Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Bachelor Degree in Informatics Engineering
Computer Engineering Specialization
Degree Final Project

Performance analysis and optimization of a combustion simulation

Final Degree Thesis

Author
Guillem Ramírez Miranda

Director
Marta Garcia Gasulla

Co-director
David Vicente Dorca

Tutor
Julian David Morillo Pozo

19th January 2021

Abstract

This work presents the analysis and optimization of chemical simulation of a combustion process using Alya, a code for high-performance computational mechanics. The work includes the performance analysis of the use case to detect its bottlenecks and possible optimizations, the proposal and implementation of the optimizations and its evaluation.

Resumen

Este trabajo presenta el análisis y optimización de una simulación química de un proceso de combustión usando Alya, una aplicación de mecánicas computacionales de alto rendimiento. Este estudio incluye: Un análisis de rendimiento para detectar los cuellos de botella y las posibles optimizaciones, una propuesta e implementación de una optimización y su evaluación.

Resum

Aquest treball presenta l'anàlisis i optimització d'una simulació química d'un procés de combustió emprant Alya, una aplicació de mecàniques computacionals d'alt rendiment. Aquest estudi inclou: Un anàlisi de rendiment per detectar colls d'ampolla i les possibles optimitzacions, una proposa i implementació d'una optimització i la seva avaluació.

Contents

List of Figures

List of Tables

Acknowledgments

Aquest text l'escriuré en Català perquè no em surt escriure-ho amb una altra llengua.

Primer de tot vull agrair als meus pares tot el suport tant econòmic com emocional i en educació. Sense el seu suport, jo no hauria arribat on sóc ara i aquest treball ni existiria.

Vull agrair a la Marta i al David per donar-me l'oportunitat de treballar i créixer professionalment al Barcelona Supercomputing Center. També agraïments a la Marta (un altre cop) i al Víctor per ajudar-me, guiar-me i estar allà quan els necessitava durant el desenvolupament d'aquest treball. Agrïments al Dani i al seu grup de treball per proposar-nos aquest projecte i ajudar-nos amb els dubtes i inqiuetuts que en tingut.

També vull agrair a l'Ari per acompanyar-me durant aquests tres últims anys de carrera i donar-me suport emocional.

Finalment però no menys important, agraïments pels meus amics de la setmana temàtica. Aina, Joan, Ferran Maria, Andrea, Pol, Pablo, Rafel, Helena i Berni moltes gràcies per fer-me desconnectar i donar-me suport quan més ho necessitava.

Chapter 1

Project management

1.1 Context and scope of the project.

1.1.1 Context

Introduction

Computational science is a growing field that allows researchers to predict the behaviour of systems. For example, a physicist can write a program to predict the trajectory of a sphere instead of calculating it by hand.

What if we bring this further? We can try to optimize the aero-dynamism of a plane to save fuel, the behaviour of the human body cells to a recently designed drug or even given a genome figure out the risk of cancer.

Computing these are at a high cost. It is not conceivable to solve the problems on our laptops as the number of calculations and data required to process is overwhelming. To solve these tasks, we need supercomputers.

Supercomputers are machines with a huge compute power oriented to scientific and technique jobs. These machines are many small machines interconnected. Developers must write their code in a manner that the program is capable of run in parallel.

Writing efficient parallel programs is a difficult task and even more so if the developer's specialization is not computer science. In this work, we will study and improve Alya [1], a real use case application. In particular, we will focus on the module in charge of computing combustion processes.

This work is under an Educational Cooperation Agreement between the Facultat d'Informàtica de Barcelona and the Barcelona Supercomputing Center.

The study has been developed within a Curricular internship in the Barcelona Supercomputing Center precisely in the High-Level Support Team (HLST) inside the Operations department and developed under a collaboration with the Best Practices for Performance and Programmability group. This work is impulsed by the Performance Optimization and Productivity [2] (POP) Center of Excellence under a Performance Analysis request.

Terms and concepts

In the following sections you will find basic concepts to better understand the project.

High-performance computing

High-performance computing (HPC) refers to the act of grouping compute power to do massive and complex computations and data processing. Nowadays HPC is associated

to supercomputers.

Supercomputer

A supercomputer is a machine designed for HPC. The basic structure of a supercomputer consists of multiple computing nodes interconnected. Usually, a node is a shared memory system which can run programs by itself and may have add-ons like accelerators.

Parallel Programming model

Parallel programming models [3] add an abstraction to parallel computer architectures. It defines constructs that allow to operate with the parallel machine performing different actions. For example, sending and receiving messages between processes, reading and writing to the shared memory or spawning tasks in form of threads, all depending on the kind of programming model.

The two main types of parallel programming models used are:

- **Shared memory:** Parallel processes share a global memory address space that they can use for sharing data, synchronization. This type of model is optimal for Multi-Processor systems and cannot be used when connecting multiple machines (i.e., among multiple nodes) as they do not share memory.

- **Message passing:** Parallel processes are independent and they use messages to synchronize and share data. This type of model is ideal to exploit the interconnection network between nodes although it can be used within a single node.

Message Passing Interface (MPI)

The message passing interface [4] is a parallel programming model that defines a standard interface for communications between processes. This allows applications to run in numerous nodes. The processes use the standard interface to send messages to each other using the underlying interconnection network of the machine. This interface allows, among other things, process synchronization, data-sharing via asynchronous and synchronous messages, gathering and scattering data, parallel input-output (I/0).

MareNostrum 4

MareNostrum 4 [5] is a supercomputer managed by the Barcelona Supercomputing Center and managed by the Operations department. It is divided in two blocks, the general purpose block which represents the majority of the computing power and the emergent technologies block that aims to test new technologies. Figure 1.1 shows a picture of the supercomputer located in the "Torre Girona" chapel.

In this work we will be running the application on the general purpose block. The general purpose block consists of 3456 nodes each node containing 2 Intel Xeon 8160 with 24 cores each running at 2.1 GHz. The interconnection network consists of an Intel Omni-Path full-fat tree at 100 Gbps.

Alya

Alya [1] is the application we will be studying. It is code oriented to high-performance computational mechanics that aims to solve engineering coupled problems. In other words, Alya consists of multiple modules that can be joined to solve a specific problem. For this work, we will focalise the analysis into a new Alya module devoted to simulate combustion processes. We will work with a real industry use case in the combustion industry.

Alya supports a wide variety of programming models but the module we are given to analyse only supports MPI. It is contemplated to add support for other programming models if we find out that doing so improves the performance.

Figure 1.1: MareNostrum 4 machine. By Gemmaribasmaspoch, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=61308843

Performance Analysis

Running real use case scientific applications in supercomputer environments is crucial to identify flaws that deplete its performance when running on several machines.

Performance analysis refers to the process of gathering data from application executions, study the data and finally identify bottlenecks of the program.

In the state-of-the-art performance analysis, there is a wide variety of methodologies. However, this work is under the POP Center of Excellence resultantly we chose its methodology [6]. The application of the methodology is tricky, [7] shows the application of the methodology.

BSC tools

As said, for a performance analysis we need data about the execution of programs and tools to visualize the data to get an insight of the bottlenecks.

The main BSC tools we are going to use are:

- **Extrae** [8]: This is the central tool of all the collection of tools. It allows the user to hook up Extrae to the application and gather data during the execution. We can trace events such as the MPI calls, PAPI [9] counters, other programming models events and even manually instrument the code using Extrae API. All the data extracted is dumped into a file we call trace.

- **Basic Analysis** is a set of scripts that using existing BSC tools automatically computes the POP methodology metrics.

- **Paraver** [10] [11] is a powerful desktop application that allows to load up traces and study them. The two main ways to study a trace are using a histogram or a timeline. Histograms allow to study the tendency of some certain counter in the execution. Figure 1.2 shows an example histogram where the *y-axis* shows MPI-processes and the *x-axis* shows different frequency values. The color represents how frequent is that frequency value during the execution, green is not frequent and blue is frequent. Time-lines allow to analyse the behaviour of the code in certain regions. For example, the pattern of MPI calls, the memory access patterns, the executions of tasks on a shared-memory programming model. Figure 1.3 shows

a time-line where *y-axis* show MPI-processes and *x-axis* show time. The color of a process at a certain time show what the process is doing. Blue means the process is computing something useful while black means the process is idle, communicating or in a synchronization. Looking at traces in the Paraver UI it is not the best visual appealing experience. For the sake of comprehensiveness we will use tracedraw [12] package for drawing more visual appealing traces. Figure 1.4 shows the imitation of Figure 1.3 with tracedraw.

Figure 1.2: Sample Paraver Histogram. Own compilation.

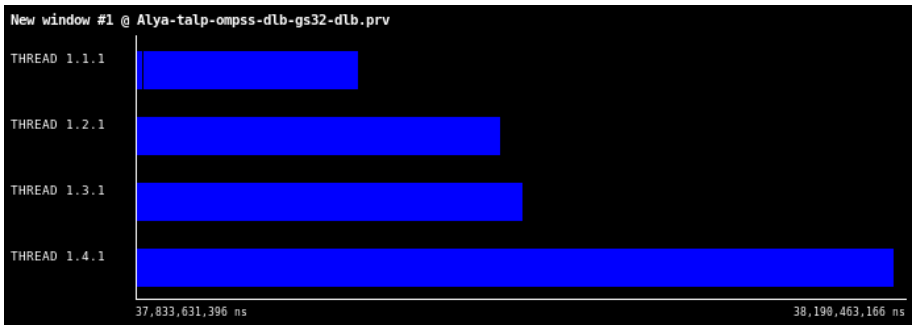


Figure 1.3: Sample Paraver trace. Own compilation

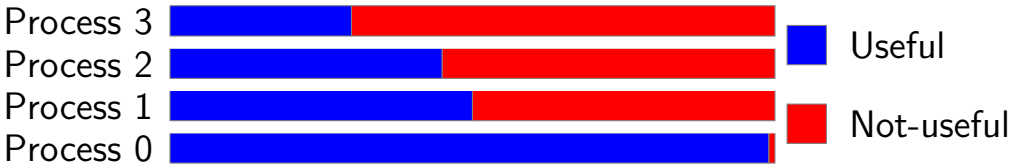


Figure 1.4: Sample trace representation with tracedraw. Own compilation

Problem to be resolved

This project is motivated by the Propulsion Technologies BSC group applying to a POP Performance Analysis assessment.

The problem to solve is that the combustion module they programmed is slow. We are asked to find out why and if possible, apply optimizations and modifications to the code to improve its performance and efficiency. In Section 1.1.4 we will discuss the solutions to the problem by defining our objectives.

Stakeholders

The following groups will benefit from the work.

Propulsion Technologies BSC group

This is the principal stakeholder as it is the developer of the module and the applicant for the analysis.

Scientific community

This beneficiary is also benefited from the work since the results and conclusions of the study are public and open to the scientific community. More in detail, the combustion community will be benefited as the results of the study and the possible optimizations will be shared and used in the newly approved Center of Excellence in Combustion (CoEC)[1].

Society

POP is an European Center of Excellence, this means that the European Union states that the work done under the Center of Excellence (POP) will have a positive impact to the European Society.

In the case of optimizing the combustion module it will impact because combustion researchers will be able to simulate faster which can lead to new discoveries on more energy efficient and pollute less combustions.

Optimizing a code also implies that the application users will use less time the machine which means that they will use less energy for their research.

1.1.2 Justification

For studying Alya as said we will use the POP methodology because:

- The study is a POP service.
- It is a methodology promoted by an European Center of Excellence, meaning it is in the state-of-the-art of performance analysis methods and it is widely use in the academia.
- It is a well-known method and has been in part developed inside BSC.
- It is the methodology I have learnt during my formation period in BSC.

We will use the BSC toolset for extracting data to apply the POP methodology because:

- They are well-integrated with the POP methodology.
- The tools are well-known and developed in BSC. In the situation that we find a bug or we have a problem using the tools, we can directly contact the application developers as they are co-workers.

[1] https://www.bsc.es/news/bsc-news/the-ec-approves-two-new-centres-excellence-high-performance-

- It is the toolset I have learnt during my formation period in BSC and during the computer engineering degree at PAR[2] subject.

1.1.3 Scope

In this section you will find a definition of the objectives of the projects and the identification of possible risks.

1.1.4 Objectives

The project is divided into two main objectives.

- Perform a rigorous Performance Analysis following the POP methodology and identify bottlenecks provided by a real and industry use case in combustion state-of-the-art.
- Knowing the bottlenecks, propose, implement and test optimizations to the code that improve the performance and efficiency of the code..

Other sub-objectives are:

- Learn and gain experience on the performance analysis and optimization field.
- Document and explain Performance Analysis finding to developers and scientists responsible of the code that are not necessary experts in the performance analysis field.

Potential obstacles and risks
The following potential obstacles and risks have been identified:

- **MareNostrum4 not available** A risk we must take into account is that MarenNostrum4 gets stopped, by a maintenance, overheating issues or other errors. If this happens the possibility of moving the study to another similar machine is contemplated.
- **Misinterpretation of a metric during the analysis:** This risk can lead the study to incorrect conclusions. However, by daily feedback with the director of the work and bi-monthly group meetings, we minimize this risk.
- **Incorrect runs:** This is a common risk that occurs when performing different tests of an application forgetting to change a parameter and leading to an incorrect run. These are usually easy to detect as numbers obtained does not match the expected. Still, if the risk is not detected the feedback received during the development should avoid this risk.
- **Machine noise:** Large supercomputer machines use complex environments that can lead to system noise. System noise refers to abnormal events that interrupt the application making counters and timings to have erratic values. Take an appropriate number of samples and doing a correct statistical analysis will avoid this risk.

[2]Parallelism: https://www.fib.upc.edu/en/studies/bachelors-degrees/bachelor-degree-informatics-engineering/curriculum/syllabus/PAR

1.1.5 Methodology and rigour

The methodology relays in constant feedback between Marta Garcia (the project director) and me. As tasks are sequential, once a task is done the results are shared with the director and conclusions and definition of the next task is discussed. The feedback and the discussion are done via email, personal meetings or group meetings.

This work has been developed during the global pandemic of COVID-19 so telematic meetings are contemplated.

Another important aspect to cover is "Version Control" of the project. A git repository has been created for keeping track of the POP deliverables. Another git repository is created for tracking the changes to the code.

For storing the data we will use a Google spreadsheet[3] data sheet with the corresponding information about the runs and the tests performed. However, trace files and run logs are kept inside the MareNostrum 4 shared storage.

An issue page in the internal GitLab[4] of the Best Practices for Performance and Programmability group has been created. In this page I will be posting updates about the work so the members of the team can be updated about the state of the project and we can keep track of it.

[3] Google office suite: https://www.google.com/intl/es/drive/

[4] Gitlab Community Edition: https://about.gitlab.com/install/ce-or-ee/

1.2 Time planning

1.2.1 Description of tasks

Project management

Project management is a compulsory task for any project.

Meetings

Regular meetings are contemplated to keep track of the progress of tasks and to make decisions. It is planned to do approximately 2.5 hour of meetings per week. Time estimated: 45h

Context and scope of the project

Definition of context and scope of the project. Time estimated: 30h

Time Planning

Definition of the tasks and the time planning of the project. Time estimated 10h

Budget and sustainability

Compute the monetary costs of the project. Definition of the sustainability report. Time estimated 10h

Integration and final document

Review, fix and integrate the last 3 tasks into a final document that groups all the information related to the project management. Time estimated 15h

Hands-on

These tasks are intended to get introduced to the application analysed and ensure we are analysing the correct thing.

Contact with application developers

We will meet with the application developers where they will introduce us to the application module. The following points will be worked:

- Basic explanation of the science between the module and the input case.
- Localization of the code and the input case.
- Explanation of the compilation process of the code.
- Explanation of how to run the code.
- Download the code and the input case
- Compilation of the code
- First runs of the code.

Time estimated: 10h

Study the compilation process

It is very important to understand the compilation process and the different options that the build chain allows. This is a simple task but helps a lot in minimizing the possible errors we can encounter adding or modifying the code in the future.

Time estimated: 15h

Run and test the application

Every user in a HPC machine has its own environment. As the code is in development process and I may need to modify it an make my own installations thus some things from the application may differ. It is a critical task to perform some runs and verify with the application developers that we are not having errors, we are running the correct thing and the intended thing to analyse.

Time estimated: 15h

Performance analysis

Trace extraction

Once everything it is sanity-checked, the environment is ready and we are familiarized with the application the next step is to extract traces of the executions. As we will use the POP method we will need traces from executions from 1 process to 48 process and from 1 node to 16 nodes. This process is long and needs to operate with data.

Time estimated: 30h

Modelfactors

Once we have the traces we have to use basic analysis tool in order the extract the POP metrics from the executions. This process require a few hours to process the traces but nothing compared to extracting them.

Time estimated: 10h

Focus of analysis

In this stage of the project we will study the results of the modelfactors. From the results we will obtain insights into what is reducing the performance of the application. With this, we will use the other BSC tools to find the bottlenecks. Once the bottlenecks are found we will need to quantize the real impact on the performance to know which bottleneck is better to optimize. This bottleneck will be called the "Focus of analysis".

Knowing the time estimated is tricky as it depends on the easy is to identify the bottlenecks and the analysis in general, but, based in previous expiriences we estimate a total of 35h

Feedback to application developers

At this point of the project, a presentation and a conclusions of the analysis will be prepared and presented to the application developers.

Estimated time: 15h

Optimization

Design implementation

Based on the results of the previous section, we will decide what we will do to attack the bottleneck. The decision and the design it is an unknown for now. It is expected to be a total of 30h long.

Implementation

This phase is the great unknown of the project since we do not know what we are going to optimize or what we are going to do to optimize it. It is also the main task of the project and the one that will surely take the most hours. However, we cannot know how long it will take, since for now what we are going to do is unknown. Based on previous experiences and the time limit of the project we expect to invest 100h in this phase.

Testing implementation

It is mandatory to test and ensure that the modifications the code still lead to a correct program. A proper testing suite in collaboration with the application developers will be done.

Time estimated: 30h

Evaluation of the improvements

Once the job is done it is very important to evaluate the performance of the application with the improvements. This task consists of gathering traces and timing data from both versions, the one which is improved and the original one and elaborating a final conclusions of how is performing the optimization.

Time estimated: 40h

Final milestone

It is necessary to write the documentation before ending the project. Two sub-tasks are expected:

- Memory redaction. Time estimated: 70h
- Presentation preparation. Preparation of the final presentation of the work, this includes the material and the training for the presentation. Time estimated : 15h

Task dependencies and summary of tasks

Table 1.1 shows a summary of tasks and it's dependencies. Almost each tasks depends on its predecessor making this project really sequential.

Task resources

Human resources

The BSC researcher will be in charge of the project and will be the main responsible of all tasks.

Other human resources are:

- The project director that is responsible of tracking the status of the project and giving feedback and suggestions to the researcher.
- The application developers are also human resources needed as they are in charge of introducing the researcher to the program (T1) and giving support if any problem with the application is encountered.

Material resources

- Dell Latitude 7490. Laptop that the project author will use for all the tasks.
- MareNostrum 4 supercomputer used for tasks T2, T3 and T4.
- Control versioning server used for keeping track of the changes to the code and the documentation. Tasks T1, T3.7, T4 and T6
- A Mailserver used for communication with the project director and the application developers.

Id	Name	Time (h)	Dependencies
T1	Project managment	110	
T1.1	Meetings	45	
T1.2	Context and scope of the project	30	T1.1
T1.3	Time planning	10	T1.2
T1.4	Budget and sustainability	10	T1.3
T1.5	Integration and final document	15	T1.4
T2	Hands-on	40	
T2.1	Contact with application developers	10	
T2.2	Study the compilation process	15	T2.1
T2.3	Run and test the application	15	T2.2
T3	Performance analysis	90	
T3.4	Trace extraction	30	T2.3
T3.5	Modelfactors	10	T3.4
T3.6	Focus of analysis	35	T3.5
T3.7	Feedback to application developers	15	T3.6
T4	Optimization	200	
T4.1	Design implementation	30	T3.6
T4.2	Implementation	100	T4.1
T4.3	Testing implementation	30	T4.2
T4.4	Evaluation of the improvements	40	T4.3
T5	Final milestone	85	
T5.1	Memory redaction	70	T4.3
T5.2	Presentation prepration	15	T5.1
Total			525

Table 1.1: Summary of tasks. Own compilation.

- LaTeXused for writing documentation. Tasks T1, T3.7, T4 and T6.
- Ganttproject for the Gantt diagram tracking. Task T1.3.
- A text editor for writing the code, the documentation and in general manipulating files. All the tasks involved.
- Office supplies.

1.2.2 Gantt

The start of the project is expected by the 13/9/2020 which is the approximate start date of the thesis management course. The project is intended to end before the 29 of January that is the date of defence.

Figure 1.5 shows the gantt diagram of the project.

1.2.3 Risk management: alternative plans and obstacles

- **MareNostrum4 not available**: The unavailability of the machine can lead to delays in almost all tasks. In general when the machine becomes unavailable in 1 day the issue is solved. We can expect a delay of maximum a 5% of the expected time. If a catastrophic event happens to the machine and it becomes unavailable during a notable period of time it is contemplated to move the study to a similar machine that BSC disposes of. This implies an increment of 20h to move the project and adapt to the new environment. Tasks affected: T.2, T.3 and T.4.
- **Misinterpretation of a metric during the analysis**: This can lead to small delays of maximum 5h. From previous experiences we can expect to not affect the project timing and this risk will be lesser as the researcher is more experienced. Tasks affected: T.3
- **Incorrect runs**: This risk affects the project for maximum 2h as it is fast to detect and fast to fix. Tasks affected: T.3 and T.4
- **Machine noise**: As this risk is contemplated and it is solved by gathering the proper number of samples it is already contemplated on the duration of the tasks and won't affect the timing of the project. Tasks affected: T.3 and T.4.

1.3 Budget and sustainability

1.3.1 Budget

Staff

The following roles are considered for developing the tasks described in the Gantt diagram.

- Junior researcher.**Cost per hour:** 10 €/ h
- Project manager. **Cost per hour:** 20.24 €/ h

Table 1.2 shows given the tasks and the time to complete defined in the Gantt the hours each role focus on and the cost.

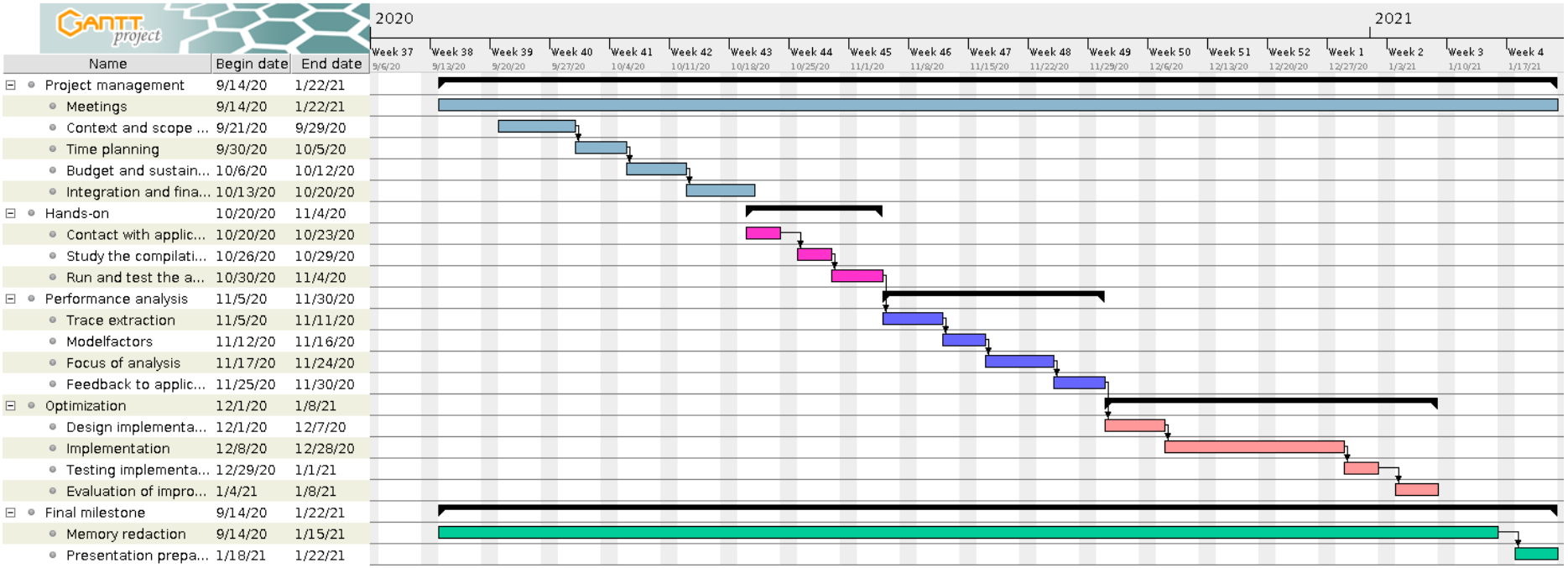


Figure 1.5: Gantt diagram. Own compilation.

Task	Junior Researcher	Project manager
Project managment	20 h	90 h
Hands-on	30 h	10 h
Performance analysis	80 h	10 h
Optimization	150 h	50 h
Final Milestone	65 h	20 h
Total (€)	3450	3643
Total aggregated (€)	7093	

Table 1.2: Cost estimate for each role in the project. Own compilation

Material resources

The estimate cost of the material resources are detailed in table 1.3. Mind that:

- Cost and amortization of MareNostrum4 machine is not included as some of the data needed to calculate that must remain confidential.
- Some of the software used in the machine it is not free, neither as in freedom and as in beer. This cost is not included as it is also confidential.
- The office costs are illustrative as the project is developed in the K2M building in Campus Nord.
- Lifetimes are calculated by considering a 8 hours per day use during 5 years for electronic devices and 10 years for the furniture. Usage time may differ from real usage due to COVID-19 global pandemic.

Resource	Cost (€)	Lifetime (h)	Usage (h)	Amortization (€)
Dell latitude 7490	2500	9600	525	136.72
Generic display	300	9600	525	16.4
Generic keyboard	20	9600	525	1.09
Generic mouse	10	9600	525	0.54
Office furniture	1000	19200	525	27.34
Total	3830	-	-	182.03

Table 1.3: Amortization estimate for material resources. Own compilation

Table 1.4 shows the fixed monthly expenses. A final calculation on the part of the costs that are related to the project is added in a column. For example, if is a cost related to all the office users is divided by the number of the workers in the office.

Resource	Monthly cost (€)	Total cost (€)	Project cost (€)
Office rent	5000	25000	250
Office supplies	100	500	5
Watter and electricity	1000	5000	50
Total	6100	30500	305

Table 1.4: Fixed costs and total for project. Own compilation

Contingencies

It is estimated a 15% of the total of the amortizations and the staff costs for any contingency we could encounter during the project development.

$$Cont = (Amort + Staff) * 0.15 = (7093 + 182.03) * 0.15 = 1091.2545\ e$$

Incidentals

In the previous chapter we commented the possible incidentals. Table 1.5 shows how the incidentals affect the budget.

Incident	Extra staff time (h)	Material costs (€)	Total (€)
MareNostrum 4 unavailable	16.5	0	165
Machine change	20	0	200
Incorrect runs	2	0	20
Total	28.5	0	385

Table 1.5: Cost prediction of incidentals during the project. Own compilation.

We don't contemplate possible material costs associated to the MareNostrum 4 unavailable as it is the BSC the responsible of the machine maintenance and we are just users.

Total

Table 1.6 shows the grouped predicted budget of the whole project. Notice that from this cost we have omitted software licenses and compute hours costs which are confidential.

Concept	Cost (€
Staff	7093
Amortizations	182.03
Fixed costs	305
Contingencies	1091.25
Incidentals	385
Total	9056.28

Table 1.6: Total costs of the project. Own compilation

1.3.2 Sustainability report

Self-assessment of the current domain of sustainability competition

The sustainability is a field where I was not conscious to think about but indeed I did.

I'm always thinking on the social and economical aspects of my projects as I'm interested on this fields. A common question I use to ask myself is how this project will affect the society? This is obviously thinking about the social and the economical aspects. Non-ethical technological products are becoming a tendency, changing people vote intention, governments tracking its citizens are examples of this and examples of developers that not thought about the social and economical aspects of their projects.

I have been ignoring or keeping apart thinking on the environmental aspects of my projects. This is usually because my projects don't use to impact on this aspect but now that I think that is the most important topic to think about it. At least on High-performance computing field that aims to solve or to find faster the solution to the problem.

I have also the concern to use and promote the free (as in freedom) software and hardware to avoid companies to be able to modify lifetime of products and therefore, generate more e-waste.

Economic Dimension

Reflection on the cost estimated

The cost estimated it is not precise as some parts are omitted for confidentiality. The estimations I made try to be realistic with the current price standard in Barcelona, Spain but I'm almost sure I miss predicted some estimations as I'm not experienced on this also not in charge of making the expenses.

Useful Life My project aims to improve the computation time of a simulation. Hopefully if we improve the simulation the computation times will be lesser, therefore, scientists will have to invert less money on computational resources or will reach faster to conclusion which leads to scientific progress.

Environmental Dimension

Environmental impact of the project development

The environmental impact estimation has not been done. It is a very complex calculation as it involves the MareNostrum4 fabrication and electrical consumption and it is used by a lot of users. Scoping this in the project means making a lot of estimations which is not possible.

Minimize environmental impact

This is not possible in the project. The machine is already built.

The only way we can act is in being conscious the electricity that consumes the machine and the laptop and try to use efficient the resources. This implies performing the appropriate number of runs.

Environmental impact of the project

The project will have a positive environmental impact. This research will be useful in order to make programs more efficient and reduce the running time of the applications which implies less power consumption.

Social Dimension

Personal growth

This project will make me grow on my young research career and will make me improve me technical and social skills. Contacting and explaining the project to the application developers, analysing, thinking and developing an optimal solution, discussing with my project manager all the aspects of the project, writing technical reports are things that will make me grow as a professional and as person.

Social improvement

This project aims to accelerate the combustion research as its objective is to make the scientific simulation to go faster. This can impact the society in many ways, from

making less pollute motors to making faster planes. We cannot predict what research we will accelerate.

Is there a real need for the project?

There is a need to improve application performance in order to use the resources efficiently. Enabling applications to scale is a strategic key to scientific progress.

Chapter 2

Background

This chapter introduces general concepts needed to understand the work. We will take a look at:

- High Performance Computing (HPC) environment basics.
- Alya and the combustion use case.
- Application analysis methodology.

At first, we will understand how an HPC environment works, and we will contextualize it within the BSC MareNostrum4 infrastructure, which is the target environment for the application. Then we introduce Alya, the HPC code we work with, and the use case we analyze and optimize in this project. Finally, we will discuss the application analysis methodology giving detail about the steps and the concepts behind.

2.1 HPC enviornment

For completeness concerning the work, I will explain the different concepts of an HPC environment, contextualized in the BSC Marenostrum4 infrastructure[13].

A supercomputer, also known as a cluster, is a set of machines interconnected, and usually coordinated to solve a problem. The machines are named nodes, and each has memory, CPU, storage, and is running an operating system. On the Marenostrum4 platform, each node consists of 2 physical Intel Xeon Platinum 8160 with 24 cores each and 96 GB of DDR4 RAM. There are also particular nodes with more memory available which are the same but with 384 GB of DDR4 RAM.

Each node can send and receive data from the other nodes from the system. Thus there is an interconnection network underlying. Networks are difficult to design as we expect low latency and high bandwidth on communication up to among 150.000 processes concurrently. The MareNostrum4 platform uses Intel Omni-Path technology for interconnection. It has six routers that are responsible for handling messages from a group of nodes, precisely one-sixth of them. Intel Omni-Path allows speeds up to 100Gbps.

To send data, applications, use the Message Passing Interface (MPI) standard. MPI allow programmers to divide the problem among the nodes in the machine and do communications and synchronizations via the interconnection network. HPC industry widely uses MPI, and therefore there are a lot of implementations. As we focus on the MareNostrum4 platform which integrates Intel technologies, we will use Intel MPI implementation and Intel compilers.

We have commented out that we will use different software and implementations. Marenostrum4 uses modules for managing the diverse software and implementations that are installed. Modules provide the users with an easy way to select which software they want to use, managing the Linux environment variables. Listing 2.1 shows an example of how to load the Intel MPI implementation and the Intel compiler suite version 2017.4 and then change to the version 2018.4.

```
1 module load intel/2017.4 impi/2017.4
2 module unload intel/2017.4 impi/2017.4
3 module load intel/2018.4 impi/2018.4
```

Listing 2.1: Example of module usage.

Another essential element of an HPC environment is the Resource Manager. The Resource Manager establishes the access policies to the resources offered by the HPC infrastructure. In the context of the MareNostrum4 infrastructure, the resource manager used is Slurm[14]. Slurm has a queue system where the users send jobs, wait for their jobs to pass through the queue, and finally, Slurm allocates the resources demanded and runs the job. The priority a user has on the queue varies on the number of resources required, the time demanded with the resources and other factors. Listing 2.2 shows how to allocate a node for ten minutes, query the job identification of the job, and cancel the job.

```
1 salloc -N 1 --time=10:00
2 squeue #look for the JOBID
3 scancel JOBID
```

Listing 2.2: Example of slurm usage.

For a more detailed description of the commands and different options visit the Slurm documentation [15].

2.2 Alya and the combustion use case

Alya is an HPC application for computational mechanics. It has diverse modules that solve a specific problem, which can be coupled to solve a general problem. For example, we can couple together the module to simulate how the temperature evolves in a system and the module that computes the particle motion in the same run.

The code is written in Fortran90 and can use MPI, OpenMP/OmpSs and GPU. The module we are asked to optimize is pure MPI and despite the code is written in Fortran, it uses Cantera library for a critical computation inside the module. Cantera[16] is an open-source suite of tools for problems involving chemical kinetics, thermodynamics, and transport processes written in C++.

The use case we will study consists of a bunsen flame in 2D. It is a simple use case although representative of the industrial real use cases of the application. Figure 2.1 a real picture of a bunsen flame.

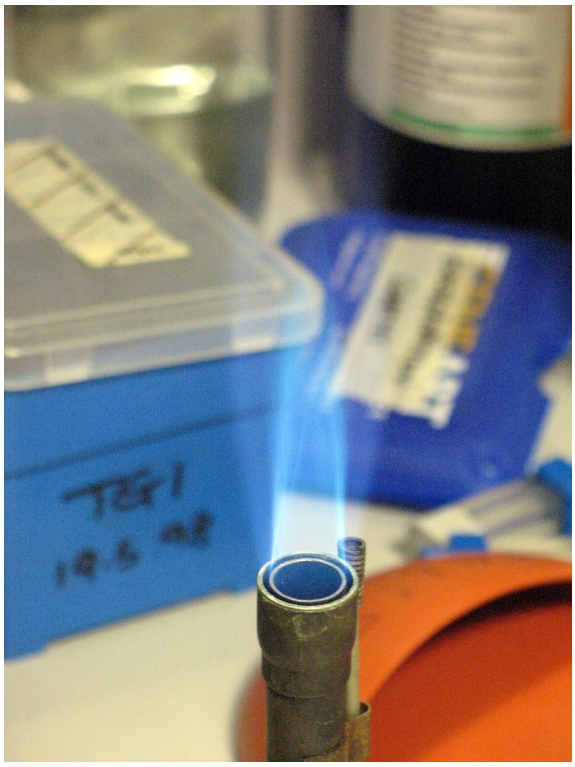


Figure 2.1: Bunsen flame by El Bingle licensed under CC BY-NC-SA 2.0

In general terms the simulation consists of:

- The centre of the 2D domain contains the bunsen flame.
- We compute environmental parameters such as temperature and pressure produced during the combustion for all the grid points.
- Following transport equations defined, compute the motion of chemical species, and if the environmental conditions are ideal, compute the chemical reactions.

Developers of the application gave us two versions of the bunsen 2D flame:

- **Detailed:** It has a significant number of transport equations, which means that the case is industry-relevant and the chemical part of the simulation, will be the most relevant part of the code.
- **Reduced:** Has a short number of transport equations, meaning the computational relevance of the chemical part of the code is low, and the case is less industry-relevant but interesting to compare to the Detailed version.

2.3 Analysis

As said in Section 3, we will follow the Performance Optimization and Productivity[6] methodology. The POP Efficiency metrics are the base of the methodology. First, we usually run the application many times in a *strong scaling* manner. Strong scaling means leaving the problem's size constant and increasing the resources, for example, running the same Alya input set with 1, 2, 4 and Nodes. Then we gather necessary data for computing the efficiency metrics from the previous runs. Once we have the metrics, we possess insight into the factors limiting the code's performance when scaling in the number of resources. From this insight, we can locate the zones in the code responsible for it and try to propose a better implementation.

2.3.1 Efficiency metrics

Pop efficiency metrics try to describe the sources of inefficiency. Figure 1 shows the representation of the metrics. The main feature of the metrics is that they are hierarchic. The hierarchy follows a multiplicative scheme. Multiplicative means that the multiplication of the children of a metric computes its value.

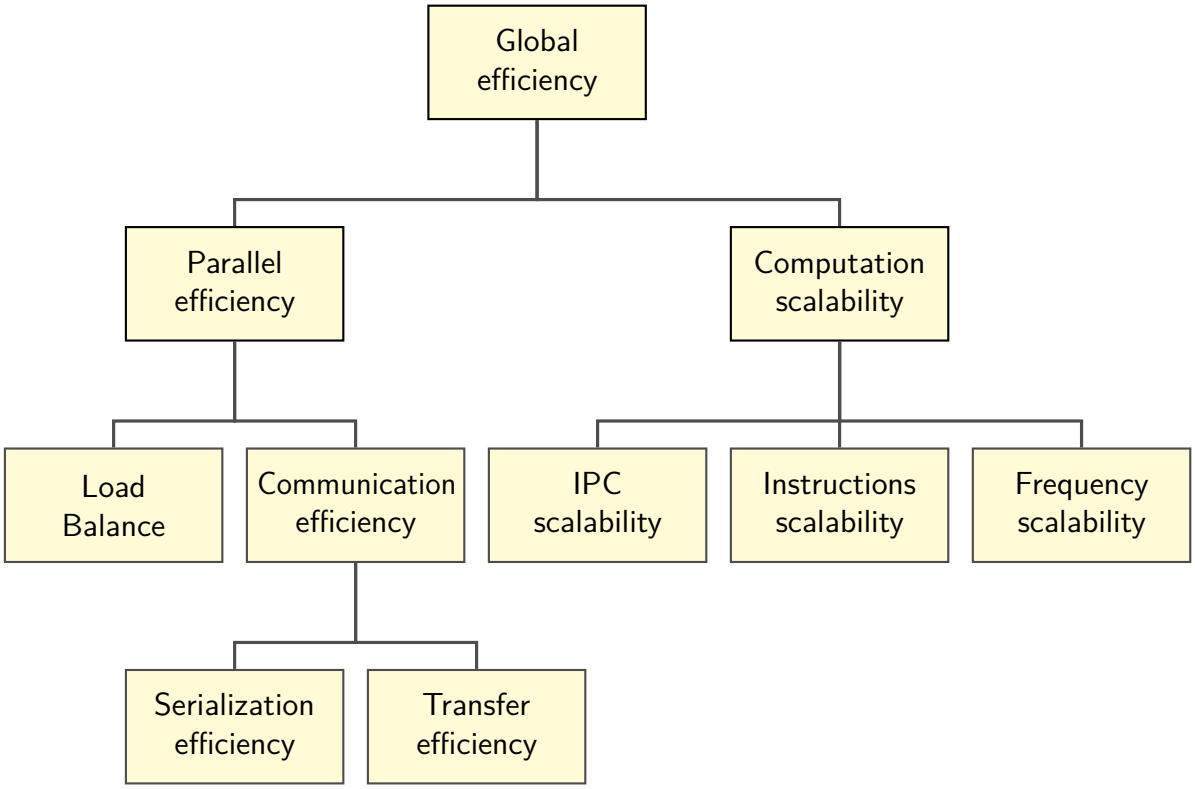


Figure 2.2: POP Efficiency metrics. Own compilation.

The global efficiency describes how good is the application performance in general. Then, the global efficiency is represented by:

- **Parallel efficiency:** Shows the efficiency lost due to splitting the work between processes and the effects of adding communications.
- **Computation scalability:** Represents the efficiency lost in useful computation. When we study a parallel application we look for computation side effects of

the parallelization. For example, dividing the data among processes adds more instructions, running with more processes in the same machine can lower the IPC because of the resource sharing.

Parallel efficiency

Before entering in-depth on the metrics, we need to explain a few assumptions. The efficiency metrics estimate that each process can be in two states:

- Useful state means that the process is performing computation. We usually represent a process in useful with the colour blue.
- Not useful state means that the process does not perform computation, such as sending an MPI message or waiting in a synchronization. We usually represent a process that is not useful with the colour red.

Figure 2.3 shows the graphical representation in a timeline of processes and their states. The x-axis represents the time, y-axis processes, and the colour of a process means the state that the process is in that time. The Figure shows four processes following the pattern: All computing, processes 3, 2 and 1 finish early and wait for process 0, then process 0 finishes and re-establish the computation.

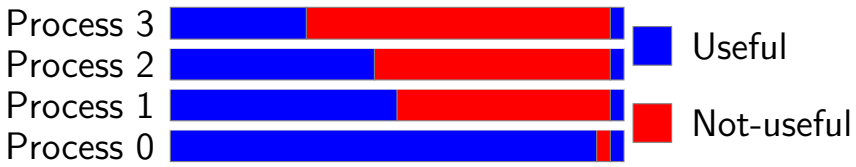


Figure 2.3: Example of useful and not useful states. Own compilation

We define E as the total execution time, $P = \{p_1, p_2, \ldots, p_n\}$ as the set of processes and for each process the set of time intervals where the process is in useful $U_p = \{u_1^p, u_2^p, \ldots, u_{|U|}^p\}$ and the same set but when the process is in not-useful $\overline{U}_p$. Equation 2.1 shows the definition of T_{U_p} which is the total time the process is in useful. Graphically expressed the sum of the blue chunks. The time the process is in not-useful $T_{\overline{U}_p}$ is defined similarly.

$$T_{U_p} = \sum_{U_p} \blacksquare = \sum_{i=1}^{|U_p|} u_i^p \tag{2.1}$$

The parallel efficiency (PE) represents the time lost due to the parallelization. Then we can express it as the factor of the total time in useful by the total CPU time consumed. As we said, as the metrics are multiplicative it can also be computed with the product of the Load Balance (LB) and the Communication Efficiency (CE). Equation 2.2 shows the formalization.

$$PE = \frac{\sum_{i=1}^{|P|} T_{U_i}}{E * |P|} = LB * CE \tag{2.2}$$

The Load Balance measures the efficiency lost due to improper work distribution. Expressed mathematically the factor between the average time in useful by the maximum time in useful. Equation 2.3 shows the formalization.

$$LB = \frac{\sum_{i=1}^{|P|} T_{U_i}}{|P| * max_{i=1}^{|P|} T_{U_i}} \tag{2.3}$$

The Communication Efficiency measures efficiency lost due to spending time in not-useful state, namely, communications. Equation 2.4 defines the CE. Expressed in words, is the maximum time in useful, divided by the total runtime. Also the Communication Efficiency is the multiplication of it's children, the Serialization Efficiency (SE) and Transfer Efficiency (TE).

$$CE = \frac{max_{i=1}^{|P|} T_{U_i}}{E} = TE * SE \tag{2.4}$$

As we said, the metrics are multiplicative. Therefore we can check that $PE = LB * CE$. Equation 2.5 shows the check, we can see that the expression of efficiency lost by distributing improperly the work by the time lost in communications, expresses the Parallel Efficiency.

$$LB * CE = \frac{\sum_{i=1}^{|P|} T_{U_i}}{|P| * max_{i=1}^{|P|} T_{U_i}} * \frac{max_{i=1}^{|P|} T_{U_i}}{E} = \frac{\sum_{i=1}^{|P|} T_{U_i}}{E * |P|} = PE \tag{2.5}$$

As said, the Communication Efficiency can be expressed in terms of the Transfer Efficiency and the Serialization Efficiency.

The Serialization Efficiency measures time lost in communications where there is not data to send or receive. Therefore, for calculating the metric we need to simulate an ideal network so no time because of transferring data is taken into account. Then the Serialization efficiency is the maximum computation time on an ideal network divided by the total runtime on an ideal network. Equation 2.6 shows the expression where the suffix ideal means the time is on an ideal network.

$$SE = \frac{max_{i=1}^{|P|} (T_{Uideal_i})}{E_{ideal}} \tag{2.6}$$

The Transfer Efficiency measure time lost in not-useful due to data transfer. This can be computed with the factor of the runtime on ideal network by the runtime. Equation 2.7 shows the mathematical expression.

$$TE = \frac{E_{ideal}}{E} \tag{2.7}$$

As done with the parallel efficiency, we can do the same check with the communication efficiency. Equation 2.8 shows the check. We can see that the communication efficiency is represented by time lost due to the network not being ideal, which may seem it is not a thing we can not improve, but packing communications can solve it. And also time lost in not-useful without sending/receiving data.

$$SE * TE = \frac{max_{i=1}^{|P|} (T_{Uideal_i})}{E_{ideal}} * \frac{E_{ideal}}{E} = \frac{max_{i=1}^{|P|} T_{U_i}}{E} = CE \tag{2.8}$$

Computation scalability

In the previous Section 2.3.1 we have been discussing about efficiency. In this section we present the Computation Scalability, which this metric and its children refer to scalability, not efficiency.

This difference is important because the numbers depend on the baseline case. The Computation Scalability in strong scaling is defined as the ratio of the sum of all the useful computation time by the number of processes divided by the baseline ratio.

The factors that affect directly the computation scalability are:

- **Instruction scalability:** Measures if the number of instructions scales with the number of processors. For example in strong scaling, if we double the number of processors, the ideal number of instructions executed should be the same. More instructions could mean that we are repeating work, or adding overhead from the parallelization.
- **IPC scalability:** Measures how the IPC evolves from the baseline, when we increase the number of processors on a run we can expect that the IPC is lower since the resources are shared with more processors.
- **Frequency scalability:** Measures how the frequency evolves from the baseline, frequency of the processors should remain stable but pre-emptions and other side effects from running the application can affect the frequency and lower the performance.

Equation 2.9 shows the execution time formula. We can observe that the execution time strictly depends on the computation scalability children.

$$T_{exe} = \frac{N_{ins}}{IPC * f} \tag{2.9}$$

2.3.2 Computing the efficiency metrics.

Now that we understand the efficiency metrics that explain the performance of the code, we need to know how to apply it in practice. All the data we need for computing the efficiency metrics is:

- Elapsed time.
- Maximum time in useful.
- Total time in useful.
- Elapsed time running in an ideal network.
- Average IPC in useful.
- Number of instructions executed in useful.
- Average frequency in useful.

Fortunately the Barcelona Supercomputing Center has a set of tools that allow us to gather the data, automatize the extraction and calculating the Efficiency Metrics. The process consists of:

1. Run the application with different number of processes with Extrae enabled.

2. Select a focus of analysis, this consists of the relevant part of the code that we want to study. Usually the program initialization and finalization is not taken into account, and if the application uses an iterative pattern, we usually choose to focus on a set of iterations that represents the average iteration to study.

3. Once we have the focus of analysis and the traces, we have to chop the traces so they just have the data of the focus of analysis.

4. Finally we pass the chopped traces to the BasicAnalysis BSC tool which automatically extracts the data (including simulating the run with an ideal network using Dimemas) and computes the metrics.

The next sections bellow include a basic background on how to use the BSC tools for extracting the data for the Efficiency Metrics.

Extrae

Extrae[17] is a set of tools concerned to generate traces of executions for post-mortem analysis. There exist multiple ways for Extrae to intercept the program execution and get information. As we will be working with MPI we will use the `LD_PRELOAD` method with intercepts the MPI calls and dumps the information.

For using this method we need:

- A valid `extrae.xml` file including the different tracing options we want.
- Choose the correct dynamic library to preload according to the programming model we are using. In our use-case we will use the pure MPI dynamic library.

Listing 2.3 shows the integration of extrae with SLURM running an application `"BINARY"`. We add the call to `mpi2prv` tool because we disable automatic merging in the `extrae.xml` configuration file because we detected sometimes the tracing is stalled with the option enabled.

```
1  #!/bin/bash
2  #SBATCH --job-name=alya-trace
3  #SBATCH --output=alya.out
4  #SBATCH --ntasks=48
5  #SBATCH --cpus-per-task=1
6  #SBATCH --time=1:20:00
7  #SBATCH --exclusive
8
9  export EXTRAE_HOME=#path to extrae installation
10 export EXTRAE_CONFIG_FILE=#path to extrae.xml
11
12 EXEC="env LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitracef.so BINARY"
13 mpirun -np ${SLURM_NTASKS} ${EXEC}
14 $EXTRAE_HOME/bin/mpi2prv -f TRACE.mpits -o Alya-trace.prv -no-keep-
       mpits
```

Listing 2.3: Running a binary with SLURM and Extrae.

Paraver

Paraver[10] is a visual inspection tool for analysis of parallel applications. The data the application needs is a trace in format `.prv` which are Extrae traces. This tool let us visually inspect the execution of applications. Therefore we can visually detect the pattern of the application, usually and in our use-case iterative and chop the trace having in the focus of analysis.

Dimemas

Dimemas[18] is a command-line tool that given an input trace of an execution allows to remake the trace changing some conditions, for example, simulating an ideal network or changing some of the computational resources conditions. We will not enter in depth on how to use Dimemas as it is automatically called by BasicAnalysis (Section 2.3.2).

BasicAnalysis

BasicAnalysis[19] is a suite of tools for, as the name says, performing a basic analysis. We will focus just on the `modelfactors.py` tool.

The `modelfactors.py` tool automatically extracts and computes all the data needed for computing the POP metrics. We just need our set of chopped traces for a set of processes P. Listing 2.4 shows how to run the tool inside a job script.

```
1 #!/bin/bash
2 #SBATCH --job-name=modelfactors
3 #SBATCH --output=modelfactors.out
4 #SBATCH --ntasks=1
5 #SBATCH --cpus-per-task=48
6 #SBATCH --time=1:20:00
7 #SBATCH --exclusive
8
9 modelfactors.py trace-48.chop1.prv trace-96.chop1.prv ...
```

Listing 2.4: Running modelfactors with SLURM.

Running it will output a table with the metrics computed and will output the following files:

- **modelfactors.csv**: The POP metrics in a csv format.
- **modelfactors.gp**: A gnuplot[1] script file that projects the efficiencies to a high number of processes. Figure 2.4 shows an example plot given by modelfactors tool.

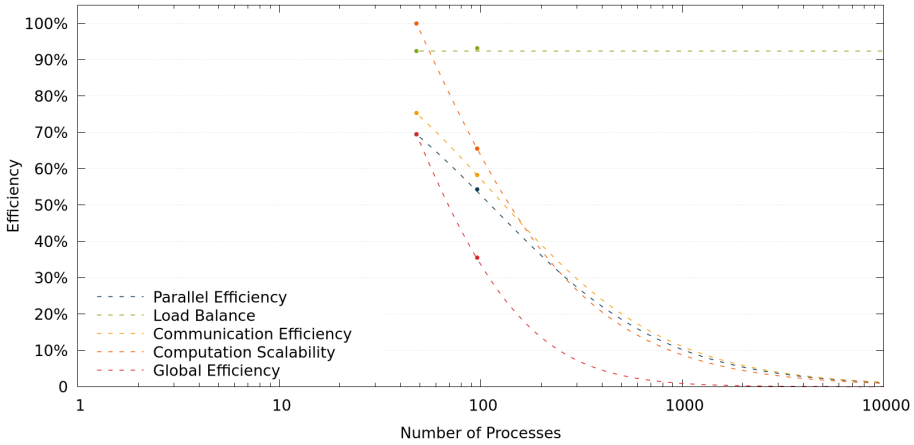


Figure 2.4: Modelfactors projected efficiencies. Own compilation.

[1]gnuplot site: http://www.gnuplot.info/

Chapter 3

Performance Analysis

This chapter introduces the performance analysis of the Alya combustion case. We follow the methodlogy explained at the Section 2.3.

- Experimental set-up.
- Identification of the code structure and the focus of analysis.
- Efficiency metrics.
- Conclusions of the study.

3.1 Experimental set-up

This section explains the experimental set-up for the study. We present the compilation process, the job scripts to run and trace the application and finally the automation scripts for running and gathering data from multiple runs.

3.1.1 Compilation

Alya is build using GNU/Make[20]. The Makefile necessary to compile the application is generated by a configure script written in Perl that parses the options and flags in the *config.in* file. Table 3.1 shows a summary of options toggled in the configuration. Notice that: we set the compiler to *mpif90*, that is, the MPI compiler that wraps the Intel compiler, we link to the Intel Math Kernel libraries[21], we explicitly tell the compiler that we want to use the Intel AVX-512 vectorial unit, to perform optimizations based on the architecture and to try to perform optimizations among all the files of the code. In summary, we tried to achieve maximum performance from just tweaking compilation parameters.

Flag	**Description**	**Value**
F77	Fortran 77 compiler	mpif90
F90	Fortran 90 compiler	mpif90
EXTRALIB	Libraries linked	-lmkl_core -lmkl_sequential
FOPT	Optimization level	-O3
OPTFLAGS	Optimization flags	-xCORE-AVX512 -mtune=skylake -ipo

Table 3.1: Configuration flags for Intel MPI Alya compilation. Own compilation.

Once we have the *config.in* file ready, we need to proceed to the actual compilation. Listing 3.1 shows the simplified build script. First, we load the Intel modules which enable the compilation using the Intel toolchain required. Then, we run the configure script, that will parse the *config.in the* file we previously set and generate the Makefile. Finally using the generated Makefile we:

- Compile the *Metis4* partitioner.
- Compile the Cantera wrapper for Alya.
- Compile Alya telling the Makefile we enabled Cantera.

```
1 module load intel mkl impi
2 ./configure # -x nastin chemic parall temper
3 make metis4
4 make cantera4alya
5 make CANTERA=1
```

Listing 3.1: Building Alya

With just that we got our Alya binary!

3.1.2 Running set-up

For running an HPC application, we need the binary and the input set. We know how to get the binary, and we are given the input set, meaning, we are ready for running the application.

An Alya input set is a folder with different files. Each file describes the mechanisms and parameters each module uses and defines the domain. All the input set files begin with the same name, and this is the case name. Listing 3.2 shows the command to launch an Alya run assuming the binary is copied to the input set folder, and we are located inside that folder.

```
1 mpirun -np 48 ./Alya.x CASE_NAME
```

Listing 3.2: Running Alya with 48 processes.

Since we are running in a production HPC environment, we have to use the resource manager to run isolated and with the requested resources. Listing 3.3 shows how to run Alya using a Slurm script, requesting one node. Remember each MareNostrum4 node has 48 CPUs, and we are running with pure MPI. Therefore we set the *cpus-per-task* flag to one and the *ntasks* to 48. Other flags present are the output flag that tells the job manager the file to write the script's output, the time flag which we must provide to make sure that the job gets killed if it gets stalled and the exclusive flag tells Slurm that we do not want to share the resources.

```
1 #!/bin/bash
2 #SBATCH --job-name=alya-test
3 #SBATCH --output=alya.out
4 #SBATCH --ntasks=48
5 #SBATCH --cpus-per-task=1
6 #SBATCH --time=1:20:00
7 #SBATCH --exclusive
8 mpirun -np ${SLURM_NTASKS} ./Alya.x CASE_NAME
```

Listing 3.3: Running Alya with SLURM.

3.1.3 Timing set-up

For studying how the application performs apart from getting traces, we need to gather the timing data from the application. The file `f1d.chm.cvg` inside the input set folder owns the timing data. The file format is a tab-separated table where the first column is the number of iteration, and the eighth field is the elapsed CPU time. The file contains comments beginning with "#" character. Listing 3.4 shows the script named `gather.sh` that parses the file and prints the average iteration time and the total iteration time to a file that stores all the results. First we define the output file which should be external to the input case and shared among executions so we store the data from all executions. We set variables `tot` and `n` to 0 which will store the total time and the number of iterations parsed respectively. For each line in the file we:

1. Check if it is a comment.
2. Parse the number of iteration.
3. If it is the first iteration, we skip it as it is longer therefore not representative for the study.

4. If the number of iteration is not a number, we also skip it.

5. Get the eighth field of the line and parse it to the same number format.

6. Add to the total time counter the timing and increment the number of iterations.

Once we computed `tot` and `n` we compute the total time `TOT_IT` and the average iteration time `AVG_IT`. Finally, we print the results to a file. We assume that the MPI variable is the first argument passed to the script. Notice that we need to lock the file because different processes may be writing concurrently.

```
1  file="../results-temp.csv"
2  tot=0
3  n=0
4  while read line; do
5          if [[ $line == \#* ]]; then
6                  continue
7          fi
8
9          nit=‘echo $line | awk ’{print $1}’‘
10
11         if [[ "$nit" == 1 ]]; then
12                 continue
13         fi
14
15         if ! [[ "$nit" =~ ^[0-9]+$ ]]
16         then
17                 continue
18         fi
19
20         time=‘echo ${line} | awk ’{print $8}’ | sed -E ’s
               /([+-]?[0-9.]+)[eE]\+?(-?)([0-9]+)/(\1*10^\2\3)/g’‘
21         tot=‘echo "${tot}+${time}" | bc -l 2> /dev/null ‘
22         n=‘echo $n+1 | bc -l‘
23 done < f1d.chm.cvg
24
25 MPI=$1
26 lockfile -r -1 $file.lock
27 echo -e "${MPI} ${AVG_IT} ${TOT_IT}" >> $file
28 rm -rf $file.lock
```

Listing 3.4: gather.sh script

3.1.4 Tracing set-up

In Section 2.3.2 we presented a basic example of running extrae. Listing 3.5 shows the integration of Listing 3.3 with extrae. Notice that the structure is the same than the extrae example substituting `"BINARY"` with the Alya execution line (Alya.x f1d)

```
#!/bin/bash
#SBATCH --job-name=alya-trace
#SBATCH --output=alya.out
#SBATCH --ntasks=48
#SBATCH --cpus-per-task=1
#SBATCH --time=1:20:00
#SBATCH --exclusive

export EXTRAE_HOME=#path to extrae installation
export EXTRAE_CONFIG_FILE=#path to extrae.xml

EXEC="env LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitracef.so Alya.x f1d"
mpirun -np ${SLURM_NTASKS} ${EXEC}
```

Listing 3.5: Running Alya with extrae and SLURM

3.1.5 Wrap-up and automation

This Section presents the scripts developed in order to gather all the data.

First, we developed a parametrized script that enables us to run with different options without manually modifying the job script. In this case, we decided to leave placeholders in the job script and then set them with `GNU/sed`[1]. Listing 3.6 shows the implementation, the convention for the placeholders is `%%PARAMETER_NAME%%`. We set the trace location outside the folder where we execute, so all traces are stored in an independent folder. If we are not tracing, we call the `gather.sh` script (Listing 3.4) to store the timing data.

```
#!/bin/bash
#SBATCH --job-name=alya-%%MPI%%-%%TRACE%%
#SBATCH --output=alya.out
#SBATCH --ntasks=%%MPI%%
#SBATCH --cpus-per-task=1
#SBATCH --time=1:20:00
#SBATCH --exclusive

TRACE=%%TRACE%%

if [[ ${TRACE}==1 ]]; then
    export EXTRAE_HOME=#path to extrae installation
    export EXTRAE_CONFIG_FILE=#path to extrae.xml
    TRACENAME="../traces/Alya-${SLURM_NTASKS}.prv"
    EXEC="env LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitracef.so Alya.x
        f1d"
else
    EXEC="./Alya.x f1d"
fi

mpirun -np ${SLURM_NTASKS} ${EXEC}

if [[ ${TRACE}==1 ]]; then
    $EXTRAE_HOME/bin/mpi2prv -f TRACE.mpits -o ${TRACENAME} -no-
        keep-mpits
else
  ./gather.sh ${MPI}
```

[1] https://www.gnu.org/software/sed/

```
26 fi
```

Listing 3.6: Parametrized alya script.

Once we have the parametrized script, we developed a *launcher* script in charge of submitting to SLURM de jobs requested. Listing 3.7 shows the implementation. For each job requested, we copy the input set so the jobs can be executed in parallel. We copy the parametrized job script into the copied input set, set the parameters using sed, and finally submit the job using SLURM.

```
1  function submit_job {
2    NAME=alya-${MPI}-${TRACE}
3    cp -r base ${NAME}
4
5    sed -i "s/%%TRACE%%/${TRACE}/g" ${NAME}/job.sh
6    sed -i "s/%%MPI%%/${MPI}/g" ${NAME}/job.sh
7
8    cd ${NAME}
9    sbatch job.sh
10   cd -
11 }
```

Listing 3.7: Alya launcher script.

Listing 3.8 shows the usage of the `submit_job` function with the purpose of executing Alya for 1, 2, 4, 8 and 16 nodes with tracing enabled and not.

```
1 for MPI in 48 96 192 384 768; do
2   TRACE=1
3   submit_job
4   TRACE=0
5   submit_job
6 done
```

Listing 3.8: Alya job launch example.

3.2 Identification of code structure. Focus of analysis.

Before proceeding into gathering the efficiency metrics, we need to study the application structure to decide the *focus of analysis*.

The Focus of Analysis is the region of the application we select to study. We choose a region representative of the application, meaning that we usually discard initialization and finalization code. When the application presents repetitive patterns, such as an iterative pattern, we select a representative part instead of the whole region.

For doing so, we take a trace with one node and then with the useful duration view in Paraver, we try to identify patterns. Figure 3.1 illustrates the useful duration view of Alya with one node. The x-axis represents time, the y-axis the processes, and the colour means the amount of time each process performs computation with a scale from green to blue, green means lower time and blue higher. We observe three phases, first an initialization phase, then an iterative pattern and finally a finalization phase. As the trace file generated is big and the iterations are constant, we decide to study just four iterations.

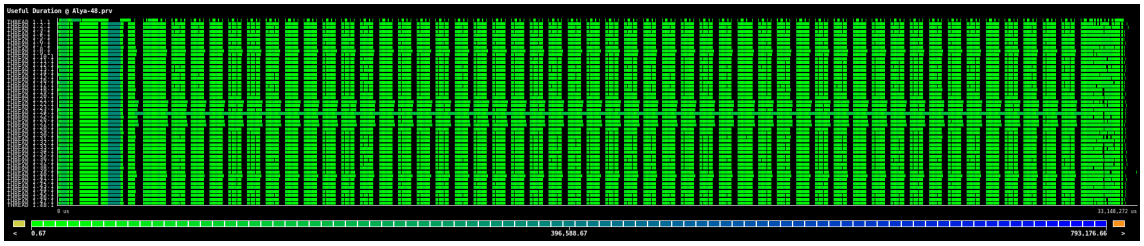


Figure 3.1: Alya structure. Own compilation.

Figure 3.2 shows the *focus of analysis*, as said, we focus on four iterations. We can see a notable difference of time in useful between processes, which suggests a load balance problem.

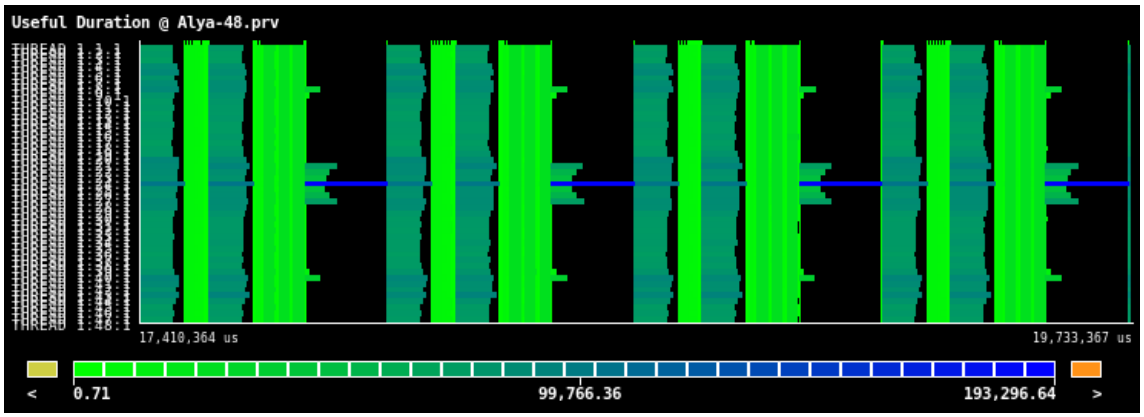


Figure 3.2: Alya focus of analysis. Own compilation.

3.3 Efficiency metrics

For gathering the efficiency metrics of the application, we need traces. For the study, we ran with 8, 48, 96, 384 and 768 processes. Table 3.2 shows the efficiency metrics for the detailed case. We can see that parallel efficiency in the base case (8 processes) is 75%, and this tells us the application scalability is terrible and is far from being able to scale to a high number of cores. The main limiting factor is the load balance going from 75% to 35%. The other two limiting factors are the transfer efficiency and the instruction scalability. These two factors will determine the performance when the load balance is solved.

Number of processes	**8**	**48**	**96**	**384**	**768**
-Parallel efficiency	75.63	59.10	49.40	36.14	26.28
-Load balance	75.82	59.99	50.83	41.71	35.75
-Communication efficiency	99.74	98.53	97.18	86.65	73.52
-Serialization efficiency	99.86	99.32	99.13	95.83	89.39
-Transfer efficiency	99.88	99.21	98.03	90.42	82.25
-Computation scalability	100.00	91.37	89.62	83.80	78.30
-IPC scalability	100.00	93.29	93.55	93.36	92.90
-Instruction scalability	100.00	97.61	95.79	89.75	84.27

Table 3.2: Efficiency metrics. Own compilation.

We can observe that the IPC scalability lowers when we jump from 8 processes to 48 it stabilizes. This happens because we are filling the nodes and the memory bandwidth, the caches, and in general, the node resources saturate.

As we saw in the *focus of analysis* section, the main limiting factor is the load balance. This issue is probably due to work not being adequately distributed among processes. We will examine this flaw in the next section.

The transfer and serialization efficiencies are also an issue we need to focus on, the application scaling to a higher number of cores will lose all the efficiency due to communications, although the load balance is addressed.

3.4 Load Balance

For addressing a load balance issue, we first need to characterize the source of the imbalance. The asymmetry can come from an improper work distribution or different instructions per cycle.

To characterize the imbalance source, we need a way to determine how the application is distributing the work.

Equation 3.1 shows the useful instruction balance (UIB) calculation, which essentially computes how the application is distributing the work.

$$UIB = \frac{avg_{i=1}^{|P|} I_{U_i}}{max_{i=1}^{|P|} I_{U_i}} \tag{3.1}$$

With Paraver, we obtained the useful instructions of each run's process and computed the useful instruction balance. Figure 3.3 shows the comparison between the useful instruction balance and the load balance. We can see that the work distribution is the main issue affecting the application balance.

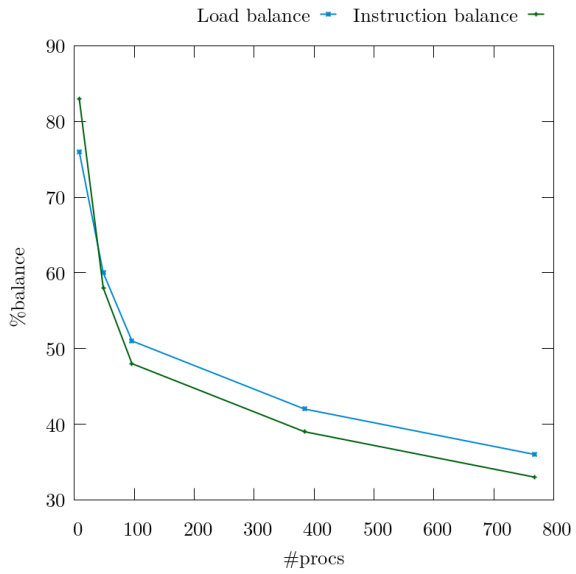


Figure 3.3: Load balance characterization. Own compilation.

3.5 Conclusions

The primary outcomes of this work are:

- The application's main flaw is the Load Balance. As the load imbalance is algorithmic, the main suggestion for addressing this is adding support for the Dynamic Load Balancing library. This workaround needs of adding a shared memory parallelism layer such as OpenMP or OmpSs.
- The communication efficiency and instruction scalability are issues that need for further analysis. Nonetheless, the priority is de Load Balance.

Chapter 4

Optimization

This chapter proposes and implements and optimization to approach the load balance issue identified in the previous chapter.

We will work on the following topics:

- At first, we propose an optimization and explain the concepts behind.
- Then we will implement and test the optimization.

4.1 Optimization proposal

As said in the previous section, the idea is to add a second level of parallelism to apply de Dynamic Load Balancing (DLB) library. This library dynamically lends the node resources of an idle process. For the shared-memory programming model, we choose OmpSs.

4.1.1 Concepts

OmpSs

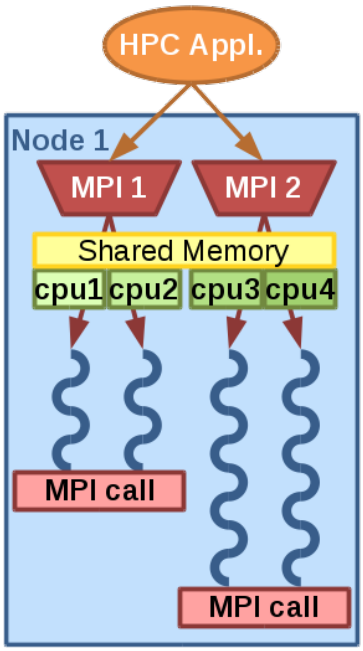
OmpSs[22] is a programming model developed at BSC, similarly to OpenMP, is a shared memory programming model based on tasks. It is also used as a forerunner to OpenMP, meaning some proposals and directives are later implemented at OpenMP.

The primary motivation for using OmpSs rather than OpenMP the standard in the HPC community is that OmpSs is better integrated with DLB.

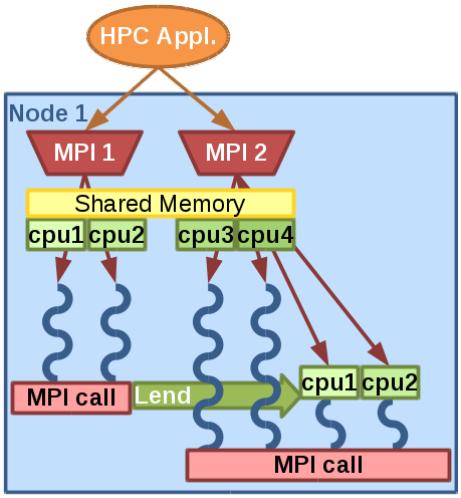
Dynamic Load Balancing library

The DLB library[23] is a collection of tools aimed to improve and track the performance of parallel applications.

As we mentioned before DLB can lend unused resources to processes with more load. This mechanism is named "Lend when idle" (LeWI); it intercepts the MPI calls that stall the processes and dynamically lends the threads to other MPI processes within the node which are computing. Figures 4.1a and 4.1b show a comparison between the same execution with 2 MPI processes, each running with two threads, with and without LeWI. Overall we can notice that the execution with LeWI enabled reaches the MPI call earlier because MPI 1 lends its cpus to the MPI 2, meaning that this process will run with 4 cpus. Thus it will finish the execution quicker.



(a) No LeWI



(b) LeWI

Figure 4.1: LeWI example. From DLB website https://pm.bsc.es/dlb

4.1.2 Proposal

The steps we will follow are:

- Hybridize the code, add necessary directives and ensure we don't add any race conditions.
- Compile and test the application with OmpSs.
- Add DLB support.
- Compile and test the application with DLB and OmpSs.

4.2 Hibridization of the code

4.2.1 Code modification

The first step is to locate in the code the source of unbalance. Fortunately, the developers hint us the code region responsible.

The region involved is responsible for computing the chemical integration of the species. The problem comes because just a few of the species of the domain reach the conditions to react. This fact makes that only a few processes compute the chemical integration. Therefore, some processes have a lot of computational load and some not.

Listing 4.1 shows a simplification of the code we will be working on. It consists of a loop that goes for all the points of the domain and checks if the conditions are adequate for the reaction, if the conditions are met, then the reaction is computed.

```
1 do ipoin=1,npoin
2   if(reaction) call cantera_integrate(ipoin)
3 end do
```

Listing 4.1: Chemical integration base loop.

Next, we need to add the directive to hybridise the code. Listing 4.2 shows the hybrid code. We use a `taskloop` directive. This construct allows us to parallelise the code easily. We specify the data sharing default to *share* and specify the copy into the `ipoin` variable as each thread must have its copy.

```
1 !$omp taskloop private(ipoin) default(shared) label(integrate) &
2 !$grainsize(GRAINSIZE)
3 do ipoin=1,npoin
4   if(reaction) call cantera_integrate(ipoin)
5 end do
```

Listing 4.2: Chemical integration hybrid loop.

We must analyse the code looking for the dependencies to ensure the parallelisation presents no data race conditions. Inside the call to Cantera, we found a data risk. During the `cantera_integrate` routine, there is a write and read access to a variable. Listing 4.3 shows the data race. We can observe threads can write and read simultaneity to `prt_gas[0]` data structure.

```
1 void cantera_integrate(...){
2   ...
3   //write
4   ptr_gas[0].setState_TPY(...);
5   ...
6   //read
7   r.insert(ptr_gas[0]);
8   ...
9 }
```

Listing 4.3: Race condition in cantera_integrate.

Listing 4.4 shows the solution. We add a copy to the shared variable `ptr_gas` and specify to be stored on the "thread local" storage using the `thread_local` keyword. Additionally, we modify the accesses to be to the copy.

```
1 static_thread_local Cantera::IdealGasMix *tmpGas=
2     new Cantera::IdealGasMix(*ptr_gas);
```

```
3
4  void cantera_integrate(...){
5    ...
6    //write
7    tmp_gas[0].setState_TPY(...);
8    ...
9    //read
10   r.insert(tmp_gas[0]);
11   ...
12 }
```

Listing 4.4: Copy of ptr_gas data structure.

4.2.2 Compilation with OmpSs

To build Cantera, we need Scons. Unfortunately, we don't have a suitable Scons installation. Listing 4.5 shows how to make Scons. We load the python module and bootstrap Scons using python. Once the bootstrapping is done, we change the directory into the bootstrapped build and install it to the desired path using the prefix flag.

```
1 module load python
2 python bootstrap.py build/scons
3 cd build/scons
4 python setup.py install --prefix=PATH
```

Listing 4.5: Scons build.

Once we have Scons built, we will use it to compile Cantera. Listing 4.6 shows the commands used for the installation. First we load the necessary modules, python, OmpSs for the mercurium compiler and the Intel MKL libraries. Then we run Scons to build the library. We specify as compilers for C and C++ the Intel compilers and for Fortran the mercurium plain profile that generates the needed files and passes the code to the Intel Fortran compiler. We define to use Intel MKL libraries as BLAS and LAPACK providers, finally the `-j48` option for enforcing parallel compilation. Finally, we call Scons to install Cantera and move Mercurium generated files to the `include` path of the installation so the Mercurium compiler recognizes it when building Alya.

```
1 module load python ompss mkl intel impi
2
3 scons build prefix=$CANTERA_FOLDER \
4 CC=icc CXX=icpc FORTRAN=plainifort \
5 blas_lapack_libs=mkl_rt blas_lapack_dir=${MKLROOT}/lib/intel64
6 -j48
7
8 scons install env_vars=all prefix=$CANTERA_FOLDER
9 cp $(find -name '*.mf03') $CANTERA_FOLDER/include/cantera
```

Listing 4.6: Cantera Mercurium build.

Next, we need to compile the code for using OmpSs. The Mercurium compiler translates the directives and passes the code to the Intel compiler (in our case). Table 4.1 illustrates the compiler flags for using Mercurium. The flags that are not present are not modified. Notice that we don't invoke the Mercurium compiler directly, we must tell the MPI compiler to invoke the Mercurium compiler, later on, will invoke the intel

Flag	Description	Value
F77	Fortran 77 compiler	I_MPI_F90=ismpfc mpif90
F90	Fortran 90 compiler	I_MPI_F90=ismpfc mpif90
FFLAGS	Fortran compilation flags	–ompss -DGRAINSIZE=1

Table 4.1: Configuration flags for Mercurium Intel MPI Alya compilation. Own compilation.

compiler. We add the `-ompss` flag to tell Mercurium to use OmpSs and we define the grain size value

Finally, following the steps of the Listing 3.1 but also loading the `ompss` module we got Alya compiled with OmpSs.

4.3 Integrating DLB

4.3.1 Code modification

For finishing the implementation, we need to enable DLB to improve the load balance within the integration loop. Since we want DLB not to intercept all MPI and act on the integration loop, we add a `DLB_Barrier` call outside the loop. The `DLB_Barrier` blocks processes and lends the threads to processes that still have not reached the barrier. Listing 4.7 shows the `DLB_Barrier` inclusion.

```
1 !$omp taskloop private(ipoin) default(shared) label(integrate) &
2 !$grainsize(GRAINSIZE)
3 do ipoin=1,npoin
4   if(reaction) call cantera_integrate(ipoin)
5 end do
6
7 call DLB_Barrier()
```

Listing 4.7: Chemical integration hybrid DLB loop.

4.3.2 Linking with DLB

To run with DLB, we have to modify the Alya compilation to link against the library. Table 4.2 shows the compilation flags needed to link against DLB. We include DLB headers for the call to `DLB_Barrier` and link to DLB using the rpath[1] mechanism to ensure the library is found.

Flag	Description	Value
INCLUDE	Headers	-I$(DLB_HOME)/include
EXTRALIB	Libraries linked	-ldlb -L$(DLB_HOME)/lib -Wl,-rpath,$(DLB_HOME)/lib

Table 4.2: Configuration flags for DLB Mercurium Intel MPI Alya compilation. Own compilation.

4.3.3 Running with DLB

Running with DLB also has its particularities. Listing 4.8 presents the modified Alya parametrized script with the DLB options. Mind that is also needed to alter the launcher script adding the DLB option. The `NX_ARGS` variable contains parameters for the Nanos runtime, which is the OmpSs runtime. With the `-enable-dlb` tells Nanos to enable DLB so the runtime can allow the resource lending. The `DLB_ARGS` variable contains parameters for the DLB runtime. We enable the LeWI and the DLB barrier mechanism. Finally depending wether, we are running with or without DLB we set the appropriate binary.

```
1 #!/bin/bash
2 #SBATCH --job-name=alya-%%MPI%%-%%TRACE%%
3 #SBATCH --output=alya.out
4 #SBATCH --ntasks=%%MPI%%
```

[1] The `-Wn` flag means to pass the following parameters to the underlying compiler.

```
5  #SBATCH --cpus-per-task=1
6  #SBATCH --time=1:20:00
7  #SBATCH --exclusive
8
9  TRACE=%%TRACE%%
10 DLB_ENABLED=%%DLB%%
11
12 if [[ ${TRACE}==1 ]]; then
13     export EXTRAE_HOME=#path to extrae installation
14     export EXTRAE_CONFIG_FILE=#path to extrae.xml
15     TRACENAME="../traces/Alya-${SLURM_NTASKS}.prv"
16     EXEC="env LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitracef.so"
17 fi
18
19 if [[ $DLB_ENABLED == 1 ]]; then
20   export NX_ARGS="--enable-dlb"
21   export DLB_ARGS="--lewi --lewi-mpi --barrier"
22   EXEC+=" Alya.dlb.x f1d"
23 else
24   EXEC+=" Alya.x f1d"
25 fi
26
27 mpirun -np ${SLURM_NTASKS} ${EXEC}
28
29 if [[ ${TRACE}==1 ]]; then
30     $EXTRAE_HOME/bin/mpi2prv -f TRACE.mpits -o ${TRACENAME} -no-
       keep-mpits
31 else
32   ./gather.sh ${MPI}
33 fi
```

Listing 4.8: Parametrized alya script with DLB.

With this we are ready to study how this optimization improved the performance.

Chapter 5

Evaluation of the improvements

To conclude the work, we will evaluate the optimization. We will study the overhead and the impact of the hybridization of the code and the ideal grainsize. Then we will determine the improvement of the DLB version against the vanilla version.

For all the data showed in this chapter, the timing is the average of the average of 4 time steps of 5 runs.

Important

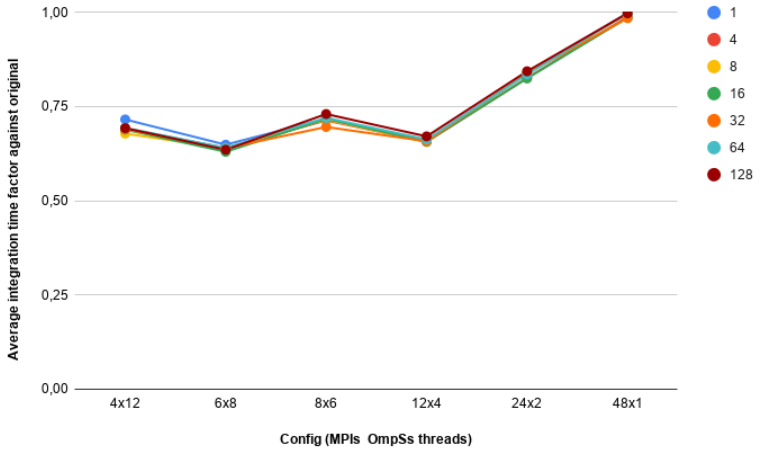
All plots and timing data is presented as a factor from the original. Absolute timing is confidential and we are not allowed to share it.

Equation 5.1 shows how we compute the timing factor, assume X is the absolute time of the Intel MPI vanilla version and X' is the absolute time for a new version we are presenting. The timing factor (F_X) is the quotient between the last and the first. Meaning that if the value is 2, it means it takes the double to compute if the value is 0.5, it takes the half to compute.

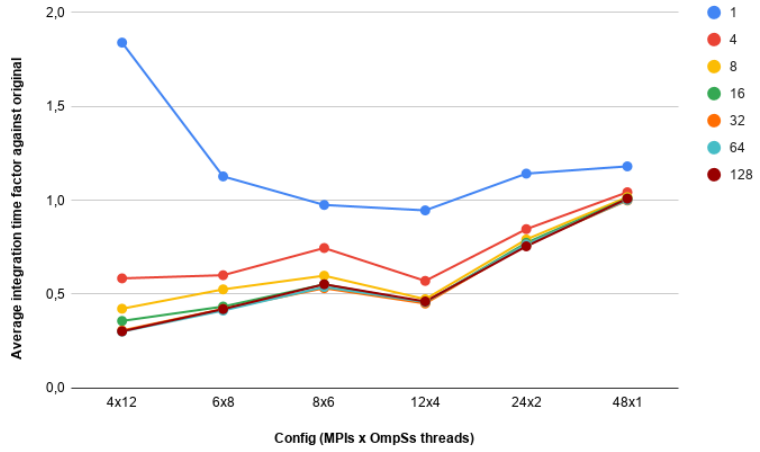
$$F_X = \frac{X'}{X} \tag{5.1}$$

5.1 Hybrid version analysis

Figure 5.1 shows a comparison between the vanilla and the hybrid version on both inputs. We tested different hybrid configurations and grain sizes. First, we can observe both cases that hybridizing does not add overheads. We can compare the *48x1* and the *48x0* configurations and see that the time is nearly equal excepting the reduced case with grain size one that the number of tasks is high and increments the execution time notably. In general terms, hybridization improves the integration time because OmpSs allows improving the load balance among threads. On the detailed case (Figure 5.1a) we perceive that the grain size does not impact the performance, although we identify the value 32 as the ideal. The best hybrid configuration for the detailed case is *6x8*. On the reduced case (Figure 5.1b) we observe grain size affects the performance. Low grainsize values make the number of tasks increase which adds overhead from making the runtime process tasks. The grainsize values that best fit is greater than 4. The ideal hybrid configuration is *4x12*.



(a) Detailed chemistry case

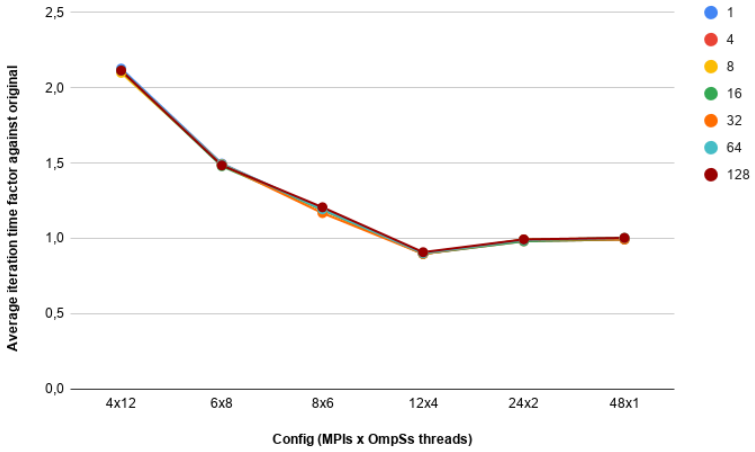


(b) Reduced chemistry case

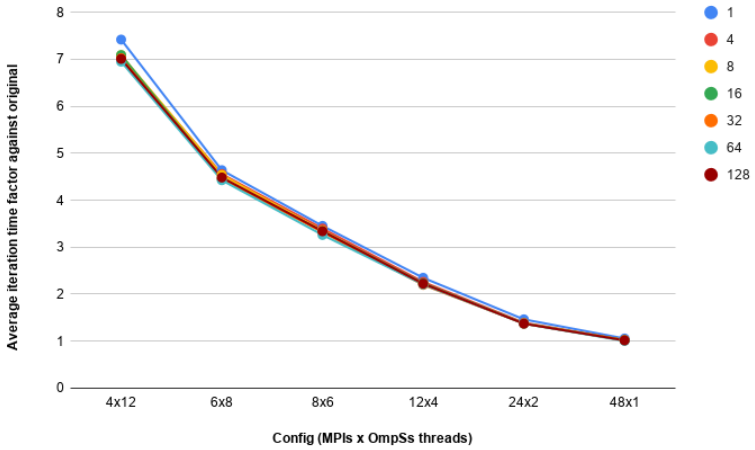
Figure 5.1: Hybrid chemical integration timing comparison. Own compilation.

Although hybridization improves the integration time sustainably, the application performance is diminished. Figure 5.2 shows the same plot but comparing against the

iteration time. We can observe that hybridization reduces overall performance. The whole application not being hybridized causes an inefficient usage of the resources and adds unnecessary overheads. This fact makes that we decide to use *48x1* version in both cases as the best for adding DLB. The negative impact of hybridization is higher on the reduced chemistry (Figure 5.2b) as the integration time weight is lower.



(a) Detailed chemistry case

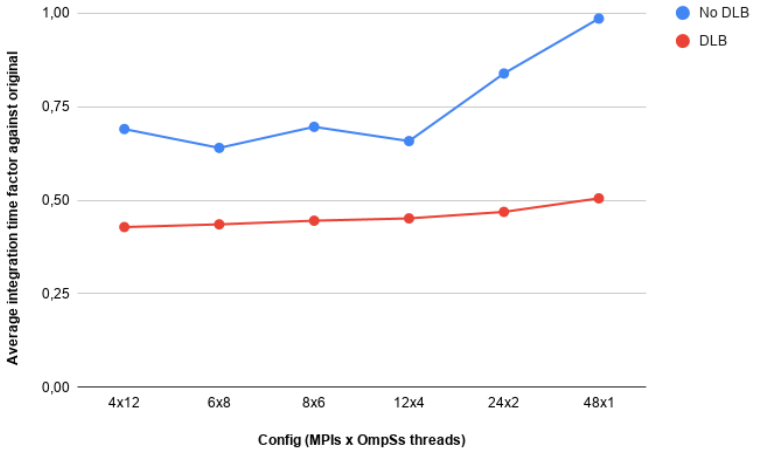


(b) Reduced chemistry case

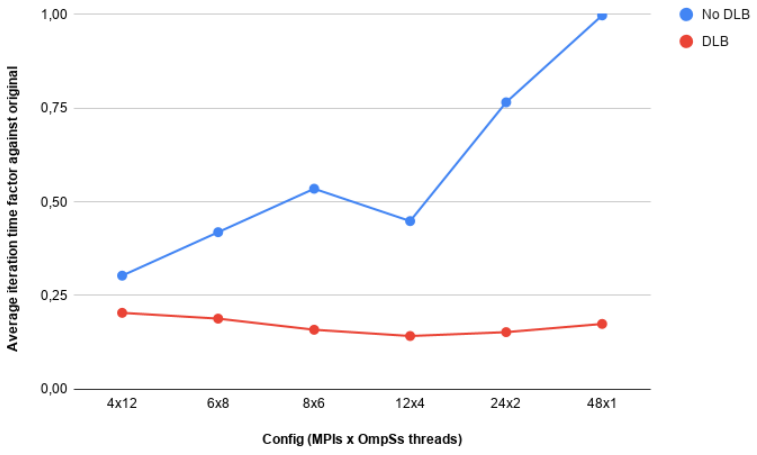
Figure 5.2: Hybrid iteration timing comparison. Own compilation.

5.2 DLB version analysis

In Figure 5.3, we compare the integration time between the hybrid version and the DLB version on the configurations we tested in the previous section. This time, we don't check all grainsize values. Instead, we choose the ideal for each input. We can observe that applying DLB makes all configurations faster than the pure MPI version and all the hybrid configurations. It is also remarkable that there is no notable difference between hybrid configurations. On the detailed chemistry (Figure 5.3a) we see that with DLB we achieve a *x2* respect the original. On the reduced chemistry (Figure 5.3b) we achieve a *x7* with DLB on the best configuration (*12x4*). Nonetheless, with the *48x1* configuration, we achieve a *x5* speedup against original.



(a) Detailed chemistry case

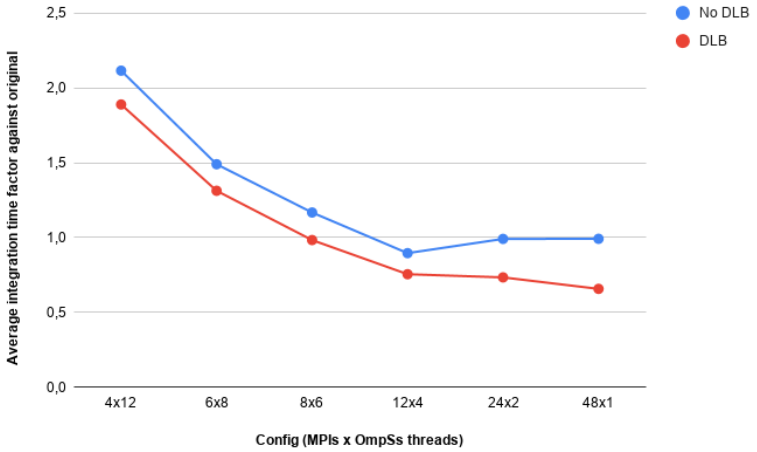


(b) Reduced chemistry case

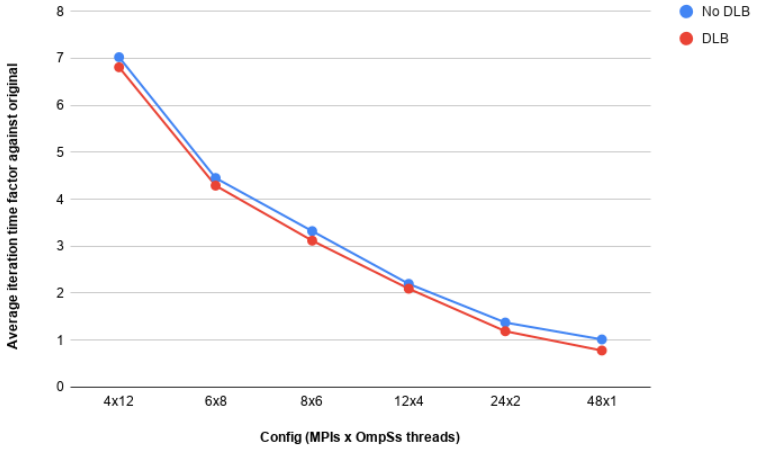
Figure 5.3: DLB chemical integration time factor on hybrid configurations. Own compilation.

To see the overall performance, in Figure 5.4 we compare the average time step with and without DLB on the hybrid configurations with the ideal grainsize. As seen in the previous section, the hybrid configurations are far from the pure MPI version since the application is not fully parallelized with OmpSs. Although, in general terms with the *48x1* configuration, we achieve a speedup in both inputs. On the detailed chemistry case

(Figure 5.4a), we earn a *1.5x* speedup. On the reduced chemistry case (Figure 5.4b), as the integration weight is lower, the speedup achieved is *1.3x*.



(a) Detailed chemistry case

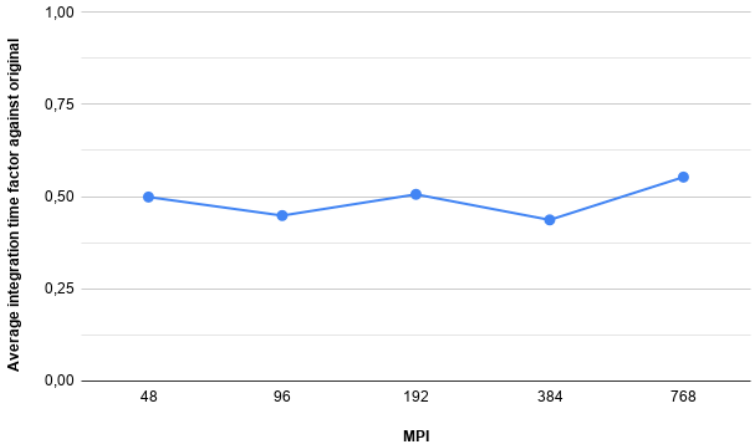


(b) Reduced chemistry case

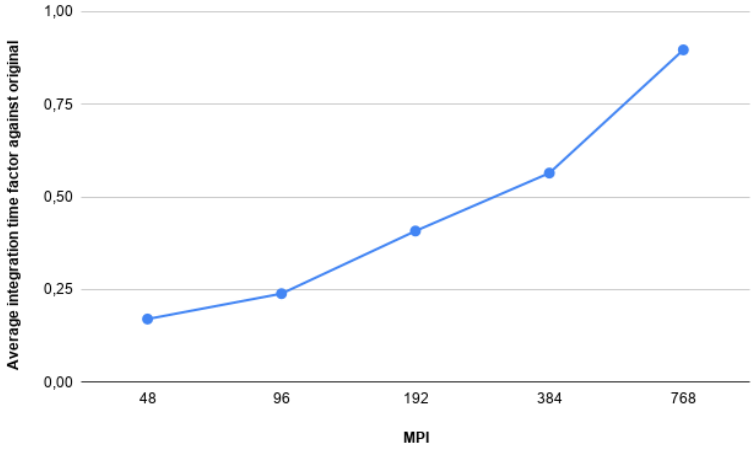
Figure 5.4: DLB iteration time factor on hybrid configurations. Own compilation.

Now we will check how the optimization behaves when running with multiple nodes. Figure 5.5 shows the integration time factor running with DLB against the pure MPI version with the same number of processes, both inputs, grainsize 16 for the complete integration and grainsize 32 for the reduced. We can see that in the complete integration case (Figure 5.5a), the speedup is constant to *2x*. On the reduced (Figure 5.5b), DLB effectiveness is lost when increasing the number of nodes. This loss is probably because when we increase the number of nodes, the balance among nodes is worse. DLB can balance the workload difference within a node because it relays on a shared memory programming model.

Figure 5.6 presents the scalability plots of the same runs, with and without DLB. On the detailed case (Figure 5.6a), we see scalability from both versions are similar. This fact makes sense since the speedup showed in the last plot is constant, the scalability between the versions is analogous. On the reduced chemistry (Figure 5.6b) we observe that the DLB version's scalability is poor because as said, DLB loses effectiveness when increasing the number of processes.

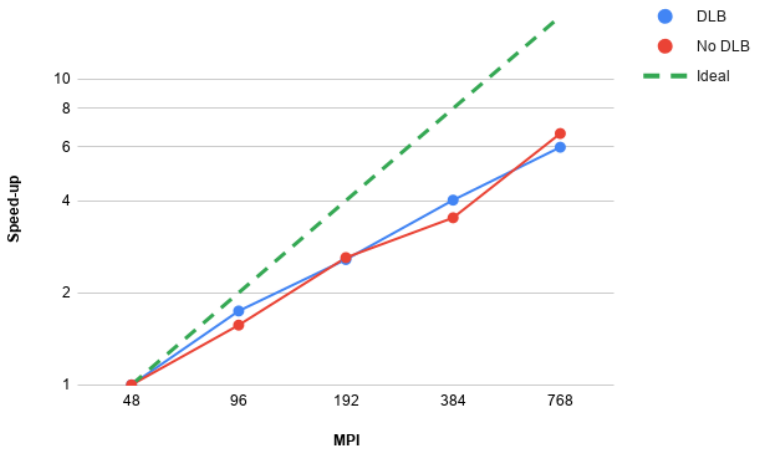


(a) Detailed chemistry case

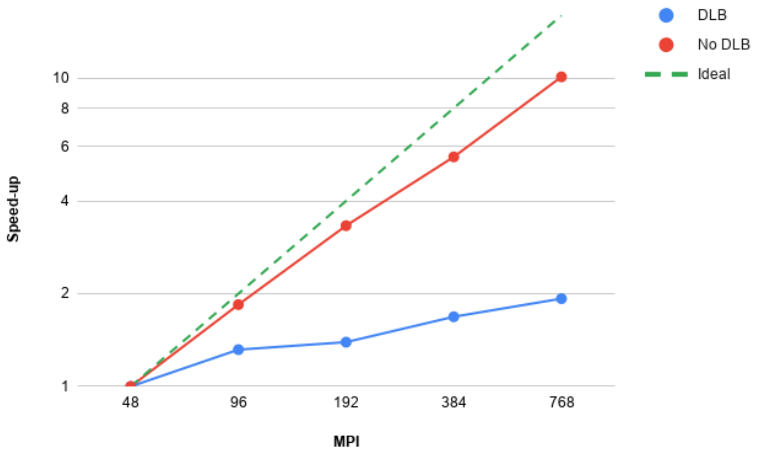


(b) Reduced chemistry case

Figure 5.5: DLB Chemical Integration time factor multinode. Own compilation.



(a) Detailed chemistry case



(b) Reduced chemistry case

Figure 5.6: Chemical integration scalability comparison. Own compilation.

Chapter 6

Conclusions

With this final chapter, we close the work. The main takeaways from the study are:

- We helped the module developers to have a better understanding of their code performance.
- We achieved a relevant speedup against the vanilla version. It will allow combustion researchers to get results faster and use fewer resources.
- This study has motivated further work on analyzing in-depth the Load Balance and getting the metrics at run-time using Tracking Application Life Performance (TALP), a DLB module. We have submitted (pending of revision and acceptance) a study to the International Supercomputing Conference, presenting TALP and demonstrating it using this Alya module.

On the personal side:

- This project has introduced me to research, precisely on the high-performance computing field.
- I have developed in-depth my performance analysis skills.
- Working in a project in conjunction with non-experts in the performance analysis field has given me experience in teamwork and communication skills.
- This project has introduced into writing academic articles.
- Applying OmpSs and DLB to the code has made me grown my technical and learning new things skills.
- Developing the analysis have grown my working methodology, specifically on managing data and version control of code.

6.1 Future work

We planned a contribution to the combustion domain's scientific community, presenting the approach to address load imbalance in the combustion using the optimization presented in this work.

Bibliography

[1] M. Vazquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Aris, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, and J. M. Cela, "Alya: Towards exascale for engineering simulation codes," 2014.

[2] "Performance optimisation and productivity," POP-coe, accesed on 26/09/2020. [Online]. Available: https://pop-coe.eu/

[3] C. Kessler and J. Keller, "Models for parallel computing: Review and perspectives," *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen*, vol. 24, p. 13–29, 2007. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-40734

[4] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[5] "Marenostrum 4 techincal information," Barcelona Supercomputing Center, accessed on 26/09/2020. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information

[6] M. Wagner, S. Mohr, J. Giménez, and J. Labarta, "A structured approach to performance analysis," in *International Workshop on Parallel Tools for High Performance Computing*. Springer, 2017, pp. 1–15.

[7] F. Banchelli, K. Peiro, A. Querol, G. Ramirez-Gargallo, G. Ramirez-Miranda, J. Vinyals, P. Vizcaino, M. Garcia-Gasulla, and F. Mantovani, "Performance study of hpc applications on an arm-based cluster using a generic efficiency model," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2020, pp. 167–174.

[8] "Bsc tools extrae website," Barcelona Supercomputing Center, accessed on 26/09/2020. [Online]. Available: https://tools.bsc.es/extrae

[9] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[10] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1. Citeseer, 1995, pp. 17–31.

[11] "Bsc tools paraver website," Barcelona Supercomputing Center, accessed on 26/09/2020. [Online]. Available: https://tools.bsc.es/paraver

[12] G. Ramirez-Miranda, "Tracedraw repository," accessed on 26/09/2020. [Online]. Available: https://github.com/teleportex/tracedraw.sty

[13] "Marenostrum4 user's guide," Barcelona Supercomputing Center, accesed on 26/12/2020. [Online]. Available: http://www.bsc.es/support/MareNostrum4-ug.pdf

[14] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[15] "Slurm documentation," Slurm, accesed on 23/12/2020. [Online]. Available: https://slurm.schedmd.com/documentation.html

[16] D. G. Goodwin, R. L. Speth, H. K. Moffat, and B. W. Weber, "Cantera: An object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes," https://www.cantera.org, 2018, version 2.4.0.

[17] H. Servat, G. Llort, K. Huck, J. Giménez, and J. Labarta, "Framework for a productive performance optimization," *Parallel Computing*, vol. 39, no. 8, pp. 336–353, 2013.

[18] S. Girona, J. Labarta, and R. M. Badia, "Validation of dimemas communication model for mpi collective operations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2000, pp. 39–46.

[19] M. Casas, R. Badia, and J. Labarta, "Automatic analysis of speedup of mpi applications," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 349–358.

[20] R. M. Stallman and R. McGrath, "Gnu make-a program for directing recompilation," 1991.

[21] "Intel math kernel library website," Intel, accessed on 23/12/2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html

[22] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[23] M. Garcia, J. Corbalan, and J. Labarta, "Lewi: A runtime balancing algorithm for nested parallelism," in *Parallel Processing, 2009. ICPP '09. International Conference on*, Sept 2009, pp. 526–533.