the load balance seems to be good, as can be seen in Figure 6.7, this supposition is explored in the next data analysis.

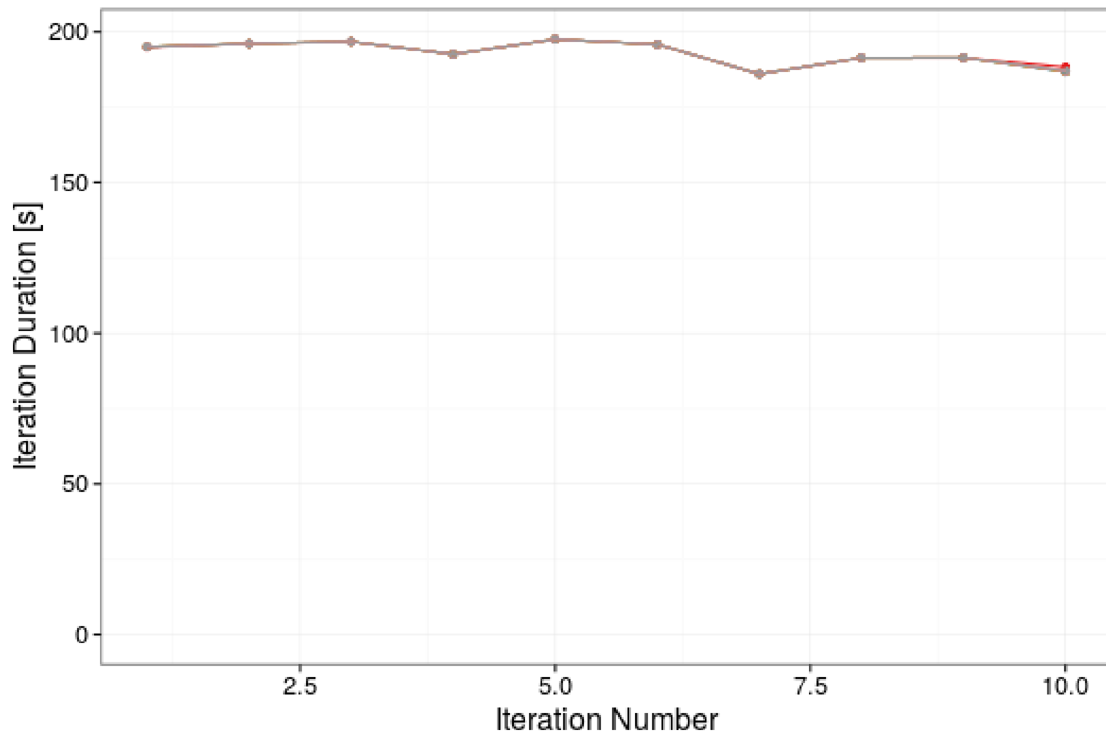
Figure 6.7: Aggregated time of effective computation (Y axis) per rank (X).



Another interesting analysis that can be done using the timesteps is to find out how long each timestep takes, and if they have similar times of execution. In Figure 6.8 each process represents a line, so since most lines overlap each other it is shown that the timesteps have roughly the same duration. It is important to

note that this analysis is only for 10 timesteps, so it is not possible to assure that this behavior would be the same for a longer execution.

Figure 6.8: Plot of duration of each timestep.

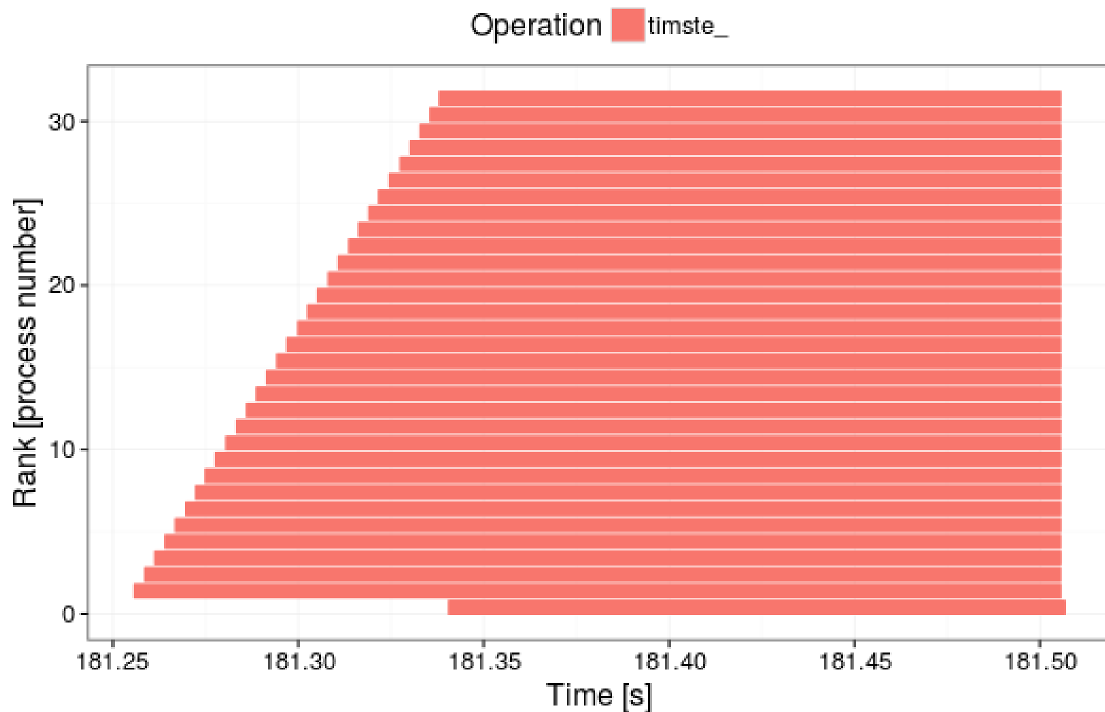


As stated before the function `timste` and `endste` mark the beginning and end of one interaction, so they were also traced in this execution. The function `timste` has been chosen as a divisor of timesteps because it is the function that starts a timestep. Analyzing the calls of the `timste` function we can see in Figure 6.9 they are all called sequentially and finish at the same time. The last one to be called is the root process. It is important to note the timesteps start roughly at the same time in all ranks.

One significant information that can be extracted from parallel applications is the communication ratio and the computation ratio. They are both presented in Figure 6.10, on top the communication ratio (a) is presented and the computation ratio is shown in the bottom (b). It is expected that both ratios to be complementary to each other, and in fact, it is possible to see that both values are correlated since when one rises the other decreases.

To get a better visualization the master rank has been excluded since it will present very high levels of communication. The initial and final iterations present higher communication and are not presented in the figure to better vi-

Figure 6.9: Space/Time plot of `timste` call event with division per rank. X axis represents time, Y axis represents rank.

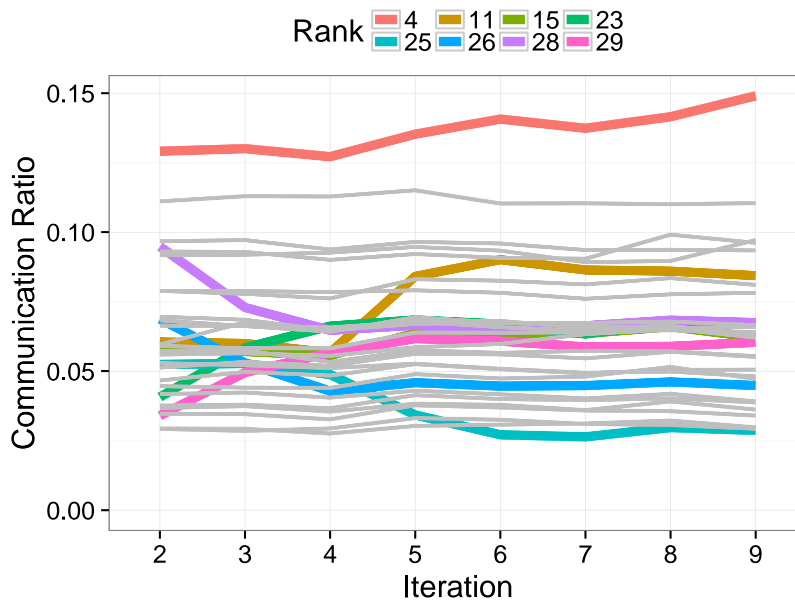


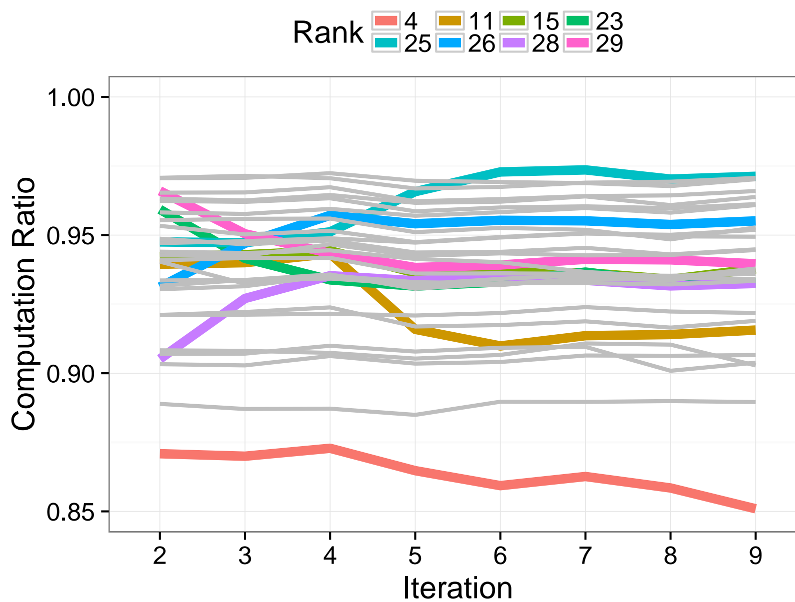
sualize the other iteration information. In the beginning of the execution (initial iteration) there is more communication required to divide and share the subdomains. Similarly, in the end (final iteration) more communication happens in order to assemble all the solution in the master rank.

The cores with higher variance are represented by thicker lines with colors while the most stable cores are gray. Looking at the communication ratio, it is possible to observe that the communication is not too high, which is beneficial for the performance of the application. There are some anomalies on communication ratios in certain cores that do not remain stable, and one, in particular presents a tendency to grow throughout the execution, finishing with more than 16% of its time communicating.

Once again using the timestep function `timste` as a timestamp it was possible to divide the data set into smaller data sets, apply local analysis and calculate local metrics. One of the metrics was the percent imbalance, calculated and plotted in Figure 6.11. As pointed out in Chapter 2 the smaller the value of the percent imbalance the better is the workload distribution. The first iteration is not shown in order to gain precision since it presented low levels of load imbalance.

Figure 6.10: (a) Communication Ratio without rank 0 and without iteration 1 and 10. (b) Computation Ratio without rank 0 and without iteration 1 and 10.



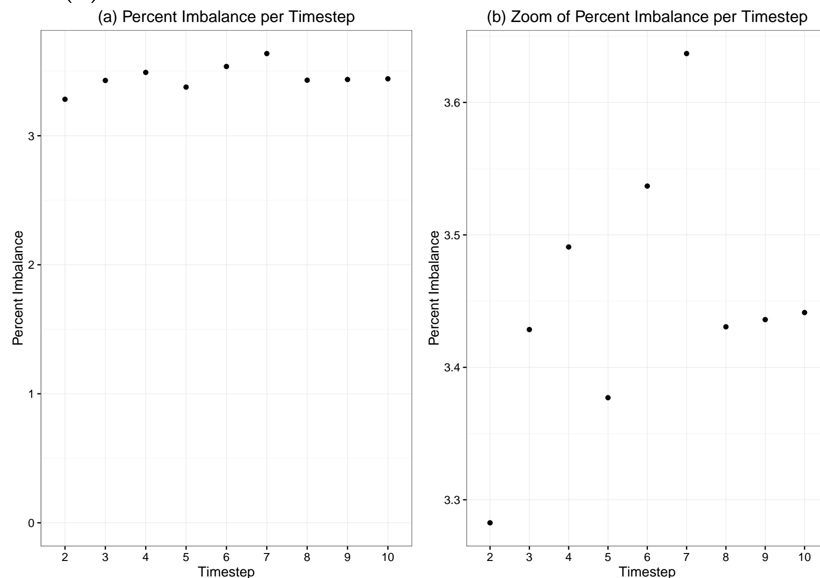


The root rank (zero) is not being considered for the calculation of the percent imbalance.

The chart on the left of Figure 6.11 (a) shows the whole plot, followed by a zoom on the right (b). In the zoomed image it is possible to see that the percent imbalance has a tendency to rise as the execution proceeds, the application starts with percent imbalance lower than 3,3 and increases reaching values higher than

3,6 in timestep 7, and finishing with load imbalance values between 3,4 and 3,5. The behavior in timestep seven is very different from other timesteps, as can be seen in Figure 6.8, timestep seven is the shortest timestep.

Figure 6.11: Local percent imbalance metric by timestep. X axis represents the percent imbalance, Y axis represents the timestep. In (a) the whole plot is presented, and in (b) a zoom is shown.



6.3 Comparison between the tracing tools

In order to evaluate the tracing tool used in this work, experiments were conducted with a different number of cores, timesteps and in shared and distributed memory environments.

One constant concern when creating the traces is the size of the files. This a serious problem because the size of the final trace quickly escalates with the number of cores as stated in Chapter 4 and in (VÁZQUEZ et al., 2014). This problem was faced in all tracing tools used, but for ScoreP selective instrumentation was used to reduced the size of the traces making it possible to run the application with more timesteps.

The usage of selective instrumentation along with ScoreP made it possible to exclude all information that was not relevant for this analysis and also to keep track of the timestep functions resulting in a more meaningful analysis. The selective instrumentation in ScoreP is simple and effectively done with a filter file as explained in Section 5.2.2. In comparison, this is a clear advantage that ScoreP

has over Extrae. The selective instrumentation in Extrae has dependencies and is not as practical as ScoreP.

Extrae deals well with distributed memory executions. On the other hand, ScoreP presented some issues. ScoreP assumes NFS to run in a shared memory environment to synchronize the data. The system that was currently available did not have NFS, and no experiments were conducted with more than one node with ScoreP.

7 CONCLUSION AND FUTURE WORK

Parallel applications have an immense potential to reduce execution time, but they do not follow regular sequential programming paradigms, and the same applies for the execution flow. For that reason debugging or even figuring out how a parallel code works might be a challenge without the use of any tool. In this work the power of tracing tools was shown. Together with data analysis in R, it was possible to explicit the behavior of a big parallel application, showing communication patterns, synchronization points, effective communication, and evaluate the performance. The usage of R with raw data made it possible to perform several kinds of analysis, providing a certain degree of freedom that a visualization tool would not be able to provide.

The experiments performed with Extrae were part of the investigation regarding distributed memory. The application could not be executed with a higher number of timesteps due to the massive size of the trace files created that raises quickly as more ranks are involved in the experiment. Therefore, a simulation with three timesteps was carried out on a platform of four computer nodes totaling 64 cores, leading to an execution of approximately 450 seconds. The execution was traced using the Extrae, enabling us to discover relevant information about the core operation of Alya in the addressed case. We observe that a significant share of the time (about 34% in a run with three timesteps) is somewhat wasted in the beginning of the execution to divide the mesh sequentially, creating considerable overhead since remaining processes are kept idle. Other results include the detection of anomalies in some processes and the perception that the root process (zero) sends the partitioned data sequentially to different nodes, making the start of application considerably slow and not scalable. One other possible reason for this is the synchronization between the nodes.

The experiments performed with ScoreP show the communication pattern and explicit the behavior of the application in terms of MPI events. They also show the pattern and load balance of each iteration. It was possible to see that the iterations have basically the same duration, and start at the same time with synchronization. The execution has been performed with 32 cores all in the same node processing 10 timesteps running for roughly 2110 seconds. It was possible to see the communication ratio was good, never being higher than 16 %. Only

8,5% of the total time is employed to divide the mesh and distribute the problem. During this time only one rank is computing and compared to the experiment with three timesteps, the overhead of the division and distribution of work is significantly smaller. This means that if more timesteps are involved, the time percentage that the application spends in a sequential execution is reduced.

Some issues have been found in this work and will be tackled in the future. One issue faced in the experiment was the environment requirements of ScoreP. It assumes the there is a Network File System(NFS) for the executions in multiple nodes. Those conditions were not met by the computers available. One future approach is to conduct a similar experiment in a system with NFS. Then multi nodal (distributed memory) execution with medium size problems will be performed. Another issue was that only one test case was considered for this work, it is expected that more test cases with different configurations will be available soon. Future experiments will comprise results from different scenarios with different parameters that will be compared with the results found in this work. Some initial research has been done towards the use of the tracing tool TAU, that seems promising. The tool presents selective instrumentation and can run multi nodal experiments without NFS. In the future, experiments will be done using this tool.

REFERENCES

ALYA. **The Alya System - Large Scale Computational Mechanics: Alya overview**. 2013. <http://www.bsc.es/computer-applications/alya-system>. Accessed: 2016-06-30.

ARTIGUES, A.; HOUZEAUX, G. Parallel mesh partitioning in alya. 2015. Acessed: 2016-11-18.

AVILA, M. et al. A parallel cfd model for wind farms. **Procedia Computer Science**, v. 18, p. 2157 – 2166, 2013. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050913005292>.

BARNEY, B.; LIVERMORE, L. **MPI - Message Passing Interface**. 2016. <https://computing.llnl.gov/tutorials/mpi/>.

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

CAJAS, J.; HOUZEAUX, G.; EGUZKITZA, B. Parallel subdomain coupling for non-matching meshes in alya. 2015. Acessed: 2016-11-3.

CAMELO, G. A.; SCHNORR, L. M. **Alya**. [S.l.]: GitHub, 2016. <http://github.com/guiacamelo/alya/>.

KARYPIS, G.; KUMAR, V. **MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0**. 2009. <http://www.cs.umn.edu/~metis>.

PEARCE, O. et al. Quantifying the effectiveness of load balance algorithms. In: ACM. **Proceedings of the 26th ACM international conference on Supercomputing**. [S.l.], 2012. p. 185–194.

PRACE. **Prace Research Infrastructure Unified European Applications Benchmark Suite - PRACE Research Infrastructure**. 2013. <http://www.prace-ri.eu/ueabs/>. Published: 2013-10-17, Acessed: 2016-07-15.

RODRIGUES, F. A. **Study of Load Distribution Measures for High-performance Applications**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brasil, 2016.

RODRíGUEZ, J. **Performance Analysis of Alya on a Tier-0 Machine using Extrae**. [S.l.], 2014.

SCHNORR, L. M. **Análise de desempenho de programas paralelos**. Porto Alegre, RS, Brasil, 2014. Disponível em: <http://www.inf.ufrgs.br/~schnorr/download/talks/erad2014-minicurso-texto.pdf>.

SCHNORR, L. M.; LEGRAND, A. Visualizing more performance data than what fits on your screen. In: **Tools for High Performance Computing 2012**. [S.l.]: Springer, 2013. p. 149–162.

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. **Int. J. High Perform. Comput. Appl.**, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 20, n. 2, p. 287–311, maio 2006. ISSN 1094-3420. Disponível em: <http://dx.doi.org/10.1177/1094342006064482>.

VÁZQUEZ, M.; HOUZEAUX, G. **The Alya System - Large Scale Computational Mechanics**. 2015. <https://computing.llnl.gov/tutorials/mpi/>. Acessed: 2016-11-3.

VÁZQUEZ, M. et al. Alya: Towards exascale for engineering simulation codes. **arXiv.org**, 2014. Disponível em: <http://arxiv.org/abs/1404.4881>.