Universidad Carlos III de Madrid
Escuela Politécnica Superior
Degree in Computer Science and Technology

B.Sc. Thesis

Development of a Performance Analysis Environment for Parallel Pattern-based Applications

Author: Nerea Luna Picón
Supervisor: Manuel F. Dolz Zaragozá

Leganés, Madrid, Spain
June 2018

"Never give up. Today is difficult and tomorrow will be worse, but after tomorrow sun will rise."

Jack Ma

Agradecimientos

Lo primero de todo, quería agraceder a mi familia todo su apoyo cuando no estaba en mis mejores momentos y veía todo muy negro. Siempre han tratado de convercerme y hacerme ver que valía más de lo que pensaba. Eso me ha animado mucho y ha hecho que cambiara ese pensamiento tan negativo que a veces se apoderaba de mi cabeza.

Por otro lado, dedicar este trabajo al departamento de ARCOS, tanto a profesores como compañeros, desde luego que sin ellos no habría sido posible sacarlo adelante.

A Javi Prieto, Saúl y Guille, nuestros consejeros, haciéndonos reflexionar sobre la universidad y sobre la vida en general, de verdad que han sido un gran apoyo y aún mejores personas.

A Javi Kernel, por ofrecerse a darnos clases de LaTeX a todos aunque no a tanto nivel como querías tú, la próxima vez será.

A Javi Doc, por su paciencia durante los primeros meses de trabajo en el departamento cuando no entendía bien las cosas. Por explicarme línea a línea archivos de más de 2000 líneas totalmente incomprensibles. Te sentabas conmigo y daba igual que fuera la hora de comer, te quedabas ahí hasta que las cosas funcionasen, eso lo valoro mucho. A David, por ayudarme también con mi trabajo y por aportar esas ideas 'eficientes' de las que yo no me había dado cuenta, gracias por todo.

Por último, quiero mencionar especialmente a Manu, mi tutor. Lo primero de todo, muchas gracias por haberme ofrecido pertenecer a este departamento, de verdad que he aprendido mucho más de lo que pensaba y no me arrepentiré nunca de haber aceptado esta oportunidad (aunque al principio no lo viese tan claro). Quiero agredecerte todo el tiempo que me has dedicado para conseguir terminar mi trabajo con éxito y por estar siempre ahí, por venir prácticamente todos los días de la semana a preguntar qué tal iba y por molestarte tanto por intentar sacar una buena nota, creo que muchos profesores deberían aprender de ti y de tu forma de enseñar.

Tampoco quiero dejar de nombrar a Florin, que por desgracia nos ha dejado recientemente provocándonos a todos ese gran vacío. Te echaremos de menos.

Development of a Performance Analysis Environment for Parallel Pattern-based Applications

B.Sc. Thesis

Nerea Luna Picón

Abstract

One of the challenges that the scientific community is facing nowadays refers to the data parallel treatment. Every day, we produce more and more overwhelming amounts of data, in such a way that there comes a point at which all those data volumes grow exponentially and can not be treated as we would desire.

The importance of this treatment and data processing is mainly due to the need that arises in the scientific area for discovering new advances in science, or simply for discovering new algorithms capable of solving experiments each time more complex, whose resolution years ago was an unfeasible task due to the available resources.

As a consequence, some changes appear in the internal architecture of these computers in order to increase their computing capacity and thus, be able to cope with the need for massive data processing. Thus, the scientific community has implemented different pattern-based parallel programming frameworks in order to compute experiments in a faster and efficient way. Unfortunately, the use of these programming paradigms is not a simple task, since it requires expertise and programming skills. This is further complicated when developers are not aware of the program internal behaviour, which leads to unexpected results in certain parts of the code. Inevitably, some need arises to develop a series of tools with the aim of helping this community to analyze the performance and results of their experiments.

Hence, this bachelor thesis presents the development of a performance analysis environment based on parallel applications as a solution to that problem. Specifically, this environment is composed of two techniques commonly used, *profiling* and *tracing*, which have been added to the GrPPI framework. In this way, users can obtain a general assessment of their applications performance and thus, act according with the results obtained.

Keywords: data processing · parallel programming frameworks · GrPPI · performance analysis · profiling · tracing

Desarrollo de un Entorno de Análisis de Prestaciones para Aplicaciones basadas en Patrones Paralelos

B.Sc. Thesis

Nerea Luna Picón

Resumen

Uno de los retos actuales al que la comunidad científica está haciendo frente hace referencia al tratamiento en paralelo de datos. Diariamente producimos cada vez más cantidades abrumadoras de datos, de tal manera que llega un punto en el que todos esos volúmenes de datos crecen desenfrenadamente y no pueden ser tratados como se desearía.

La importancia de este tratamiento y procesamiento de datos se debe, principalmente, a la necesidad que surge en el ámbito científico por descubrir nuevos avances en la ciencia, o, simplemente, por descubrir nuevos algoritmos capaces de resolver experimentos cada vez más complejos cuya resolución años atrás era una tarea inviable debido a los recursos disponibles.

Como consecuencia, surgen cambios en la arquitectura interna de estos ordenadores con el fin de aumentar su capacidad de cómputo y así poder hacer frente a dicha necesidad de tratamiento masivo de datos. Así, la comunidad científica ha implementado distintos *frameworks* de programación paralela basados en patrones paralelos con el fin de computar esos experimentos de una manera más rápida y eficiente. Desafortunadamente, la utilización de estos paradigmas de programación no es una tarea sencilla, ya que requiere experiencia y destrezas en programación. Esto se complica aún más cuando los desarrolladores no son conscientes del comportamiento interno del programa, lo cual conlleva a obtener resultados inesperados en ciertas partes del código. Inevitablemente, surge la necesidad de desarrollar una serie de herramientas con el objetivo de ayudar a esta comunidad a analizar el rendimiento y resultados de sus experimentos.

Así pues, este trabajo fin de carrera presenta el desarrollo de un entorno de análisis de rendimiento de aplicaciones paralelas como solución al ese problema. Concretamente, este entorno está compuesto de dos técnicas comunmente utilizadas, *profiling* y *tracing*, las cuales han sido añadidas al *framework* de programación paralela GrPPI. Así, los usuarios podrán recibir una valoración general sobre el rendimiento de sus aplicaciones y actuar conforme a los resultados obtenidos.

Palabras clave: tratamiento de datos · *frameworks* de programación paralela · GrPPI · análisis de rendimiento · *profiling* · *tracing*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This first chapter is divided into four sections. Section 1.1 motivates the problem and presents the main challenges that today's scientific and industrial applications are facing. Section 1.2 presents some theoretical concepts that need to be taken into account in order to understand the proposed solution to a problem that will be next explained. Section 1.3 refers to the main objectives of the project. Finally, in Section 1.4, we will show a brief description of the different chapters that this document is composed of.

1.1 Motivation

We live in a world that is constantly being adapted to technology advances. Specifically, this refers to the development of more powerful computers which face the computational demands of nowadays scientific experiments. Data is generated at every moment day-to-day, and this situation requires each time the development of experiments able to support higher computational workloads. Consequently, scientific progress aims for solving each time more complex experiments, and High-Performance Computing (HPC) community is doing his bit in such a way that those experiments (applications) are completed faster.

These changes, motivated by the huge amount of data generated and the associated computational complexity, have moved computer to a new era in which the operations can be performed much faster than before. That is, computers have now the capacity to compute from 100 megaflops to a petaflops level (which is a measure of the machine performance equivalent to 10^{15} floating point operations per second, also known as FLOPS [1]). However, we are still far from getting supercomputers able to perform operations at least at the order of one exaflop (10^{18} FLOPS).

The reason why we need exascale computers is because each time, we need to treat larger data volumes and also compute more complex and precise calculations in order to discover new phenomena and advances in science [2]. Exascale computers are envisioned to happen by 2020, and by that time, computers will make use of exaflops instead of petaflops, which is the same as saying that computers will be 1000 times faster than the ones currently used.

This change to the exascale era will also imply some changes at an architectural computer level. Working with bigger amounts of data in new experiments will inevitably require exploiting parallelism levels in order to obtain faster results in an effective way. Consequently, we will also need much more multiprocessors working in parallel and more memory capacity to store those data volumes [3].

HPC systems refer to those that compute constant parallel processing techniques to solve nowadays complex computational problems from a wide range of scientific and industrial applications, for instance, in astronomy, cosmology, computational chemistry or computational engineering. They are prepared to deliver a high and continued performance through a concurrent use of computing parallel resources. However, the complexity to develop and run applications on these systems also increases. This complexity has to do with the number of heterogeneous processing elements that these platforms offer, and how applications make use of them in an efficient manner.

However, there exist some limitations regarding exascale platforms with respect to its energy consumption [3]. The problem is that there is no an equitable balance between the computational capacity of a computer and the amount of energy it requires. At the end of the day, energy is money, and there is an urgent need for developing more energy-efficient processors and accelerators which help at reducing costs and energy to make possible the exascale goal. Moreover, as stated in [4], in the future computers are very likely to consume more energy than now, and so, their use will be restricted by the provided power draw. For this reason, it is recommended to have in mind one initial optimization goal: trying to consider energy consumption figures when working with computers at high computing levels. Something that is quite often in order to achieve it, is by means of improving the performance of our applications, in such a way that their time-to-solution is reduced, and so, less energy would be consumed.

To achieve the performance and energy constraints of exascale platforms, it has been envisioned that parallel programming models will play a very important role nowadays. They will allow the process and treatment of all our data in a more efficient way. Without these parallelism paradigms, computations would be much slower and even we might not exploit all available resources provided by such HPC platforms. Nevertheless, we also need to say that the development of parallel applications is not a straightforward process, since data races, poor data locality, starvation, etc. are some situations very likely to appear and it also requires further expertise on the area.

One solution to cover this necessity is by means of pattern-based parallel programming frameworks. This type of frameworks offer some mechanisms that allow users parallelize their code in a very simple way. These mechanisms refer to parallel patterns. Thanks to them, users are able to develop more readable and portable solutions with a high-level of abstraction, in such a way that they hide away the complexity behind concurrency mechanisms.

They help users to better understand the code and to build it in a flexible, robust and portable fashion. However, the code gets encapsulated in such a way that developers lose track of the inner behaviour of these building blocks in the program, that is, parallel patterns are presented to users as mere black boxes, and also the optimization process when using them gets harder. For this reason, users are likely to not fully understand and follow the program internal flow, and this has caused the development of appropriate support techniques (profiling and tracing) and tools that can help them to make an exhaustive analysis of parallel applications performance.

Profiling and tracing tools demonstrate their importance not only by means of understanding the parallel behaviour of our applications, but also for identifying hot spots and potential bottlenecks, that are likely to happen when using parallelization mechanisms such as C++ threads, OpenMP, Intel TBB, etc. In this way, there exist two main performance approaches in order for us to have an idea of the internal behaviour of our applications [5]:

- **Profiling**: obtains performance metrics when as more profiles of the application behaviour has been aggregated during execution. It reduces the amount of data that needs to be stored at run-time, but however, it lacks some dynamic information in comparison to event tracing.
- **Tracing**: records and tracks different types of events in a precise and detailed time stamp during run-time. This information has to do with significant points (events) that may happen during execution. By collecting data at run-time, any dynamic interaction between concurrent processing elements may be captured.

Having reached this point, and briefly defined these two concepts (which will be explained in detail in Section 1.2), we will study in this project the development and testing of a performance analysis environment. This work describes the development and implementation of both profiler and tracing approaches in GrPPI, a pattern-based parallel programming framework which is able to switch from one back end (C++ threads, OpenMP, Intel TBB) to the other in a very flexible way.

Data is being produced at every time in different scopes, but we are not always noticing about what is really happening inside when running parallel applications, because they can become complex at some points. Faster applications will also be needed to treat those data, but actually, we are not really aware of the benefits we could obtain if we could get feedback from some performance information

during execution. By having some type of performance tool, it will help us to detect bottlenecks and unexpected situations that can massively slowdown the execution of our applications, so, in the end, its use is not a waste of time although it would seem. Performance tools will provide us with some idea of the workflow of our applications, not only statistically but graphically.

1.2 Theoretical background

In this section, some fundamental concepts will be explained since they will be used along the whole document, so we will give an introduction to them so that any reader can acquire at least some basic knowledge of them.

This project puts together both aforementioned topics: *parallel patterns* and *analysis environment*. Within the *environment* part, we need to distinguish between two sub-parts or submodules: profiling and event-tracing. For this reason, an explanation of these three concepts will be next given.

1.2.1 Parallel programming

The main objective of this project is the development of a performance analysis environment for GrPPI framework. The two aforementioned submodules will compose this environment, and in order to use it, we need to make use of a set of different application examples, concretely, parallel applications. The following will serve to introduce parallel programming to get some basic overview.

Parallel programming is a computation modality in which multiple compute resources are used concurrently in order to solve a problem. With multiple resources, it can refer to the number of processors or to the number of computers directly connected throughout a network [6]. In this way, the problem is broken down into several parts, and each computing entity (thread or process) takes care of a single one. Some advantages of this programming modality are:

- Complex problems can be solved.
- We can save time and money since computations will be completed faster.
- We can take advantage of non-local resources.
- Parallelism exploits better computers in terms of reducing time-to-solution or increasing the performance which usually leads to less energy consumption.

One way to exploit parallelism with high-level abstraction is the use of parallel patterns. Parallel patterns are a compact and comfortable strategy to deal with a specific parallel programming problem (see

Section 2.1). Therefore, our parallel performance analysis environment is going to be applied to those applications that contain parallel patterns.

1.2.2 Performance analysis techniques

As stated in Section 1.1, understanding how applications are executed and leverage computational resources is a really complex task that demands additional tools. To get insights into the behavior of applications, the HPC community has developed profiling and tracing tools that aid at performing this task.

A program is composed of a set of instructions. When these instructions are executed, different types of *events* can be emitted. An *event* is any action or occurrence emitted by the program. If the program is instrumented through a profiling or tracing library, then, after the termination of this event, its response can be kept for a future: precisely, by means of one of the two implemented submodules (profiling or tracing). After the program execution, they will show that recorded information from events by means of a execution plot or graph, or a summary table.

They are very useful and can help us to parallelize our code since we can learn from the feedback obtained when using them. Two different ways to measure the efficiency of our program can be achieved by means of profiling and tracing. In both of them, a set of three phases need no be accomplished [7]:

1. **Instrumentation** – is the process of adding to our code some specific routines or functions that are the responsible for tracking all the actions that can take place in the program. These routines can be added at any stage in the program, trying always to locate those interesting parts.
2. **Measurement** – with both techniques (profiling and tracing), we can measure those interesting aspects of our code via some specific and defined metrics (that we will explain in Section 4.1). After instrumentation, we can run the program and obtain the results from these measurements.
3. **Visualization and Analysis** – finally, the most important part from the user side is visualizing and trying to understand/conclude what is the application behaviour.

With this, a brief explanation of each approach is going to be explained [7] below:

Profiling

With the profiling technique, we can obtain information about our program in terms of a statistics of performance summary. The usual process is that recorded information after instrumentation is kept

during execution and stored in some data files, that, afterwards, will show the user a statistical summary depending on the used metric. We will see that in our model, we will make use of some specific *data structures* that will save the events instead.

Profiling is as important as event-tracing (which is going to be explained below) since by using it, we can have another point of view from the same concept: performance analysis. Instead of getting some graphical representation of our program's behaviour (as tracing offers), a specific and concrete summary will be provided. Within this approach, two main procedures can be followed [7]: *sampled process timing* or *measured process timing*. In the first one, hardware counters are the responsible for interrupting the program execution from time to time. Regarding the second one (which is the chosen approach), events are tracked at the beginning and end of some routine so that at the end of execution we can collect those stored data.

Examples of statistical metrics used in profiles can be: *accumulated mean*, *standard deviation*, *mean throughput*, *total execution time*, *maximum and minimum values*, *cache misses*, *CPU time*, etc. most of them used in our proposed solution.

Tracing

Tracing is an alternative to profiling which pursues the recording of information about significant points (events) during the execution of a program [8]. With this other technique, a detailed log of time-stamped events is tracked with this information, users are able to see not only independent events but the interactions between them (communications and routine's transitions).

As with profiling, we need previously to instrument our code at those desired regions, and after executing our program, that tracked or recorded information will be stored by means of *event records*. Event records are composed of a set of different fields providing each one concrete information. Typically, these fields represent information such that CPU identifier, the thread identifier, time at which the event has occurred, event type, etc. Therefore, when applying tracing, several *event trace records* are generated after execution (one at each time-stamp) and these ones will be the ones that will feed a concrete trace visualization tool able of showing those objects in a graphical way [8].

By graphically observing the results, we can easily detect possible bottlenecks by looking at the graphical representation to identify which parts are conflicting or the most time-consuming. It is like a way to identify or detect the potential bottlenecks of our implementation, during the optimization stage, a developer can directly focus on that part trying to fix it or optimize it.

However, it is not worth it if tools only show low-level information that is completely ignored by users because they just do not understand it, so with our proposed model, profiling and tracing submodules

for GrPPI, applications parallelized using patterns will be able to address this possible situation and make the user observe which portions of the code should be improved, have low performance, or are, for instance, not well-balanced in terms of parallel programming. Later on, we will be able to determine how these techniques have been integrated into this pattern-based parallel programming framework, GrPPI, and the results obtained from its use.

1.3 Objectives

The main aim of this project is to design and implement a performance analysis environment that serves us as a tool that can help us to make an exhaustive analysis of pattern-based applications performance. This goal can be translated into the following objectives:

- **O1. Promote Parallel Pattern Programming:** our solution is directly related to parallel applications. Specifically, it focuses on those that make use of different parallel patterns structures, as we will see along next chapters. Parallel patterns are a very compact and useful strategy to deal with a specific parallel programming problem while hiding complexity to users. Besides, the use of our implemented environment will justify the improvement obtained in applications when using these pattern structures.
- **O2. Flexibility:** our solution should provide the user to choose between the *profiling* submodule (which offers a statistical summary of the execution) or the *event-tracing* one (in charge of showing a graphical representation of the computation carried out) since both have been implemented. These are two aspects that users do not notice about, but knowing how things are computed in a low-level way is actually important in order to assess efficiency. More details about the whole module can be found in Chapter 4.
- **O3. Efficiency:** thanks to our model, users will be able to easily measure application performance. Our model will serve as an application's efficiency verification tool.
- **O4. Reusability:** the two submodules developed for the pattern-based parallel programming framework have been developed to be as flexible as possible, so accomplishing the reusability goal. That is, since two submodules are going to be added (profiling and tracing), the provided interface will need to be as modular as possible so that in the future other modules can be easily incorporated.
- **O5. Usability:** our solution will be aggregated to a pattern-based parallel programming framework (GrPPI) that supports different back ends (sequential, ISO C++ threads, OpenMP and Intel TBB). Therefore, users could also compare different implementations of the same program by using those

back ends in order to identify which one performs better for a specific application. Alternatively, it can serve to examine the differences when programming the same piece of code with different programming approaches.

1.4 Document structure

This document has been divided into different chapters. Next, we will present a short description of each one:

- Chapter 1, *Introduction*, presents a brief description of the document contents. Besides, it includes the motivation, a brief theoretical background and the objectives of this project.
- Chapter 2, *State of the Art*, presents the related work and it includes both, a description of today's parallel programming frameworks along with their principles and limitations, and the currently used profiling and tracing tools. Besides, we will dedicate a subsection to parallel and modern architectures.
- Chapter 3, *System Architecture*, describes, on the one hand, the complete requirements specification catalog according to the elicitation process, and on the other hand, it shows in a detailed way the set of different software diagrams designed for this specific project.
- Chapter 4, *Implementation*, describes the proposed environment design. On the one side, the profiler part, which, in a table-based shape, it will statistically summarize the performance of our application. On the other side, the event-tracing part, that will register the whole inner activity of the application in terms of different execution traces according to the used metric. For each already mentioned part, one simple example will be given. In order to better understand the applied statistical measures, we will provide a formal definition of the implemented metrics at the beginning of this chapter.
- Chapter 5, *Evaluation*, specifies the performed experiments in both profiler and tracing submodules, accompanied by a set of figures so that demonstrate the results obtained with it. Concretely, we have evaluated eight different case studies and a more and real data stream processing application. This application uses some additional features developed in the profiling submodule that allows observing the application behavior at real-time.
- Chapter 6, *Legal Framework*, explains the actual applicable regulations to this project, as well as the different licenses through which this project is subject to.

- Chapter 7, *Project Planning*, explains the methodology put into practice as well as a description of the project planning by means of a Gantt diagram.
- Chapter 8, *Social-Economic Environment*, contains a summary of the total project budget as well as explaining the social and economic impact this project will produce to society.
- Chapter 9, *Conclusions and Future Work*, collects the set of conclusions once finished this project, including some of the problems that we have faced along it. We will mention as well some future work still to develop.

Chapter 2

State of the Art

This chapter presents the state of the art of nowadays parallel programming frameworks, together with profiling and tracing techniques. We will call into question the actual problem and the proposed solution, as well as naming a set of similar entities that also work with the explained topics in Chapter 1.

As mentioned in Chapter 1, this project has been developed as a solution to applications performance analysis. It is not only a question about how to make our programs more efficient, but even more important, trying to understand the performance obtained after execution. If we were able to visualize and abstract the obtained results, we could learn from them and maybe this would lead us optimize some parts of the code to finally become more efficient than before.

Two differentiated concepts have been mentioned in the previous paragraph, *efficiency* and *performance analysis*, which precisely are the two main topics involving this project. Regarding efficiency, say that this performance analysis environment has been developed over parallel applications, since they are usually more complex in terms of behaviour understandability. For this reason, a set of different parallel programming frameworks will be next shown in Section 2.2. However, all these frameworks work under a set of parallel patterns, so we will show first in Section 2.1 how these patterns are defined and how they work.

On the other side, we present different profiling and tracing tools available nowadays (Section 2.3), in contrast with the performance analysis module developed in this project over the just mentioned GrPPI framework. Taking a look to some currently used performance analysis tools, we will also be able to learn about the advantages each one provides so that we can make a general conclusion about them in Section 2.5.

2.1 Introduction to parallel patterns

Before going deeper in this subsection, it is interesting to know beforehand about parallel patterns. A parallel pattern can be defined as a compact and flexibly strategy to deal with a specific parallel programming problem. Throughout the definition of concrete building blocks, we will be able to provide parallelism to our applications in a very flexible way, since with these patterns, our code gets encapsulated in such a way that programmers do not have to worry about any possible concurrency issue during execution. Consequently, as we do not need to care about concurrency problems, neither we do not need to make use of any additional communication nor synchronization mechanism to avoid it.

2.1.1 Data parallel patterns

Next, we describe the Data Parallel Patterns that will be used in forthcoming chapters of this work. This strategy refers to any entity that is independent between the current and following executions. That is, computations over each element are *"stateless"*, as different entities processes elements independently. Below we describe the associated patterns to this strategy:

- **Map:** belonging to the Data Pattern model, this pattern is based on the application of one operation in parallel over one or several data sets, giving as a result of this transformation, another different data set. There are two variations of this pattern, the one taking as input one single data sequence, namely *unary-map*, or the one taking multiple data sequences as input, *N-ary map*. Then, it could be considered the *"transformation"* operator as the main element in this pattern.

  For instance, if we had a sequence of ten natural numbers, with this pattern we could get another sequence of the same length but containing the squares of the initial values of each element in that sequence. This operation will be done in parallel for each element and the result sequence will contain the new values after this transformation. A graphical representation of this pattern is shown in Figure 2-1:

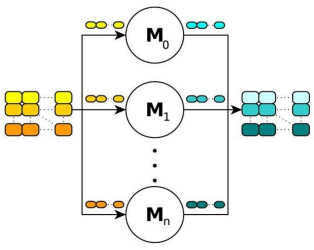


Figure 2-1: *Map pattern scheme [9].*

- **Reduce:** this is another type of Data Pattern. It consists of the combination of the values of one data set by means of a binary operation (usually the associative or commutative one). As a consequence, a unique element is obtained as the result of this operation. There also exists two versions of this pattern: the one in which you can previously set the initial value of the reduction, or the case in which you do not make use of any initial value. The *"combination"* is, thus, the element that roles the main function here, as we need to combine the elements of a sequence in one way or the other.

  For instance, given again a random sequence of natural numbers, we can obtain by applying this reduction, the aggregation of all of them in such a way that we would get a final value (element) as a final result (see Figure 2-2).

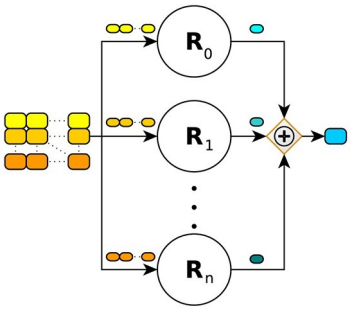


Figure 2-2: *Reduce pattern scheme [9].*

- **Map-Reduce:** also belonging to the Data Pattern group, this pattern is just a mixture of the two former patterns, the map and the reduce ones, but joined together in just one building block. In this way, two phases can be observed when making use of this pattern: a *"transformation"* phase just followed by a *"combination"* one. As with the previous ones, we can have not only the *unary-map* variant but also the *N-ary map*. In Figure 2-3, a picture of this pattern is presented.

  Continuing with the previous examples, the use of this pattern is able to produce as output one number, for instance. This number will result after going through the two aforementioned phases, so it will be the result of the square sum of the elements that become part of that specific number sequence.

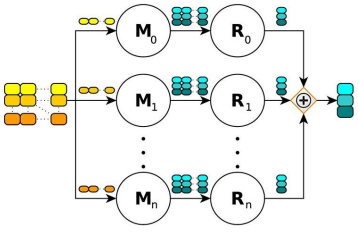


Figure 2-3: *Map-Reduce pattern scheme [9].*

- **Stencil:** this pattern uses the concept of *neighbourhood*. Here, elemental functions are not only applied over one element at a time, but also taking into account their *"neighbours"*. That is the reason why this pattern can be considered as a generalization of the map pattern. Two key elements make up this pattern are: a transformer operation and a second neighbourhood operation.

  From any number sequence, and previously defined the specific neighbourhood to be used (in this case, for instance, neighbours can be the element just on its left and the one on its right), we could add in parallel (for each item in the sequence), its value plus the values of the two previous neighbours. As a result, another sequence will be obtained, but values will logically differ from the initial ones. Its corresponding scheme is shown in Figure 2-4:

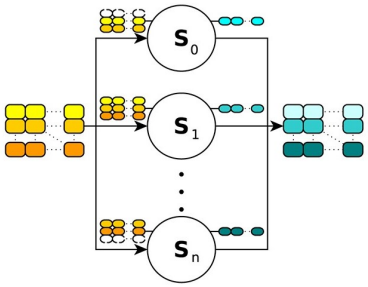


Figure 2-4: *Stencil pattern scheme [9].*

2.1.2 Task parallel patterns

Now, we present the Task Parallel Pattern strategy. This type of parallelism refers to a pair of entities instead of just one. In this way, each actor (task) operates over a different branch (from the same start point) in parallel, in such a way that the output of one of them can never reach the input of the other one. Both tasks can then communicate each other by defining a task dependency graph. The only Task Parallel Pattern we highlight is the following:

- **Divide and Conquer:** this pattern is a well known and used approach when dealing with problems that may be complex at the first sight. It solves a particular problem by means of partitioning it in two or more sub-problems of the same type. The solutions of each subproblem can be afterwards joined together so that they can give us the solution of the initially treated problem. Main elements in this pattern are:

  - *Divider:* it is the function in charge of splitting the original problem into several and small subproblems.
  - *Solver:* it starts working from one of the generated subproblems and finds a solution to it, which is going to be sent to the combiner.

– *Combiner:* it gets together the pairs of received solutions and it aggregates them all in order to give a final solution to the problem.

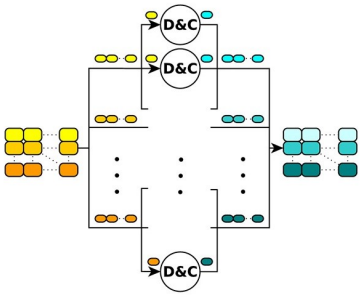


Figure 2-5: *Divide and Conquer pattern scheme [9].*

Important to mention that both data and task patterns functions must be pure, that is, they have to operate only making use of the specified input parameters and not turning to any other element outside them.

2.1.3 Stream parallel patterns

Finally, we describe the Stream parallelism strategy. It deals with the processing of some given data workflow in parallel. The data stream computation is partitioned into several tasks, and each task acts over the elements one by one (if a sequential task is applied) or in a multiple way (parallel task). As a difference from Data Parallelism, in this case, we sometimes do not have beforehand the input data stream, elements are generated and available as time passes.

From now on until the end of this subsection, the set of all Stream Parallel Patterns will be explained. *Stream* means that a continuous flow of items is going to be processed. Stream parallel patterns can be considered as the followings:

- **Pipeline:** this pattern receives a concurrent data workflow that will be processed over the different stages in the pipeline. Each stage is produced by its preceding stage, and it sends its result to the next stage until reaching the last one, where final result will be got. Key elements in this pattern are:

  – *Generator:* this operator produces the set of data items that will be treated along the different stages.

  – *Transformer:* it applies any transformation to each of the data items being produced by the generator.

This pattern is commonly used since it acts as the primary pattern of other patterns that are going to be explained below. That is why the following patterns are known as *composable streaming patterns*, and since they all depend on the pipeline one, they must be always composed with it. The basic structure of the pipeline pattern is represented in Figure 2-6:

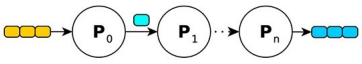


Figure 2-6: *Pipeline pattern scheme [9].*

- **Farm:** it consists on the application of a single operation over each of the items received in the stream (actually the pipeline) in one stage. It is needed to specify the cardinality of this pattern (number of replicas that are going to process concurrently the given operation), and the *transformation* operation carried out. Therefore, this operation will be parallel applied to each item in the input stream and an output value will be returned. It is important to say that the performed operations must be independent of each other, since in any other case, this pattern can not be applied. The graphical representation of the farm pattern is shown in Figure 2-7:

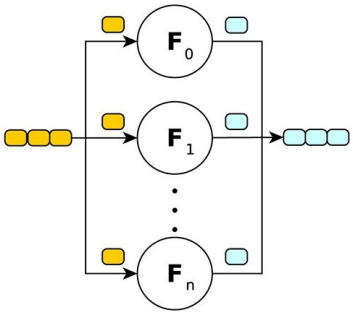


Figure 2-7: *Farm pattern scheme [9].*

- **Stream Filter:** the main element of this pattern is the *predicate*. A *predicate* has two possible values as a result, true or false, so that when applied will give as a result a *boolean* answer that will determine whether the processed element has to pass to the next pipeline stage or not (only the ones accepted will be sent to the output). We can see in Figure 2-8 how only those items that fulfill the predicate are sent to the output stream.

  As a simple example, consider a list of items (words) that we would like to apply a filter to them in such a way that we only want to keep the words that have a minimum length. In that case, a stream filter could be easily applied.

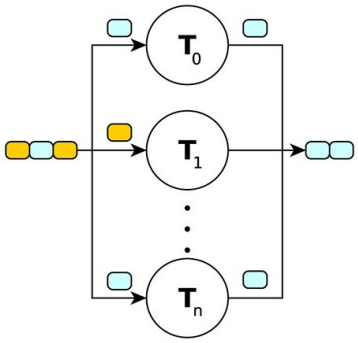


Figure 2-8: *Filter pattern scheme [9].*

- **Stream Reduction:** regarding streaming reduction, this pattern makes use of the aforementioned reduction operation over a set of streaming data set. Each item is processed by means of this operator, getting a new output as a result. It is composed of the following parts:

  - *Window size:* size of the aggregation window, that is, the specified chunk size from the total sequence.
  - *Offset:* distance between two consecutive windows.
  - *Size:* indicates the number of elements initially generated.
  - *Combiner:* this operation gets together a pair of values into a single one, in such a way that each time we are getting closer to the final result.

- **Stream Iteration:** this pattern allows performing inner loops within a data workflow. This pattern applies continually a function over a same item until the predicate is satisfied. Hence, this pattern is composed by a *transformation* operation over the present elements, and its associated predicate. Once the predicate is met, the result is sent to the output stream.

At this point, we are now able to differentiate between 10 different types of parallel patterns, each one belonging to a different type of parallel programming model (Data,Task or Streaming). These basic structures can easily provide parallelism in our applications in a very flexible and direct way.

Concretely, the GrPPI framework has been selected in order to test the profiler and event-tracing modules. Afterwards, we will describe some further characteristics about his library. Besides, in Chapter 5, a set of real use cases from this framework will be provided to give a more detailed idea about all these concepts.

2.2 Parallel pattern-based programming frameworks

Parallel programming frameworks are a way to aid programmers to implement their parallel applications. However, the difficult part is trying to reuse a specific parallel model into another similar one, since some expertise and a deep knowledge on the area is needed. This gets harder since there are not so many high-level parallel pattern abstractions, so complexity when developing parallel applications also increases [9].

In this section, four different parallel programming frameworks are explained in detail. Before going deeper into the implementation of the proposed solution in the following chapters, it is important to know to which framework has our solution been applied, so we will briefly explain in Section 2.2.1 some aspects related to the environment where this work has been developed. Afterwards, we present other similar approaches to ours that work over the same scope in order to identify the (dis) similarities among them.

2.2.1 GrPPI

The Generic Reusable Parallel Pattern Interface (GrPPI) is the framework over which we have implemented and tested our performance analysis tool, this is why it is explained the first one.

GrPPI is an open source generic and reusable parallel pattern programming interface developed at University Carlos III of Madrid. As its formal definition says, it is a reusable platform that gives support to both data-intensive C++ applications and streaming processing [9], since it supports other existing parallel programming frameworks as back ends such as C++ threads, OpenMP and Intel TBB. It acts like an envelope containing those three architectures into just one single platform, so this can help us to solve the reusability problem we suggested before in Chapter 1.

For this reason, this parallel programming interface offers quite flexibility so that users can switch from any of the aforementioned frameworks to another in order to execute their applications in an easy way. Users only have to change the execution policy inside their applications. Hence, this required expertise before mentioned is quite reduced when using GrPPI, since this platform lets us encapsulate the code in such a way that users do not need to worry about using common parallel programming frameworks directly. Definitely, GrPPI is an innovative solution for programmers (at not so much high level of experience), to implement their applications in a flexible and easier way, without being concerned about parallelization mechanisms, data races, correctness, etc.

Say that all the previous parallel patterns, both data and streaming ones, are exactly those that this parallel framework supports and over which the evaluation process has been carried out.

One clarification regarding these patterns is that this library is currently being modified by researchers at the university in order to provide more flexibility to users, and more freedom to compose these patterns and build more complex structures. For instance, it has been worked on composable structures such that farm of pipelines, pipelines of farms, etc. but this version has not been published yet.

Next, we will show two simple examples in order to see how can we make use of these patterns inside our parallel applications. Listing 2.1 shows a data parallel pattern example whereas Listing 2.2 shows a streaming parallel pattern:

Listing 2.1: *Example header of the reduce data pattern.*

```
1 grppi::reduce(ex, begin(v), end(v), 0L,
2     [ ](auto x, auto y) { return x+y; }
3 );
```

Listing 2.2: *Example header of the farm streaming pattern.*

```
1 grppi::pipeline(ex, // pipeline
2     [&]() -> optional<string> {
3       //...
4     },
5     grppi::farm(4, // farm
6       [ ](auto word) {
7         // ...
8       })
9 );
```

Next, as argued before, we will explain several similar approaches that also work with these parallel patterns, and we will state which ones of the before explained patterns in Section 2.1 do they support.

2.2.2 FastFlow

Another open-source parallel programming framework is Fastflow (FF). This framework was developed at the Departments of Computer Science of the Universities of Pisa and Torino, Italy [10].

This open-source framework was developed because of several reasons. One of them has to do with computer architectures. Nowadays, manufacturers are in favour of pending multi-core systems so that programmers can write more efficient and portable code. However, these architectures are not fully exploited as expected, and consequently, these systems are still viewed as unfamiliar to programmers from the HPC point of view. Apart from that, there is a need to develop mechanisms that allow machines a perfect concurrent access to the shared-memory of the system. Besides, regarding this memory, we

also need such environments that perfectly hide the complexity offered to developers [11].

For this purpose, FastFlow comes up as a solution to address the former needs and to provide users with a suitable parallel pattern environment that eases the parallel programming while providing efficiency (coming from their optimizations regarding basic communication mechanisms and its layered design explained below) [10].

FastFlow bases itself in a set of basic principles that are going to be briefly explained [11]:

- *Layered Design* - FastFlow aims for a structural layer design in such a way that different layers of progressive abstraction are given to the user to offer portability, extensibility and performance. The first layer is the core of the framework, which provides several techniques to efficiently support fine-grain parallelism (for instance, FIFO queues). Above this layer, we have an extension of the previous one by means of adding synchronization mechanisms and data flows over these queues, also providing less overhead.
- *Efficiency of Base Mechanisms* - FastFlow is internally based on Single-Producer-Single-Consumer (SPSC) queues that serve as base mechanisms over which everything is assembled. Thanks to these queues, programmers can build networks between a batch of threads that work over a concrete data while communicating with each other.
- *Stream parallelism* - this is the pattern-based type through which Fastflow develops its parallel patterns, although this does not mean that the rest of parallelism models are not being implemented.

  Any program based on data streaming can be represented as a graph whose edges refer to the stages of the stream. Any stage can read the data from the input stream, apply some computation to it and finally deliver the modified data to the next stage.
- *Parallel Design Patterns* - refers to the available parallel pattern catalog. In this context, this term refers to a set of different *algorithmic skeletons* structures that can derive the whole set of available parallel patterns. In that sense, FastFlow offers flexibility in such a way that patterns can be freely modelled in different heterogeneous scenarios in order to exploit parallelism.

Regarding the model it proposes, say that it has a structural shape. That is, the set of parallel patterns it offers allows the user to compose them as independent structures so that more complex and customizable structures can be built *algorithmic skeletons*.

After describing the basics of this parallel framework, we need to say that from all the parallel patterns established in Section 2.1, Fastflow supports the followings: *pipeline*, *farm*, *map*, and several complex versions regarding the *pipeline* which allows the stream data to be fed itself with the output obtained results.

2.2.3 SkePU

SkePU is a multi-backend skeleton programming framework for multi-core CPU and multi- GPU systems. As its name says, it is based on skeleton structures as FastFlow that can be modified as the user wishes. Several versions have been developed over time. Improvements from the first version were joined together and know they are working on SkePU2 [12].

This open-source framework offers both data-parallel and task-parallel skeletons. Due to a change from version to version, some of these patterns have been removed or modified [13], [14]. For instance, *Map Array* and *Generate* skeletons have been removed in favour of getting a generalized *Map*, and the *Call* skeleton has been also added in this second version. Therefore, the current used patterns are *Map*, *Map Overlap*, *Reduce*, *Scan*, *Farm* and *Call*, so we can observe how some of them coincides with the patterns list described in Section 2.1.

Regarding its design principles as with FastFlow, we can distinguish the following ones [13] (apart from the set of parallel patterns listed above):

- *Multiple back-ends and multi- GPU support* - with the above defined skeletons, each one can support executions for sequential C, OpenCL, CUDA, OpenMP and OpenCL over multi- GPU systems.
- *Tunable Context-Aware Implementation Selection* - from these back ends, SkePU offers the possibility to automatically choose between the fastest one for each skeleton call (map, reduce, etc.).
- *Smart Containers* - these concretely refer to vector and matrices containers. These containers act as a wrapper of operand elements when passing parameters inside the skeleton call so that they can automatically optimize communication, perform memory management and synchronize asynchronous skeleton.

SkePU was created in 2012 with the objective of improving portability between different parallel programming platforms. However, the C++ language has also suffered from different changes and improvements during this time. This has caused the generation of several limitations [12] that should be managed:

- *Type safety* - this aspect has to do with *macros* variables. The problem is that these are not type-safe since when using them, errors will not appear until reaching the run-time phase.
- *Flexibility* - SkePU has increased and allows the definition of a higher amount of *macros* just mentioned above. The problem arises because this increasing use of macros also implies modifying the run-time system. For this reason, more flexibility is preferred to be given in real-world applications.

- *Optimization opportunity* - currently, some optimizations or specializations cannot be offered since the C preprocessor is being used for code transformation and consequently, this can not be possible.
- *Implementation verbosity* - there are available different skeleton configurations that can be applied whenever the user wants. They are defined separately but they can also be used in a combined way.

2.2.4 HPX

HPX is a general purpose C++ run-time system that currently makes use of two different already implemented versions from these two groups: Stellar HPX and Crest HPX. As the rest of frameworks, HPX emerges to fight the complexity and size of modern architectures [15].

Regarding Stellar Group, say that it is based on a new model of computation, ParallelX, which gives support to dynamic and adaptive resource management as well as scheduling [16]. HPX can work with parallel and distributed applications of any scale. Besides, it has extended its library so that remote asynchronous functions can also be used. HPX is quite easy to understand. Its API provides an interface that serves for the integration of multi-core multi-threaded heterogeneous architectures and user-level software applications [17]. One of the main design features of HPX is the *asynchrony of tasks*. It supports a set of asynchronous functions called "Actions", and a set of data constructs such as "Futures" and "Dataflows" are also requested due to these functions.

The second implementation, HPX-5 was developed by Center for Research in Extreme Scale Technology (CREST). Developed at Indiana University, new approaches regarding petascale/exascale computing are currently carried out so that supercomputers platforms can achieve the highest levels of performance. This model is considered as a reduction to practice of the Stellar Group model. It is composed of a set of different classes for local synchronization, a global address space, an active-message parcel transport and a thread scheduler. Besides, it is internally built over some data structures that try to minimize the overhead produced when dealing with shared-memory synchronization [16].

2.3 Profiling and tracing tools

Having analyzed several parallel programming frameworks, in this section we scan the different profiling and tracing tools (and libraries) available in order to apply them to those applications from the before explained parallel frameworks. As we said, if no parallelism is presented in applications, then these performance analysis mechanisms would not have too much importance and results would not be so compelling, hence the previous exaplanation of those frameworks.

2.3.1 Extrae and Paraver

Extrae

As Virtual Institute – High Productivity Supercomputing affirms [18], Extrae can be defined as a dynamic instrumentation package that tracks information from an application in order for then being able of generating traces. The information collected by Extrae may include entry and exit to the programming model run-time, hardware counters (PAPI), call stack reference, user functions, periodic samples and user events [18], so we can observe all the functionalities we can have when using it.

Extrae library was developed by the Barcelona Supercomputing Centre team. Extrae supports different programming models such as MPI, OpenMP, CUDA, OpenCL, pthreads, OmpSs, Java and Python. Besides, this tool makes use of different interposition mechanisms to inject timestamps to our code so as to gather information regarding the application performance [19]. These mechanisms are the following ones: Linker Preload (LD_PRELOAD), Dyninst, Additional instrumentation mechanisms and the API of Extrae. No matter which one is used since all of them have in mind the same target: obtaining performance metrics at some points in our applications for us to analyze it later on, and get some conclusions of our work.

In this project, it has been employed the last mechanism from the previous list, since, after carrying out a meticulous study of all of them, it has been considered that one to be the most convenient one. Besides, since there were others instrumentation packages (for instance TAU, explained later in 2.3.2), that also possessed an API for instrumenting, then, a comparison among all of them was performed for finally selecting the most appropriate one, which indeed has been the API of Extrae.

Regarding Extrae API, we can distinguish between two of them. The first one, the *basic* one, is focused on the elemental functionalities of this software. Among these functionalities, we can stand out the emission of events functions, source code tracking and be changing instrumentation mode. The second one, the *extended*, has been expanded from the basic one to provide more powerful calls. In this project, however, it has been sufficient for us the fact of using the simplest one [20].

One important aspect regarding this instrumentation package is that, in order to use it, it needs to be previously installed on your computer. Besides, this instrumentation process requires the program to be written in C or C++ programming languages.

Next, we will show two simple examples so that we can have a clear idea of how this library works. The following examples have been taken and exposed just to clarify the explanation, but this does not mean these are the only functions available, they are simply the most used ones in this project.

Listing 2.3: *For loop instrumentation with Extrae library [20].*

```
1   for (i = 1; i <= MAX_ITERS; i++){
2    // ...
3    Extrae_event(1000, i);
4    [rest of loop code here]
5    Extrae_event(1000, 0);
6    // ...
7   }
```

Listing 2.4: *Routines instrumentation with Extrae library [20].*

```
1  Extrae_init();
2   // ...
3
4   void routine1(){
5    Extrae_event(6000019, 1);
6    [code of routine1...]
7    Extrae_event(6000019, 0);
8   }
9
10  void routine2(){
11   Extrae_event(6000019, 2);
12   [code of routine2...]
13   Extrae_event(6000019, 0);
14  }
15
16  // ...
17 Extrae_fini();
```

Listing 2.3 shows how we can instrument a `for` loop with Extrae. Lines 3 and 5 show the required functions to instrument it. Notice that the point is surrounding the lines that are the most significant to us. As we are in a loop, we would like to track every single iteration, that is why we place that pair of functions inside the loop, at the beginning and at the end of it. The first parameter in the function `Extrae_event()` means the event identifier or type (which can be any number specified beforehand by us), while the second one refers to the value that event has, starting in this case by the value of `i` and always finishing with value 0 (which marks the end), as it can be identified in line 5.

However, Listing 2.4 could apparently be a more difficult example, but in fact, it does almost the same as the previous one. The most relevant difference is the pair of functions at the beginning and end of that algorithm. This just initializes the library and finalizes it when desired. Regarding the `Extrae_event` functions, these are the same as above but as we are now inside a function, the second parameter is an explicit value and not an iterator. In the same way, final values need to be zero in order to ease the analysis to Paraver tool.

Extrae can be configured manually by users through a specific XML file. In this way, thanks to an environment variable, we can activate or deactivate it when desired [20]. Several file configuration models are available for users (starting from the most basic one). Inside them, users can *enable* or *disable* any performance metric and customize it as required.

This configuration file is quite important in the sense that depending on the metrics configuration established, then, Paraver (a performance analysis tool still to explain) will allow users to visualize the different metrics of the application according to what he established on that file.

Paraver

Extrae uses Paraver as its trace visualization tool. It is a very powerful and easy-to-use performance tool developed in order to inspect the application behaviour (in both quantitative and qualitative way), and focus to those points that can be optimized in some way [20].

By a widely amount of useful information this program offers, the user can identify possible bottlenecks in parallel executions. As a brief summary, we list some of the most important functionalities this tool supports [20]:

1. Detailed quantitative/qualitative analysis of program performance.
2. Concurrent comparative analysis of several traces.
3. Fast analysis of very large traces.
4. Support for mixed message passing and shared memory.
5. Customizable semantics of the visualized information.

Those can be combined with the following modules that it also offers: tracing for sequential and parallel applications, tracing for multiprogrammed environments and tracing for hardware counters.

Going a bit deeper into the explanation, it is important to mention that this program needs as input a trace file (apart from making use of the configuration one). Its extension is `.prv` and it is generated once we run our application instrumented with Extrae library. Later on, we will identify the specific contents this trace file holds. An advance is given in Figures 2-14, 2-15, 2-16. Once the file has been loaded, we can start customizing our trace by means of a large amount of available configurations files, that allows us changing the colours, the units, the metrics, etc. Paraver is very flexible in the sense that it perfectly works with any trace size and with traces coming from different environments.

Next,we show a general scheme of how this tool is built. As we can identify in 2-9, Paraver is divided into three different modules [21]. What it does first is filtering the set of tracefiles records (record types explained afterwards), by the ones we are interested in. In our case, we focus on the ones regarding event emissions. That is, from the initial trace file, this module gives a partial view of the former one.

Then, the second module, the *semantic* one, is in charge of taking those filtered tracefiles and interpreting them (by giving sense to the computed tracefiles values). Thus, a transformation to time-dependent values can be performed, and consequently, passed this tracefile to the following and last module, the *representation* one.

The objective of the *representation* module is trying to show all the previous information in different ways by means of a graphical display. Paraver does not only give a graphical feedback of the collected data (*visualization submodule*), but also in a *textual* and *analytic* (providing quantitative data) way as it can be observed in Figure 2-9, since we need users to be informed about everything at every moment [22].

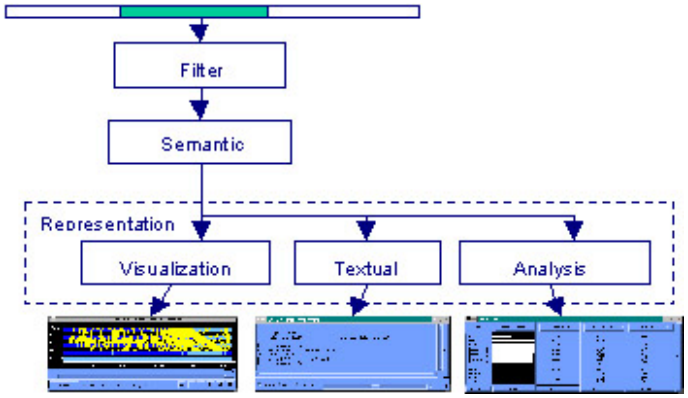


Figure 2-9: *Paraver structure scheme [21].*

Within the visualization module, Paraver can also differentiate between a set of three main elements (or record types) within a single trace file: *state records*, *events* and *communications with timestamps* [20]. Those elements are the ones used to build and generate the final graphical trace. In Figure 2-9, this trace file that we are talking about would correspond with the first horizontal bar placed on top of this picture.

Below, we provide a description of each of the mentioned object record types [21], [23], as they compose the basis for understanding the program behaviour.

- **State records,** represent a state value for one thread during different time intervals.
- **Event records,** represent punctual time points (events) in the code. They are encoded by two parameters: the first one refers to the type, and the second one to the value. As stated before, these objects have been the ones used in this project.

- **Communication records,** relate the logical and physical communication process between two points (representing both sender and receiver) over time. Indeed, this communication takes place between two different threads that at some point they need to exchange some information.

We introduce the Paraver Object Model. Paraver works with two different an orthogonal object models as exposed in [22], the Process Model and the Resource Model. Regarding the first one, this is composed by three different abstraction levels. From the outermost to the innermost: application, process and thread objects, as depicted in figure 2-10. Each parallel application (namely *APPL* in the picture), is composed by a set of different processes (*TASKS*). Subsequently, each task can be assigned to one (sequential application) or more than one threads (parallel application). On top of these layers, we have the *WORKLOAD*, which refers to the set of applications that can be executed at the same time making use of the same resources.

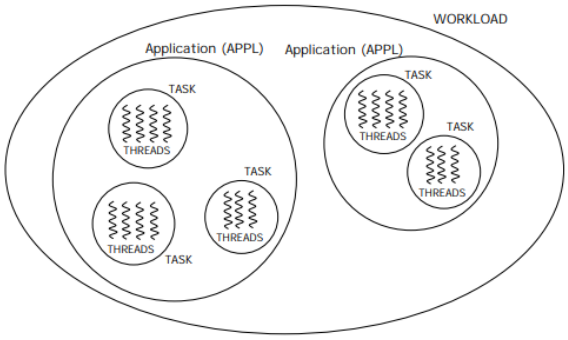


Figure 2-10: *Paraver Process Model [22].*

However, the Resource Model layered is different from the previous one (see picture 2-11). Here, we have the physical *SYSTEM* throughout our applications (*TASKS*) are going to be run. Specifically, any *TASK* can be mapped to any *NODE*, which is precisely the next layer defined in this model. Inside its node, one or more processors can be available depending on the machine resources.

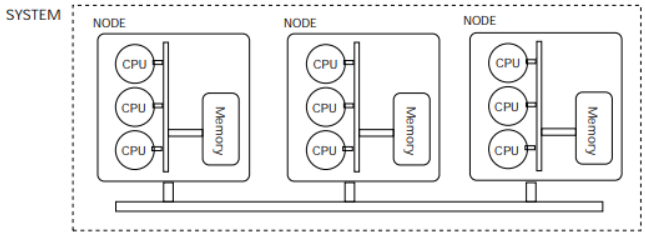


Figure 2-11: *Paraver Resource Model [22].*

Paraver Trace File Format Once studied the three basic record types in a trace and the two models this tool offers, we now show how a Paraver trace is presented, i.e., format used to store record objects in the `.prv` file. Next, we will show different simple pictures describing the appearance of this file once executed our application.

This file is composed of two main parts. The header (just representing the first line of the file), and the body, which constitutes basically the rest of lines. The next picture shows better this explanation:

```
#Paraver (22/05/01 at 16:20):1021312:2(16,16):1:2(1:1,1:2)
1:1:1:1:1:0:100:4
1:2:1:2:1:0:200:4
1:1:1:1:1:100:300:1
1:1:1:1:1:200:500:4
3:1:1:1:1:300:325:2:1:2:1:200:330:10:3000
2:1:1:1:1:300:60000000:1
```

Figure 2-12: *Paraver trace file format (.prv) [22].*

Figure 2-12, the line starting by `#Paraver` refers to the header and the rest to the body. Referring to the body lines, say that each one belongs to a different record type, and within a record, information is stored in different fields separated by ":". In order to understand the meaning of all those numbers (fields), we show it through a graphical explanation:

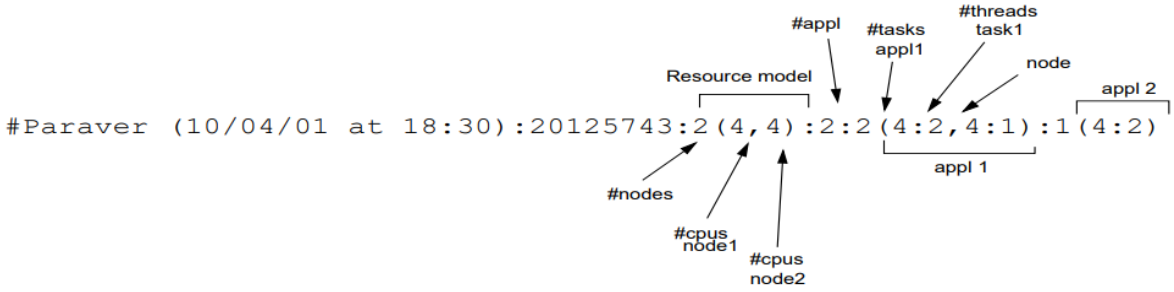


Figure 2-13: *Header fields meaning in a Paraver trace [22].*

```
1:cpu_id:appl_id:task_id:thread_id:begin_time:end_time:state
```

Figure 2-14: *State record representation in a Paraver trace [22].*

```
2:cpu_id:appl_id:task_id:thread_id:time:event_type:event_value
```

Figure 2-15: *Event record representation in a Paraver trace [22].*

```
3:object_send:lsend:psend:object_recv:lrecv:precv:size:tag
```

Figure 2-16: *Communication record representation in a Paraver trace [22].*

2.3.2 TAU and Jumpshot

Tuning and Analysis Utilities (TAU), is another performance technology developed at Oregon University and used to analyze complex parallel systems. Programming languages that this software can instrument include Fortran, C, C++, UPC, Java and Python [24].

This performance framework supports a profiling and tracing toolkit that allow developers discover the hot spots in their code. It also presents some instrumentor tool that automatically instruments the code instead of doing manually.

As with Extrae software, which uses Paraver as trace visualization tool, TAU makes use of `paraprof` to visualize profiles and Jumpshot to visualize traces. Thanks to its graphical interface, we can also observe the results obtained once an application has been executed and it will help the user to detect possible bottlenecks or even correlations and dependencies between some parts of the code [25]. Additionally, among all the functionalities this framework can offer, we can underline the possibility to generate a trace file that can be afterwards analyzed with Paraver, Vampir or Jumpshot (both explained below).

Several goals [26] were present when developing this toolkit. The main ones are:

- Support from multiple parallel programming paradigms.
- Allowing a flexible and configurable performance analysis.
- Supporting multi-level performance instrumentation (manually and automatically).
- Providing a widely-ported parallel profiling system.

Concerning about its computational model [26], say that this software is based on three main elements: **node**, which refers to the physical machine, **context**, which refers to the virtual memory space inside that node and **thread**, which are the workers performing tasks.

Apart from that, it is interesting to study the system architecture that it presents. As a whole, each different part from its structure is associated to one different phase [26]. In this way, we can difference between:

- **Instrumentation:** performed by means of the previously mentioned API. There exist several types of instrumentation techniques depending on the list of predefined requirements we have.
- **Measurement:** by both profiling or tracing approaches, TAU supports a robust timing and hardware performance counters such as PAPI ones. We can also obtain information about performance events, use a timer library to get measures at a high-resolution level, etc.
- **Analysis and Visualization:** as said before, we can use some tools such as `paraprof`, Jumpshot or Vampir. These will help us to carry out trace measurements and future analysis of results.

Again, similar to Extrae, TAU presents an API composed of a set of functions that can be freely inserted in our code. These functions will be the ones in charge of recording all the information of the program flow during run-time. We can distinguish between two different API's: the basic one and the extended, the latest compatible with C and Fortran programming languages. A simple example of its use will be given in Listing 2.5:

Listing 2.5: *Instrumentation process with TAU library [27].*

```
1   void routine1(){
2    TAU_PHASE_CREATE_STATIC(t2, ‘‘IO Phase’’, ‘‘ ’’, TAU_USER);
3     TAU_PHASE_START(t2);
4     [code of routine1...]
5     TAU_PHASE_STOP(t2);
6    return 0;
7   }
```

Regarding the Jumpshot tool, say that it is a Java-based tool capable of generating some specific logfiles when this type of programs are executed [28]. From those scalable logfiles, users are able to visualize their program inner flow and behaviour, as well as inner communications between active threads. Different versions have been developed over time, each one providing much more functionalities.

2.3.3 Vampir

This is the other visualization tool TAU framework supports. It is used for both profiling and tracing analysis purposes. As with the previous performance analysis tools, Vampir lets the user dig up in detail all the dynamic processes that massively take over parallel systems. Again, say that more complex applications are being developed each time, and we need reliable tools so that we can get accurate and valid information from programs inner behaviour.

Vampir is able to analyze the generated trace files, but these ones need to be created and available beforehand. However, this is only possible by making use of a working monitoring system. In some

computers, this facility was already integrated, but in Linux systems nowadays, it needs to be added in a separated way [29].

We are concretely referring to VampirTrace. This is an open source software able of generating Open Trace Format (OTF). This trace has a well-defined and concrete format as a result of using several public domains libraries for reading and writing [29]. After its creation, the Vampir tool is ready to analyze it and display it in a graphical way. VampirTrace library also allows MPI communications between processes and special events from programs. Notice that this trace file generation is obtained once the code has been instrumented. In this case, VampirTrace gives the possibility to perform that manually or in an automatic way.

2.3.4 perf

`perf` is a performance analyzer tool, formerly called *Performance Counters for Linux*. This tool is available in Linux systems from versions 2.6.31 on. This performance monitor is based on two major subsystems [30]. The first one, the `perf` event subsystem. Through it, a set of different commands can be used to obtain the desired results. All of them precede a userspace controlling utility called `perf`, which can be followed by one of the following subcommands [31] ready to use in the command line interface this tools offers (the most important ones):

- *stat*: it tracks all measures taken into account from one point in time to another.
- *top*: to show a dynamic view of some functions.
- *record*: it measures and saves the information obtained from a single program. It generates a file.
- *report*: it analyzes the previous generated file.
- *annotate*: it annotates sources.
- *sched*: it measures the actions performed by the scheduler.
- *list*: it list the available events.

The second subsystem defines a collection of userspace tools that aggregate, display and analyze that collected data. At the beginning of its development, *perf* was directed to work with the performance counters Linux subsystem, but recently, it has enhanced some functionalities. Some of them are: support for instrumentation CPU performance counters, tracepoints, kprobes, and uprobes [32].

Regarding performance counters, say that they are CPU registers that count hardware events and that forms a basis for profiling the applications. Among them, we can have counters that track the instructions executed, the cache-misses suffered or even some mispredicted branches. On the other side,

tracepoints are no more than points (by previous instrumentation) placed at strategic locations in our code such that information is generated while execution. One very interesting advantage of this profiler is that users do not need to have instrumented your code beforehand, hence, it is very precise and works really fast.

2.3.5 Gprof

`Gprof`, also known as GNU profiler, is a performance analysis tool for Linux systems that is able to analyze our input program and determine which parts in there are the most time-consuming. Besides, potential bugs can also be detected if behaviour throughout a set of predefined metrics is not the expected one.

The process is similar to the previous ones. We first need to compile our program and link it with profiling enabled. Execution will be done as in a usual way. The difference would be that more time will be required to plot the output since the profiler would be collecting all the data and writing it to a profile data, concretely, a `gmon.out` file [33].

At this point, we are now able to run that generated file with `gprof`, since it is able to interpret it. The following command allows us to do it [33]:

```
gprof options [executable-file [profile-data-files...]]  [> outfile]
```

Gprof can return different output measures [33]. Among all, the following are the most common ones:

- **Flat Profile:** this output plots the time interval spent on each function in the program.
- **Call Graph:** by means of a graph, we can identify dependencies between functions within a program, that is, we will be able to know which function called which others.
- **Line-by-Line:** it analyzes independent lines from a program.
- **Annotated Source:** this approach displays in a list some source code that is recorded with some execution labels.

2.3.6 Scalasca

Modern architectures support bigger numbers of processors since nowadays, there is a growing demand for more complex applications. Consequently to this, applications require higher degrees of parallelism and therefore, its behaviour gets even harder to understand [34]. As a solution to this, several organiza-

tions in Germany developed a tool that eases these problems. It scales very well any type of platform, not only large-scale systems but also small ones.

Scalasca is another profiling optimization tool. It measures the performance of codes in a wide variety of HPC platforms [35]. It is very intuitive, so this is a positive point for beginners that are about to use it. The measure and analysis of the parallel behaviour of applications by means of Scalasca are mainly focused on OpenMP, MPI frameworks and hybrid applications that combine both frameworks [35] but we can also use it with any parallel application that does not refer to that (there is even analysis support for sequential application executions).

The first thing to do when using Scalasca is carrying out a previous instrumentation of our code. Afterwards, when running the target application into a parallel machine, the user would be able to choose between the generation of a summary report and/or the generation of event-trace records that can be later visualized with an appropriate tool, for instance, TAU or `paraprof` [36].

2.3.7 Score-P

Scalable Performance Measurement Infrastructure for Parallel Codes (Score-P), is also a very powerful tool able to understand the inner behaviour of an application during run-time. It was created in the German BMBF project SILC and the US DOE project PRIMA [37] and is mainly designed to parallel applications executed on top of HPC platforms [38].

In this way, this software supports different performance analysis: sampling, instrumentation, profiling and tracing. Score-P offers a quite flexible measurement process of applications without the necessity of compiling more than once the application. In addition to that, it supports different programming paradigms such as MPI, OpenMP, CUDA, OpenCL, etc.

Six basic steps [38] need to be carried out in order to obtain some performance feedback from our parallel application:

1. Instrumentation of the application with Score-P.
2. Perform a measurement run with profiling enabled.
3. Perform an analysis on profile data.
4. Definition of a suitable filter.
5. Perform a measurement run with that previously selected filter.
6. Make use of Vampir performance visualization tool.

To finish this section, we also need to mention hardware counters. For many years, different techniques have been used to track performance data. Unfortunately, those techniques were not precise at all or the timers used offered a poor resolution, so vague information could be obtained from them [39].

Nowadays, we can affirm that almost current machines support hardware counters. Thanks to them, users can precisely obtain performance reports from applications (and also detection of critical parts, bottlenecks, etc.). The problem is that there is not enough documentation for developers about this topic and this produces certain ignorance. Performance Application Programming Interface (PAPI) [39] is a specification of a cross-platform providing hardware counters as a solution to that problem. PAPI library consists of a set of standard events together with several high or low-level routines accessible to users, so with this facility, they can measure the applications performance in a much more realistic way.

2.4 Parallel programming models

In this section, we are going to discuss several current parallel architectures. Then, we will explain some characteristics regarding current architectures to have an idea of which new features in architectures have been developed over time.

2.4.1 POSIX threads

POSIX thread library is a standard based on the C/C++ APIs [40]. This technique is quite used in multicore systems where the process flow can be scheduled to run on another processor, and consequently, we can gain speed through this distributed processing approach.

C++ language gives support for standard-library threads by means of POSIX threads. Threads is a execution modality that allows the program to execute across several multiprocessors [41]. Below, we describe the main characteristics of this approach.

As a difference to a process, threads share the address space. Therefore, there could be some situations in which the pool of launched threads access the data at the same time (also named concurrently) causing unexpected results. To avoid this, several mechanisms are present to ease this problem. We can stand out *mutexes*, which define mutual exclusion regions in the code and assure exclusive access to that critical section, leaving blocked the rest of threads. This can be performed by two special directives that protect that section from the rest of threads and that are associated to one *mutex*. These are *lock( )* and *unlock( )* directives [41].

2.4.2 Intel TBB

Performance of applications is now one of the most important challenges in computing field. Two main optimization targets are defined to accomplish this performance improvement: graphics and low-level instructions [42].

Apart from that, the increasing number of multi-processors in computers has caused the consumer market the development of different optimizations techniques. For this purpose, Intel developed the Threading Building Blocks library to achieve that objective.

The main advantages that [43] that TBB offers are:

- High parallelization intensive work by making use of C++ standard.
- A more comprehensive solution for parallel programming applications development.
- Highly portable and affordable for future-proof scalability.

Intel TBB [43] differs from the rest of threading packages in the following aspects:

- It makes possible to determine a logical parallelism into threads instead of using directly threads. As a consequence, we can perform a more efficient use of processor resources.
- It mainly focuses on scalability aspects based on data-parallel programming.
- It allows heterogeneous computing.

Some areas where TBB multi-threading is now applied are, for instance: artificial intelligence and automation, seismic exploration, medical applications or energy resource exploration.

2.4.3 OpenMP

This section presents the OpenMP parallel model. This architecture, as said in [44], is a portable, scalable programming model for parallel approaches on shared memory platforms.

Its main goal is trying to promote parallelism in a simple and efficient way in programs for shared memory machines. By the existence of a set of threads, we can distribute tasks along them (basing on the fork-join model) in this shared memory-based platforms.

C, C++ and Fortran are the programming languages that OpenMP API supports. This API, developed by a group of hardware vendors, offers users the possibility to convert their serial applications into parallel ones in an easy way.

The main relevant characteristic from this architecture is the use of compiler directives (`#pragma`) in charge of exploiting parallelism. These directives are simply specific instructions placed at some points in the code, so that the block of code they envelop would behave in a parallelized mode.

These directives [44] can be classified into the following categories:

1. **Parallel regions:** as the name defines, these directives refer to some block of code that is going to be executed in parallel by a group of threads. Within that group, we need to distinguish between the master thread, as it is the first one reaching that directive in the code.

2. **Worksharing:** this directive is similar to the previous one but here there is no creation of a bunch of threads in the parallel region: they just divide or share the work into all the participants in that pool in such a way that each member carries one task. A typical example of this directive can be observed in the parallelization of a for loop.

3. **Data Environment:** it comprises the set of data-sharing attributes that define the environment of some task

4. **Synchronization:** these constructs bring a certain order through which a pool of threads will operate to complete an assigned task. This can be guaranteed, for instance, by a barrier, which is a point in the code where all threads must wait until the arrival of the lowest thread at that point, so thanks to it, we can assure the termination of some task.

Among these advantages [45] this programming model presents, we can underline two of them: OpenMP requires less code modification than MPI for instance and also the prevalence of multi-core systems. On the other side, however, there are also some limitations regarding the number of processors that are available on a single machine and we need to be sure that our machine compiler is compatible with this architecture.

2.5 Summary

In this chapter, two different topics have been studied, on one side, commonly used parallel programming frameworks (2.2) together with the most used parallel patterns, and on the other side, different nowadays profiling and tracing tools (2.3). The main goal is to develop a performance analysis environment on top of GrPPI. For that reason, we have explained the different parallel programming models so that we can compare and have an idea of the basic functionalities and capabilities developed by other libraries with respect to the proposed in this work.

Regarding the parallel frameworks described, it can be checked that not only Fastflow but Skepu work on top of these pattern-based structures, and that they are almost similar to the programming framework already studied (GrPPI). Therefore, we conclude that there is a real need to face this large-treatment of data volumes and that parallel patterns come out as a solution to satisfy this problem.

Among the different approaches presented, we have considered choosing the GrPPI interface in our project as the programming parallel framework due to its flexibility. This interface allows users to change from one parallel back end to another in a very simple way, so in that sense, it offers more flexibility than other approaches. Besides, regarding the profiling submodule development, we can also find some advantages with respect to the tracing submodule developed with Extrae library: in the tracing submodule, we have installed Extrae in our machine so that we could obtain trace files, however, with our solution regarding the profiling submodule, we did not make use of any additional (and beforehand installed) software to obtain a report summary, since we implemented it from scratch. This can be a benefit in the sense that we do not depend on any other tool that could occupy large amounts of memory in our machines, and you do not need to have some expertise on that tool since you just need to run the program to obtain your statistical summary.

With respect to profiling and tracing tools already presented, the same situation takes place here. We have reviewed a series of performance tools that target the same goal: offering the user some kind of analysis and visualization tool that helps them understand the complex behaviour of parallel applications.

At the beginning of the project, we decided to use both Extrae and TAU libraries to instrument our code and obtain further trace results. However, due to time limitations, we have reduced the project extension to Extrae library along with the usage of the Paraver tool to visualize the results. We do not disregard the rest of performance visualization tools, but after documenting ourselves about Paraver, it is a more intuitive and easy-to-use tool.

Chapter 3

System Architecture

This chapter presents the software development process carried out in this project regarding the design and analysis in a high-level abstraction. For that reason, Section 3.1, we will provide the general capabilities and constraints of the project, as well as the potential user characteristics along with the set of requirement catalogue. From that analysis, in Section 3.2 will provide the complete system design by means of a set of software diagrams (components, relationships, etc.) that will help us to understand the system behaviour from different perspectives. From this last argument, we will provide more specifically the user and system requirements in order to check the traceability between them.

3.1 Analysis

This section comprises all aspects related to the developed system itself. As in any software project, we need to capture and consider all user needs so we can focus the project according to what we have been told. Next, we will briefly describe the set of general capabilities and constraints affecting the project implementation, as well as the conditions and characteristics any user must have when testing our system. Finally, we will finish by showing the complete requirements catalogue obtained from the elicitation process.

3.1.1 User characteristics

This performance analysis environment will be directed to both ordinary users and developers working at professional high-computation levels, including enterprises. However, for potential users, it is true that they will need to show some basic expertise in this area. First, they should have enough skills in pro-

gramming, and secondly, they need to have some previous knowledge about the parallel computations. In any other situation, users will not be able to exploit the purpose of the system we propose.

Focusing on the profiling submodule, users will need to have some theoretical background on statistics, since the report summary given as a result contains some statistical metrics regarding the program execution. Concerning the tracing submodule, we would advise users testing this system to have used Paraver software, as it is the performance visualization tool through which our system works. Anyway, this tool is not so complex to understand, and if users know about computer architecture, they will not face problems at all when using it, since they would be familiarized with the features and technical terms that Paraver makes reference to. However, once having a general view of the results it offers, it would be a good idea for users to read some manuals and tutorials about Paraver in order to discover further features, since it is a very powerful performance tool.

3.1.2 User requirements

User requirements comprise needs and desires from the user point of view. At the end, our aim is satisfying potential users with our product, and this can only be done by having in mind at every moment what they have asked us to develop in the elicitation process. By means of our proposed solution, we need to assure the covering of those specific needs.

User requirements can be divided into two main categories: *capabilities* and *constraints*. The requirements representing capabilities refer to those actions that the user can perform within our system, as the name says, what the user is capable to do. On the other hand, constraints requirements make reference to the user constraints when solving a problem, what things he can not do or can be done with some limitations according to the product design.

From now on, we list , the full user requirements catalogue, including not only the *capabilities* ones but the *constraints* ones. Requirements, whatever the type it is, must follow some guidelines to fully address all aspects related to the software product to develop: functionalities, restrictions, external interfaces, etc. Besides, requirements should fulfill the following characteristics [46], that is, they should be:

1. **Verifiable:** it can be tested and evaluated in some way.
2. **Clear and Concise:** it specifies a single and concrete idea.
3. **Complete:** it contains all needed information so that you do no need to suppose further assumptions.
4. **Consistent:** the requirement does not repeat the same idea as any other requirement and does not conflict with them.

5. **Traceable:** each requirement can be traced to system functions.

6. **Viable:** it is helpful to build the complete system and affordable within the specified budget.

7. **Unambiguous:** it does not generate more than one interpretation.

Next, we provide the full set of user requirements specification (both types) using the template shown in Table 3.1. This template has been applied for each user requirement about *capabilities* and *constraints*.

UR-XX-YY	
Priority:	Low | Medium | High
Necessity:	Essential | Non-essential
Verifiability:	Low | Medium | High
Stability:	Stable | Unstable
Status:	Proposed | Verified | Validated | Rejected | Suspended
Definition:	Requirement description text

Table 3.1: *User Requirements Template Table.*

In Table 3.1, **XX** refers to the requirement type whose possible values are: *CA* (capability requirements) or *CO* (constraints requirements). **YY** is the requirement identification number, whose values comprise the range from 01 to 99. An example can be *UR-CA-05*, which refers to the fifth user capability requirement.

The following tables expose the user capabilities requirements:

UR-CA-01	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to compile his application.

Table 3.2: *User Capability Requirements UR-CA-01.*

UR-CA-02	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to execute his application.

Table 3.3: *User Capability Requirements UR-CA-02.*

UR-CA-03	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to obtain distinct statistical summary metrics.

Table 3.4: *User Capability Requirements UR-CA-03.*

UR-CA-04	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall obtain a execution summary of the different tasks.

Table 3.5: *User Capability Requirements UR-CA-04.*

UR-CA-05	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to use the profiling submodule.

Table 3.6: *User Capability Requirements UR-CA-05.*

UR-CA-06	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to use the tracing submodule.

Table 3.7: *User Capability Requirements UR-CA-06.*

UR-CA-07	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to obtain performance information about data patterns.

Table 3.8: *User Capability Requirements UR-CA-07.*

UR-CA-08	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	High
Status:	Stable
Definition:	The user shall be able to obtain performance information about streaming patterns.

Table 3.9: *User Capability Requirements UR-CA-08.*

The following tables show the constraints from the user perspective:

UR-CO-01	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	Stable
Status:	Validated
Definition:	The user shall run his application within GrPPI framework.

Table 3.10: *User Constraint Requirement UR-CO-01.*

UR-CO-02	
Priority:	High
Necessity:	Essential
Verifiability:	High
Stability:	Stable
Status:	Validated
Definition:	The user shall make use of C++ programming language.

Table 3.11: *User Constraint Requirement UR-CO-02.*

UR-CO-03	
Priority:	Medium
Necessity:	Essential
Verifiability:	High
Stability:	Stable
Status:	Validated
Definition:	The user shall specify the type of parallel programming framework used.

Table 3.12: *User Constraint Requirement UR-CO-03.*

3.1.3 System requirements

To this type of requirement belongs the set of requirements that describe what the system shall do to accomplish the previous necessities from users. In other words, the capabilities the system offers to users.

As with the user requirements, we can also distinguish between two subtypes: the *functional* requirements and the *non-functional* ones. The former refers to those that exhibit the functionalities of the system, what the system is able to do. Therefore, for considering that user needs are being satisfied, each capability use requirement must be associated (or covered) to at least one system functional requirement. The latest refers to the ones that establish the restrictions or limitations of the system, which also will restrict the actions from the user's point of view.

We will now show the template (Table 3.13) that has been applied for this other type of requirement.

SR-XX-YY	
Definition:	Requirement description text
Priority:	Low | Medium | High
Necessity:	Essential | Non-essential
Verifiability:	Low | Medium | High
Stability:	Stable | Unstable
Status:	Proposed | Verified | Validated | Rejected | Suspended
Origin:	UR-XX-YY

Table 3.13: *System Requirements Template Table.*

XX in *SR-XX-YY* refers to the requirement type and possible values are: FR (functional requirement) or NF (non-functional requirement), and **YY** is the requirement identification number, whose values comprises the range from 01 to 99. An example can be like *SR-FR-06*, which refers to the sixth system functional requirement. The origin field makes a reference to the user requirement that produced the creation of that system requirement (this field allows more than one user requirement).

The following tables show the system functional requirements:

SR-FR-01	
Definition:	The system shall let the user compile his application.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-01

Table 3.14: *System Functional Requirement SR-FR-01.*

SR-FR-02	
Definition:	The system shall let the user execute his application.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-02

Table 3.15: *System Functional Requirement SR-FR-02.*

SR-FR-03	
Definition:	The system shall notify the user in case some error appears while executing.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-02

Table 3.16: *System Functional Requirement SR-FR-03.*

SR-FR-04	
Definition:	The system shall calculate the accumulated mean.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.17: *System Functional Requirement SR-FR-04.*

SR-FR-05	
Definition:	The system shall calculate the accumulated mean maximum value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.18: *System Functional Requirement SR-FR-05.*

SR-FR-06	
Definition:	The system shall calculate the accumulated mean minimum value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.19: *System Functional Requirement SR-FR-06.*

SR-FR-07	
Definition:	The system shall calculate the standard deviation.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.20: *System Functional Requirement SR-FR-07.*

SR-FR-08	
Definition:	The system shall calculate the standard deviation minimum value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.21: *System Functional Requirement SR-FR-08.*

SR-FR-09	
Definition:	The system shall calculate the standard deviation maximum value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.22: *System Functional Requirement SR-FR-09.*

SR-FR-10	
Definition:	The system shall calculate the throughput value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.23: *System Functional Requirement SR-FR-10.*

SR-FR-11	
Definition:	The system shall calculate the throughput maximum value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.24: *System Functional Requirement SR-FR-11.*

SR-FR-12	
Definition:	The system shall calculate the throughput minimum value.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.25: *System Functional Requirement SR-FR-12.*

SR-FR-13	
Definition:	The system shall compute the total execution time.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-03

Table 3.26: *System Functional Requirement SR-FR-13.*

SR-FR-14	
Definition:	The system shall provide the execution of the profiling submodule.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-04

Table 3.27: *System Functional Requirement SR-FR-14.*

SR-FR-15	
Definition:	The system shall create a extrae-profiler object if the profiling submodule has been chosen.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-05

Table 3.28: *System Functional Requirement SR-FR-15.*

SR-FR-16	
Definition:	The system shall provide the execution of the tracing submodule.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-06

Table 3.29: *System Functional Requirement SR-FR-16.*

SR-FR-17	
Definition:	The system shall generate a trace file in case the tracing submodule is selected.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-06

Table 3.30: *System Functional Requirement SR-FR-17.*

SR-FR-18	
Definition:	The system shall notify the user when the trace file has been generated.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-06

Table 3.31: *System Functional Requirement SR-FR-18.*

SR-FR-19	
Definition:	The system shall create a native-profiler object if the tracing submodule has been chosen.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-06

Table 3.32: *System Functional Requirement SR-FR-19.*

SR-FR-20	
Definition:	The system shall run for applications containing map patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-07

Table 3.33: *System Functional Requirement SR-FR-20.*

SR-FR-21	
Definition:	The system shall run for applications containing map-reduce patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-07

Table 3.34: *System Functional Requirement SR-FR-21.*

SR-FR-22	
Definition:	The system shall run for applications containing stencil patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-07

Table 3.35: *System Functional Requirement SR-FR-22.*

SR-FR-23	
Definition:	The system shall run for applications containing divide and conquer patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-07

Table 3.36: *System Functional Requirement SR-FR-23.*

SR-FR-24	
Definition:	The system shall run for applications containing farm patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-08

Table 3.37: *System Functional Requirement SR-FR-24.*

SR-FR-25	
Definition:	The system shall run for applications containing stream-filter patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-08

Table 3.38: *System Functional Requirement SR-FR-25.*

SR-FR-26	
Definition:	The system shall run for applications containing stream-reduction patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-08

Table 3.39: *System Functional Requirement SR-FR-26.*

SR-FR-27	
Definition:	The system shall run for applications containing stream-iteration patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-08

Table 3.40: *System Functional Requirement SR-FR-27.*

SR-FR-28	
Definition:	The system shall provide the opportunity to run the application with the `feedgnuplot` software.
Priority:	Essential
Necessity:	Low
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-08

Table 3.41: *System Functional Requirement SR-FR-28.*

However, these ones are the non-functional system requirements:

SR-NFR-01	
Definition:	The system shall work within the GrPPI framework.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CO-01

Table 3.42: *System Non-Functional Requirement SR-NFR-01.*

SR-NFR-02	
Definition:	The system shall work with the C++ language.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CO-02

Table 3.43: *System Non-Functional Requirement SR-NFR-02.*

SR-NFR-03	
Definition:	The system shall work with ISO C++ Threads parallel programming framework.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CO-03

Table 3.44: *System Non-Functional Requirement SR-NFR-03.*

SR-NFR-04	
Definition:	The system shall work with the OpenMP parallel programming framework.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CO-03

Table 3.45: *System Non-Functional Requirement SR-NFR-04.*

SR-NFR-05	
Definition:	The system shall work with sequential GrPPI patterns.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CO-03

Table 3.46: *System Non-Functional Requirement SR-NFR-05.*

SR-NFR-06	
Definition:	The system shall calculate the total accumulated mean.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-05

Table 3.47: *System Non-Functional Requirement SR-NFR-06.*

SR-NFR-07	
Definition:	The system shall calculate the total standard deviation.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-05

Table 3.48: *System Non-Functional Requirement SR-NFR-07.*

SR-NFR-08	
Definition:	The system shall calculate the total throughput.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-05

Table 3.49: *System Non-Functional Requirement SR-NFR-08.*

SR-NFR-09	
Definition:	The system shall calculate the total execution time.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-05

Table 3.50: *System Non-Functional Requirement SR-NFR-09.*

SR-NFR-10	
Definition:	The system shall have installed the Extrae library.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-06

Table 3.51: *System Non-Functional Requirement SR-NFR-10.*

SR-NFR-11	
Definition:	The system shall output Paraver compliant files.
Priority:	Essential
Necessity:	High
Verifiability:	Stable
Stability:	High
Status:	Validated
Origin:	UR-CA-06

Table 3.52: *System Non-Functional Requirement SR-NFR-11.*

After listing the set of different requirements, we will show the traceability matrix (see Table 3.53) so that we can check the *traceable* property of each requirement before stated. Concretely, this matrix will assure if every user requirement is covered by at least one system requirement. Only in that way, we could conclude that all user needs had been satisfied. The matrix looks as follows:

	UR-CA-01	UR-CA-02	UR-CA-03	UR-CA-04	UR-CA-05	UR-CA-06	UR-CA-07	UR-CA-08	UR-CO-01	UR-CO-02	UR-CO-03
SR-FR-01	✗										
SR-FR-02		✗									
SR-FR-03		✗									
SR-FR-04			✗								
SR-FR-05			✗								
SR-FR-06			✗								
SR-FR-07			✗								
SR-FR-08			✗								
SR-FR-09			✗								
SR-FR-10			✗								
SR-FR-11			✗								
SR-FR-12			✗								
SR-FR-13			✗								
SR-FR-14				✗							
SR-FR-15					✗						
SR-FR-16						✗					
SR-FR-17						✗					
SR-FR-18						✗					
SR-FR-19						✗					
SR-FR-20							✗				
SR-FR-21							✗				
SR-FR-22							✗				
SR-FR-23							✗				
SR-FR-24								✗			
SR-FR-25								✗			
SR-FR-26								✗			
SR-FR-27								✗			
SR-FR-28								✗			
SR-NFR-01									✗		
SR-NFR-02										✗	
SR-NFR-03											✗
SR-NFR-04											✗
SR-NFR-05											✗
SR-NFR-06					✗						
SR-NFR-07					✗						
SR-NFR-08					✗						
SR-NFR-09					✗						
SR-NFR-10						✗					
SR-NFR-11						✗					

Table 3.53: *Traceability Matrix*.

3.2 Design

Together with the implementation, the design phase has been the most time-consuming one. A good design is key to success in further phases (such as implementation and evaluation), so we need to list our goals and think how they can be achieved using the available resources.

From the first moment, our main objective was to develop the most flexible possible design, in such a way that at any moment, we would be able to modify some part of it according to what the user asked for. With this, two different submodules will provide the most compact and reusable interface, allowing users to switch between them in the easiest way, which can be translated into flexibility issues.

Below, we will describe the working environment used to develop this project. We will also explain in a global way how the system architecture has been designed by means of a set of diagrams that represent the activities and actions we can perform within the system.

3.2.1 Work environment

The starting point of this project is GrPPI. The goal is to add some module to this library in order to serve as a performance analysis tool, since this will also add new functionalities to the aforementioned pattern-based parallel framework. It took us several days in order to concretely define the range of the project. Having in mind a rough draft of what the system should look like. Therefore, a general description of the environment set up is now provided.

The title of this project focuses on parallel pattern-based applications and performance analysis environments. The question now is how can we add that aforementioned performance analysis functionality to GrPPI-based applications. By including in our code those parallel patterns as explained in 2.1, we will be able to track the pattern behaviour used within the applications.

This new performance analysis module added to GrPPI framework will be divided into two different submodules, since we need to provide users with the opportunity to choose according to their needs and preferences. The first submodule is in charge of giving users the *profiling* functionality, whereas the second submodule will refer to the *tracing* functionality. In order to clearly understand this, Figure 3-1 shows a simple schema describing our proposed solution to the problem suggested in Section 1.1.

This schema shows (in a very high-level abstraction), the main goal of this project, but we need to dig up this pattern-based parallel framework in order to better understand how we will implement this module as a whole. Therefore, GrPPI architecture will be also exposed in a graphical way together with this newly added module.

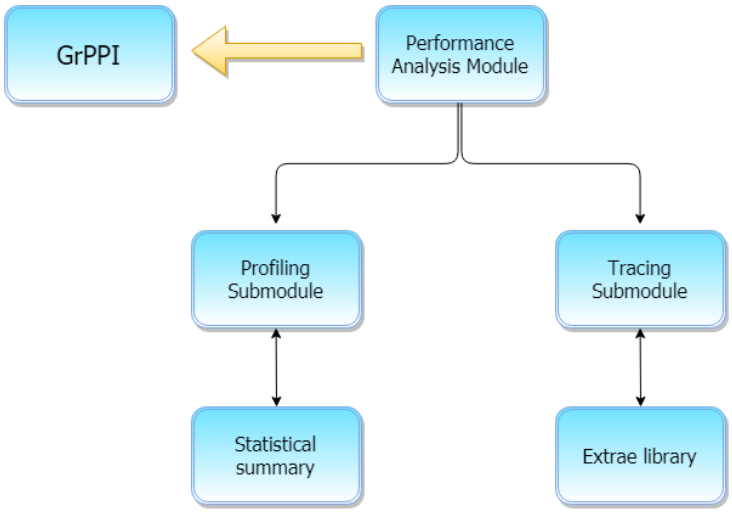


Figure 3-1: *Global description of the project.*

Figure 3-2 shows the internal files organization of the GrPPI framework. For simplicity not every subcomponent has been shown in this structure schema. Therefore, only the most important parts along with the ones that we have been working with are shown there.

We can observe first the *docs* folder, which contains a set of documentation files as bibliography. Then, we have the *include* folder. This has been the most important one for us, since as can be observed, the core of the project is included there. There, we can distinguish between the different back ends already mentioned (sequential, parallel_native with ISO C++ threads, OpenMP and Intel TBB). Each one is composed of one different file according to that architecture. Apart from those folders, we can also identify the *common* folder, which is the one where all the aforementioned modules have been added (by means of the five subsequent files).

The *samples* folder is just the one that encloses the set of examples that this framework has in order to test the parallel patterns (actually, some of those applications have been used to test the new performance analysis environment). Finally, different test cases are present for each parallel pattern just for testing purpose. This picture has been shown in order to explain the interactions between this framework and the new aggregated module. For that reason, two squared green boxes have been drawn: they just select the main components where this project has been implemented. On one side, we have the three GrPPI back ends[1], and on the other green square, the two performance analysis submodules. The combination of each back end with each submodule is just the same as Table 3.54 shows, which corresponds to the parts that have been implemented.

After defining the performance analysis metrics we needed to use, we started by instrumenting the classes involving the left big box (*sequential_execution*, *parallel_execution_native* and *parallel_execution_omp*).

[1]TBB back end has been left for future work

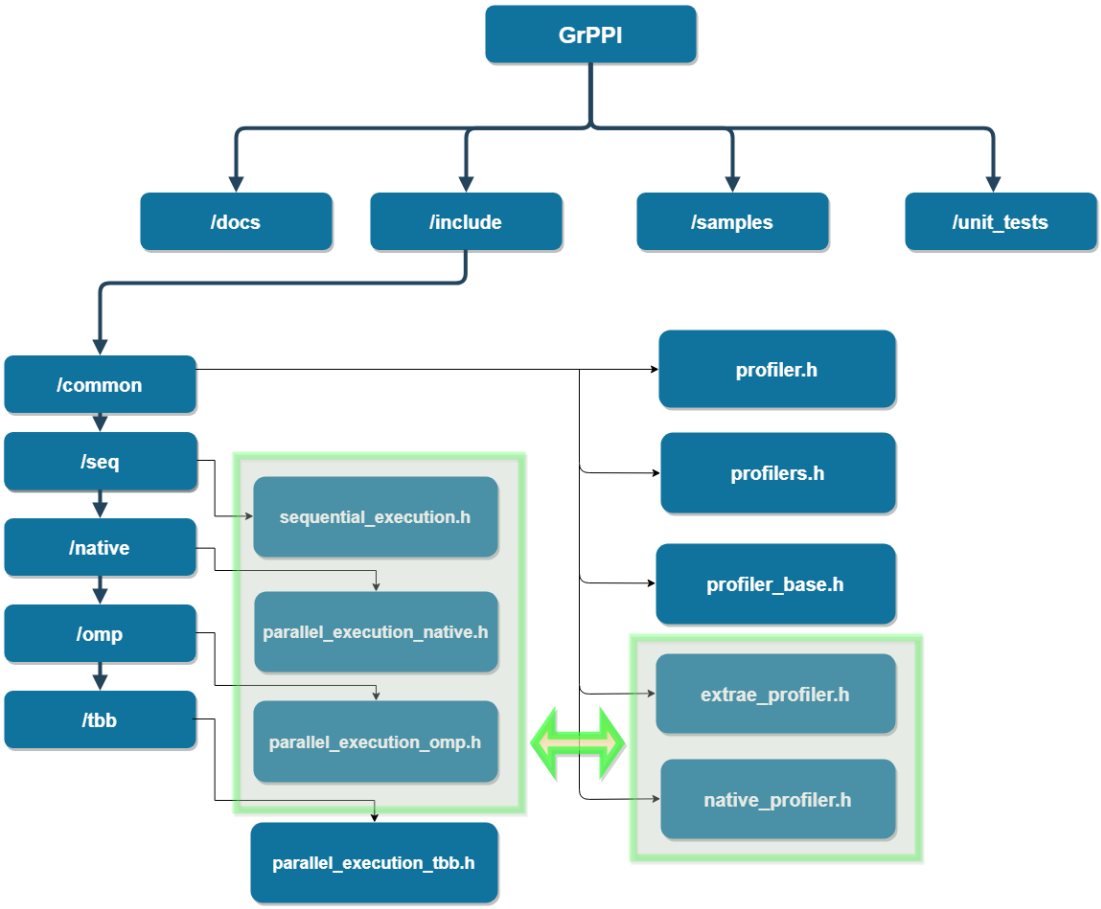


Figure 3-2: *GrPPI internal structure.*

After that, those metrics were implemented through a set of different functions in the two classes on the right green box (*extrae_profiler* and *native_profiler*). With all this, when running our application by using one of the submodules, different performance results would be obtained.

Once depicted the working environment, we list the project range by means of a table, since not all the combinations of submodules and available back ends in GrPPI have been implemented. Let us look at Table 3.54 to observe the project implementation in a summarized way. Say that cells with a "✓" mean

	Profiling (Native)	Tracing (Extrae)
Sequential	✓	✓
Native (ISO C++ Threads)	✓	✓
OpenMP	✓	✓
Intel TBB	✗	✗

Table 3.54: *Implemented backends in this project.*

that the given combination (row-column) has been implemented, while cells containing the "✗" symbol have not been implemented and are left for future work.

3.2.2 Software architecture model

Within this section, we provide a set of multiple views coming from the system design, by means of a set of different software diagrams.

In order to do so, the *4+1* architecture view model has been followed. This model is a complete and compact way to show, at the same time, the architecture of a system from different points of view [47]. In this way, thanks to this *4+1* graphical representation of the system, it would be possible to strengthen any possible incomplete information by the rest of diagrams.

Its name comes from the number of diagrams this model is composed of. Four different software diagrams offer its corresponding information about the system, and a fifth one is the result of an illustration of a real use case in a specific scenario where all interactions regarding the user take place.

Concretely, this model is composed of the present views in Figure 3-3. A short description of them is now shown [47]:

- **Use case view:** also called the scenario view. It needs to be coherent, and its function is to validate the completeness of the system.
- **Development view:** it represents the static behaviour of the system. The diagram associated with it is the *component diagram* one.
- **Logical view:** it describes the object model (components) of the system. The diagram associated to it is the *class diagram* one.
- **Process view:** it describes the interactions and activities within the system, that is, it represents the dynamic behaviour of the system. The diagram associated with it is the *activity diagram* one.
- **Physical view:** it maps the software part into the hardware one. The diagram associated with it is the *deployment diagram* one.

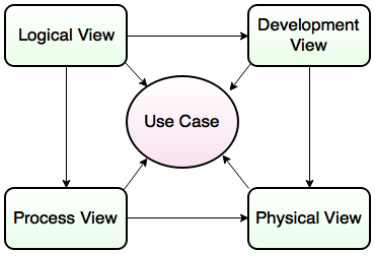


Figure 3-3: *4 + 1 Architecture Model [48].*

As it can be identified in the previous list, each view is related to one specific diagram. In this project,

the set of diagrams have been developed according to the Unified Modeling Language (UML) standard. UML is a visual language that combines the natural language with graphs. With this standard, anyone with a basic expertise would be able to understand everything.

They have been developed with the Visual Paradigm tool, since it supports the aforementioned standard. Next, we will explain, for each view, its associated diagram in order to fulfill this architecture with the proposed design.

Use case diagram

The use case diagram is the result of the following four diagrams and it is strongly related to the user actions within the system, since it shows every possible interaction between them. Therefore, all possible actions that the user can carry out should be coherent with the aforementioned user requirements.

Next, we show the result of this diagram:

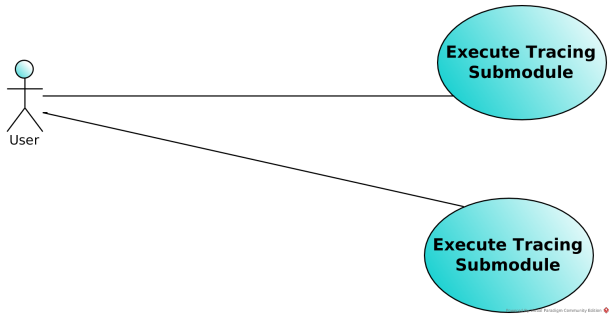


Figure 3-4: *Use Case Diagram.*

As a general overview, this diagram is quite simple. At the end, what user can perform when testing the implemented system corresponds to each of the coloured circles.

On one hand, the user can execute his application by means of the profiler submodule. On the other hand, the user can do the same but with the other submodule. Any other possible action to carry out is actually performed by the machine and not to our system, that is why this diagram has been reduced to what it can be observed in Figure 3-4.

To fulfill the proposed use case design, we need to define a set of use cases. A use case is defined by:

UCA-XX	
Name:	Name
Actor:	User | Administrator
Description:	Text that describes the use case
Preconditions:	Set of preconditions to be fulfilled
Postconditions:	Set of postconditions to be fulfilled

Table 3.55: *Use Case Template Table*

XX refers to the unit test identification number. It comprises values from 00 to 99.

UCA-01	
Name:	Execute Profiling Submodule
Actor:	User
Description:	The user implements its code within the GrPPI framework and once executed its parallel application, he will obtain a report performance summary according to his application behaviour.
Preconditions:	• The application shall run within the GrPPI parallel framework.
Postconditions:	• The user obtains a report performance summary.

Table 3.56: *Use Case UCA-01.*

UCA-02	
Name:	Execute Tracing Submodule
Actor:	User
Description:	The user implements its code within the GrPPI framework and once executed its parallel application, a trace file will be generated, allowing the user to further use Paraver tool to visualize its application performance.
Preconditions:	• The application shall run within the GrPPI parallel framework.
Postconditions:	• A trace file is generated and user is able to use Paraver tool.

Table 3.57: *Use Case UCA-02.*

Component diagram

The component diagram refers to the set of subsystems that the whole system is composed of. It is not difficult to determine that only two main subsystems are presented in this project: on one side the GrPPI framework, and on the other the performance analysis module.

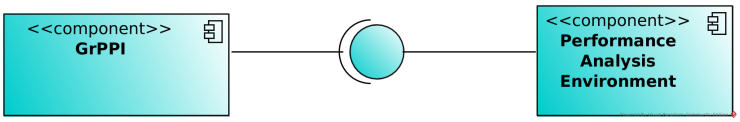


Figure 3-5: *Component Diagram.*

As shown in Figure 3-5, two components make up this diagram. The difference between them (talking about component terms) is the type of interface defined on each one, that is, the relation that makes being connected. From the performance analysis module perspective, the symbol needs to be a straight line joined with a circle (it *provides* the interface). However, from the GrPPI framework perspective, this framework needs this module to work in the way we want, so it *requires* the interface (a straight line attached to a half circle).

Class diagram

Regarding the class diagram, it expresses a generic and descriptive overview of the system talking about programming terms. We say *programming* terms since each entity (box) in this diagram is composed of three main subparts.

On top of each, we find the name of the class (compulsory section). Then, a middle section which lists the set of attributes (qualities) belonging to that class. Finally, the bottom part includes the set of methods of that class, that is, the operations or interactions with the data stored in the previous attributes. For simplicity purposes, only the main methods (not even attributes) shared in the common interface by these two submodules are going to be placed inside its corresponding box, because in any other way, the size of each box would be too large. In Figure 3-6 the diagram that we are referring:

This diagram is similar to the one shown above (see Figure 3-1) but with more detail and with the formal relationships between classes.

A starting point class is denoted by *profiler*. This is the main class where the two submodules will be implemented. Hence, it contains the general interface shared by those two. These two submodules inherit from the parent class (the *profiler* class that we have just commented). The only difference between them, is just the set of attributes defined inside each class, since as previously said, the set of methods

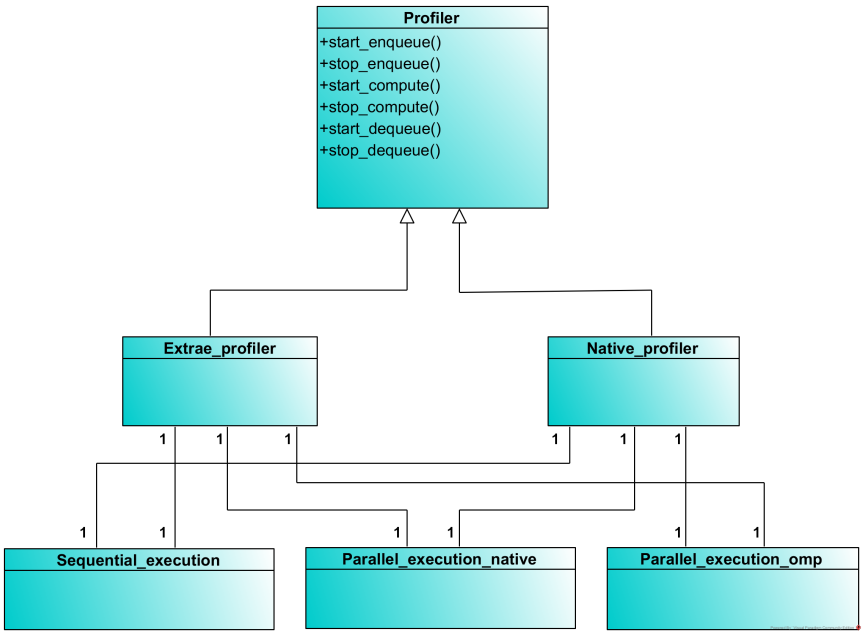


Figure 3-6: *Class Diagram.*

are equal for both (interface is shared between them), but as stated before, we are not showing them for space purposes.

For each submodule, there is a relationship *1:1* to each of the available back ends in GrPPI[2]. Again, these six relationships by means of arrows is what Table 3.54 shows by means of "✓" symbols.

Sequence diagram

Now, we continue with the set of diagrams in order to fulfill the software *4+1* model. For that, we specify the sequence of actions our system must perform from the user point of view, that is why it is the main actor in that diagram. This type of diagrams is important since it shows the interactions (represented by arrows) from one point to another. It basically explains the functionality of the system from the first moment the user tries it, until the end of the execution: this is called a usage scenario.

Above, we show the pair of sequence diagrams for this system. The only difference between them is just the type of created object, that is, the type of submodule the user wants to use and the action carried out once data has been recorded, so we will explain both at the same time since the basis is the same.

There, we can distinguish between one single actor, the user, which is going to interact with the rest of entities in the system (see the three labelled rectangles on top of the diagram). The first thing the user does is just executing his application once included in the GrPPI framework. Automatically,

[2]notice that TBB is not taken into account since it has not been implemented

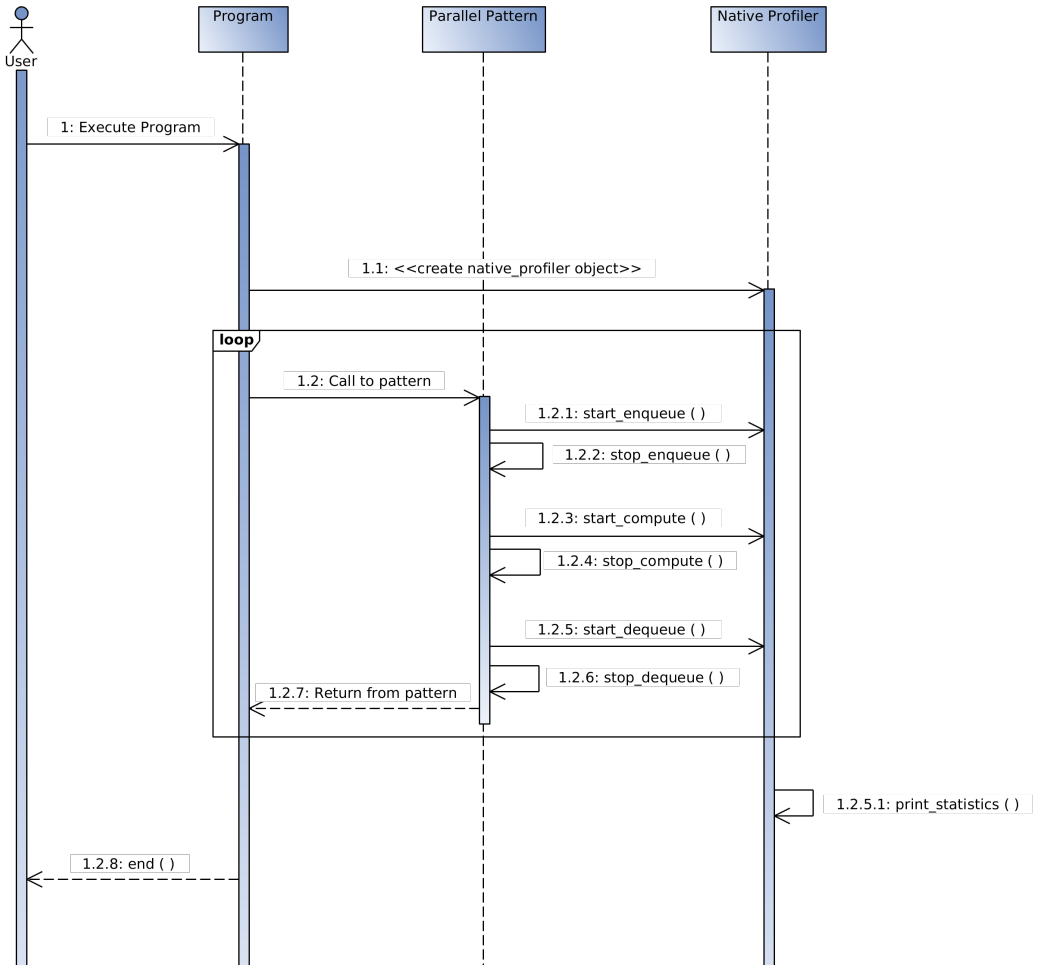


Figure 3-7: *Sequence diagram for profiling submodule.*

the program will create a *native* or *extrae* object depending if the user wants to use the profiling or the tracing submodule respectively. After the object creation, the system will detect the type of pattern used in the program, and independently of whatever pattern it is, a set of different functions (concretely six) will be called in a loop until processing all the input data. Notice that these six functions coincide which the ones shown in the class diagram.

These functions are the ones established in the class diagram. They belong to a common interface shared by these two submodules, in such a way that they can be reused for any pattern and for each of the submodules. Then, data from the input application should be stored and metrics should have the proper data in order to plot the registered information.

If the tracing submodule is used, a file with extension *.prv* will be created. This file format is the one that Paraver tool supports for plotting the data, so once this file has been generated, we will be able to observe the information by means of graphs. After the generation of this file, the program will send a notification telling that this file has been already generated and execution will finish then.

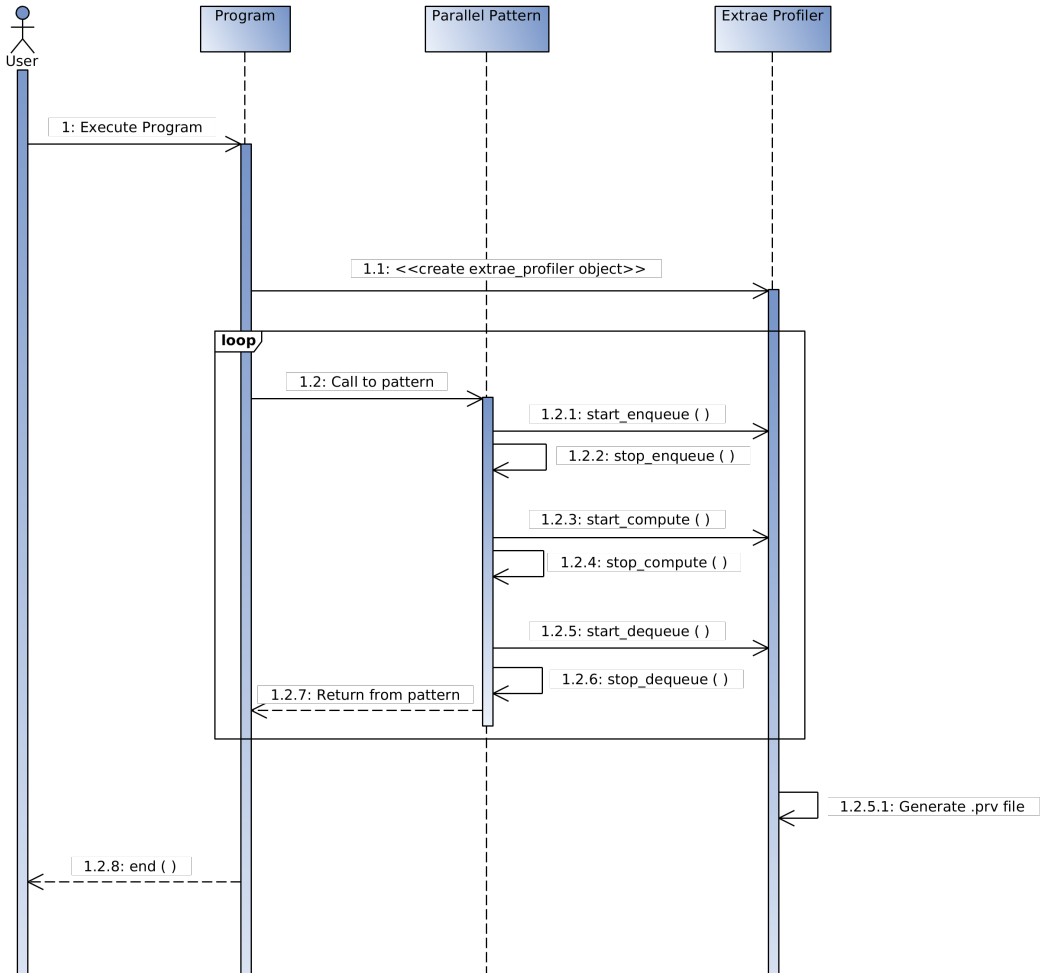


Figure 3-8: *Sequence diagram for tracing submodule.*

In case the profiling submodule has been the selected one, instead of generating that file, a call to an additional method will be needed instead. This method is in charge of computing the set of statistical metrics that have been thought beforehand: accumulated mean, standard deviation, etc. which are explained in detail in next chapter, section 4.1. Once obtained all the results from those metrics, information will be returned to the user by means of statistical summary tables and the execution will finally terminate.

Deployment diagram

A deployment diagram is no more than a graphical representation of the hardware topology obtained from software. In this project, not much resources have been required to carry out the implementation, so everything can be summarized in one single box as shown in Figure 3-9:

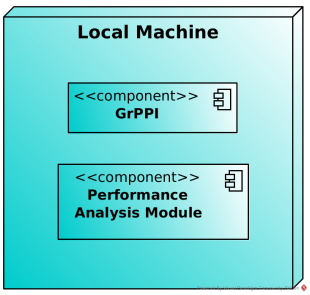


Figure 3-9: *Deployment Diagram.*

The resulting diagram is very simple: it just references the local machine where all aspects of the system have been built. Inside this hardware element, we can differentiate between the GrPPI component together with the performance analysis module.

Chapter 4

Implementation

In this project two main parts can be differentiated, the profiling and the event-tracing submodules, which have been implemented using different modules. A very important point to consider when using performance analysis approaches is the measurement of interesting aspects of the program, also called metrics: depending on the chosen ones, different measures have to be captured. Afterwards, these metrics can be visualized and analyze what it has been recorded and make some conclusions about it.

The implementation process corresponds with the three phases listed in Section 1.2.2. Next, in Section 4.1, we focus on the theoretical aspects of the measurement phase. We have temporarily skipped the first phase (instrumentation) since the explanation of those metrics is essential to understand the instrumentation process. Along next section, we describe the set of statistical metrics that have been chosen during the implementation. After the metrics explanation, the complete performance analysis module (with its two submodules) is explained in detail. For concluding this chapter, a section dedicated to future work will also be given.

4.1 Introduction to metrics

Performance analysis is carried out over applications by means of a set of statistical metrics previously studied. For this reason, several days were spent defining the metrics that were specific in the requirements of the project, as well as defining the project range.

This section explains the set of metrics and the implementation of the profiling and tracing submodules. Parallel patterns internally behave as a continuous flow of data that need to be processed, as shown in the graphical schemes in Section 2.1 for each pattern. Depending if it is a data or streaming, data flow

is defined differently inside the pattern.

Concretely, if we are facing a data pattern, only some operations over that data volume need to be carried out: this is reflected in a ***compute stage***. However, for stream patterns, we have also to consider a ***enqueue stage*** of data, a ***compute*** one (as with the data) and a ***dequeue stage***. These three basic stages, that do not need to be performed in order and that can be performed several times, are the ones associated to a pipeline structure. A pipeline is composed of several stages, and so, intermediate data in the stream has to be stored in some way: this is done by means of special data structures called *queues*. Thanks to them, we can push/pop items to/from it until there is not left data.

Below, we detail the profiling metrics:

1. **Accumulated Mean:** this metric is similar to the typical mean calculation, but it differs by the fact. The difference is that this calculation can be performed at run-time without the need of store all recorded samples into an additional data structure.

   The next formula shows how to calculate this metric:

$$\bar{X} = \frac{(total\_number\_items * \bar{X}) + elapsed\_time}{number\_items} \tag{4.1}$$

   where:

   - *total_number_items:* is the current number of recorded items at some point in time.
   - $\bar{X}$*:* is the value of the mean total value at some timestamp. It could refer to *mean_enqueue*, *mean_compute* or *mean_dequeue*.
   - *elapsed_time:* time interval between the beginning of an operation and the completion of it. It can refer to: enqueue elapsed time, compute elapsed time or dequeue elapsed time.
   - *number_items:* is the *total_number_items* plus one, since we start tracking this metric by having one element.

   This formula is the general one, but since we face three different phases during execution, this will lead to the development of three different versions from the general formula:

   - Accumulated mean time of data items when being enqueued into a queue:

$$\bar{X}_{enqueue} = \frac{(total\_accumulated\_mean * \bar{X}_{enqueue}) + elapsed\_enqueuing\_time}{number\_enqueued\_items} \tag{4.2}$$

   - Accumulated mean time of data items when being computed:

$$\bar{X}_{compute} = \frac{(total\_accumulated\_mean * \bar{X}_{compute}) + elapsed\_computing\_time}{number\_computed\_items} \tag{4.3}$$

- Accumulated mean time of data items when being dequeued from a queue:

$$\bar{X}_{dequeue} = \frac{(total\_accumulated\_mean * \bar{X}_{dequeue}) + elapsed\_dequeuing\_time}{number\_dequeued\_items} \tag{4.4}$$

For data patterns, only Formula 4.3 will be needed, since those patterns only perform the second phase, the *compute* one. Nevertheless, the three previous formulas will be applied for streaming parallel patterns, since we will need to *enqueue*, *compute* and *dequeue*. Besides, we need to say that this metric has been calculated per each stage in the stream.

2. **Standard Deviation:** this other metric is based on the previous metric. It can be defined as how far one element is from the mean total value, that is why it is considered to be a dispersion statistical measure.

   As with the mean, we will show the standard deviation general formula:

$$\sigma = \sqrt{\frac{x_1^2 + x_2^2 + x_1^2 + ... + x_i^2}{total\_number\_items} - \bar{X}^2} \tag{4.5}$$

   where:

   - x_i*:* is the i-th item in the data stream. Each x represents the elapsed time between the beginning of an operation and the completion of it. Operations can be: enqueue, compute or dequeue.
   - *total_number_items:* is the current number of recorded items at some point in time (this number is equal to n).
   - $\bar{X}$*:* is the value of the mean total value at some timestamp. It could refer to *mean_enqueue*, *mean_compute* or *mean_dequeue*.

   The same happens here for the three versions from this formula: only the second phase (*compute*) will be calculated for the data patterns and every phase (*enqueue, compute* and *dequeue*) will be applied to the streaming ones. Besides, for this last pattern type, this metric has been calculated per each stage in the stream.

3. **Mean Throughput:** this has to do with the data volume being processed during execution. That is, we could track the number of items that enter some phase (enqueue, compute or dequeue) per unit time. This is called the throughput, and it is a quite interesting measure to identify the parts in our code where there is more/less data traffic. It is important to mention that this metric is only applicable for streaming parallel patterns. It is calculated as follows:

$$throughput = \frac{1}{(\bar{X}_{enqueue} + \bar{X}_{compute} + \bar{X}_{dequeue}) * number\_threads} \tag{4.6}$$

where:

- $X_{enqueue}$ is the total mean value of items when been enqueued.
- $X_{compute}$ is the actual value of the mean at compute phase.
- $X_{dequeue}$ is the actual value of the mean at dequeue phase.
- *number_threads* is the current number of workers (in a specific stage) performing some tasks.

Important to say that this formula is only applicable for the streaming patterns since they have more phases through which data is distributed (throughput will be calculated per each stage).

4. **Minimum/Maximum Values:** we write them in plural since these values have been calculated for every single metric already explained. That is, we have recorded the maximum/minimum values for:

   - *mean_enqueue values*
   - *mean_compute values*
   - *mean_dequeue values*
   - *standard_deviation_enqueue values*
   - *standard_deviation_compute values*
   - *standard_deviation_dequeue values*
   - *throughput*

5. **Total Execution Time:** is simply a measure that computes the time interval since the beginning of the execution until the end of it.

The following metrics have been the ones used for the tracing submodule thanks to Paraver tool. Important to mention that from all the metrics explained next, we distinguish between the type of pattern used.

In case it is a data pattern one, only the MFLOPS and L3 TCM (explained just below), are more relevant to apply, and so, the four remaining ones (Event Trace, Execution Time, Throughput and Active Threads) will be applied to the streaming patterns:

1. **MFLOPS:** this metric (*megaflops*), is equivalent to 10^6 Floating-point Operations per Second (FLOPS). It records the number of floating point operations that are being performed per unit of time, in this case, seconds.

2. **L3 Total Cache Misses:** shows the number of L3 total cache misses per second occurring during the execution. Total includes both instructions and data.

3. **Event Trace:** a different identification number has been associated to each parallel pattern. Besides, it has also be recorded the different phases through which data passes when execution takes

place. Hence, this trace shows a complete summary of the different phases associated to a concrete parallel pattern.

4. **Execution Time:** by means of bar graphs (one per thread), it shows the time interval through which the application is running and shows the specific periods in which threads are performing some tasks.

5. **Throughput:** this is the same as with the profiling submodule, but this is now represented in a graphical way.

6. **Active threads:** as the name says, it marks with a continuous line the activity of each specific thread in the application, so we are able to determine how much work is associated to some concrete thread.

4.2 Performance analysis module

This section describes as a whole overview the module that has been developed. Since two different and independent parts have been implemented (each one corresponding to Section 4.2.2 and 4.2.3), we will explain them by separately, since this will show the work done during several months.

In Section 3.2 from the system architecture chapter, we have observed in a high-level abstraction the shape our system presents by means of different schemes and diagrams. However, that was explained in a superficial way, without going deeper into the exact content of each component within those diagrams. Therefore, over this chapter, we will try to describe what those boxes in the diagrams represented and their characteristics.

Before continuing with each submodule, we will explain the content and purpose of the *parent* class through which the submodules have inherited from (which at a high-level, it would correspond with the *Performance Analysis Module* box in Figure 3-1). This *parent* class, called `profiler_base` in our system, will be the one that behaves as the general interface. We wanted the system to be as reusable as possible, that is the reason why this general interface has been developed. With this, we explain what this *parent* class contains.

First of all, insist on the role this class takes. It is a general class, and as in any other inheritance case, the *parent* class contains the declaration of all the structures and functionalities that its *children* need. Therefore, we will find both variables and functions that will need the profiling and tracing submodules, but we could also find specific aspects that maybe the other submodule does not need at all. In any case, we will explain all of them, first the common aspects and afterwards the specific ones for each submodule.

We can distinguish between three different main "blocks". The **first** one and possibly the most important one, comprises the set of all functions in charge of tracking the inner flow of the program (according to the parallel pattern used). We are going to show an example of two of them:

```
virtual void start_enqueue (unsigned int stage_id) { };
virtual void stop_enqueue (unsigned int stage_id) { };
```

We can proceed in the same way with the *compute* and *dequeue* phases that were also mentioned in Section 4.1 (introduction to metrics). These six functions include as parameter an identification for the stage in the stream, since we are going to track performance per each stage. In the case of data parallel patterns, as we already stated before, they would only perform the *compute* phase (two functions), and consequently, they will only have associated one identification number for that concrete stage.

As it can be observed, functions have been stated as *virtual*, since in that way, they will be overwritten by its children, profiling and tracing submodules.

Apart from those statements, we need to specify how these *stages* work in our system (which corresponds to the **second** block). Different events in a trace need to be distinguished from the rest, they need to be unique. Therefore, we have assigned one different identification number to each data pattern, so that we can better follow the track of the program even if we use the profiling or tracing submodule:

- **Map:** 2
- **Parallel-For**[1]**:** 3
- **Reduce:** 4
- **Map-Reduce:** 5
- **Stencil:** 6
- **Divide and Conquer:** 7

But again, streaming patterns behave in a different way. They are composed of more than one stage and for that reason, we thought of starting assigning to whatever the streaming pattern was, the value 10 in the first stage, and increasing this value until reaching the last stage of the data stream. Besides, for these pattern type, as they deal with *queues*, we also need to track when items are enqueued and dequeued. Therefore, we have added two specific variables for tracking these actions: number 8 for enqueue and number 9 for dequeue. This identification assignment number is very important, since it will guide us when obtaining results. For instance, in the profiling submodule when Paraver filters those numbers with colours so that by knowing this number, we can determine to which pattern it is referring to in the bar graph.

[1]We have also worked with it but as it had been recently added to GrPPI and being subjected to different tests, it has not been considered in the evaluation part.

Associated to those numbers, we have also defined the **third** block. These functions just return the values already specified according to the pattern type. An example of one of them looks like:

```
virtual int get_stencil_start() { return 0; };
```

In that concrete example, when calling that function in any of the submodules, we will get the number 6, as this is the specified number. The same function has been defined for the rest of patterns.

Now, we will refer to the specific functionalities that this *parent* class defines for each part. Regarding the profiling one, we have defined one additional method. It is `print_statistics()`, whose name says, it will show the summary report to users. We will not go deeper into this since a detailed explanation will be given when describing the corresponding submodule. With respect to the tracing one, since we have made use of Extrae library, this fact has forced us to define an identification event number so that in this way, Paraver would filter all the different events by the one specified by us:

```
static constexpr int GRPPI_EVT = 6000000;
```

4.2.1 Instrumentation phase

We have argued before that the entire implementation of this project is directly associated to the three main phases we need to carry out when applying profiling or tracing techniques (instrumentation, measurement and visualization).

Now, after explaining the content of the module *parent* class (basically the *environment set up* task stated in the Gantt Diagram), we will explain how we proceeded with the instrumentation, the first phase in Section 1.2.2.

Implementation phase was common for both submodules, since they share a common interface already explained (*parent* class), so we just needed this process to do it once. Our idea when developing this project was to instrument some specific internal classes of GrPPI. These classes are `sequential_execution`, `parallel_execution_native` and `parallel_execution_omp`, and are marked with a big square in the left part of Figure 3-2. They are the ones that internally work with all the parallel patterns, so we just needed to modify that classes, one by one. However, the instrumentation process is quite delicate since you need to be sure where to place these record-based directives that envelop interesting parts of our code.

Since the three classes have been instrumented in the same way, we are going to focus on just one of them, the `parallel_execution_native` since it has been the one with which evaluation has also been performed.

Hence, we start this phase from the usual behaviour that GrPPI offers. GrPPI is written in C++ and internally works by means of metaprogramming instructions. Metaprogramming is a quite advanced programming modality that allows the generation of code at compile-time.

Leaving apart this metaprogramming topic, since it is out of the project range, we will start describing a bit the different structures and functions we can find inside that class. As in any other class, the `parallel_execution_native` one has a set of constructors at the beginning of it, and then, one function representing each possible parallel pattern. In this way, users execute their applications (this one containing whatever the pattern it is), his application will call the `parallel_execution_native` class and this last one will execute the function associated to the pattern used in the user program. For instance, if his application contains a *map-reduce* pattern, the `parallel_execution_native` class will execute the associated *map-reduce* function.

Each parallel pattern was already build internally (by metaprogramming directives), and so, it was time to start placing the aforementioned six functions (`start_enqueue`, `stop_enqueue`, `start_compute`, `stop_compute`, `start_dequeue` and `stop_dequeue`) inside each of the different pattern (inside each associated pattern function).

Next, we will show two pieces of instrumented code extracted from two different data patterns:

Listing 4.1: *Piece of code extracted from the map data pattern.*

```
1  stage_id_= profiler_.get_map_start();
2  auto process_chunk = [&transform_op, this](auto fins, std::size_t size, auto fout){
3    profiler_.start_compute(stage_id_);
4    const auto l = next(get<0>(fins), size);
5    while (get<0>(fins)!=l) {
6      *fout++ = apply_deref_increment(
7         std::forward<Transformer>(transform_op), fins);
8    }
9    profiler_.stop_compute(stage_id_);
10 };
```

Listing 4.2: *Piece of code extracted from the map-reduce data pattern.*

```
1  stage_id_= profiler_.get_map_reduce_start();
2  auto process_chunk = [&](auto f, std::size_t sz, std::size_t id) {
3    profiler_.start_compute(stage_id_);
4    partial_results[id] = seq.map_reduce(f, sz,
5        std::forward<Identity>(partial_results[id]),
6        std::forward<Transformer>(transform_op),
7        std::forward<Combiner>(combine_op));
8    profiler_.stop_compute(stage_id_);
9  };
```

As can be observed in the two pieces of code, we have surrounded the part of code where the *map* and *map-reduce*, respectively, perform all the computations.

This pair of functions is called throughout a *profiler* object which has been initialized in the two aforementioned new constructors. The parameter `stage_id_` is the one that we have explained before and that has a value associated to the pattern type. If we look at line 1 in both pieces of code, we can see that this value is obtained from the value the *get* method returns respectively (in the first case we will get the number 2 and in the second case, the number 4).

Next, we will show another example of the instrumentation carried out, but this time, focusing on a streaming parallel pattern, a bit complex one:

Listing 4.3: *Piece of code extracted from the farm stream pattern.*

```
1  auto farm_task = [&,stage_id__ = stage_id_](int nt){
2    profiler_.start_dequeue(stage_id__);
3    auto item{input_queue.pop( )};
4    profiler_.stop_dequeue(stage_id__);
5
6    while (item.first) {
7      profiler_.start_compute(stage_id__);
8      farm_obj(*item.first);
9      profiler_.stop_compute(stage_id__);
10
11     profiler_.start_dequeue(stage_id__);
12     item = input_queue.pop();
13     profiler_.stop_dequeue(stage_id__);
14   }
15   profiler_.start_enqueue(stage_id__);
16   input_queue.push(item);
17   profiler_.stop_enqueue(stage_id__);
18 }; // end farm_task function
19 // ...
20 stage_id_++;
```

In the previous example, we can distinguish a streaming pattern. We said that these patterns were internally composed of more than one stage, so consequently, those patterns have associated three different functions in the `parallel_execution_native` class instead of just one. The first method is the *Generator*, a defined function which is executed one thread in any of the streaming parallel patterns, it is a common method. Then we have the associated pattern function itself (which is specific for each stream pattern as with the data ones), and finally, the *Consumer* method that treats all processed data in the previous stages (also common for every streaming parallel pattern).

As a difference to the previous examples, here we have made use of the two other types of functions, the enqueue and the dequeue ones (both *start* and *stop*). We can see that, when in the farm structure an item is being *popped* in line 3 (which means extracted from the queue), we need to surround that line by the pair *dequeue* functions, and when an item is *pushed* to a queue in line 15 (inserted to a queue), we need to apply the pair of *enqueue* functions.

Another different aspect is the `stage_id` variable. When we reach line 1 the first time in that piece of code, `stage_id` variable already has a value. This value is the one obtained in the *Generator* function (that we are not going to explain). Therefore, at the end of that code, `stage_id` will be incremented by one since we need to go ahead the next stage in the pipeline.

In the way that we have explained the three previous examples, we then continued with this instrumentation process, but regarding the rest of patterns and also regarding the other two classes, the `sequential_execution` and the `parallel_execution_omp`. During the following two subsections, we will be able to describe the content of the two children classes, which correspond with the *Profiling Submodule* and *Tracing Submodule* boxes in Figure 3-1.

4.2.2 Extrae tracing submodule

In this subsection, we explain the tracing submodule. It is not the order that has been followed along the document (profiling was always first), but we will follow the real order during the development of this project. Besides, this part was easier than the profiling one, so we will start from here.

Having clarified that, we will continue with the explanation of this class. Notice that now, we are going to focus on the *Tracing Submodule* box from Figure 3-1.

The purpose of this submodule was to use Extrae library. In that way, after the execution of any application, the `.prv` file would be generated and by means of Paraver tool, we would be able to visualize the results. Therefore, we needed to use the set of functions that Extrae API provides. Specifically, we have applied three from the whole set:

- `Extrae_init()`: it initializes the tracing library.
- `Extrae_fini()`: used to finalize the tracing library.
- `Extrae_event()`: it adds a single timestamped event to the tracefile.

The first and second ones were called just one in the constructor and destructor respectively as follows:

Listing 4.4: *Initialization and finalization of Extrae library in the tracing submodule.*

```
1    extrae_profiler(){ // constructor
2       Extrae_init();
3    }
4    ~extrae_profiler(){ // destructor
5       Extrae_fini();
6    }
```

However, the third one should be called per each of the six functions inherited from the *parent* class. This has been the definition of them:

Listing 4.5: *Implementation of the six functions within the tracing submodule.*

```
1    void start_enqueue (unsigned int stage_id){
2      Extrae_event(GRPPI_EVT, ENQUEUE_VAL) // GRPPI_EVT = 6000000 , ENQUEUE_VAL = 8
3    }
4    void stop_enqueue (unsigned int stage_id){
5      Extrae_event(GRPPI_EVT, 0);
6    }
7    void start_compute (unsigned int stage_id){
8      // stage_id can take any of the aforementioned values
9      Extrae_event(GRPPI_EVT, stage_id);
10   }
11   void stop_compute (unsigned int stage_id){
12     Extrae_event(GRPPI_EVT, 0);
13   }
14   void start_dequeue (unsigned int stage_id){
15     Extrae_event(GRPPI_EVT, DEQUEUE_VAL); // GRPPI_EVT = 6000000 , DEQUEUE_VAL = 9
16   }
17   void stop_dequeue (unsigned int stage_id){
18     Extrae_event(GRPPI_EVT, 0);
19   }
```

Now, we show two simple examples regarding the usage of this submodule. The first picture below corresponds to a data parallel pattern while the second one refers to a streaming pattern.

Data pattern example

This first example belongs to a simple exercise proposed for testing within GrPPI framework. It is called *add_sequences* and the reason is because it adds two sequences element by element. The first one contains the `n` first natural numbers and the second the square of those natural numbers, given as a

result another output sequence. In Figure 4-1 we can observe the results with Paraver tool.Each coloured bar represents one thread. Therefore, we can observe that we have launched 4 threads in the execution. We have established the n value (sequence size) to 50000, so that we could identify more activity in the figure.

The only colour hue that really interests us is the green one. The mustard one just appears to fill in the white colour from the background. In that picture, we have recorded the MFLOPS metric (which was already explained in Section 4.1), by selecting the corresponding configuration file that Paraver offers. With that metric, we can identify the parts where threads compute a larger number of operations per second.

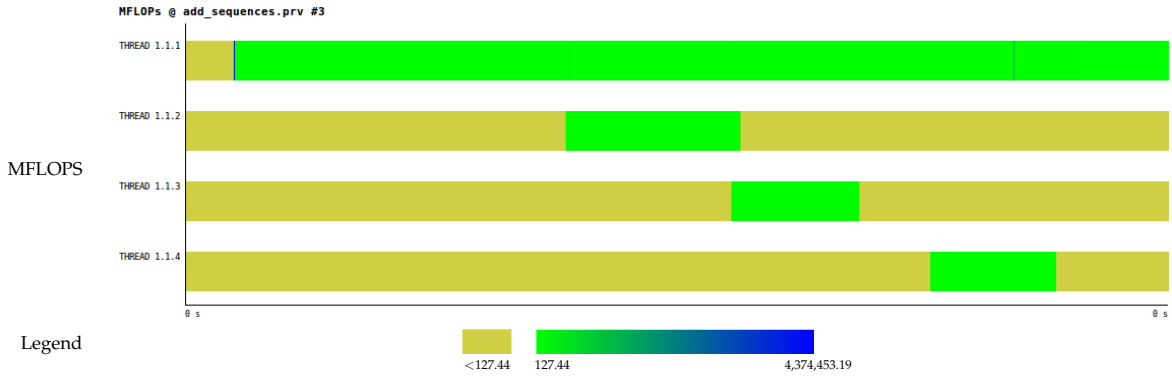


Figure 4-1: *Results obtained with the tracing submodule. MFLOPS metric has been used.*

Stream pattern example

This other example refers to a farm streaming pattern. Concretely, this application is composed of 3 stages: the first one belongs to the main function of the program (first bar/thread in the graph), the second one is the pipeline stage (second bar/thread in the graph) and the third stage corresponds with the farm stage. Since 4 threads (cardinality of the farm pattern) have been set, that makes a total of 6 threads as can be observed in the picture.

This application also deals with sequences, but in this case the computation has to do with rational numbers. Given a sequence of `n` natural numbers, for instance 5, the output sequence will look like as:

$$\frac{1}{1^2} \quad , \frac{1}{2^2} \quad , \frac{1}{3^2} \quad , \frac{1}{4^2} \quad , \frac{1}{5^2} \tag{4.7}$$

At first sight, we can identify much more colours with respect to the previous image. This is because here, there is a larger computation process, and since it is also a streaming application, items are not only computed by also enqueued and dequeued. That figure represents the normal trace, that is, over there we can identify the different actions performed by those 6 threads (enqueue, dequeue, compute, first stage, second stage, etc.), represented each with one different colour.

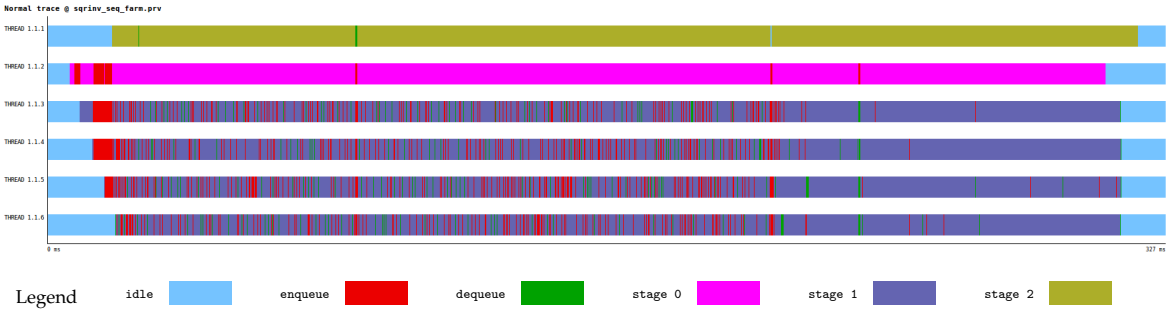


Figure 4-2: *Results obtained with the tracing submodule. The normal trace has been obtained.*

4.2.3 Native profiler submodule

Now, we will focus on the second submodule, the profiling, a more time-consuming process with respect to the already described tracing submodule, since as argued before, we needed to implement everything from scratch without the use of any additional tracing/profiling library.

The structure of this class comprises two main sections. The first one, the implementation of the six functions inherited from the *parent* class (as in the tracing submodule), but additionally, we implemented a second method (`print_statistics()`) which is in charge of plotting all the statistics recorded in those six functions.

Let us focus first on the six functions. As explained during the instrumentation process, each parallel pattern had associated one function. Inside each, we made calls to these six functions, but as a difference with the other submodule, now we do not instrument the patterns with Extrae functions but with timestamped utilities, that is, the *time elapsed* between the start and the end of some operation and the *current time*, both making use of the `chrono` library.

In that way, we could be able to track the activity of each thread during run-time. To do so, inside each of the six functions, we made use of several data structures to store that information regarding time. Specifically, we used the *map* data structure. A *map* container works like a vector but in each position it stores two fields: a *key* that helps us to index some element in the map, and a second field that stores the value itself (an integer, string, etc.).

In this class, it has been the most used data structure: to store the number of threads per stage, the total number of items been enqueued, computed or dequeued, the values of the mean, standard deviation and throughput per stage, etc. To understand how we proceeded with the implementation process and the usage of this type of container, we will show the implementation of two of the six functions:

Listing 4.6: *Implementation of the start_compute function within the tracing submodule.*

```
1 void start_compute (unsigned int stage_id){
2   // we get the thread identification number:
3   std::thread::id thread_id = std::this_thread::get_id();
4   // we store the current time in the times_thread_list map container
5   times_thread_list[thread_id] = std::chrono::steady_clock::now();
6 }
```

Listing 4.7: *Implementation of the stop_compute function within the tracing submodule.*

```
1  void stop_compute (unsigned int stage_id){
2    // we get the thread identification number:
3    std::thread::id thread_id = std::this_thread::get_id();
4    //we get the time at a current point in time:
5    std::chrono::time_point<std::chrono::steady_clock> end_time =
6         std::chrono::steady_clock::now();
7    mtx_.lock(); // critical section, blocked with a mutex
8      // time interval between the end and start of the compute operation:
9      auto elapsed_msecs = std::chrono::duration_cast<std::chrono::microseconds>
10         (end_time - times_thread_list[thread_id]).count();
11     // thread counter in that stage:
12     threads_per_stage[stage_id].insert(thread_id);
13     // mean calculation (see formula in~\ref{sec:metrics})
14     mean_values_comp[stage_id] = (((total_num_computes[stage_id] *
15         mean_values_comp[stage_id]) + elapsed_msecs) /(total_num_computes[stage_id] + 1));
16     // needed to calculate the std-deviation formula in print_statistics():
17     size_comp = ++total_num_computes[stage_id];
18     // needed to calculate the std-deviation formula in print_statistics():
19     total_cuadratic_value_compute[stage_id] += pow(elapsed_msecs, 2);
20   mtx_.unlock(); // we unlock the mutex
21 }
```

We have just shown the *compute* implementation functions. Each pair of functions (start/stop) sharing the same operation (in this case compute), are "connected" through the corresponding STL map containers. For example, the time stored in line 5 from first algorithm is also used in line 10 from the `stop_compute` algorithm to calculate the time spent between line 5 and line 10 in both pieces of code.

From line 14 in the second algorithm forward, we can observe how we proceed with the calculation of the accumulated mean, which was theoretically explained in Section 4.1. Besides, in line 17 and 19, two

additional steps are performed in order to store partial results that afterwards, in the `print_statistics` method, we will use to finally obtain the standard deviation. We proceeded with the four remaining functions exactly the same as with the two already explained, so we will not show them here.

Thanks to the map containers we could have information ready at any moment. By those six functions, we stored all the necessary information about the inner behaviour on the program talking in statistical terms, and then, by means of the other method `print_statistics()`, we could be able to plot a table summarizing that data. Within that method, we also need to distinguish between a data or streaming parallel pattern, since as we will identify in the two following figures, results are very different.

At the beginning of it, we computed the total throughput, the standard deviation at enqueue, compute and dequeue and the total standard deviation, this last is just the mean of the three previous values (the accumulated mean was already calculated in the six functions as we saw before). Besides, during those calculations, we tracked the maximum and minimum values by successive comparisons with some fixed values that were updated each time a new maximum or minimum was found.

Nevertheless, all those calculations would be computed according to the parallel pattern type. In case a data pattern was identified during the execution, we will only show the accumulated mean and the standard deviation (see Figure 4-3) in the result table, but only referring to the start/stop compute functions. In case a streaming pattern was identified, the set of its different stages are well differentiated in the resulted table (see Figure 4-4). We can distinguish between two different tables, the mean summary and the standard deviation summary. Within each, we can find one row per each stage, and within each row, we can observe the total enqueue, compute and dequeue times, both accumulated mean and standard deviation. On the rightmost part, we can also observe the total throughput and the total standard deviation. Apart from that, we have also implemented a table that comprises all the maximum/minimum values, but that table will be shown along with the applications in the evaluation chapter.

Finally, once having stored all the values, we just needed to print it. We designed a table format and applied and readjusted it for each of the different patterns. Afterwards, in the destructor of that class, we call this `print_statistics` method and users then could be able to see the final results.

As with the tracing submodule, we are going to show the results our profiler offers when executing applications. We will present next two pictures that correspond to the data pattern and streaming pattern used as an example in the other submodule respectively.

Data pattern example

As a data pattern is identified in the system, it will only execute the two functions above presented, since `start_compute ()` and `stop_compute ()` are the only functions executed by a data parallel pattern. We are only interested in the inner activity that takes place in a single stage. Therefore, the only metrics we can obtain now are the total accumulated mean and the standard deviation in that compute phase.

```
        ***** DATA PATTERNS STATISTIC SUMMARY *****
================================================================================
|                   Pattern |           Mean Time (µs) |       Std_Deviation (µs) |
--------------------------------------------------------------------------------
|                       Map |                     1273|                   323.192|
================================================================================


Total Execution time: 23 milliseconds
```

Figure 4-3: *Results after applying the profiling submodule to the data-pattern application.*

Throughput and maximum/minimum values make no sense here, since we only have one stage, compute stage, therefore they are not shown either calculated. At the bottom of the picture, we can observe the total execution time.

Stream pattern example

In this case, this pattern would execute the six functions, and a larger table will be obtained.

```
                    ***** STREAMING PATTERNS STATISTIC SUMMARY *****
 MEAN - THROUGHPUT SUMMARY:
================================================================================================================================
|               Pattern |               Stage_id |    Enqueue Mean Time (µs)|    Compute Mean Time (µs)|    Dequeue Mean Time (µs)|     Throughput (items/µs)|
--------------------------------------------------------------------------------------------------------------------------------
|              Pipeline |                     0|                   17.0593|               0.000999833|                         0|                 0.0586155|
--------------------------------------------------------------------------------------------------------------------------------
|              Pipeline |                     1|                   95.7853|                         0|                   2.22685|                 0.0025507|
--------------------------------------------------------------------------------------------------------------------------------
|              Pipeline |                     2|                         0|                   11.2877|                   2.04516|                 0.0750029|
--------------------------------------------------------------------------------------------------------------------------------

 STANDARD_DEVIATION SUMMARY:
================================================================================================================================
|               Pattern |               Stage_id | Enqueue Std_Dev Time (µs)| Compute Std_Dev Time (µs)| Dequeue Std_Dev Time (µs)|       Total Std_Dev (µs)|
--------------------------------------------------------------------------------------------------------------------------------
|              Pipeline |                     0|                   7.55221|                 0.0774532|                         0|                   2.54322|
--------------------------------------------------------------------------------------------------------------------------------
|              Pipeline |                     1|                   31.8233|                         0|                    7.2132|                   13.0122|
--------------------------------------------------------------------------------------------------------------------------------
|              Pipeline |                     2|                         0|                   1.90418|                   2.59352|                   1.49923|
--------------------------------------------------------------------------------------------------------------------------------
```

Figure 4-4: *Results after applying the profiling submodule to the stream-pattern application.*

4.2.4 Visualizing real-time data with the profiling submodule

Having explained the functioning of this submodule, we will show next an implemented additional tool within this class that we already mentioned before in this document.

This tool fits very well with any type of application for showing the application behaviour at real-time. Therefore, we decided to use this tool with the Mandelbrot set application, which is further explained in the evaluation phase in Chapter 5. This tool is called `feedgnuplot`. It is a general purpose pipe-oriented plotting tool able to read an input file and plot its content by means of a graph. Hence, we needed to create and fill in that file with the recorded information from the set of map containers before explained.

Specifically, we focused on the throughput metric. In the constructor of that class, we launched an auxiliary thread in charge of calculating the throughput and writing that result into a file, the one that `feedgnuplot` will use afterwards. One aspect to take into account was the refresh rate, that it, how much time did we want the graph to be updated. We set that value to 500 milliseconds so that we gave time to our eyes to see the evolution of the throughput values. However, the formula computed every 500 milliseconds by the aforementioned thread was not the one stated in Section 4.1, it looked as follows:

$$throughput = \frac{(number\_items(n) - number\_items(n-1))}{t(n) - t(n-1)} \tag{4.8}$$

where:

- *number_items (n)* is the number of stored items in the map container at point *n* in time (*number_items (n-1)* is the same but in the *n-1* instant).

- *t(n)* is the actual point in time and *t(n-1), the previous instant in time*.

Thus, we will show next in a graphical way, the process carried out when making use of this external software together with the profiling submodule:

By looking at Figure 4-5, during the execution of this type of application, an auxiliary thread is generated. This thread is in charge of calculating the throughput at each point in time, as well as writing that result in a specific file. This process is continuously repeated until the end of the execution application.

As a difference to the usually described throughput metric, here we are not focusing on the number of items being processed along the data stream, but throughput here it is a measure of the number of frames per second able to generate.

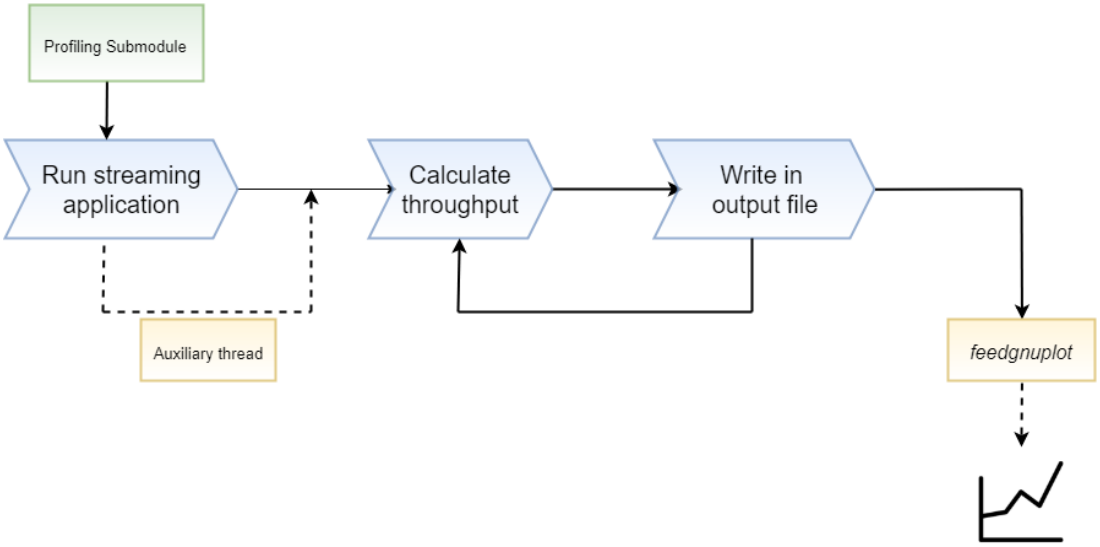


Figure 4-5: *Streaming application workflow with the profiling submodule.*

This process has been done in such a way that after the execution completion, that file contained 4 different columns: the first one regarding the subsequent values of the refresh rate, starting from 0 and updated each 500 milliseconds, and the three remaining ones, each one corresponding to a different stage, and representing the throughput value at that stage and in that refresh rate respectively.

Chapter 5

Evaluation

In this chapter, we evaluate the developed profiler-tracing environment through a set of different examples using the GrPPI parallel pattern interface. Through the application of these parallel patterns in different programs, we are able to analyze the applications' performance using different patterns and metrics. Thanks to that framework, we are able to identify potential bottlenecks that may limit at some point the performance. In the following sections, we will explain by separately the results obtained after the execution of each application.

The evaluated examples are key to demonstrate the feasibility and application of the proposed framework. The idea was to understand the results of these applications even though their descriptions might seem not complex talking about mathematical comprehension. For the sake of simplicity, this section only evaluates the GrPPI C++ threads back end.

The rest of this chapter is organized as follows. Section 5.1 describes the characteristics of the environment where this project has been accomplished. Section 5.2 lists a set of validation tests that will verify the system behaviour. Once been sure that our system reacts in a positive way to those tests, we will cover in Section 5.3 the followed experimentation regarding data parallel patters. Section 5.4 shows an experiment through the use of a task parallel pattern, and Section 5.4 experiments regarding stream parallel patterns. Finally, in Section 5.6, shows an additional analysis of a complex application making use of a stream parallel pattern, since as described in last chapter, `feedgnuplot` tool been used.

5.1 General working environment

As argued over the whole document up to this point, we have stated what our project would consist of. Therefore, in this section, we would like to briefly explain all the characteristics we can get from it, that is, under which resources specification this project has been carried out and where do we have tested our applications.

The system has been designed and implemented in a platform with the following specific characteristics:

- Processor Model: Intel Xeon 2.40GHz
- Sockets: 2
- Cores per socket: 12
- NUMA nodes: 2
- CPUs per Node: 12
- Total CPUs: 24
- Number of Hardware Counters: 50

Besides, the machine run over a Linux Ubuntu distribution with Version 14.04 and applications were compiled with the `gcc` compiler, Version 4.8.2.

With respect to the profiling and tracing tools, say that we have made use, first, of Extrae (Version 2.5.0) and Paraver software, and for the profiling one, `feedgnuplot` (Version 4.0), that we will later explain. The implemented performance analysis environment will work for those applications written in C++ language. Moreover, this system has been designed in a pattern-based parallel programming framework (GrPPI), that also works with several back ends, C++ threads, OpenMP and Intel TBB.

5.2 Validation tests

This section presents a battery of validation tests so that we can verify the list of requirements catalog described in Chapter 3. This is important to check if our system fulfills the set of functionalities before stated in the requirements list, or, in any other case, we need to take a step backwards and review the design decisions. Next, we will show the template used to verify the requirements throughout unit tests:

Test Case Id	XX
Description	Description of the test
Source Requirement	SR-FR-ZZ
State	Validated | Not Validated

Table 5.1: *Template for each unit test.*

XX means a number between 01 until 99, corresponding to the test identification number. Within each SR-FR-ZZ (system requirement that verified that proposed test), **ZZ** means a number between 01 and 99.

Having explained that, we list below the set of proposed tests:

Test Case Id	**UT-01**
Description	The system compiles an application
Source Requirement	SR-FR-01
State	Validated

Table 5.2: *Unit Test UT-01.*

Test Case Id	**UT-02**
Description	The system executes an application
Source Requirement	SR-FR-02
State	Validated

Table 5.3: *Unit Test UT-02.*

Test Case Id	**UT-03**
Description	The system notifies the user when an error appears during execution
Source Requirement	SR-FR-03
State	Validated

Table 5.4: *Unit Test UT-03.*

Test Case Id	**UT-04**
Description	The system calculates the accumulated mean, standard deviation, throughput and total execution time values
Source Requirement	SR-FR-04, SR-FR-07, SR-FR-10, SR-FR-13
State	Validated

Table 5.5: *Unit Test UT-04.*

Test Case Id	**UT-05**
Description	The system calculates all maximum and minimum values
Source Requirement	SR-FR-05, SR-FR-06, SR-FR-08, SR-FR-09, SR-FR-11, SR-FR-12
State	Validated

Table 5.6: *Unit Test UT-05.*

Test Case Id	**UT-06**
Description	The system executes the application and creates a extrae-profiler object when using the profiling submodule
Source Requirement	SR-FR-14, SR-FR-15
State	Validated

Table 5.7: *Unit Test UT-06.*

Test Case Id	**UT-07**
Description	The system executes the application and creates a native-profiler object when using the tracing submodule
Source Requirement	SR-FR-16, SR-FR-19
State	Validated

Table 5.8: *Unit Test UT-07.*

Test Case Id	**UT-08**
Description	The system generates a trace file and notifies the user when it has been generated
Source Requirement	SR-FR-17, SR-FR-18
State	Validated

Table 5.9: *Unit Test UT-08.*

Test Case Id	**UT-09**
Description	The system runs if the application contains map patterns
Source Requirement	SR-FR-20
State	Validated

Table 5.10: *Unit Test UT-09.*

Test Case Id	**UT-10**
Description	The system runs if the application contains map-reduce patterns
Source Requirement	SR-FR-21
State	Validated

Table 5.11: *Unit Test UT-10.*

Test Case Id	**UT-11**
Description	The system runs if the application contains stencil patterns
Source Requirement	SR-FR-22
State	Validated

Table 5.12: *Unit Test UT-11.*

Test Case Id	**UT-12**
Description	The system runs if the application contains divide and conquer patterns
Source Requirement	SR-FR-23
State	Validated

Table 5.13: *Unit Test UT-12.*

Test Case Id	**UT-13**
Description	The system runs if the application contains farm patterns
Source Requirement	SR-FR-24
State	Validated

Table 5.14: *Unit Test UT-13.*

Test Case Id	**UT-14**
Description	The system runs if the application contains stream-filter patterns
Source Requirement	SR-FR-25
State	Validated

Table 5.15: *Unit Test UT-14.*

Test Case Id	**UT-15**
Description	The system runs if the application contains stream-reduction patterns
Source Requirement	SR-FR-26
State	Validated

Table 5.16: *Unit Test UT-15.*

Test Case Id	**UT-16**
Description	The system runs if the application contains stream-iteration patterns
Source Requirement	SR-FR-27
State	Validated

Table 5.17: *Unit Test UT-16.*

Test Case Id	**UT-17**
Description	The system runs the application with the `feedgnuplot` software
Source Requirement	SR-FR-28
State	Validated

Table 5.18: *Unit Test UT-17.*

Next, we will show in a tabular way the correspondence between each designed unit test and the system requirement it is based on:

	UT-01	UT-02	UT-03	UT-04	UT-05	UT-06	UT-07	UT-08	UT-09	UT-10	UT-11	UT-12	UT-13	UT-14	UT-15	UT-16	UT-17
SR-FR-01	✗																
SR-FR-02		✗															
SR-FR-03			✗														
SR-FR-04				✗													
SR-FR-05					✗												
SR-FR-06					✗												
SR-FR-07				✗													
SR-FR-08					✗												
SR-FR-09					✗												
SR-FR-10				✗													
SR-FR-11					✗												
SR-FR-12					✗												
SR-FR-13				✗													
SR-FR-14						✗											
SR-FR-15						✗											
SR-FR-16							✗										
SR-FR-17								✗									
SR-FR-18								✗									
SR-FR-19							✗										
SR-FR-20									✗								
SR-FR-21										✗							
SR-FR-22											✗						
SR-FR-23												✗					
SR-FR-24													✗				
SR-FR-25														✗			
SR-FR-26															✗		
SR-FR-27																✗	
SR-FR-28																	✗

Table 5.19: *Traceability matrix for unit tests.*

Once checked that our system passes all the proposed tests, and that every system requirement is associated to at least one unit test, we can now continue with the evaluation of the different proposed parallel applications.

5.3 Data parallel applications

In this section, we evaluate three different applications making use of the patterns: map, map-reduce and stencil. The evaluation is performed in terms of profiles and traces as we will observe later on.

5.3.1 Matrix-vector product

This application assesses the matrix-vector product. Sometimes, machines need to compute large amounts of data based on almost similar numerical operations. Most of the times, 10% of the total execution time is dissipated in these kinds of operations, so, if we were able to optimize this percentage, faster results will got [49]. The scalar product, dot product, rotations, etc. are some of the most commonly used operations involving this 10% of numerical computations.

For this reason, a generic specification of the combination of these transformations over numerical data has been developed to make programming easier for users. Concretely, we can talk about Basic Linear Algebra Subprograms (BLAS). BLAS is composed of a variable set of low-level subroutines that compact operations like the former ones in order to gain efficiency. It has been proved that these operations are used and available for any type of high-quality linear algebra software, since they are all competent [50].

It provides basic vector and matrix operations by defining a set of different subroutines divided into three level types. Level 1 refers to subroutines regarding simpler operations, in particular, vector, scalar and vector-vector operations. Level 2, a bit complex one, performs matrix-vector operations and Level 3, performs matrix-matrix operations. Therefore, this exercise will refer to the second level type.

From BLAS, we can focus on the matrix or vector product multiplication (its associated subroutine in BLAS is also known as `gemv`). There exist two variants from this subroutine depending on which precision we want to use, single-precision or double one. This application focuses on the double-precision, so its subroutine real name is `dgemv`.

In this way, we will proceed with the product between a matrix called A and a vector x. The only restriction is that this operation can be performed only in the case in which the number of columns in A equals the number of rows in x, as it is a basic algebra property regarding the product. The product can be defined as Ax, where A is a $m \times n$ matrix and x is a $n \times 1$ column. The matrix-vector product results in b, which is a $m \times 1$ column vector. The general formula for a matrix vector product is given by:

$$Ax = b \implies \begin{pmatrix} a_1 & a_2 & a_3 & \cdots & a_n \\ b_1 & b_2 & b_3 & \cdots & b_n \\ c_1 & c_2 & c_3 & \cdots & c_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_1 & m_2 & m_3 & \cdots & m_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_1x_1 & a_2x_2 & a_3x_3 & \cdots & a_nx_n \\ b_1x_1 & b_2x_2 & a_3x_3 & \cdots & b_nx_n \\ c_1x_1 & c_2x_2 & a_3x_3 & \cdots & c_nx_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_1x_1 & m_2x_2 & a_3x_3 & \cdots & m_nx_n \end{pmatrix}.$$

Recall that this matrix-vector product is based on the scalar/dot product (`ddot` in BLAS): this transformation sequentially takes a pair of vectors (one row in the matrix `A` and the vector `x` itself) and multiplies both until reaching the last row of the matrix, giving always as a result a number. Hence, our aim is trying to simulate these two subroutines, `dgemv` and `ddot`, by means of data parallel patterns, in order to fulfill the aforementioned request. Specifically, the *map* pattern will be used.

After this theoretical explanation, we show a brief piece of code to see how do we call this specific pattern within our application:

Listing 5.1: *Example header of the map pattern.*

```
1  grppi::map(ex, begin(mat), end(mat), begin(res),
2    [&] (std::vector<double> row) {
3      double res= 0;
4      for (int i = 0; i < row.size(); i++)
5        res += row[i] * vec[i];
6      return res;
7    }
8  );
```

The set of parameters this program requires to run are five, as detailed in Table 5.20:

Parameter	Description	Used Parameter
Column Number	An integer number greater than 0	20000
Row Number	An integer number greater than 0	20000
Execution Mode	Available options are “seq”, “omp”, “thr”	thr
Number of Threads	An integer number greater than 0	8

Table 5.20: *Required parameters to run the matrix-vector product application.*

The command line used to execute the program looked as follows:

```
./dgemv 20000 20000 thr 8
```

We now leverage the proposed profiling-tracing framework to analyze the performance of the matrix-vector product. This analysis is key to gain insights about the application performance and to improve potential bottlenecks.

First, we have obtained a statistic summary by using the implemented profiling submodule. The following table shows the statistical information recorded by this matrix-vector product in a matrix of 20000 x 20000 by a vector of 20000 elements:

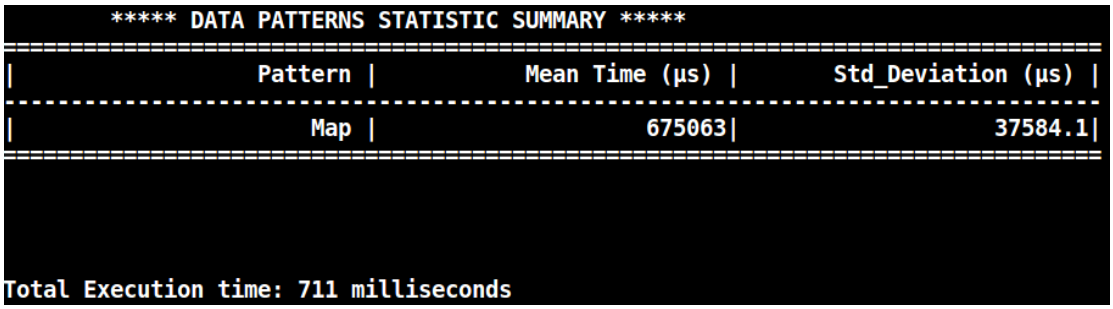


Figure 5-1: *Statistical summary after the execution of the matrix-vector application.*

Since this application makes use of the map parallel pattern, the profiling metrics represent total accumulated mean at compute phase and its corresponding standard deviation. The total mean is 0.675 seconds, so we can observe that actually, it is a very short application in terms of duration.

Once obtained this general profile (in terms of execution time), we now show an execution trace of the same application using the developed tracing submodule.

Focusing on the execution trace of this application with respect to MFLOPS (see Figure 5-2), we can observe the behaviour of the different threads. In the legend, the number of operations per second (FLOPS) from the green colour to the blue one is represented. Eight different threads have been launched to compute that matrix-product, and we can observe by showing at the blue colour when threads start and stop computing. We can also observe that the longest bar is the first one: this is because this thread is in charge of initializing the matrix, thus having much more work than the rest.

Regarding the figure at the bottom, we can observe that colours are just opposite with respect to the previous picture. This is because when the computation process is larger, that implies we dispose all the data and we do not spend time acquiring it, so L3 TCM (Total Cache Misses) consequently, decrease (see metrics explanation in Section 4.1).

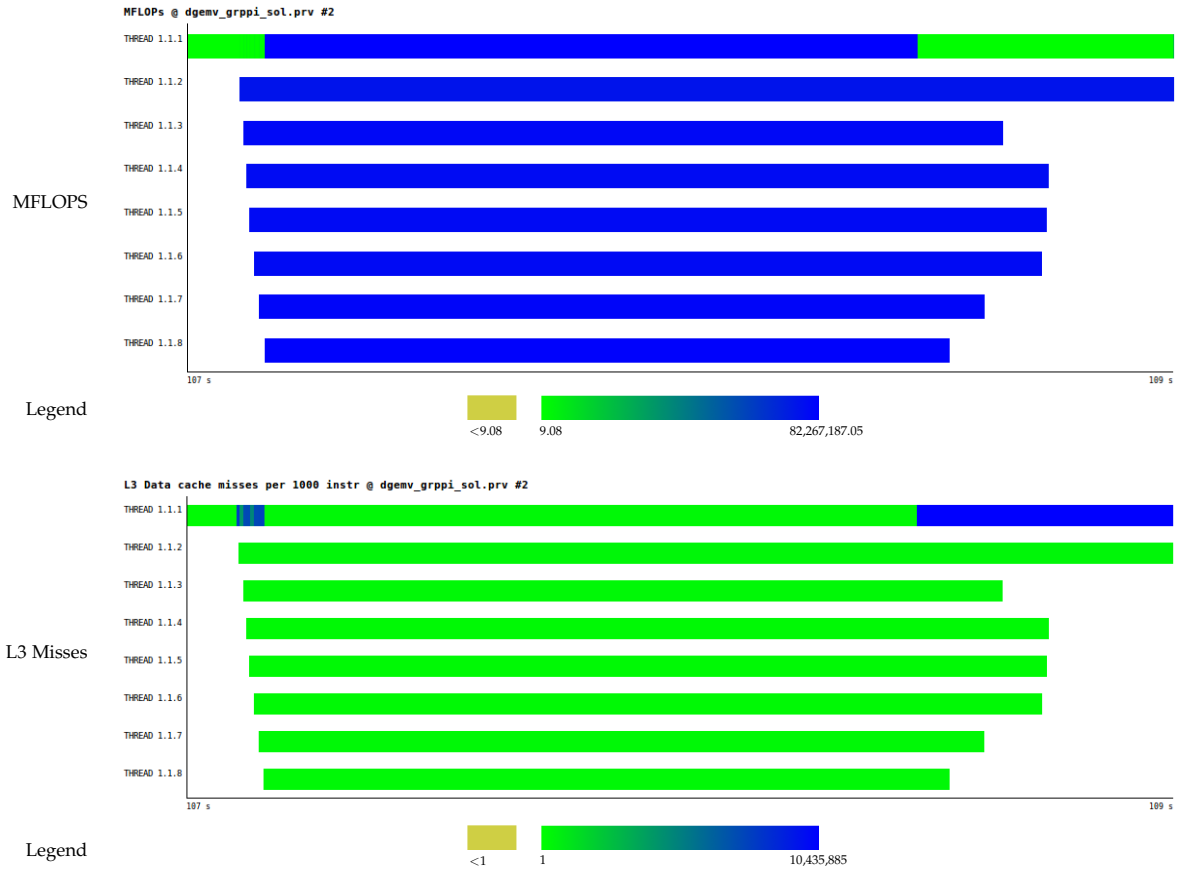


Figure 5-2: *Trace of matrix-vector product with Paraver tool. Top: MFLOPS metric. Bottom: L3 TCM.*

5.3.2 Word-count

This concrete application receives an input file with a specific length in order to compute the number of times each word appears within that text. To do so, this application makes use of the *map-reduce* pattern.

First, the *map* part is in charge of storing the first appearance of each word in a data container. This data container is the *"map"* structure that we already studied in Section 4.2.3 (do not confuse with the name of the map pattern). Here, this map stores per each element, the name of the word (key), and an integer that indicates the number of times that the word appears in the text. The declaration of this container is:

```
std::map <string, int> total_words;
```

where the `string` refers to the word itself and the `int` to the number of occurrences.

Once the *map* part of this pattern has tracked one appearance of each word, then, the *reduce* part will be the one that accumulates the following appearances of each word (in case they are more) into its

corresponding position in the map container. After the application execution, we will have collected a number per each word in the text.

This is a quite simple but powerful example regarding *Text Mining*. Text Mining is the process of extracting information from unstructured data coming from different texts written in plain text. Since this process manually will take us so much time finding a pattern or some interesting part in a text, actually there exist some NLP algorithms that avoid us performing that task [51]. This activity is commonly used, for instance, in business or marketing. From online reviews, emails, tweets, comments, etc. we can know what customers demand and what they think about some concrete products. Next, we show a brief piece of code to see how do we call this specific pattern within our application:

Listing 5.2: *Example header of the map-reduce pattern.*

```
1  auto result = map_reduce(ex, words.begin(), words.end(), std::map<string,int>{},
2    [ ](string word) -> std::map<string,int> { return {{word,1}}; }, // map
3    [ ](std::map<string,int> & lhs, const std::map<string,int> & rhs)
4       -> std::map<string,int> & { // reduce
5       for (auto & w : rhs) {
6         lhs[w.first]+= w.second;
7       }
8       return lhs;
9    }
10 );
```

To run the previous pattern-based application, the following parameters have to be provided:

Parameter	Description	Used Parameter
Input file name	The name of the input file	input.txt
Execution Mode	Available options are “seq”, “omp”, “thr”	thr

Table 5.21: *Required parameters to run word-count application.*

The command line used to execute the program looked as follows:

```
./word_count input.txt thr
```

Let us look at Figure 5-3, which corresponds with the profiling submodule. As a comparison with the previous values, here the compute mean time and the standard deviation are not so large. As a consequence, the total time execution is very low, 2 milliseconds.

The input file used to run this program was a piece of news extracted from a sports newspaper, concretely it contained 1006 characters, but any other input text is also valid. Besides, we launched 8 threads to complete the execution of this program.

```
        ***** DATA PATTERNS STATISTIC SUMMARY *****
=============================================================================
|                  Pattern |          Mean Time (µs) |       Std_Deviation (µs) |
-----------------------------------------------------------------------------
|               Map-Reduce |                  35.375|                  16.0619|
=============================================================================


Total Execution time: 2 milliseconds
```

Figure 5-3: *Profiling statistical summary of word-count application.*

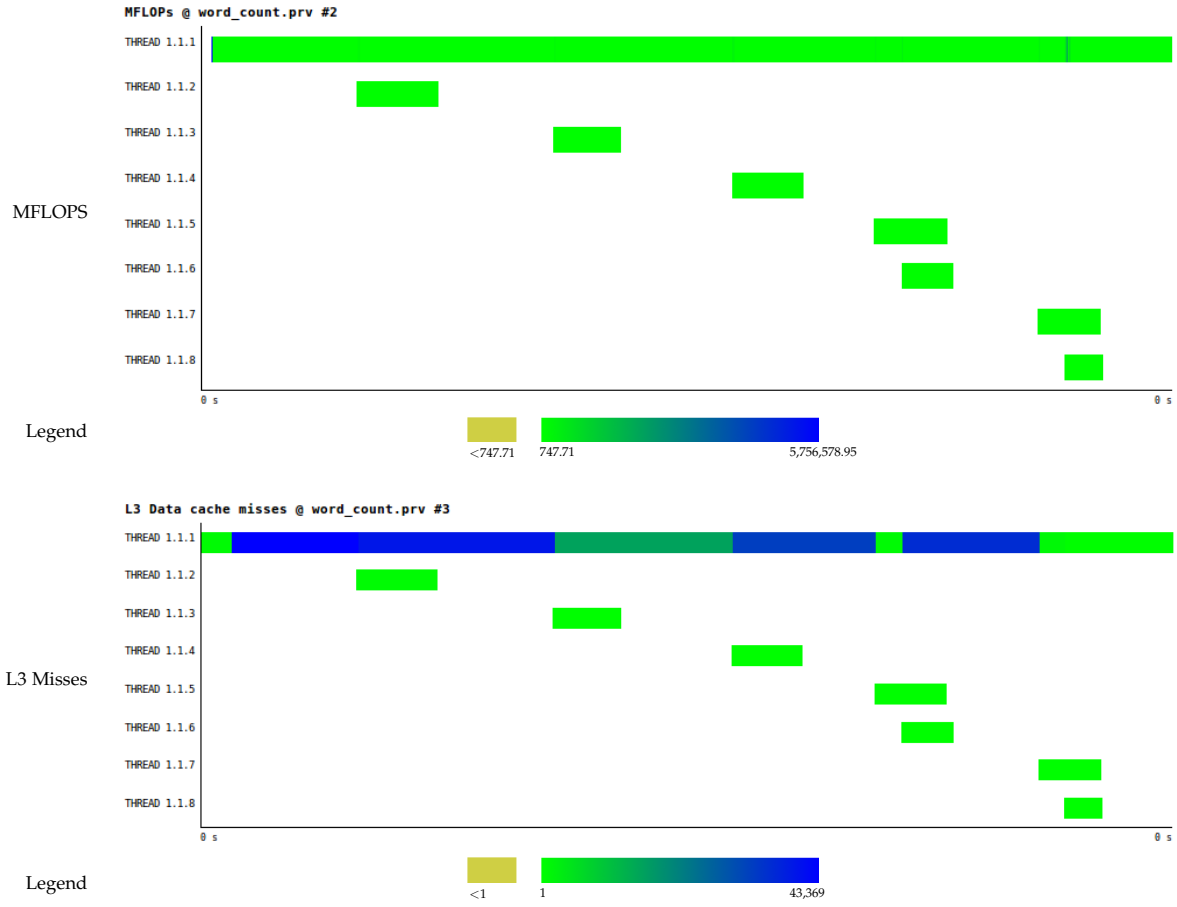


Figure 5-4: *Trace of word-count with Paraver tool. Top: MFLOPS metric. Bottom: L3 TCM.*

Now, by looking at the two figures below, we can observe the use of the same tracing metrics as in the previous pattern example (MFLOPS and L3 TCM). Here again, 8 threads have been launched, but now,

we can identify shorter bar colours from the second thread until the eighth, which means that the time threads are computing is less than before (that is logic since only 2 milliseconds are needed to complete the total execution of the program). If we take a look at the first thread, however, a much larger bar is observed. Besides, several different hues are observed in the figure on the bottom, not only green ones but also blues. Values are indicated in the corresponding legend.

5.3.3 Blur filtering operation

This application is related to image processing, specifically with a Gaussian blur filter (that is the reason for the exercise name). To implement that Gaussian filter, we have made use of the *stencil* parallel pattern. Any Gaussian filter is the result of applying a Gaussian function. A unidimensional Gaussian function is given by the following formula [52]:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} * e^{\frac{-x^2}{2\sigma^2}} \tag{5.1}$$

where σ is the standard deviation of the distribution. It is a very practical technique, since only the modification of one variable, the standard deviation, needs to be taken into account.

The Gaussian function is used in numerous research areas [52]: it defines a probability distribution for noise or data, it is a smoothing operator and it is used in mathematics.

This application focuses practically on all the previously mentioned areas, since this effect is widely used in graphics software with a main objective: reducing the image noise and detail of the image in such a way that the result is more suitable than the original image for a specific application. As a consequence of applying this filter, the input image gets blurred in a smooth way once we look at that image throughout a translucent screen [53]. As explained in Section 2.1, *stencil* pattern makes use of a neighbourhood so that it can compute. In mathematical terms, the action of applying any blur filter to some image is equivalent to performing a convolution to the image with a Gaussian function or kernel.

Given a neighbourhood, there are many variants regarding the operations we can apply over the pixels of an image. The possibles ones are: sum, weighted sum, average, weighted average, min, max and median [53]. However, the most common operation is to multiply each of the pixels of the previously defined neighbourhood by a specific kernel and adding them all values into a single one. That final result corresponds to the final value in the already filtered image, as we can observe in Figure 5-5:

As it can seem, the convolution computations regarding each pixel of the image is an embarrassingly parallel process, given that each pixel of the resulting image can be computed independently. For that reason, this pattern is applied here.

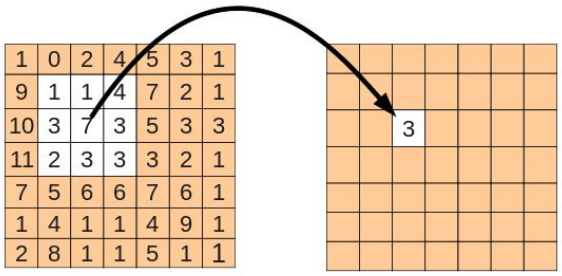


Figure 5-5: *Application of a 3x3 filter to one specific pixel in the image [53].*

Next, we show a brief piece of code to see how do we call this specific pattern within our application:

Listing 5.3: *Example header of the blur pattern.*

```
grppi::stencil(ex, begin(frame), end(frame), begin(result),
    [&](auto it, auto neighbours) {
      // apply kernel to one pixel plus its neighbours
    },
    [&](auto it) {
      // return neighbours
    }
);
```

Regarding the parameters this application requests, they are:

Parameter	Description	Used Parameter
Filter	The kernel used to modify the original image	kernel_avg7.txt
Input file name	The name of the input file	terragen.bmp
Output file name	The name of the output file	out.bmp
Execution Mode	Available options are “seq”, “omp”, “thr”	thr
Number of Threads	An integer number greater than 0	7

Table 5.22: *Required parameters to run blur application.*

The command line used to execute the program looked as follows:

```
./blur kernel_avg7.txt terragen.bmp out.bmp thr 7
```

where the file used for applying the filter (`kernel_avg7.txt`) occupies 14 bytes, both input and output images are 900.05 KB and the number of threads is 7.

Let us see an example of a real case. In Figure 5-6, we can observe two different subfigures. The one on the left corresponds to the original one, while the one on the right is the result of applying the blur filter implemented by means of the *stencil* pattern. As a consequence, the image gets blurred and shows that appearance.

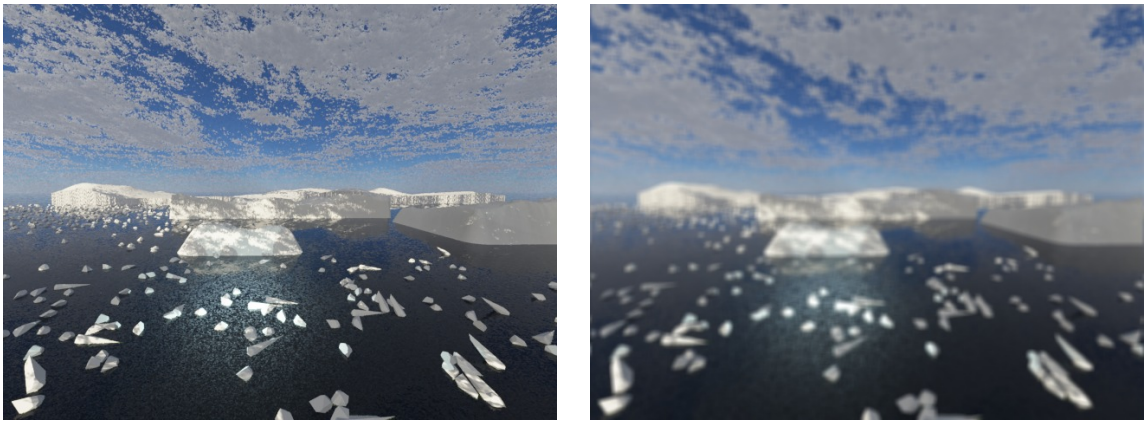


Figure 5-6: *Differences between applying (or not) a Gaussian filter in blur application.*

Next, we will show the results obtained with the profiling submodule. Figure 5-7 shows that the used pattern is the stencil one, and that the total mean at compute is 0.353 seconds. This implies that from the total execution time, almost every millisecond is expired in computation tasks. Besides, the standard deviation is quite low in comparison to the mean, so this is a good result. Mean time is the average time spent to compute all tasks from this applications by using this concrete pattern (the same happens with the standard deviation).

```
       ***** DATA PATTERNS STATISTIC SUMMARY *****
================================================================================
|                 Pattern |          Mean Time (µs) |      Std_Deviation (µs) |
--------------------------------------------------------------------------------
|                 Stencil |                  353050|                  1906.13|
================================================================================



Total Execution time: 358 milliseconds
```

Figure 5-7: *Statistical summary obtained when running the blur application.*

Let us focus now on the tracing part, where MFLOPS and L3 TCM have been applied. With respect to the MFLOPS metric, first picture, it seems only green colour is present there, but actually different green hues are camouflaged.

In this example, we have launched 7 threads, and we can observe that the last six ones almost perform the same amount of work, only that each one starts computing a bit earlier than the following one. However, the first thread does a higher amount of work, this is because it needs to calculate not only the value of the pixel itself but its neighbours around it. Regarding the L3 TCM, we can observe that the graph shows a similar shape than the MFLOPS one. However, according to the legend, we can identify a greater number of hues. Again, we can observe how green colours with lower values in the figure on the top, are much smaller than the same parts of the graph but in the figure on the bottom (which represents highest values in blue hues).

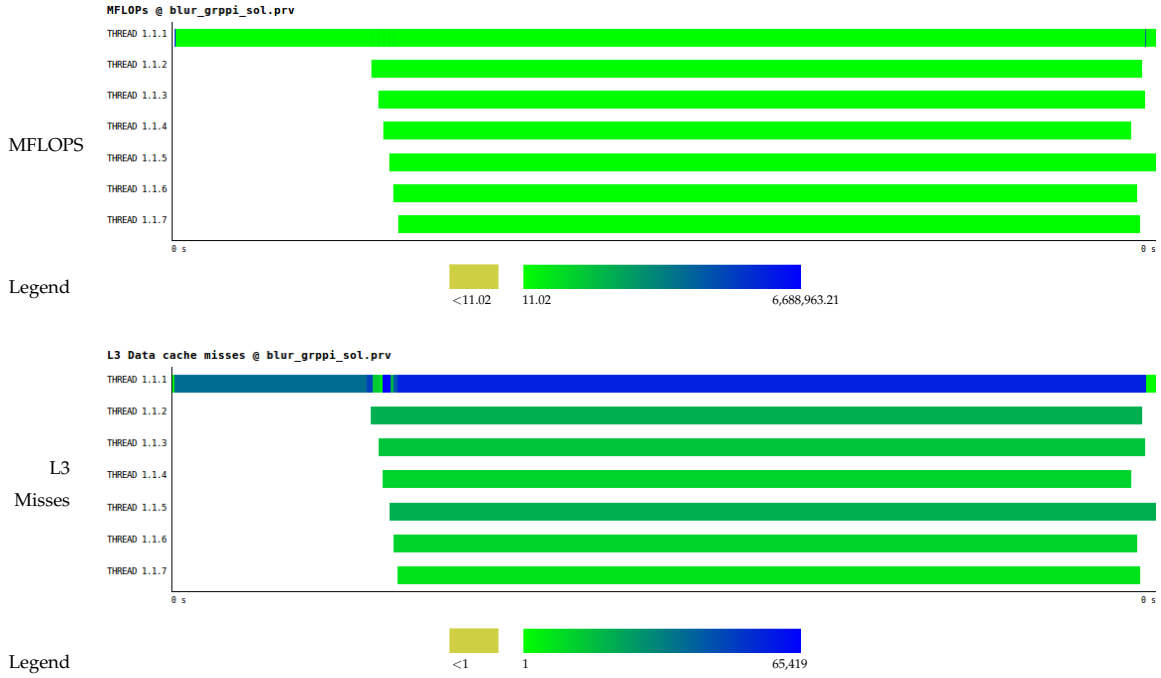


Figure 5-8: *Trace of blur with Paraver tool. Top: MFLOPS metric. Bottom: L3 TCM.*

5.4 Task parallel applications

In this section, we present an example of this type of parallelism by means of the divide and conquer parallel pattern. As with the previous patterns, both profiles and traces will be obtained and commented.

5.4.1 Mergesort

The *Mergesort* problem is a well-known comparison-based sorting algorithm. It was invented by the mathematician John von Neumann in 1945 [54]. This problem can be solved using the *divide and conquer* parallel pattern.

As its own name says, the objective of this algorithm is to reorder the elements of an unsorted array. The way it is done is by means of recursive divisions of the array A into two subarrays of equal size until their size is 1. At that point, a recombination (merge) of those single elements is carried out, so that the final and sorted array is obtained.

This pattern is composed of three parts, so, in order to apply it, these three phases need to be distinguished in the corresponding problem (*mergesort* here). Regarding the *divider*, this is the one in charge of taking the initial array and dividing it into two subproblems each time (this process ends when the subarray has been split down into a single item). Next, the *solver* phase (or *conquer*), is in charge of solving the pair of previous generated subproblems. Finally, the *combiner*, it just aggregates the partial solutions from the previous step into one final result, getting in this way the desired sorted array. This process can be shown in Figure 5-9:

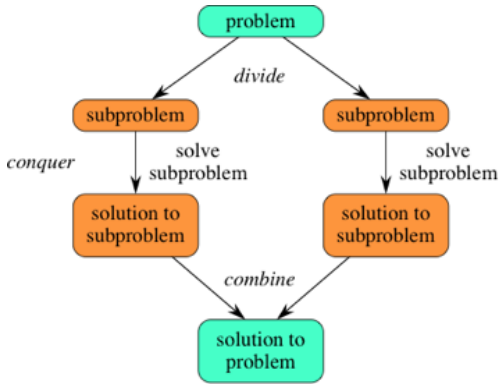


Figure 5-9: *Scheme showing the three phases of the divide and conquer algorithm [55].*

Another interesting topic regarding this algorithm is its computational complexity. We can analyze it from three distinct points of view: the best case, the average case and the worst case, in all cases complexity is the same. Specifically, time complexity takes `n * log (n)` to solve the problem, where `n` is the input size of the problem (in this case, the array length).

That complexity can be considered as one of the advantages of this sorting algorithm, since not all of them takes that time to finish. Besides, the number of comparisons it performs is quite optimal. On the contrary, the disadvantage is that it continuously needs to dynamically allocate memory, action that may penalize the total time to get elements ordered [56].

The whole scheme of *mergesort* algorithm can be represented as a binary tree, since each time a new level of recursion is reached. An example of this structure can be observed in Figure 5-10.

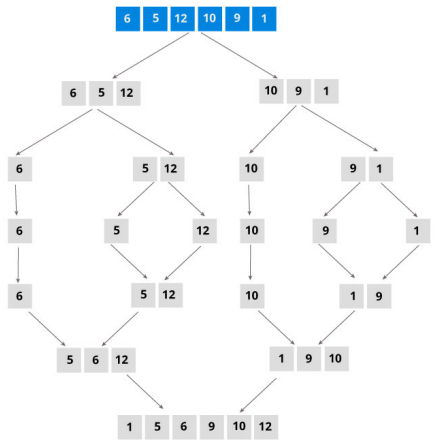


Figure 5-10: *Process of a recursively sorting algorithm with a concrete example [57].*

As we can identify in that example, the input size of the vector is 6. We start splitting the array into two (if it was an odd number, it would not matter, we would take one more element into one arbitrary side). We continue dividing both sides until size 1 is obtained on each one. At that point, we compare those numbers by parts and directly positioning them in a sorted way.

Next, we show the corresponding piece of code so that we can make use of this concrete pattern:

Listing 5.4: *Example header of the mergesort pattern.*

```
1  grppi::divide_conquer(ex, r,
2      [ ](auto &sequence) -> std::vector<range> {
3        // Divider
4      },
5      [ ](auto &sequence) -> std::vector<int> {
6        // Solver
7      },
8      [ ](auto &first, auto &second) {
9        // Combiner
10     }
11 );
```

The following table shows the needed parameters for executing the application:

Parameter	Description	Used Parameter
Vector size	An integer number greater than 0	5000
Output size	The name of the file to which result is stored	output_file.out
Execution Mode	Available options are “seq”, “omp”, “thr”	thr
Number of Threads	An integer number greater than 0	6

Table 5.23: *Required parameters to run the mergesort application.*

The command line used to execute the program looked as follows:

```
./mergesort 5000 output_file.out thr 6
```

where the vector is composed of 5000 unsorted elements, `output_file.out` is the output file where result is stored, we have used the `thread` execution mode with a number of 6 threads.

Given the behaviour of this algorithm, where different subarrays in the process can be sorted independently, the sorting process can be parallelized by means of the aforementioned task parallel pattern. Figure 5-11 shows the statistical report obtained with the profiling submodule. Total execution time is 11 ms, and the total mean at compute stage is 1.55 μs. Standard deviation is 0.95 units separated from the mean value.

```
        ***** DATA PATTERNS STATISTIC SUMMARY *****
================================================================================
|                  Pattern |           Mean Time (µs) |       Std_Deviation (µs) |
--------------------------------------------------------------------------------
|       Divide and Conquer |                  1.55556|                  0.955814|
================================================================================


Total Execution time: 11 milliseconds
```

Figure 5-11: *Statistical summary obtained after running mergesort application.*

In the following two pictures we can distinguish between 10 threads unless we had stated 6 in the command line. This is because apart from those 6 threads, the divide and conquer part of the program uses 3 more threads (one per phase), plus the one regarding the main thread in the program, which makes a total of 10. Again, the first thread is the one that makes more computations, since it needs apart to initialize the whole input vector.

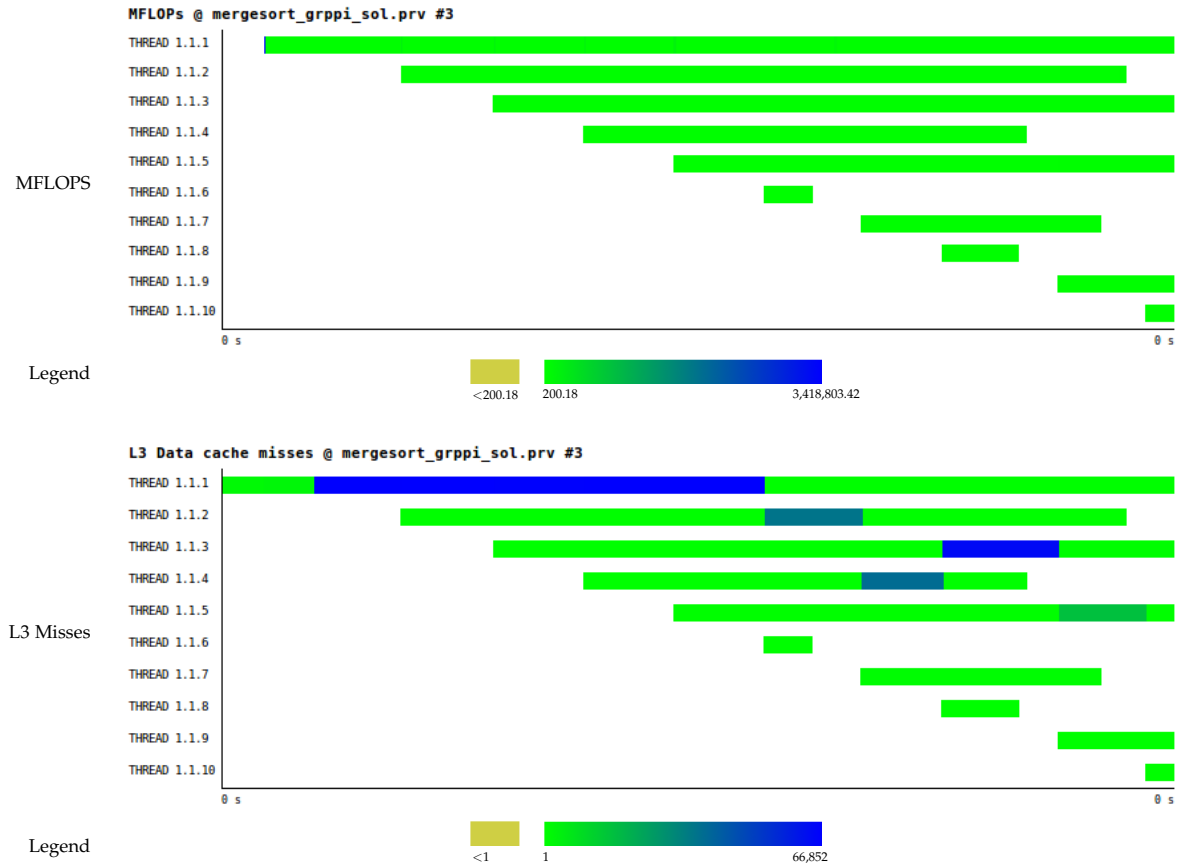


Figure 5-12: *Trace of mergesort with Paraver tool. Top: MFLOPS metric. Bottom: L3 TCM.*

5.5 Streaming parallel applications

Regarding streaming parallel patterns, we will give different examples based on a *pipeline* structure. Firstly, the results of the *farm* pattern will be exposed. Then, the *stream filter* followed by the *stream reduction* and *stream iteration*. And finally, in a separate subsection, an explanation of the Mandelbrot fractal will also be given.

5.5.1 Swaptions

The swaptions application is focused on the economic world and is based on the Heath Jarrow Morton (HJM) method in order to price a portfolio of financial options [58]. This example has been adapted and comes directly from GrPPI library, but originally it was developed by Princeton Application Repository for Shared-Memory Computers (PARSEC) team [59].

As defined in [60], HJM model is a very general framework used for pricing interest rates and credit derivatives. There exist different types of derivatives, and banks need to have a technological framework that accommodates new payoffs. This financial framework is simulated by means of the Monte Carlo (MC) method. Monte Carlo simulation is a stochastic (random) method very used when we need to solve a statistical problem [61]. By using it, a distribution of all the statistical results can be obtained. This method has numerous applications, one of them regarding the portfolio management. It allows analysts to know the size of a portfolio and get the expected distributed values this algorithm gives back. Concretely, in this example, this method has been used to price fixed interest rate derivatives. Therefore, swaptions employs MC simulation to compute the prices of a portfolio [62].

Leaving apart the theoretical concepts, we will explain how this methodology is applied to our parallel program by means of the aforementioned parallel pattern. Basically, the program stores a portfolio into an array. Each swaption (and thread) will correspond to a different position in that array, and consequently, each thread will be focused on computing one specific block for getting the final price. The array will be initially divided into an equal number of blocks, each one associated with one thread. Each thread will also invoke another method so that a HJM path for each MC run can be generated. Moreover, as a result of those computations, not only the prices will be obtained, but also the standard error.

Specifically, this pattern has to be defined over an outer pattern (in this case the pipeline one), so this application has been designed as a composable function of these two patterns. Next, we show the corresponding piece of code at a high-level so that we can know how to use this pattern:

Listing 5.5: *Example header of the swaptions pattern.*

```
1  grppi::pipeline(ex,
2      [&]() -> optional<parm> {
3        // pass vector index to the farm stage
4       grppi::farm(4,[&](parm swaptions) { // cardinality: 4 threads
5         // calculate swaption price
6        [&](parm swaptions) {
7         // print swaptions
8        }
9       }
10     }
11 );
```

Several configuration parameters are needed in order to get these prices. They can be obtained in the following list:

Parameter	Description	Used Parameter
Number of simulations	An integer number greater than 0 (-sm)	3
Number of threads	An integer number greater than 0 (-nt)	1
Number of swaptions[1]	Number of swaptions (blocks) in which the array will be divided (-ns)	4
Seed	A random seed that produces this randomness of MC (-sd)	1

Table 5.24: *Required parameters to run swaptions application.*

The command line used to execute the program looked as follows:

```
./swaptions -ns 4 -sm 3 -nt 1 -sd 1
```

where `-ns` is the number of swaptions (vector size), `-sm` is the number of simulations (like trials), `-nt` is the number of threads used to run the application and finally, `-sd`, which corresponds to the seed (used to simulate the MC random process).

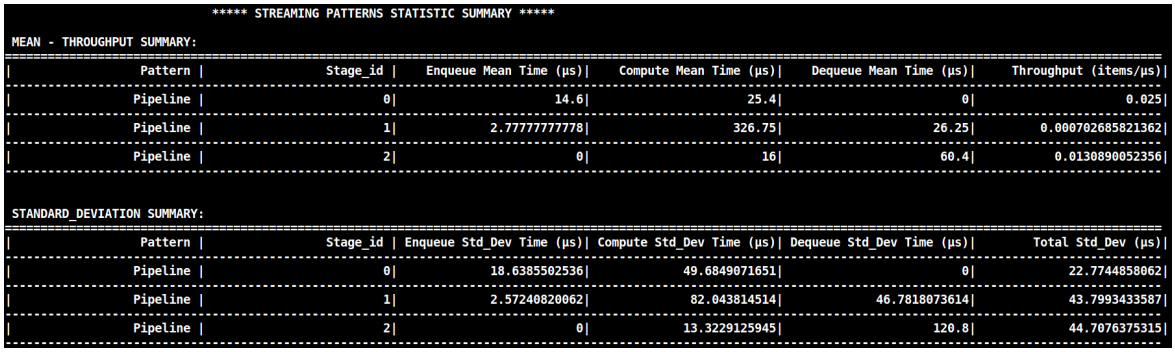
From now on, the set of figures obtained thanks to the profiling submodule will differ from the previous examples. This is because now we will focus on streaming patterns, and as argued in the previous chapter when showing the pair of examples, here we will get a large table together with the one regarding the maximum and minimum values.

In Table 5-13, we distinguish between the three phases (enqueue, compute and dequeue), for each stage in the pipeline with respect to the mean and standard deviation tables. Analyzing its figures, we can observe that the most time-consuming part during the whole execution takes place in the second stage at compute phase, with a mean value of 326.75μs. However, since it takes much time to compute, that is because it lasts much time when processing the elements, which is the same as saying that the throughput is smaller, with a value of 0.007μs.

Regarding the standard deviation, it happens the same since that metric totally depends on the

[1]Important to say that the number of swaptions must be greater than the number of threads.

accumulated mean. Moreover, we can observe the maximum and minimum values tracked during that execution, which are shown in Table 5-14.



***** STREAMING PATTERNS STATISTIC SUMMARY *****

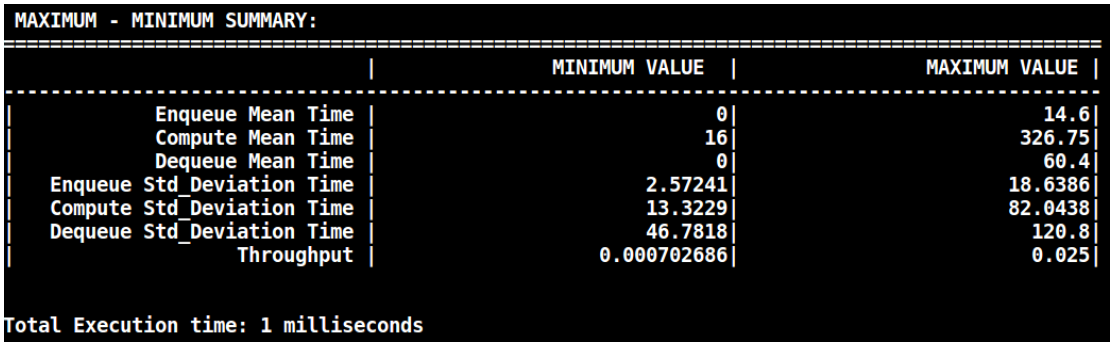
MEAN - THROUGHPUT SUMMARY:

Pattern	Stage_id	Enqueue Mean Time (µs)	Compute Mean Time (µs)	Dequeue Mean Time (µs)	Throughput (items/µs)
Pipeline	0	14.6	25.4	0	0.025
Pipeline	1	2.77777777778	326.75	26.25	0.000702685821362
Pipeline	2	0	16	60.4	0.0130890052356

STANDARD_DEVIATION SUMMARY:

Pattern	Stage_id	Enqueue Std_Dev Time (µs)	Compute Std_Dev Time (µs)	Dequeue Std_Dev Time (µs)	Total Std_Dev (µs)
Pipeline	0	18.6385502536	49.6849071651	0	22.7744858062
Pipeline	1	2.57240820062	82.043814514	46.7818073614	43.7993433587
Pipeline	2	0	13.3229125945	120.8	44.7076375315

Figure 5-13: *Profiling statistical summary of swaptions. Mean and standard deviation values represented.*



MAXIMUM - MINIMUM SUMMARY:

	MINIMUM VALUE	MAXIMUM VALUE
Enqueue Mean Time	0	14.6
Compute Mean Time	16	326.75
Dequeue Mean Time	0	60.4
Enqueue Std_Deviation Time	2.57241	18.6386
Compute Std_Deviation Time	13.3229	82.0438
Dequeue Std_Deviation Time	46.7818	120.8
Throughput	0.000702686	0.025

Total Execution time: 1 milliseconds

Figure 5-14: *Profiling statistical summary of swaptions. Maximum/Minimum values represented.*

Next, a set of four different figures concerning the execution of *swaptions* application by using the tracing submodule will be shown. Each of them shows the program inner behaviour from different points of view, since different metrics have been used. All of them have been synchronized so that we can observe exactly the same parts at the same time.

The first one (see Figure 5-15) shows the usual trace. The light blue colour, as shown in the legend, is just for the graph visibility, but there, threads do not perform any work. The last four threads (horizontal bars) correspond with the farm stage, since we had specified it by the `-ns` parameter. From those four, almost all work is performed by the first thread as we can observe a larger coloured bar. The three remaining just perform a small piece of the work, which corresponds with that small vertical shape in each bar.

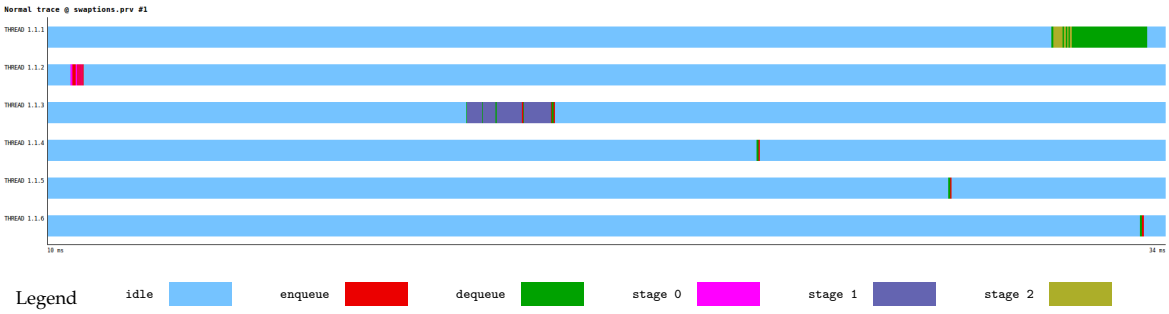


Figure 5-15: *Collected trace after swaptions execution with Paraver tool.*

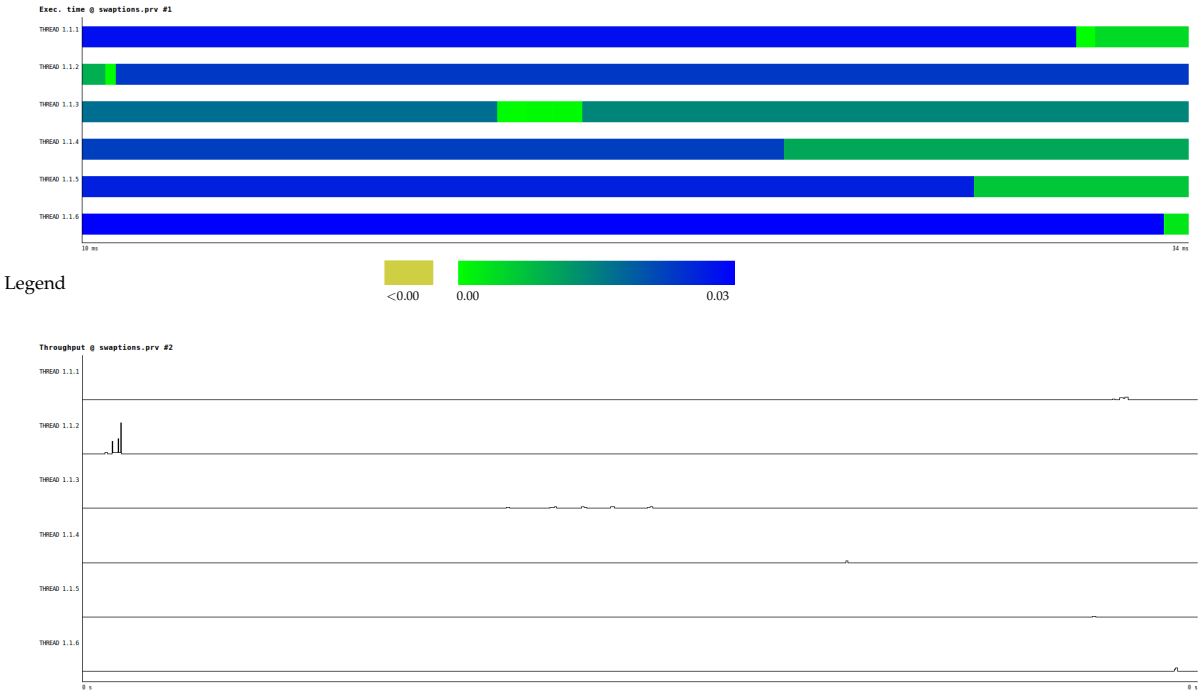


Figure 5-16: *Trace of swaptions with Paraver tool. Top: Execution time metric. Bottom: Throughput [Min. value: 52,109 - Max. value: 632,911.392].*

Figure 5-17: *Thread activity during the execution of swaptions application with Paraver tool.*

Along the two last figures (5-16, 5-17) we can observe the execution, the throughput and the active threads metrics with respect to that program, also synchronized, so we can observe how bars coincide

into the different graphs. On one side, we can distinguish the complete execution trace, followed by another graph that records the throughput at every single point during the execution. Besides, we can observe another type of graph that, by means of a simple line, it shows the whole activity, each thread presents. In that way, we can observe at the beginning that this line increases a little, then it remains constant, and finally, it decreases again. Then, we have a considerable piece of graph in where threads are not computing anything (line is completely horizontal), and this can be checked by looking at the throughout graph just above this one, since we can observe that there are no peaks in the horizontal bars of each thread.

5.5.2 Discard-words

We will now explain one example that also deals with Text Mining but, in this case, related to a streaming parallel pattern instead of a data one. By means of the *stream-filter* pattern, we will be able to discard those words in a given text that does not match with a proposed predicate. That is, given an input file of words, we could filter those words by some word length measure. Therefore, from all the listed words, we will just keep those that fulfill that length condition.

This application is very flexible, since it works with any input length, and it can be changed depending on the user needs. As the previous streaming examples, this pattern has also to be defined inside another one. The code associated to this pattern looks like the following:

Listing 5.6: *Example header of the discard-words pattern.*

```
1 grppi::pipeline(ex,
2    [&file]() -> optional<string> {
3      // read words from input file
4    },
5    grppi::keep([](string w) -> bool { return w.length() < 5;}), //stream filter
6      [ ](string w) {
7        // print those filtered words
8      }
9 );
```

The following table shows the needed parameters for executing the application:

Parameter	Description	Used Parameter
Input file name	The name of the input file	input.txt
Execution Mode	Available options are “seq”, “omp”, “thr”	thr

Table 5.25: *Required parameters to run the discard-words application.*

The command line used to execute the program looked as follows:

```
./discard_words input.txt thr
```

This parallel pattern has been tested with an input file containing 96 different size words, and those have been filtered by those containing less than 5 characters, as can be observed in line 5 from the previous piece of code. Besides, 4 threads have been launched, although this is not an explicit parameter, it is configured inside the user application.

In the following pair of Figures, 5-18 and 5-19, we can show the statistical report obtained when executing this application. We can observe that figures are quite small, this is because this application has less computational load. Again, we spent more time on the computation part.

One interesting aspect regarding the mean table is the appearance of zero values at some points in the stages. We can observe that the first zero value appearing in the first column is because the pipeline at that last stage does not need to enqueue items since it is the last stage. The same happens in the third column in the first stage: we get the zero value since we are still in the first stage.

```
                    ***** STREAMING PATTERNS STATISTIC SUMMARY *****
MEAN - THROUGHPUT SUMMARY:
|              Pattern |              Stage_id |    Enqueue Mean Time (µs)|    Compute Mean Time (µs)|    Dequeue Mean Time (µs)|     Throughput (items/µs)|
|              Pipeline |                   0|                  0.804124|                   2.41237|                         0|                  0.310897|
|              Pipeline |                   1|                  0.115702|                 0.0208333|                  0.587629|                  0.690451|
|              Pipeline |                   2|                         0|                   3.30435|                  0.166667|                    0.2881|

STANDARD_DEVIATION SUMMARY:
|              Pattern |              Stage_id | Enqueue Std_Dev Time (µs)| Compute Std_Dev Time (µs)| Dequeue Std_Dev Time (µs)|       Total Std_Dev (µs)|
|              Pipeline |                   0|                   1.04351|                    19.585|                         0|                    6.87617|
|              Pipeline |                   1|                  0.344738|                  0.416508|                   6.49908|                    2.42011|
|              Pipeline |                   2|                         0|                   5.90825|                   0.62361|                    2.17729|
```

Figure 5-18: *Profiling statistical summary of discard-words. Mean and standard deviation values represented.*

```
MAXIMUM - MINIMUM SUMMARY:
=============================================================================================
                                |              MINIMUM VALUE  |              MAXIMUM VALUE |
---------------------------------------------------------------------------------------------
|            Enqueue Mean Time |                          0|                   0.804124|
|            Compute Mean Time |                  0.0208333|                    3.30435|
|            Dequeue Mean Time |                          0|                   0.587629|
|    Enqueue Std_Deviation Time |                   0.344738|                    1.04351|
|    Compute Std_Deviation Time |                   0.416508|                     19.585|
|    Dequeue Std_Deviation Time |                    0.62361|                    6.49908|
|                  Throughput |                     0.2881|                   0.690451|

Total Execution time: 4 milliseconds
```

Figure 5-19: *Profiling statistical summary of discard-words. Maximum/Minimum values represented.*

Focusing now on the tracing submodule, we will show below two different perspectives from the same image, the second one will be an enlarged view from the first one so that we can better distinguish the colours. The activity of the four threads can be identified in the first picture. However, in the second picture, we have just focused on the first half of the execution so we can only observe thread 2 and 3 activities.

By looking at the enlarged figure, we can distinguish colour red and pink in a higher amount: this is because we are at the beginning of the execution (stage 0 as the legend states) and that thread is in charge of enqueuing the items further to be processed. Afterwards, we can observe dark colours like green and even purple (stage 1): this is because we have changed to the second stage. Notice that thread 3 also proceeds with the enqueuing of items, as we can also identify red colour there.

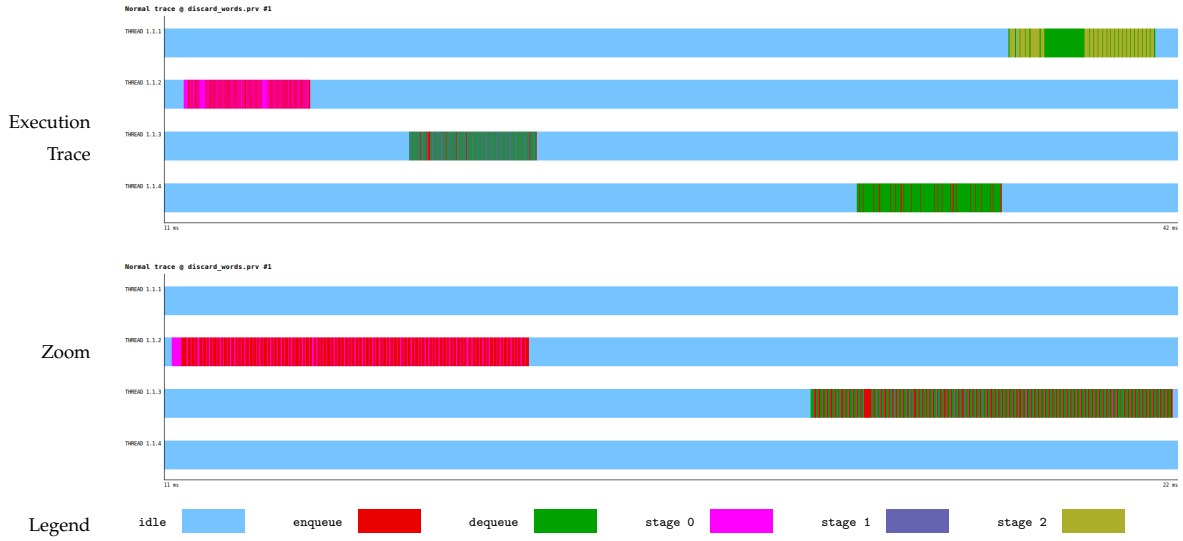


Figure 5-20: *Top. Generic trace representation after discard-words execution with Paraver tool. Bottom: Amplified results from previous image.*

With respect to this three graphs, we can observe quite active in all senses. As already observed in the two previous graphs, each thread contributes a little so that this filter of words can be properly applied.

In the first figure, the execution trace is shown. We can observe that it presents a similar shape with respect to the top part in Figure 5-20. Besides, regarding Figure 5-21, we can conclude that the sector where more activity takes place is in the first part of the execution (stage 0) when we enqueued items, since we can distinguish higher and very concentrated peaks in the graph.

In the last graph, the one that shows the thread activity, we can observe, in the same way, how the part where threads are more active also coincides with the first part of the execution.

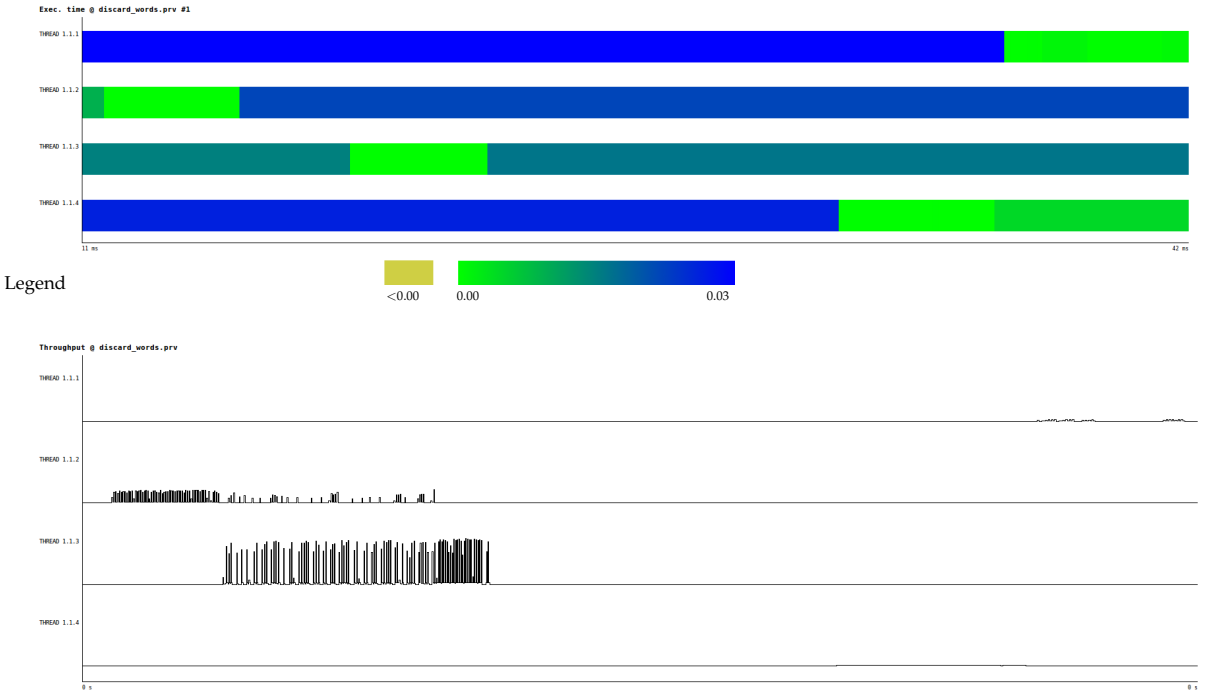


Figure 5-21: *Trace of discard-words with Paraver tool. Top: Execution time metric. Bottom: Throughput [Min. value: 63,486368 - Max. value: 1223990,2080].*

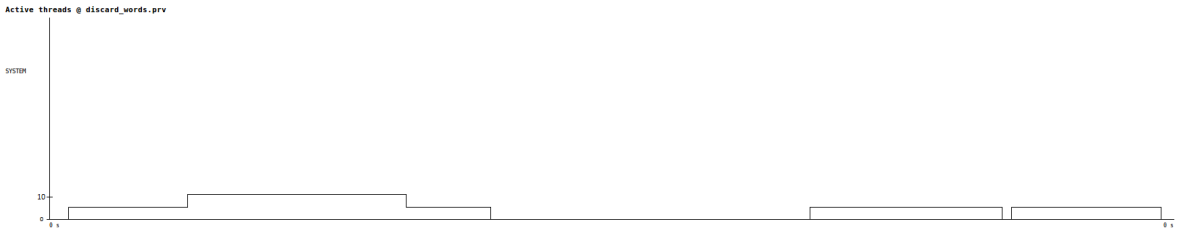


Figure 5-22: *Threads activity representation after discard-words execution with Paraver tool.*

5.5.3 Chunk-sum

This example makes use of the *stream-reduction* parallel pattern, that is, we will apply the already studied *reduce* pattern but this time over a data stream.

In this concrete exercise, we will work with vectors. Therefore, the aim of this streaming application is, given a sequence of consecutive natural numbers, apply the sum operation over the windows (each one composed of n numbers) starting every k items in that sequence. As output, we will obtain another sequence where each element is the sum of the elements of the corresponding window from the input sequence of natural numbers. An example will be given afterwards.

We will now show the corresponding basic piece of code to make use of this pattern. Again, a composable structure of a pipeline with this stream-reduce pattern inside:

Listing 5.7: *Example header of the chunk-sum pattern.*

```
1 grppi::pipeline(ex, generator,
2   grppi::reduce(window_size, offset, 0, // stream-reduce pattern
3     [ ](int x, int y) { return x+y; }
4   ),
5   [ ](int x) { cout << x << endl; }
6 );
```

In the following table, we can identify the parameters we need to execute the application:

Parameter	Description	Used Parameter
Vector size	Number of elements initially generated in the vector	900
Window size	Number referring to the aggregation window size	5
Offset	Distance between the start of one window to the following one	10
Execution Mode	Available options are “seq”, “omp”, “thr”	thr

Table 5.26: *Required parameters to run the chunk-sum application.*

We will show a simple example before showing the real command line used to run this program, since higher figures have been used:

If we had a vector size of 10 elements like:

0 1 2 3 4 5 6 7 8 9

a window size of 2, and an offset also of 2, then we will have:

$$0+1, \quad 2+3, \quad 4+5, \quad 6+7, \quad 8+9$$

where the result of 0+1 is the first window, and the second one (2+3) begins two positions (offset) to the right starting from the 0 position in the vector.

The final command line used to execute the program looked as follows:

```
./chunk_sum 900 5 10 thr
```

that is, we have used a 900 element vector with a window size of 5 elements and the distance between two windows is now 10. Besides, this application has been launched with 3 threads.

This is the statistical summary obtained with the execution of this application:

```
                    ***** STREAMING PATTERNS STATISTIC SUMMARY *****
 MEAN - THROUGHPUT SUMMARY:
|               Pattern |               Stage_id |    Enqueue Mean Time (µs)|    Compute Mean Time (µs)|    Dequeue Mean Time (µs)|     Throughput (items/µs)|
|               Pipeline |                     0|                  0.129856|                         0|                         0|                   7.70085|
|               Pipeline |                     1|                 0.0989011|                 0.0222222|                  0.134295|                   3.91514|
|               Pipeline |                     2|                         0|                   7.56667|                  0.010989|                  0.131967|

 STANDARD_DEVIATION SUMMARY:
|               Pattern |               Stage_id | Enqueue Std_Dev Time (µs)| Compute Std_Dev Time (µs)| Dequeue Std_Dev Time (µs)|       Total Std_Dev (µs)|
|               Pipeline |                     0|                  1.63568|                         0|                         0|                  0.545226|
|               Pipeline |                     1|                 0.298529|                   1.27347|                   2.55781|                    1.3766|
|               Pipeline |                     2|                         0|                   3.65164|                  0.104251|                   1.25196|
```

Figure 5-23: *Profiling statistical summary of chunk-sum. Mean and standard deviation values represented.*

```
MAXIMUM - MINIMUM SUMMARY:
                                |               MINIMUM VALUE  |               MAXIMUM VALUE |
|            Enqueue Mean Time |                           0|                    0.129856|
|            Compute Mean Time |                           0|                     7.56667|
|            Dequeue Mean Time |                           0|                    0.134295|
|   Enqueue Std_Deviation Time |                    0.298529|                     1.63568|
|   Compute Std_Deviation Time |                           0|                     3.65164|
|   Dequeue Std_Deviation Time |                    0.104251|                     2.55781|
|                  Throughput |                    0.131967|                     7.70085|

Total Execution time: 20 milliseconds
```

Figure 5-24: *Profiling statistical summary of chunk-sum. Maximum/Minimum values represented.*

Total execution time is 20 milliseconds, which is something normal within the set of proposed examples. If we look at `enqueue_mean_time` column, we can observe how the mean decreases from stage 1 to stage 3 until reaching the zero value. However, just the opposite happens in the column on the right,

values start from zero in the first stage and it becomes 7.56 in the last stage.

Again, this happens because, at the last stage of the pipeline we do not enqueue more items, it is the last stage. The zero value in the compute stage could be because at that point, items only need to be enqueued, it makes no sense computing items that still have not been enqueued in the queue. Maximum and minimum values are also presented in Table 5-24.

Let us look at what tracing submodule has provided once executed this application. As with the previous examples, we have increased the zoom of the first figure (see Figure 5-25). In this concrete example, we can observe that the predominant colour is the red one, as well as the dark green ones, which means that threads have been enqueuing and dequeuing lots of items. Consequently, the three below graphs (see Figures 5-26, 5-27) show more activity (throughput and active threads tables) in those mentioned parts, concretely, along all the second bar associated to thread 2, and from the second half referring to thread 3.

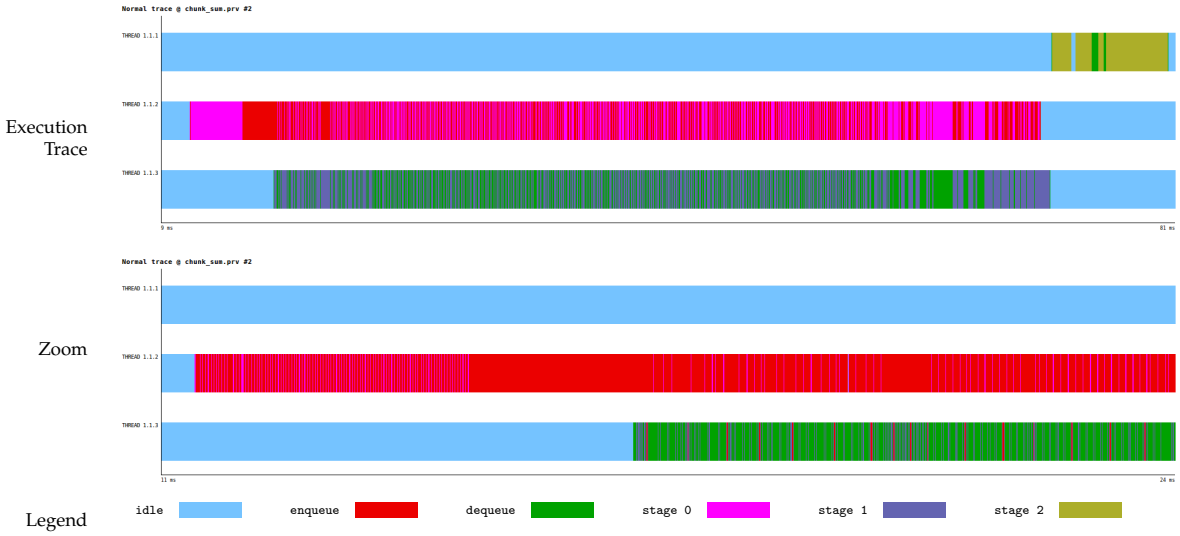


Figure 5-25: *Top. Generic trace representation after chunk-sum execution with Paraver tool. Bottom: Amplified results from previous image.*

Figure 5-26: *Active threads graphical representation after chunk-sum execution.*

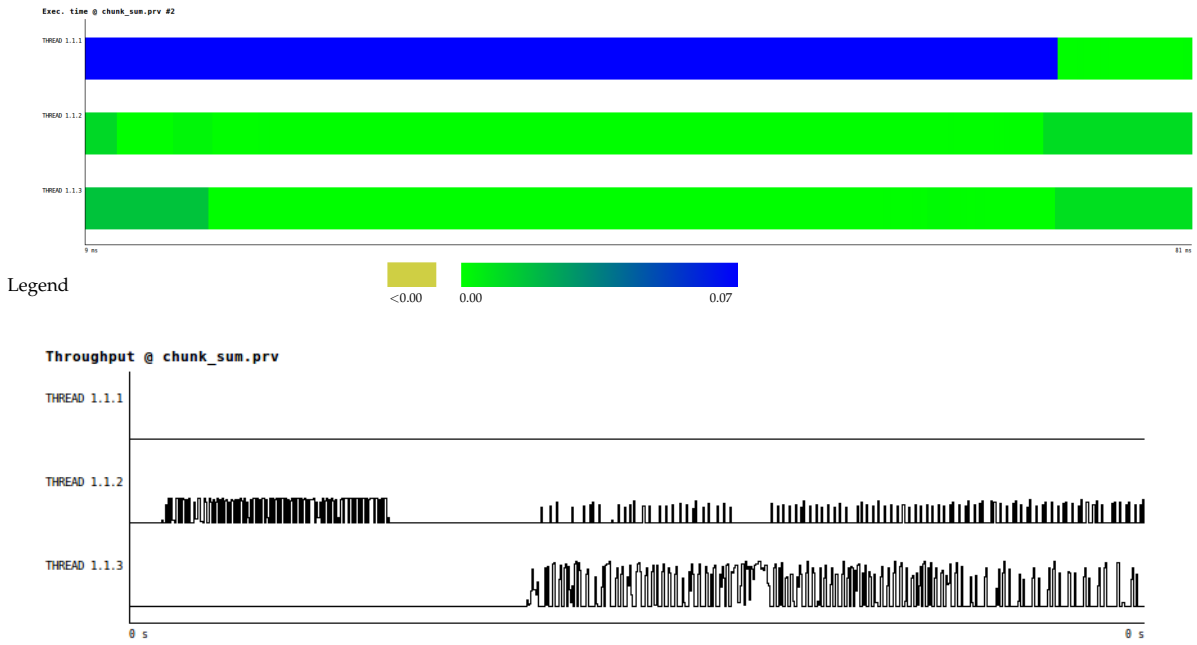


Figure 5-27: *Trace of chunk-sum with Paraver tool. Top: Execution time metric. Bottom: Throughput [Min. value: 102,889307 - Max. value: 1239157,3729].*

5.5.4 Print-power

Thanks to this streaming parallel pattern, we can perform *loops* inside the stream of input data. This pattern is based on a predicate. If at each iteration from the *loop*, that predicate is satisfied, result will be sent to the output stream. In any other case, result will be discarded. As with the three previous examples, this pattern can not be used as an independent pattern, so it will be developed as a composable function where a pipeline structure surrounds it.

In this specific example, our main objective is the generation of an output sequence. That sequence will be composed of the numbers that have previously met the predicate. We will establish a parameter that refers to the initial input sequence. Starting from number 1 in the sequence until the end of it, we will compute the double of those numbers and check if the predicate is fulfilled or not. Concretely, the predicate just verifies that the number after applying that product is larger than 1024. Only in that case, we will print that number.

Next, we show the associated code for this specific pattern. A composable structure of a pipeline with this stream-iteration pattern inside is presented.

Listing 5.8: *Example header of the print-power pattern.*

```
grppi::pipeline(ex, generator,
   grppi::repeat_until( // stream-iteration pattern
     [ ](int x) { return 2*x; },
     [ ](int x) { return x>1024; }
   ),
   [ ](int x) { cout << x << endl; }
);
```

In the following table, we can identify the parameters we need to execute the application:

Parameter	Description	Used Parameter
Sequence size	Number of sequence elements initially generated	90
Execution Mode	Available options are “seq”, “omp”, “thr”	thr

Table 5.27: *Required parameters to run the print-power application.*

The command line used to execute the program looked as follows:

```
./print_power 90 thr
```

where 90 value corresponds with the vector size to which this pattern is applied. The number of threads used to execute this application is 3.

We will now analyze what the profiling submodule has generated. Total execution time was 9 milliseconds, and in general, values have not been so high. We can distinguish again those zero values in the corresponding stages due to the aforementioned reasons, as well as minimum and maximum values.

```
                    ***** STREAMING PATTERNS STATISTIC SUMMARY *****
MEAN - THROUGHPUT SUMMARY:
|              Pattern |              Stage_id |    Enqueue Mean Time (µs)|    Compute Mean Time (µs)|    Dequeue Mean Time (µs)|     Throughput (items/µs)|
|              Pipeline |                   0|                  0.934066|                  0.010989|                         0|                   1.05814|
|              Pipeline |                   1|                 0.0983607|                         0|                  0.057377|                   6.42105|
|              Pipeline |                   2|                         0|                   11.8333|                  0.021978|                 0.0843504|

STANDARD_DEVIATION SUMMARY:
|              Pattern |              Stage_id | Enqueue Std_Dev Time (µs)| Compute Std_Dev Time (µs)| Dequeue Std_Dev Time (µs)|       Total Std_Dev (µs)|
|              Pipeline |                   0|                   1.38732|                  0.104835|                         0|                  0.497384|
|              Pipeline |                   1|                  0.304605|                         0|                   2.72897|                   1.01119|
|              Pipeline |                   2|                         0|                   6.33202|                  0.208502|                   2.18017|
```

Figure 5-28: *Profiling statistical summary of print-power. Mean and standard deviation values represented.*

```
MAXIMUM - MINIMUM SUMMARY:
=============================================================================================
                            |              MINIMUM VALUE  |              MAXIMUM VALUE |
---------------------------------------------------------------------------------------------
|           Enqueue Mean Time |                        0|                       0.934066|
|           Compute Mean Time |                        0|                       11.8333|
|           Dequeue Mean Time |                        0|                       0.057377|
|   Enqueue Std_Deviation Time |                 0.304605|                       1.38732|
|   Compute Std_Deviation Time |                        0|                       6.33202|
|   Dequeue Std_Deviation Time |                 0.208502|                       2.72897|
|                  Throughput |                0.0843504|                       6.42105|

Total Execution time: 9 milliseconds
```

Figure 5-29: *Profiling statistical summary of print-power. Maximum/Minimum values represented.*

Regarding the tracing submodule, we can observe in Figure 5-30 the inner flow of the program by distinguishing the three stages painted in pink (thread 2), purple (thread 3) and pistachio green (thread 1). We can also observe how the whole work is distributed into this threads, in such a way that when one of them finishes its work, the following one starts itself.

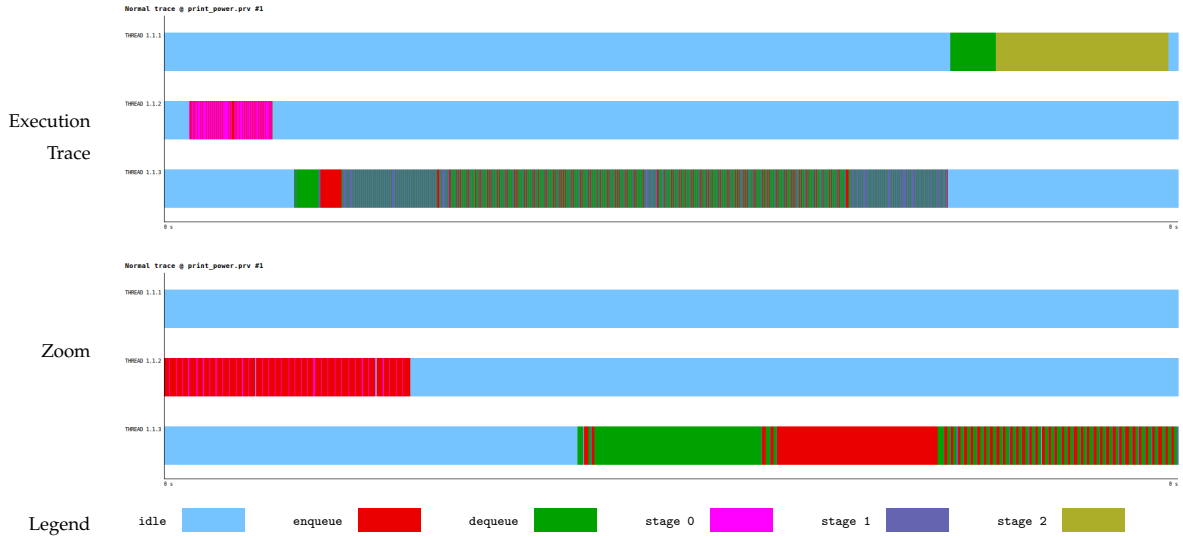


Figure 5-30: *Top. Generic trace representation after print-power execution with Paraver tool. Bottom: Amplified results from previous image.*

Finally, regarding the three figures below, we can check how the most computational activity takes place in the third thread (see peaks in the throughput graph). The first part regarding the items enqueue process in thread 2 also shows a higher activity there, however, we can identify how the work done by thread 1 at almost the end of the execution (stage 2), is not so high since peaks in the throughput graph hardly increase.

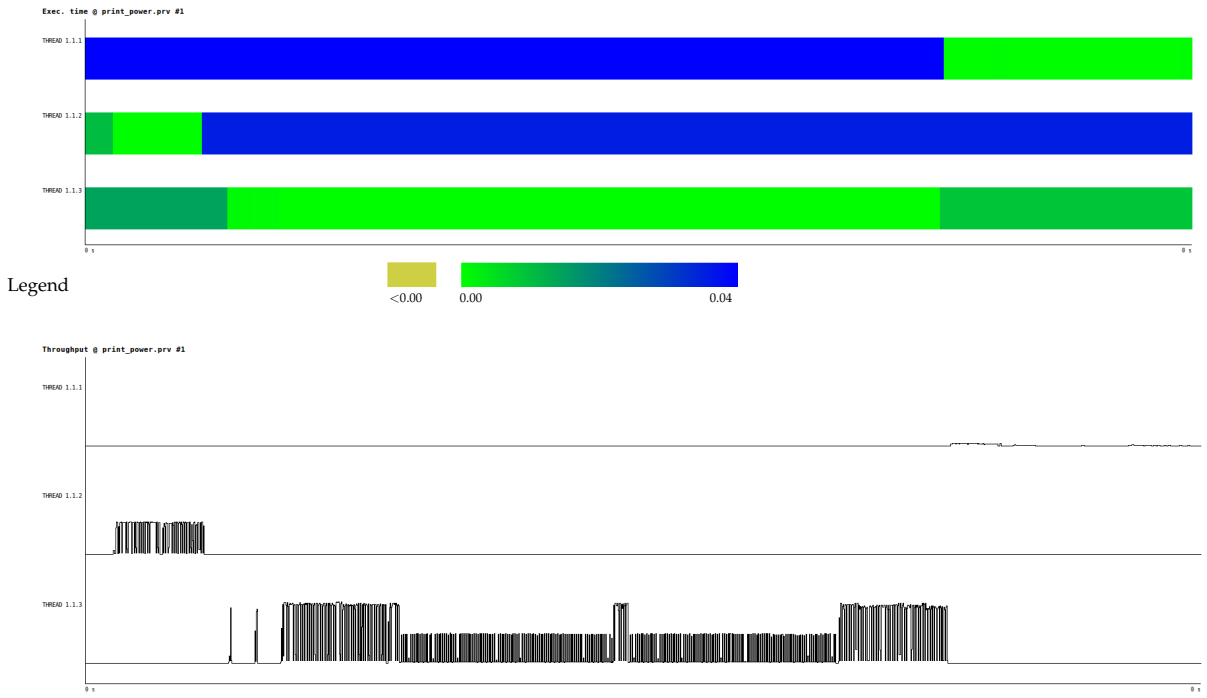


Figure 5-31: *Trace of print-power with Paraver tool. Top: Execution time metric. Bottom: Throughput [Min. value: 28,203022 - Max. value: 1282051,2820].*

Figure 5-32: *Active threads graphical representation after print-power execution.*

5.6 Mandelbrot set

This section presents another example of streaming application included in the evaluation samples set. However, as already stated in this document, we will also explain how we proceeded in (Section 5.6.1) with the application of an additional software to this specific program.

Mandelbrot Set is the given name to a special type of fractal. A fractal is no more than the repetition of a geometric base figure until reaching the infinite, in other words, fractals can be thought of as being never-ending patterns.

Fractals are recognized because of two main characteristics: they present infinite detail although we get very close to one concrete point in the figure, and also due to its self-similarity. The study of fractals dates back to the seventies thanks to the suggestion of the French mathematician Benoit Mandelbrot. The idea of knowing and discover new properties of the natural shape produced scientists and mathematicians to explore and research different theories about this topic.

Among all different fractals, we can underline some classic ones such as Peano, Cantor, Koch or the Julia Set. This last one is based on the following formula:

$$f_c(z) = z_n^2 + C \tag{5.2}$$

From the Julia Set emerged the Mandelbrot Set, concretely, when in the previous formula, the set of results of z (bounded in absolute value) diverge.

Mandelbrot images may be created by sampling the complex numbers and determining, for each sample point C, whether the result of iterating the above function goes to infinity. Treating the real and imaginary parts of C as image coordinates (x + y_i) on the complex plane, pixels may then be coloured according to how rapidly the sequence z_n^2 diverge. The Mandelbrot Set has become popular outside mathematics both for its aesthetic appeal and as an example of a complex structure arising from the application of simple rules. Specially, it became prominent in the mid-1980s as a computer graphics demo, when personal computers became powerful enough to plot and display the set in high resolution.

Given the inherent embarrassingly parallel nature of this problem, where each pixel in the image can be computed independently, this application intends to make use of the *farm* streaming pattern in order to accelerate the computation of the basic Mandelbrot image. Concretely, 3000 frames have been set in order to complete the program execution, but this value could be changed inside the main program by any other one.

This exercise depends on the value of a variable that allows us to observe more or less detail in the image, that is, it acts like a zoom variable. Therefore, finer details of the Mandelbrot Set can be observed. Thus, we can generate an animation of the Mandelbrot images at increasing zoom values.

This streaming application consists of the following stages:

1. **Generator:** returns monotonically linear increasing zoom values, which are passed to the Mandelbrot stage.

2. **Mandelbrot:** takes the zoom received by the generator stage and computes the Mandelbrot image corresponding to such zoom value and the coordinates of the point of interest.

3. **Printer:** prints by the standard output the image computed by the Mandelbrot stage.

Next, we show the associated code for this specific pattern. A composable structure of a pipeline with a farm pattern inside is presented:

Listing 5.9: *Example header of the print-power pattern.*

```
grppi::pipeline(ex,
   [&]() -> std::experimental::optional<double> {
      // computes zoom values and pass them to the farm stage
   }
   grppi::farm(4, [&](auto zoom){
      // computes Mandelbrot image of that received pixel
   }),
   [&](auto image){
      // prints the frame (we pass over here 3000 times)
   }
);
```

The following parameters should be stated to run this application:

Parameter	Description	Used Parameter
Execution Mode	Available options are “seq”, “omp”, “thr”	thr

Table 5.28: *Required parameters to run the Mandelbrot Set application.*

The command line used to execute the program looked as follows:

```
./mandelbrot thr
```

Next, we show below a pair of figures regarding the execution of this application. On the left part [63], we can observe the real resulted image of this specific fractal, while on the right, we show the simulation of this fractal thanks to the farm parallel pattern from GrPPI framework. As it can be observed, this simulation has been performed by means of ASCII code. That picture represents only one frame out of the 3000 already mentioned, since its shape during execution is updated per frame.

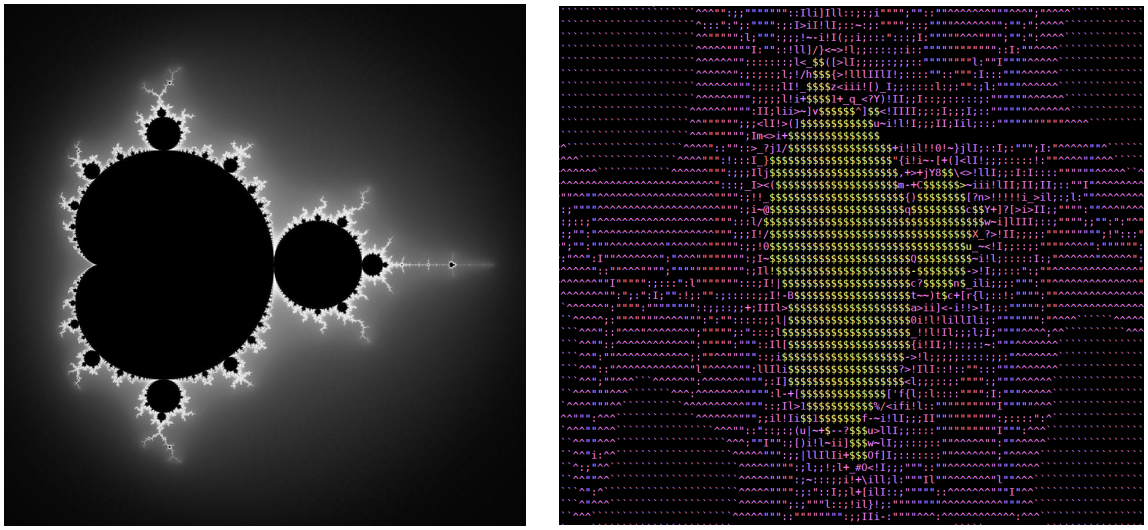


Figure 5-33: *Real Mandelbrot Set and its simulation.*

******* STREAMING PATTERNS STATISTIC SUMMARY *******

MEAN - THROUGHPUT SUMMARY:

Pattern	Stage_id	Enqueue Mean Time (µs)	Compute Mean Time (µs)	Dequeue Mean Time (µs)	Throughput (items/µs)
Pipeline	0	19122.8	33068.7	0	1.91602e-05
Pipeline	1	3.2985	194158	37821.4	1.07766e-06
Pipeline	2	0	784.632	57257.8	1.72288e-05

STANDARD_DEVIATION SUMMARY:

Pattern	Stage_id	Enqueue Std_Dev Time (µs)	Compute Std_Dev Time (µs)	Dequeue Std_Dev Time (µs)	Total Std_Dev (µs)
Pipeline	0	57017.5	603.86	0	19207.1
Pipeline	1	1.87489	203080	47733.8	83605
Pipeline	2	0	294.123	73636.9	24643.7

Figure 5-34: *Profiling statistical summary of Mandelbrot set. Mean and standard deviation values represented.*

MAXIMUM - MINIMUM SUMMARY:

	MINIMUM VALUE	MAXIMUM VALUE
Enqueue Mean Time	0	19122.8
Compute Mean Time	784.632	194158
Dequeue Mean Time	0	57257.8
Enqueue Std_Deviation Time	1.87489	57017.5
Compute Std_Deviation Time	294.123	203080
Dequeue Std_Deviation Time	47733.8	73636.9
Throughput	1.07766e-06	1.91602e-05

Total Execution time: 174230 milliseconds

Figure 5-35: *Profiling statistical summary of Mandelbrot set. Maximum/Minimum values represented.*

From this point on, we will show the results obtained by means of the profiling submodule. In Table 5-34, we can observe that in general, values are much higher than in the rest of applications. This is because as we have argued, this application has a higher computational load, and this is reflected in

those values. However, the same program behaviour appears here with the aforementioned zero values, in both enqueue and dequeue phases. Maximum and minimum values are also presented in Table 5-35.

Now, we will talk about the tracing results. At first sight, we can observe how blue colour does not appear in both figures in Figure 5-36. This is because as already said, this application requires much computational work and threads are always performing some work. Concretely, we can identify how the second thread is dedicated only to compute items at stage 0, while the four last threads (corresponding to the farm stage), each one performs a piece of work. Besides, the first thread is the one in charge of computing stage 2 in the stream, that is why that pistachio green appears every now and then.

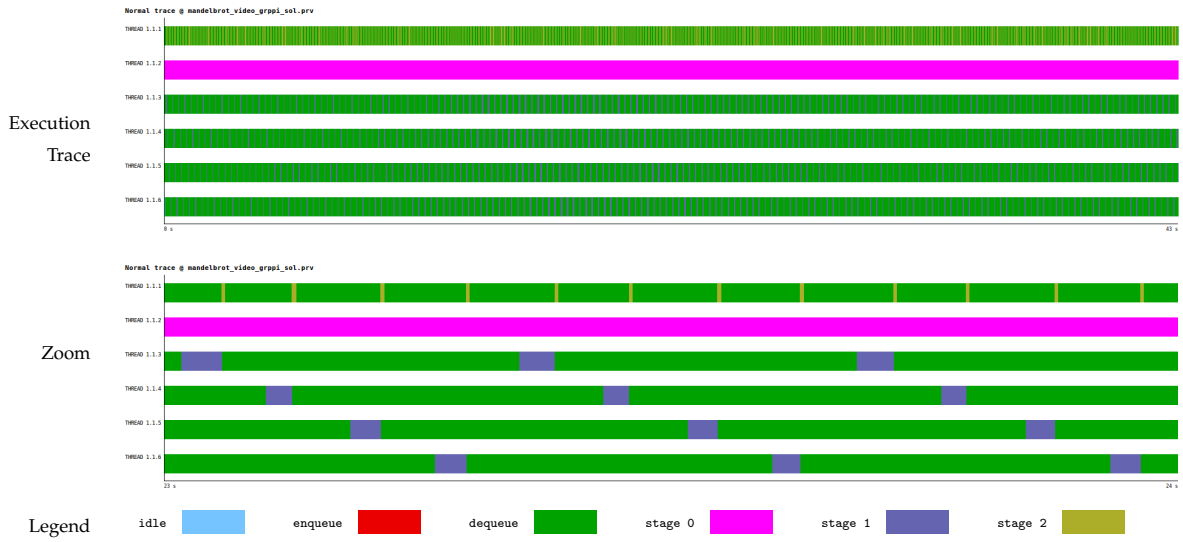


Figure 5-36: *Top. Generic trace representation after Mandelbrot Set execution with Paraver tool. Bottom: Amplified results from previous image.*

With respect to the execution trace as shown in Figure 5-37, we can identify that the burst time in the last 4 threads (corresponding with the cardinality of the farm stage) takes the highest values (see blue colour in the legend).

We can also observe that none of the throughput lines in the throughput graph takes a horizontal line, but almost they present some minimum peaks (except the first thread that is a bit special). Moreover, it can be checked in the active threads graph that the line increases in a higher proportion with respect to the rest of applications already described.

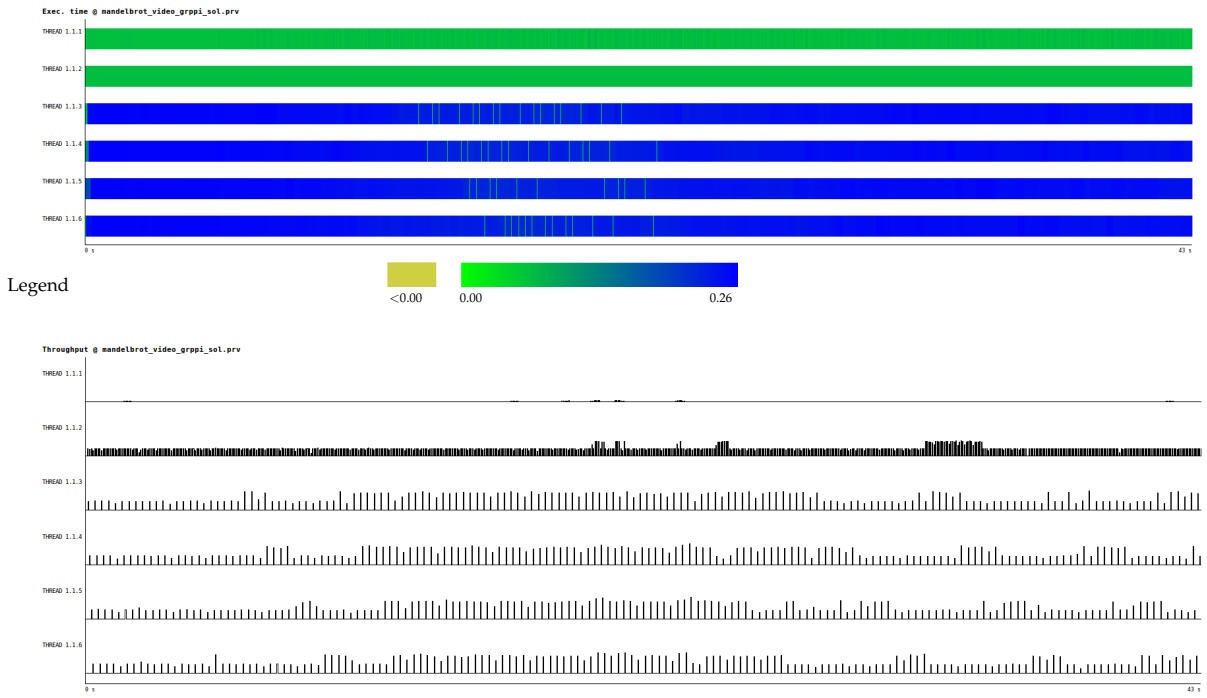


Figure 5-37: *Trace of Mandelbrot Set with Paraver tool. Top: Execution time metric. Bottom: Throughput [Min. value: 3,839266 - Max. value: 52446,6355].*

Figure 5-38: *Active threads graphical representation after Mandelbrot execution*

5.6.1 Feedgnuplot visualization tool

Afterwards, we will feed `feedgnuplot` with the ready file by means of the following command:

```
tail -f default.out | feedgnuplot
```

where `default.out` is the file we were just talking about.

We fed `feedgnuplot` together with all the needed parameters (legend, x-axis unit, y-axis unit, etc.) and obtained at the end what Figure 5-39 shows. Important to say that the aforementioned auxiliary thread generates the throughput metric and plots results into the graph in real time, together with an updating of it each 500 milliseconds as already said.

As can be observed in the legend that `feedgnuplot` generates, three different stages correspond to one different coloured line. Each stage, therefore, corresponds with each row in Table 5-34. At first sight, we can only distinguish in the graph two single colours, blue and red. Nevertheless, if we zoom in the image, we can observe that also green colour appears, it is always hidden by these two colours.

Although we can identify repeated peaks along the execution, we can say throughput is constant along the first 80500 milliseconds. Afterwards, the red line continues being constant while the blue one drops sharply at a 0.015 throughput level. Once the execution is almost finished, we can observe how now is the red line the one that drops with that slope.

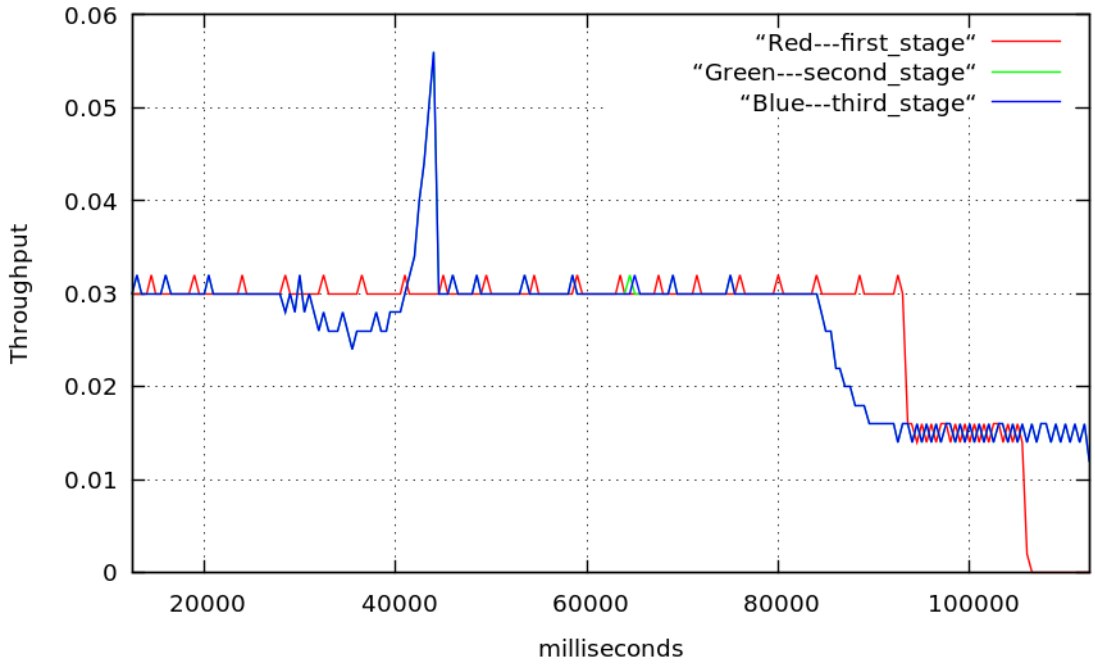


Figure 5-39: *Mandelbrot set execution with feedgnuplot tool.*

`Feedgnuplot` has been used with this exercise only because it can be considered the more complex one among all the presented ones. Besides, the result of that program execution (sequence of American Standard Code for Information Interchange (ASCII) symbols that make our eyes feel like watching a video), fitted very well with this plot-tool. With this, we can say this plot-performance tool could perfectly be used with the rest of proposed exercises using streaming parallel patterns.

Chapter 6

Legal Framework

This chapter summarizes the set of applicable regulations in the developed project. We will describe and discuss the different legal aspects in Section 6.1 in such a way that we can then conclude this project complies with the current legal framework. Apart from that, we will mention the set of licences applicable to this project in Section 6.2.

6.1 Applicable laws

This section brief analyzes possible regulations that may affect this project. In the first place, we would like to analyze the General Data Protection Regulation (GDPR) law. This regulation has recently become applicable and compulsory over all individuals belonging to the European Union, concretely since 25^{th} of May 2018.

Specifically, this law stated in 2016 (when becoming effective), that citizens in the European Union should be subject to data protection and privacy terms.

From the aforementioned date on, this new law will offer a set of different tools so that users perfectly know which personal data we are providing third parties, where this data is being transferred and for which purpose, in other words, this law gives more duties to users over their personal data, and agencies will acquire more authority to protect the data of each European country. Every enterprise will be subject to that new law whatever the country origin is, and large fines must be paid if they do not obey the law.

In our case, we are not directly dealing with sensible personal data at any moment, we just offer the user to make use of our module, but it does not need any further personal information to work. In that way, we can conclude this law is not applicable to this project. Therefore, users are responsible for their

data manipulation and we do not take responsibility for any unexpected incident.

Another law to be considered is the LOPD (*Ley Orgánica de Protección de Datos*). This is a fundamental duty that every Spaniard citizen should have. This law provides users a total control of his personal information in such a way that this control tries to avoid any possible treatment of our data that could affect our privacy. In this work, any application used in the evaluation process treats with sensitive information, so users will not be affected by the law.

6.2 Work environment licenses

This section comprises the set of licenses through which this project is submitted to. That licenses have to do with both with the developed technology, and those additional tools required for the project to work, as we will explain later on.

6.2.1 GrPPI license

In this section we will refer to possible licenses that affect the framework where this project has been based on, concretely, we will talk about GrPPI framework. It is redistributed under the Apache License[1], Version 2.0. This current version was approved in 2004. Among its main goals, we can stand out the reduction of the number of frequently asked questions and the fact to allow the license to be included as a reference and not having to make a list in every file. Besides, that license is compatible with any other open source license. Apache License 2.0 allows:

- Definitions
- Grant of Copyright License
- Grant of Patent License
- Redistribution
- Submission of Contributions
- Trademarks
- Disclaimer or Warranty
- Limitation of Liability
- Accepting Warranty or Additional Liability

[1] `http://www.apache.org/licenses/LICENSE-2.0`

6.2.2 Extrae and Paraver license

We will now talk about Extrae and Paraver license. These are the two software employed in this project and they both work under the same license, since Paraver is the associated performance visualization tool of Extrae software.

Hence, both are submitted to the Open Source (GNU LGPL) license. GNU Lesser General Public License (LGPL), was published in 1991 and its latest version was updated in 2007 (Version 3.0). It allows enterprises and developers to freely use and modify the software that has been released under the LGPL license.

LGPL allows:

- Additional Definitions
- Exception to Section 3 of the GNU GPL
- Conveying Modified Versions
- Object Code Incorporating Material from Library Header Files
- Combined Works
- Combined Libraries
- Revised Versions of the GNU Lesser General Public License

More information can be found at LGPL[2].

[2] `https://opensource.org/licenses/lgpl-3.0.html`

Chapter 7

Project Planning

This chapter presents a detailed planning of the project. Section 7.1 explains which type of methodology has been adopted, since this decision will afterwards affect the dependencies between the project tasks. Next, Section 7.2 shows a complete description of the different project phases and the distribution of the subsequent tasks associated to them.

7.1 Developed methodology

One important thing regarding a project development, has to do with the methodology carried out in order to get the project off the ground. Different phases need to be defined beforehand so that we can follow some guidelines and have in mind some kind of track that helps us to know if we are doing things well and on time, or, in any other case, to know if some aspects should be changed to improve its development.

There exist so many types of project methodologies, but it is important to determine which one is the most convenient one for us. We will need to choose the one that it is supposed to give us the best results in a future, since we have some predefined contract with the user and the product must be delivered on time. The objective of any type of methodology is showing the way the project has been developed: the distribution of tasks, the assigned period for each one, the dependencies between them, etc. We can distinguish between the following types of methodologies:

1. *Prototype Model:* this one is based on the construction of some prototype in a very short period of time. The main objectives of this model is to guarantee that elicitated requirements are being fulfilled and to minimize that uncertainty provoked in the user during the project development.

2. *Waterfall Model:* this model requires the division of the project phases into dependent tasks. That is, the start of one phase can only be possible unless the completion of the previous one.

3. *Spiral Model:* this model is quite similar to the software life cycle of a project. By applying it, the project activities are distributed into a spiral shape in such a way that tasks are fulfilled and improved by means of different iterations.

In this concrete project, we have not been able to develop a quick prototype since it demanded so many functionalities in a very short period of time. The same happened with the waterfall model, we had some clear ideas about the different parts of the project, but as the design decisions were defined not so strict from the beginning, that would imply making some modifications at some points in the project. Actually, we have performed several optimizations and changes along the implementation once the first version was completed, so this model was not the chosen one either. Therefore, the only model that satisfied our needs was the spiral one.

This model divides the project tasks into several parts, in such a way that each part needs to go through a set of different phases. Our research project included two main differentiated parts, the profiling phase and the tracing one. Since the same activities needed to be applied for each phase, we defined two different iterations in the whole planning diagram.

7.2 Planning

This section shows in a diagram the detailed description of the project in terms of performed activities along time. But before that, we will point some general considerations.

The project has been developed and implemented at University Carlos III of Madrid. It began on 23 January 2018, and ended on 17 June 2018, making a total of six months of work. Within those months, it is important to mention that only partial-time job days have been considered (without including weekends), which makes a total of 20 hours per week. However, from the end of March until the end of the project, more hours than the estimated ones have been dedicated to it.

As stated before, the beginning of the project took place the 23^{th} of January 2018. However, since September 2017, several months were dedicated to introduce ourselves into the new pattern-based parallel programming framework, GrPPI, since we will work within that environment, but that part has not been taken into account in the final chart.

We can say that the real date where the project implementation started coincides with the starting date in the first iteration in the Gantt diagram, as shown in Figure 7-1. The first iteration lasted 9 days (tracing

submodule), and when it was already finished, we started the second iteration (profiling submodule). The latest lasted more days that the profiling submodule, since we did not have any additional tool or library ready to use, but everything was built from scratch. Apart from that pair of iterations, we can distinguish in the diagram the evaluation and documentation processes. Note that evaluation took place before finishing the profiling submodule, since we already had completed the tracing one and we decided to start testing some part of the system by means of Paraver tool.

In the following page, we will show the Gantt diagram (see Figure 7-1). It shows all the tasks performed during the development of this work by means of coloured bars that mark the start and end of each one. Besides, in Table 7.1, we can observe in a more detailed way, all information related to that task, each one associated to one identification number.

Tasks information				
Task Id	**Task Name**	**Duration**	**Start Date**	**End Date**
1	Problem identification	2 days	23/1/18	24/1/18
2	Objectives identification	2 days	25/1/18	26/1/18
3	Documentation (profling and tracing tools)	10 days	29/1/18	9/2/18
4	Simple testing with Extrae library	6 days	12/2/18	19/2/18
5	Environment set up	9 days	20/2/18	2/3/18
6	Instrumentation process (common interface)	11 days	5/3/18	19/3/18
7	**Iteration 1: Tracing submodule (Extrae)**	**9 days**	**20/3/18**	**30/3/18**
8	Implementation	4 days	20/3/18	23/3/18
9	Validation	3 days	26/3/18	28/3/18
10	Maintenance	2 days	29/3/18	30/3/18
11	**Iteration 2: Profiling submodule (Native)**	**40 days**	**2/4/18**	**25/5/18**
12	Implementation	30 days	2/4/18	11/5/18
13	Validation	4 days	14/5/18	17/5/18
14	Maintenance	6 days	18/5/18	25/5/18
15	Evaluation	10 days	7/5/18	18/5/18
16	Project Documentation	31 days	7/5/18	18/6/18

Table 7.1: *Information related to each performed task.*

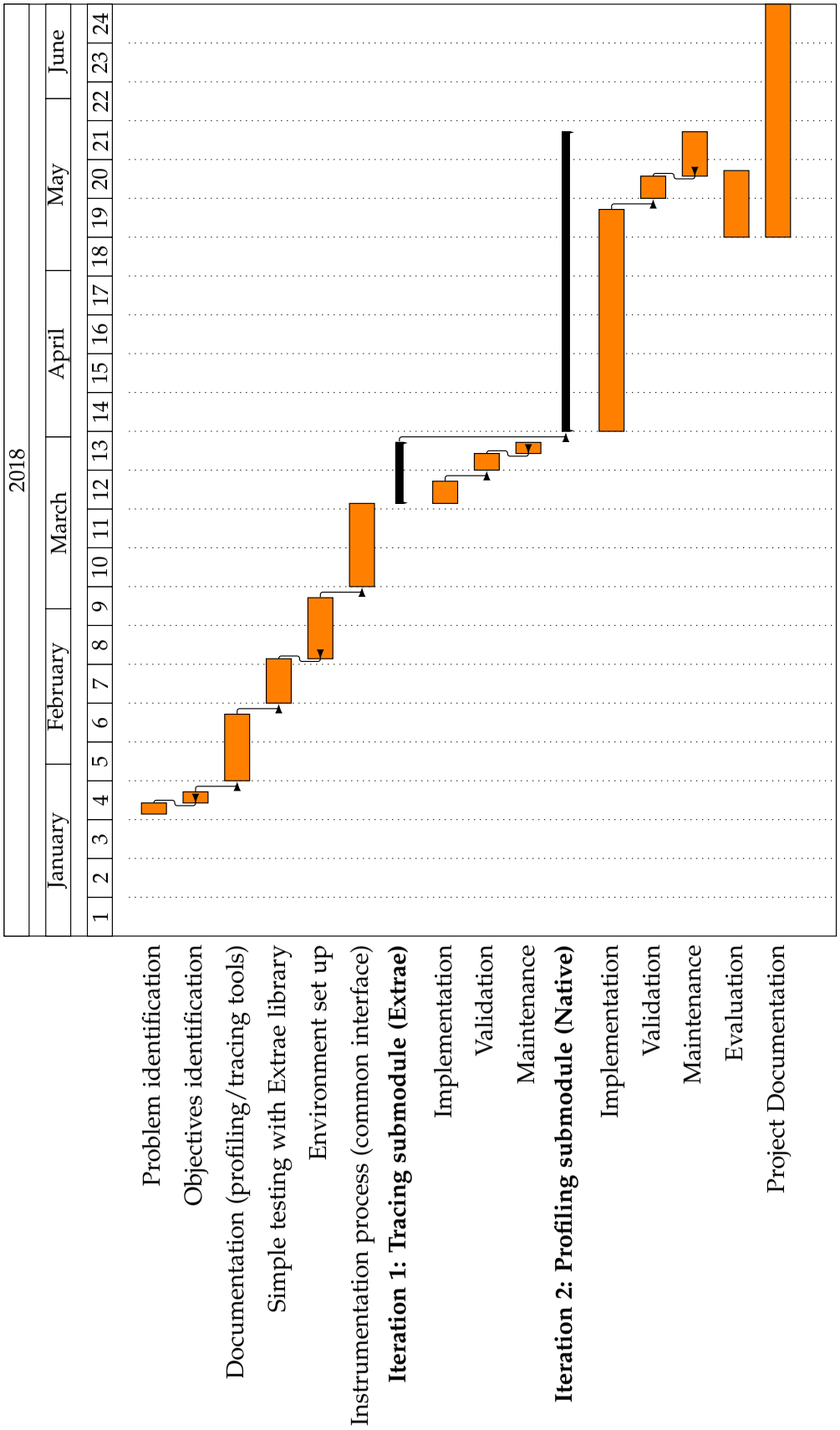


Figure 7-1: *Gantt Diagram.*

Chapter 8

Social-Economic Environment

This section summarizes the set of total costs project execution. From Section 8.1.1 to Section 8.1.4 both included, we will calculate the project budget relative to direct costs, while in Section 8.1.5, we will show the total indirect costs, which have been set to the 21% of the total direct costs. Finally, we will show a general table in Section 8.1.6 that summarizes both direct and indirect costs of the project.

8.1 Project budget

Next, we show the set of different costs taking into account to develop the whole project.

8.1.1 Human resources

Regarding human resources costs, say that the base prices from each profile have been obtained according to what the Government of Spain states in [64]. Therefore, from the minimum and maximum salary figures, we have based ourselves on the average salary they will receive. The development of this project has lasted 6 months, and 22 days per month. Then, the total human resources costs are detailed in Table 8.1.

Human Resources			
Role	*Cost/hour*	*Hours*	*Total*
Analyst	24.00 €	88	2,112.00 €
Designer	25.00 €	88	2,200.00 €
Programmer	20.00 €	264	5,280.00 €
Tester	22.00 €	88	1,936.00 €
		Total:	11,528.00 €

Table 8.1: *Human resources costs.*

8.1.2 Equipment

This section includes the total expenses of this project distributed in both hardware and software subsections.

Hardware

Regarding hardware expenses, say that prices for physical equipment have been calculated according to an amortization period of 5 years. Then, prices have been recalculated for a concrete period of 6 months, since that has been the total development period of this project.

Hardware Owned			
Concept	*Price/unit*	*Amortization (5 years) /month*	*Cost (6 months)*
Personal Computer	500.00 €	8.33 €	50.00 €
Dodo Remote Platform	4,500.00 €	75.00 €	450.00 €
		Total:	500.00 €

Table 8.2: *Owned hardware costs.*

Software

In Table 8.3, we show a list of the different elements this project has required to be completed. This list includes the *Operating System*, development and testing software and some additional software needed to develop the project.

Software			
Concept	*Price/unit*	*Amortization (5 years) /month*	*Cost (6 months)*
Linux Ubuntu 16.04	0.00 €	0.00 €	0.00 €
ShareLatex	0.00 €	0.00 €	0.00 €
Visual Paradigm Community Edition	0.00 €	0.00 €	0.00 €
Extrae	0.00 €	0.00 €	0.00 €
Paraver	0.00 €	0.00 €	0.00 €
Feedgnuplot	0.00 €	0.00 €	0.00 €
Microsoft Office	150.00 €	2.50 €	15.00 €
		Total:	15.00 €

Table 8.3: *Software costs.*

8.1.3 Consumables

Table 8.4 comprises the set of total consumable costs. Within office material, we can distinguish between pens, highlighters, sheets, printer ink, etc.

Consumables	
Concept	*Cost*
Office Material	30.00€
Total:	30.00€

Table 8.4: *Consumables costs.*

8.1.4 Travel costs

This refers to expenses related to the travels that have been needed to arrive at the work place during 6 months.

Travel Costs		
Concept	*Cost /month*	*Cost (6 months)*
Monthly travel pass	16.00€	96.00€
	Total:	96.00€

Table 8.5: *Travel costs.*

8.1.5 Indirect costs

With indirect costs we directly refer to those costs that are not accountable to the project cost. This includes, among others, electricity, central heating, water, internet, etc.

Hence, we have established indirect costs to be a 20% rate from the total direct costs. More details can be found in Table 8.6:

Indirect Costs	
Concept	*Cost*
Indirect Costs	2,433.80€
Total:	2,433.80€

Table 8.6: *Indirect costs.*

8.1.6 Total costs

This final section collects all previous prices (both direct and indirect costs), so that we can get an estimation of the total project budget. Besides, we have also included the price before and after the application of the 21% VAT tax. Results are shown in Table 8.7.

Total	
Concept	*Cost*
Human Resources	11,528.00 €
Hardware Owned	500.00 €
Software	15.00 €
Consumables	30.00 €
Travel Costs	96.00 €
Margin of Benefit (15%)	1,825.35 €
Other Costs (indirect costs)	2,433.80 €
Total before taxes	16,428.15 €
Total after taxes (21%):	**19,878.06 €**

Table 8.7: *Total costs.*

8.2 Social-economic impact

This section comprises an explanation of the social-economic impact this project could have on the society. We will talk not only about social terms but also about economic ones, since these are the two main aspects to which our proposed solution would affect. Next, we describe in Section 8.2.1 the set of involved aspects from the social point of view and in Section 8.2.2, we will proceed in the same way but from an economic perspective.

8.2.1 Social impact

This section collects different considerations that have been taken into account with respect to the social impact this project can cause.

As argued over this document, two main topics have been studied in this project. On one side, parallelism based on the use of parallel patterns, on the other side, the performance analysis module of parallel applications thanks to the developed profiling/tracing submodule. In this way, we could

considerate and analyze the social impact of these two different points of view.

With respect to the parallelism part, say that nowadays, scientific communities are more in favour to use parallelism paradigms in their applications. As argued in the introduction of this document, this is because nowadays larger amounts of data are generated, and developers are worried in the sense that they need much more resources in order to treat those data volumes. Parallelism is, therefore, a quite optimal solution to overcome that problem.

With the accomplishment of this project, we would like to motivate developers and users involved in the scientific area to use this programming modality. It is true that parallelism is not an easy task and requires some basic expertise, but currently, several parallel frameworks have been developed to ease parallelism to users by means of the aforementioned parallel patterns (in our case, we have used GrPPI). Apart from that, modern HPC architectures have been developed to be used according to some parallelism paradigms, since in any other case, they will not exploit all the computational load they offer. Parallel patterns hide away the complexity behind concurrency mechanisms and users can develop their applications in a very flexible way.

On the other side, regarding profiling and tracing approaches, we would like to mention possible benefits our proposed solution can give.

By using our performance analysis environment, we could be able to help the scientific community working on HPC parallel frameworks, since as it has been checked in the evaluation process, from a simple graph we can extract lots of conclusions about our implemented work.

This developed system may help them to detect possible bottlenecks and hot spots in their codes, so it will definitely ease their lives. We can give a simple example: in any research project, is very likely for anyone to spent many hours trying to understand some senseless behaviour in his program. By using our profiling/tracing module, they can analyze and inspect the application behaviour and directly detect the critical part, so he will save time in the sense that from that moment, that developer will only focus on that part, saving his time in that way.

8.2.2 Economic impact

Regarding the economic impact, say that this project is not oriented for commercial purposes. It has been developed just for research purposes and it will be published in the official page of the research group where it has been accomplished. Therefore, it is an open source software that will be available for any user interested in this area.

Chapter 9

Conclusions and Future Work

This chapter gathers together the set of conclusions extracted from this project after its accomplishment. Section 9.1 collects several concluding aspects together with some encountered problems along its development. Section 9.2 proposes future lines of work and possible extensions to this system trying to discover new lines of research in this area.

9.1 Conclusions

Nowadays generation of heavy data volumes is a done deal. It is true that devices costs regarding the storage of those data volumes have been decreased along time in the previous years, and besides, the memory capacity of them has increased enormously, but still total amounts of data produced are on top of this capacity. We are not noticing about such amounts of data that we generate day-to-day, but each time, there are more electronic devices capable not only producing data but capturing and transferring it, which is even a more complex situation for taking control of that data.

The HPC community, thus, is facing each time more computational load experiments due to the generation of those data volumes. To accelerate this processing and treatment process, modern architectures supporting several parallelism paradigms are being developed to get results in a more efficient way. The treatment of this data is actually a very important issue, since through it, scientists from any area will be able to discover new advances in life and new ways to face large and complex problems that before were not able to be solved with the available resources.

In this document, we have argued the actual state of modern computers and the challenges that are being proposed for a near future to develop even more powerful computers able to treat the aforemen-

tioned data volumes. We have also discussed about current pattern-based parallel frameworks that give developers alternatives to solve complex problems from different areas (text mining, economy, computer graphics, etc.) in a very flexible and robust way. However, we have stated that the usage of these parallelism paradigms is not always a straightforward task and causes developers lose the track of the inner application behaviour. Many different tools to examine the behaviour of parallel applications have been released to deal with this situation as we argued in the state of the art. They all are based on what is called profiling and tracing techniques, which in a different way (statistically and graphically respectively) they show some performance information about the application been executed.

As a solution to this general problem, we have proposed the development of a performance analysis environment, both profiling and tracing techniques, that aids these developers working at higher computational levels with the usage of different parallel paradigms, in such a way that after the execution of their applications, they will be able to receive performance feedback from it. Concretely, we have added it to an existing pattern-based parallel framework, namely GrPPI. This framework allows users to execute their applications into several different parallel back ends, switching from one to another in a very simple way. Therefore, with the two aforementioned performance profiling and tracing submodules, we will provide users another alternative to measure the application performance and also helping them to detect potential bottlenecks and hot spots present in their codes.

We have explained in detail the implementation of each performance submodule and checked in the evaluation part of this document the results obtained after the application of our system. We have observed how in a very simple way, we can obtain such amount of information that, in any other situation, we would not be able to notice about, and consequently, we could not be able to optimize our applications in the same way, mainly because parallel patterns are presented to users as mere black boxes that prevent them notice about what is really happening inside the code.

Next, we list the set of objectives proposed in Chapter 1 and review how we have addressed them in the project:

- **O1. Promote Parallel Pattern Programming:** As we have already checked in the evaluation chapter, we have observed that parallel patterns are a very flexible way for users to develop parallel applications without caring about concurrency mechanisms.
- **O2. Flexibility:** We have provided users to make use of two different submodules according to their needs, on one side profiling and on the other tracing.
- **O3. Efficiency:** Users can easily measure the application performance.
- **O4. Reusability:** Although in the evaluation chapter we have tested our system with one of the available back ends, we have justified in the implementation chapter that other back ends have also

been instrument to offer users different alternatives.

- **O5. Usability:** Users can check different versions from the same code once tested our system, since it can be used to improve pieces of code regarding parallel applications.

Therefore, we can conclude that all objectives have been fulfilled with success.

9.1.1 Personal conclusions

This section describes the main considerations and problems faced along this project. Say that although this project started in January 2018, several previous months were needed to fully understand the structure and behaviour of the parallel framework, where our system has been applied, GrPPI.

It took its time to understand it, since GrPPI is based on metaprogramming concepts that, as already stated in Chapter 4, making more difficult its understandability and source code. Notwithstanding, after many hours of work and dedication, the initial steps of the project were carried out in an effective way.

Besides, we need also to consider the amount of time that was spent in order to design both submodules in the most flexible and modular way within the GrPPI interface, since that was one of our main objectives. Design decisions were established at the beginning of the project, but several modifications were applied according to faced issues during the development of this system.

In general, we have faced some problems regarding the linking of libraries, understanding the way applications were compiled, maintenance, etc. However, it has been checked that with great effort and patience, things move on with success.

9.2 Future work

This section proposes additional lines of work so that this project can be completed in a future.

We did satisfy the list of elicitated user requirements, but the whole set of general ideas we had at the beginning of this project have not been fulfilled due to time limitations. However, we also need to mention that we have added an extra functionality to this project (`feedgnuplot` software) that actually was not expected to be finished, so we can not say we are not satisfied with our work.

In this project, we selected to candidates regarding the implementation of the tracing part: Extrae and TAU. Each of them presents two versions from its API, the basic one and the extended one. We familiarized ourselves with both basic APIs, first TAU and then Extrae, so that we could identify the

similarities and differences they showed and make some conclusions of it.

Finally, we opted to start the instrumentation process with Extrae library. The reason why that choice, was because after comparing them both, we concluded that Extrae library allowed the best trade-off between simplicity and offered functionalities. Finally, we just needed to use some of the total functions, so the way Extrae provided what we wanted was the most convenient for us.

Therefore, from this point on, we will list the set of additional lines of work that are compatible with this project and that are left for a future. We can stand out the followings:

1. Instrumentation of the Intel TBB back end offered by GrPPI. GrPPI acts as an intermediate layer between users and different parallel frameworks, therefore, if we also get instrumented that back end, we will give users just another alternative to measure its application performance.

2. Inclusion of new performance metrics to both tracing (in Paraver) and profiling submodules.

3. Show users other real-time metrics regarding `feedgnuplot` software different from the *throughput* one.

4. Inclusion of metrics regarding energy consumption of applications when being executed.

5. Instrumentation of the tracing *submodule* by means of the TAU library. Next, we will provide next a simple example to determine how we could proceed with the instrumentation process if this API was used instead:

Listing 9.1: *Routines instrumentation with TAU library [20].*

```
1   void routine1(){
2    TAU_PHASE_START(t2); // instead of Extrae_event(6000019, 1);
3    [code of routine1...]
4    TAU_PHASE_STOP(t2); // instead of Extrae_event(6000019, 0);
5   }
6
7   void routine2(){
8    TAU_PHASE_START(t2);
9    [code of routine2...]
10   TAU_PHASE_STOP(t2);
11  }
```

That was the same example proposed in Section 2.3.1. Here, however, we have used some directives from TAU API. TAU library offers a wide set of functions that may provide developers the measurement of very specific and variable metrics, so this would be another interesting alternative in order to fight for this need of performance analysis of applications.

6. GrPPI parallel framework has been designed to be the most extensible and modular library. Therefore, the inclusion of new tracing frameworks (different from the TAU library) is also another future line of work.

Acronyms

API Application Programming Interface. 22, 23, 30, 34, 35, 82, 151, 152

ARCOS Grupo de Aquitectura de Computadores, Universidad Carlos III de Madrid. v

ASCII American Standard Code for Information Interchange. 128, 132

BLAS Basic Linear Algebra Subprograms. 98, 99

BSC Barcelona Supercomputing Centre. 23

CPU Central Processing Unit. 6, 31

CREST Center for Research in Extreme Scale Technology. 22

CUDA Compute Unified Device Architecture. 21

FF Fastflow. 19

FIFO First In First Out. 20

FLOPS Floating-point Operations per Second. 1, 76, 100

GDPR General Data Protection Regulation. 133

GPU Graphics Processing Unit. 21

GrPPI Generic Reusable Parallel Pattern Interface. vii, ix, xiii, 3, 4, 7, 11, 17, 18, 36, 37, 40–42, 45–47, 50, 55, 66, 68, 78–80, 83, 91, 92, 111, 128, 134, 138, 147, 150–153

HJM Heath Jarrow Morton. 111, 112

HPC High-Performance Computing. 1, 2, 5, 19, 33, 147, 149

HPX High-Performance ParalleX. 22

LGPL GNU Lesser General Public License. 135

MC Monte Carlo. 112, 113

MPI Message Passing Interface. 31, 33, 36

NLP Natural Language Processing. 102

OTF Open Trace Format. 31

PAPI Performance Application Programming Interface. 23, 30, 34

PARSEC Princeton Application Repository for Shared-Memory Computers. 111

Score-P Scalable Performance Measurement Infrastructure for Parallel Codes. 33

SPSC Single-Producer-Single-Consumer. 20

TAU Tuning and Analysis Utilities. 23, 29, 30, 33, 37, 151–153

TBB Threading Building Blocks. 3, 18, 35, 47, 92, 152

UML Unified Modeling Language. 44

VI-HPS Virtual Institute – High Productivity Supercomputing. 23

Bibliography

[1] Exascale Team, "Exascale Computing." http://www.eesi-project.eu/about-eesi/what-is-exascale/what-is-exascale/, Last visited, April 2018.

[2] Association for Computing Machinery (ACM), "Why We Need Exascale Computing." https://www.huffingtonpost.com/entry/why-we-need-exascale-computing_us_58c94f59e4b0009b23bd94c4?guccounter=1.

[3] M. Wright and al., *The opportunities and challenges of exascale computing*. http://science.energy.gov/, U.S. Department of Energy, 2010.

[4] Department of Computer Science, "How to program large-scale heterogeneous parallel computers." https://www.inf.ethz.ch/news-and-events/spotlights/thoefler-erc.html.

[5] M. Wagner, A. Knüpfer, and W. E. Nagel, "Otfx: An in-memory event tracing extension to the open trace format 2," in *Algorithms and Architectures for Parallel Processing* (J. Carretero, J. Garcia-Blas, V. Gergel, V. Voevodin, I. Meyerov, J. A. Rico-Gallego, J. C. Díaz-Martín, P. Alonso, J. Durillo, J. D. Garcia Sánchez, A. L. Lastovetsky, F. Marozzo, Q. Liu, Z. A. Bhuiyan, K. Fürlinger, J. Weidendorfer, and J. Gracia, eds.), (Cham), pp. 3–17, Springer International Publishing, 2016.

[6] Blaise Barney, Lawrence Livermore National Laboratory, "Introduction to Parallel Computing." https://computing.llnl.gov/tutorials/parallel_comp/.

[7] S. Shende, "Profiling and Tracing in Linux." https://www.cs.uoregon.edu/research/paraducks/publ/htbin/bibify.cgi?cmd=list&coll=TALK.

[8] S. Shende, A. D. Malony, and R. Bell, "The TAU Performance Technology for Complex Parallel Systems." https://www.cs.uoregon.edu/research/paracomp/papers/talks/nrl-dc04/tautalk.pdf.

[9] del Rio Astorga David, D. M. F., F. Javier, and G. J. Daniel, "A generic parallel pattern interface for stream and data processing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, p. e4175, 2017. e4175 cpe.4175.

[10] Massimo Torquati, Computer Science Department, University of Pisa), "Parallel Programming Using FastFlow." http://calvados.di.unipi.it/storage/tutorial/fftutorial.pdf, September 2015.

[11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimo Torquati, "FastFlow: high-level and efficient streaming on multi-core." https://pure.qub.ac.uk/ws/files/815060/Wiley.pdf, Last visited, July 2011.

[12] A. Ernstsson, L. Li, and C. Kessler, "Skepu2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *International Journal of Parallel Programming*, vol. 46, pp. 62–80, Feb 2018.

[13] S. Team, "Skepu2, Autotunable Multi-Backend Skeleton Programming Framework." https://www.ida.liu.se/labs/pelab/skepu/.

[14] S. Team, "Skepu Autotunable Multi-Backend Skeleton Programming Framework." https://www.ida.liu.se/labs/pelab/skepu/skepu1/.

[15] "National Energy Research Scientific Computing Center." http://www.nersc.gov/users/software/programming-models/hpx/.

[16] C. Team, "CREST HPX." http://www.nersc.gov/users/software/programming-models/hpx/crest-hpx/.

[17] N. Team, "Stellar Group." http://www.nersc.gov/users/software/programming-models/hpx/stellar-hpx/.

[18] V.-H. Team, "VI-HPS, Extrae." http://www.vi-hps.org/Tools/Extrae.html.

[19] B. Team, "BSC, Extrae." https://tools.bsc.es/extrae.

[20] B. Team, "BSC Extrae Documentation." https://tools.bsc.es/sites/default/files/documentation/pdf/extrae-3.5.2-user-guide.pdf, Last visited, October 2017.

[21] B. Team, "BSC Paraver Tutorial." https://tools.bsc.es/sites/default/files/documentation/1367.pdf, Last Visited, November 2000.

[22] B. Team, "BSC Paraver Tracefile Description." https://tools.bsc.es/sites/default/files/documentation/1370.pdf, Last Visited, June 2001.

[23] B. Team, "BSC Paraver Tool Structure." https://tools.bsc.es/paraver/tool_structure.

[24] D. of Computer and U. o. O. Information Science, "TAU Performance System." https://www.cs.uoregon.edu/research/tau/home.php.

[25] P. Dagna, "Hybrid MPI+OpenMP Profiling." https://hpc-forge.cineca.it/files/ScuolaCalcoloParallelo_WebDAV/public/anno-2013/advanced-school/Profiling_9th_advanced_school_on_parallel_computing.pdf.

[26] S. Shende, A. D. Malony, and R. Bell, "The TAU Performance Technology for Complex Parallel Systems." https://www.cs.uoregon.edu/research/paracomp/papers/talks/nrl-dc04/tautalk.pdf.

[27] D. of Computer and U. o. O. Information Science, "TAU Instrumentation API." https://www.cs.uoregon.edu/research/tau/docs/tauapi/rn01re15.html.

[28] R. Filgueira, "Jumpshot , Vampir and TAU ." http://research.nesc.ac.uk/files/Jumpshot-Vampir-tau.pdf.

[29] F. Team, "Vampir and VampirTrace." http://futuregrid.github.io/manual/vampir.html.

[30] P. Drongowski, "PERF tutorial: Finding execution hot spots." http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/, Last visited.

[31] Wikipedia contributors, "Perf (linux) — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Perf_(Linux)&oldid=831169470, 2018. [Online; accessed 28-May-2018].

[32] "perf: Linux profiling with performance counters." https://perf.wiki.kernel.org/index.php/Main_Page, Last visited, 2015.

[33] "GNU prof." https://sourceware.org/binutils/docs/gprof/.

[34] S. Team, "Scalasca Motivation." http://www.scalasca.org/about/motivation/motivation.html.

[35] S. Team, "Scalasca Introduction." http://www.scalasca.org/about/about.html.

[36] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 702–719, Apr. 2010.

[37] V.-H. Team, "Score-P, Scalable Performance Measurement Infrastructure for Parallel Codes." http://www.vi-hps.org/projects/score-p/.

[38] S.-P. Team, "Score-P: Performance Measurement Infrastructure." https://www.olcf.ornl.gov/wp-content/uploads/2015/02/Score-P-OLCF-Tutorial.pdf, Last Visited, February 2015.

[39] J. Dongarra, K. London, S. Moore, P. Mucci, , and D. Terpstra, "Using PAPI for hardware performance monitoring on Linux systems." https://moodle.rrze.uni-erlangen.de/pluginfile.php/11671/mod_resource/content/4/01_IntroArchitecture.pdf.

[40] "POSIX Threads." http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html.

[41] C. Team, "C++11 Standard Library Extensions — Concurrency." https://isocpp.org/wiki/faq/cpp11-library-concurrency#std-lock.

[42] I. Team, "Optimizing Game Architectures with Intel® Threading Building Blocks." http://www.vi-hps.org/projects/score-p/.

[43] I. Team, "Intel Threading Building Blocks (Intel TBB)." https://software.intel.com/en-us/intel-tbb.

[44] O. Team, "OpenMP Introduction." https://www.sharcnet.ca/help/index.php/OpenMP.

[45] O. Team, "Pros and Cons Of OpenMP." https://www.dartmouth.edu/~rc/classes/intro_openmp/Pros_and_Cons.html#top.

[46] P. G. Team, "Characteristics of good user requirements." https://es.slideshare.net/guest24d72f/8-characteristics-of-good-user-requirements-presentation.

[47] T. P. Team, "Architecture Models." https://www.tutorialspoint.com/software_architecture_design/architecture_models.htm.

[48] C. S. Team, "Design Phases." https://www.careerride.com/page/design-phases-637.aspx.

[49] "BLAS Subroutines Levels." https://www.csie.ntu.edu.tw/~cjlin/courses/nm2013/part2.pdf.

[50] "BLAS and its Subroutines in Linear Algebra Subprograming." https://spectrum.ieee.org/computing/hardware/power-problems-threaten-to-strangle-exascale-computing.

[51] C. Team, "What is Text Analytics?." https://www.clarabridge.com/text-analytics/.

[52] U. of Auckland, "Programiz Team." https://www.cs.auckland.ac.nz/courses/compsci373s1c/PatricesLectures/Gaussian%20Filtering_1up.pdf, 2010.

[53] K. S. Praveen, G.Hamarnath, K. P. Babu, M. Sreenivasulu, and K.Sudhakar, *Implementation of Image Sharpening and Smoothing Using Filters*. Prentice Hall, 2016.

[54] "Basecd Team." https://medium.com/basecs/making-sense-of-merge-sort-part-1-49649a143478.

[55] "Khan Academy Team." https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms.

[56] "UCSB Computer Science Team." https://www.cs.ucsb.edu/~bboe/cs32_m12/slides/sort/.

[57] "Programiz Team." https://www.programiz.com/dsa/merge-sort.

[58] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto, "Bringing parallel patterns out of the corner: The p3 arsec benchmark suite," *ACM Trans. Archit. Code Optim.*, vol. 14, pp. 33:1–33:26, Oct. 2017.

[59] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[60] D. Gorokhov, "HJM Model for Interest Rates and Credit." https://ocw.mit.edu/courses/mathematics/18-s096-topics-in-mathematics-with-applications-in-finance-fall-2013/lecture-notes/MIT18_S096F13_lecnote24.pdf.

[61] K. Agarwal, "Monte Carlo Simulation." https://www.investopedia.com/articles/investing/112514/monte-carlo-simulation-basics.asp.

[62] P. Team, "P3arsec Benchmark." http://wiki.cs.princeton.edu/index.php/PARSEC#Swaptions.

[63] M. Hauser, "Mandelbrot Set." https://fineartamerica.com/featured/mandelbrot-set-black-and-white-fractal-art-matthias-hauser.html.

[64] S. Government, "Basis and Contributions in 2018." http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm.