Porting and tuning of the Mont-Blanc benchmarks to the multicore ARM 64 bit architecture

Ying hao Xu Lin

Director: Xavier Martorell Bofill (DAC)
Codirector: Filippo Mantovani (BSC-CNS)

Computer Engineering
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

This dissertation is submitted for the degree of
Grau en Enginyeria Informàtica

Universitat Politècnica de Catalunya
(UPC) - BarcelonaTech

October 2016

"No one knows what the future holds, that’s why its potential is infinite."

Acknowledgements

I would like to thank my director, Xavier Martorell for guiding and supporting me over these months. I will miss the weekly meetings discussing the optimizations! You have set an example of excellence as a researcher and mentor.

I would also like to thank my codirector Filippo Mantovani, for bringing me the opportunity to work in the BSC and to be part of the UPC team at the Student Cluster Competition (SCC) held in the ISC16. That conference really changed my mind!

To my SCC advisor and coworker, Dani Ruiz, for helping me every time I had a problem with the clusters. Without you, the thunder cluster would have never booted up during the competition!

Finally, I would especially like to thank my family for the support and constant encouragement I have gotten over the years. In particular, I would like to thank my grandfather and my grandmother, you have taught me to be passionate and to love what I do, the only way to do great work. Thank you very much!

Abstract

This project is about porting and tuning the Mont-Blanc benchmarks to the multicore ARM 64 bits architecture. The Mont-Blanc benchmarks are part of the Mont-Blanc European project and they have been developed internally in the BSC (Barcelona Supercomputing Center).

The project will explore the possibilities that an ARM architecture can offer running in a HPC (High Performance Computing) setup, this includes to learn how to tune and adapt a parallelized computer program and analyze its execution behavior.

As part of the project, we will analyze the performance of each benchmark using instrumentation tools such like *Extrae* and *Paraver*. Each benchmark will be adapted, tuned and executed mainly in the three new Mont-Blanc mini-clusters, Thunder (ARMv8 custom), Merlin (ARMv8 custom) and Jetson TX (ARMv8 cortex-a57) using the OmpSs programming model. The evolution of the performance obtained will be shown followed by a brief analysis of the results after each optimization.

Resum

Aquest projecte es basa en adaptar i afinar els Mont-Blanc *benchmarks* a l'arquitectura multinucli ARM 64 bits. Els Mont-Blanc *benchmarks* formen part del projecte Europeu Mont-Blanc i han estat desenvolupats internament en el BSC (Barcelona Supercomputing Center).

Aquest projecte explorarà el potencial d'usar l'arquitectura ARM en un entorn HPC (High Performance Computing), això inclou aprendre a adaptar i afinar un programa paral·lel, i analitzar el seu comportament durant l'execució.

Com a part del projecte, s'analitzarà el rendiment de cada *benchmark* usant eines d'instrumentació com *Extrae* o *Paraver*. Cada *benchmark* serà adaptat, afinat i executat en els tres nous mini-clústers de Mont-Blanc, Thunder (ARMv8 personalitzat), Merlin (ARMv8 personalitzat) i Jetson TX (ARMv8 cortex-a57) usant el model de programació OmpSs. Es mostrarà l'evolució del rendiment, seguit d'una breu explicació dels resultats després de cada optimització.

Resumen

Este proyecto se basa en adaptar y afinar los Mont-blanc *benchmarks* a la arquitectura multi-núcleo ARM 64 bits. Los Mont-Blanc *benchmarks* forman parte del proyecto Europeo Mont-Blanc y han sido desarrollados internamente en el BSC (Barcelona Supercomputing Center).

Este proyecto explorará el potencial de usar la arquitectura ARM en un entorno HPC (High Performance Computing), esto incluye aprender a adaptar y afinar un programa paralelo, y analizar su comportamiento durante la ejecución.

Como parte del proyecto, se analizará el rendimiento de cada *benchmark* usando herramientas de instrumentación como *Extrae* o *Paraver*. Cada *benchmark* será adaptado, afinado y ejecutado en los tres nuevos mini-clústeres de Mont-Blanc, Thunder (ARMv8 personalizado), Merlin (ARMv8 personalizado) y Jetson TX (ARMv8 cortex-a57) usando el modelo de programación OmpSs. Se mostrará la evolución del rendimiento obtenido, y una breve explicación de los resultados después de cada optimización.

Contents

Chapter 1

Introduction

1.1 Context

This Project is a TFG (in Catalan Treball Final de Grau) developed in the UPC FIB (Facultat d'Informàtica de Barcelona) with the support of BSC (Barcelona Supercomputing Center) and the Mont-Blanc Project which have given me access to the Mont-Blanc prototype clusters. The main idea behind this project is to port and tune a set of benchmarks to the multicore ARM 64 bit architecture using several programming models like OpenMP and OmpSs.

A benchmark is a program which main goal is to quantify the performance of a computer and to be able to compare it performance with other computers. This is achieved by making mathematical operations or doing basic operations like accessing to the memory and analyzing the time it takes.

The different programming models like OpenMP or OmpSs (used to parallelize the benchmarks) are API's (Application Programming Interface) that supports multi-platform and shared memory multiprocessing programming. These programming models consists of a set of compiler directives, library routines and runtime variables. They offer an easy way to parallelize any code in order to achieve a better execution time and they are widely used in supercomputers, where the number of CPU cores are about 50.000.

The main problem that nowadays supercomputer suffers is the power consumption. For example, the MareNostrum supercomputer allocated in the UPC Campus Nord it consumes 1MW/hour which is approximately 1.4 million of euros per year. This also limits the performance of the machine, because some cities like Barcelona can't deliver much more power than what MareNostrum already consumes, so the possibility of expanding the supercomputer

with more processors is also limited.

The Mont-Blanc Project is a project coordinated by the BSC which main goal is to develop a supercomputer based on mobile technology. To achieve this, Mont-Blanc supercomputers use ARM processors, the same ones that we can find in smartphones or tablets. The results obtained after the execution of the benchmarks in these machines can be used to compare the real performance of the new Mont-Blanc project prototype clusters with the performance of other existing supercomputers.

In order to tune the different benchmarks to the Mont-Blanc clusters, specific tools like paraver, extrae or gprof will be used. These tools help us to understand the behavior of the applied parallel strategy and to discover how to maximize the performance in each case [1].

Each benchmark will be adapted, tuned and executed mainly in the two new prototype clusters from the Mont-Blanc project that are called Thunder and Merlin. However, all the work will be done through a personal computer connected remotely to the clusters. Also we will evaluate the Jetson TX1 board from NVIDIA.

The three clusters are based on ARM 64bit technology. The merlin one is made by AppliedMicro and is the X-GENE2[1] model which has 8 cores with 128 GB of RAM DDR3 . The thunder one is made by Cavium and it is the ThunderX[2] model which has 48 cores and 128 GB of RAM DDR3 . Both processors uses a custom implementation of the ARMv8 cores, focusing its features for enterprise server solutions. The Jetson TX1[3] is an embedded system-on-module (SoM) with quad-core ARM Cortex-A57, 4GB LPDDR4 and an embedded Maxwell GPU with 256 CUDA cores.

1.2 Stakeholders

1.2.1 Developers

The developers will be responsible of all the technical part, this includes the development of each benchmark for the ARM architecture and it corresponding documentation that details the parallelization techniques used for this architecture.

[1]More information available at https://www.apm.com/products/data-center/x-gene-family/x-gene
[2]More information available at http://www.cavium.com
[3]More information available at http://www.nvidia.com/object/jetson-tx1-dev-kit.html

1.2.2 Director and Codirector

The director of this project is Xavier Martorell, he is an associate professor from the computer architecture department from UPC, postdoc researcher and manager of the parallel programming models group inside the BSC. The codirector of the project is Filippo Mantovani, a postdoc researcher and the project coordinator of the Mont-Blanc project. Their roles will be essential for the correct development of the project and their knowledge will be key for the analysis of the results.

1.2.3 Users

We consider as a user anyone who will benefit of the final product of this project, this means that any developer who works with an ARM architecture machine will have a detailed guide of how to code their applications in parallel for this type of architectures.

1.2.4 Beneficiaries

The direct beneficiary will be the BSC. Nowadays, the usage of ARM super computers is not as extended as the usage of conventional supercomputers and the results of this project will help to prove the benefit of using ARM architectures. This will help the BSC to make more advanced project proposals in order to achieve more funding to support the Mont-Blanc project.

Chapter 2

State of the art

The ARM architecture has been a revolution in the embedded systems where the power efficiency is critical. The company behind this architecture is ARM Holdings, a British multinational semiconductor and software design company headquartered in Cambridge, England. The architecture name, ARM, belongs to Advanced RISC Machine. The RISC (Reduced Instruction Set Computing) processors requires less transistors than the typical x86 CISC (Complex Instruction Set Computing) processors that can be found in personal computers [2]. This approach helps to reduce costs and heat of the device without sacrificing performance. Such reductions has contributed in the development of powerful devices that are more power-efficient and requires passive dissipation like smartphones, tablets and other embedded systems.

2.1 Computing on Mobile SoC's

The industry of the mobile SoC's (System on a Chip) is growing really fast. The market demands more and more powerful devices with less consumption and the interest of green-computing has also grown up since the fast-growing adoption of smart gadgets that uses batteries.

Nowadays, an x86 based processor can easily outperform a processor based on the ARM architecture, however, the market trend of this last years has shown that the performance of the mobile technology is growing much faster (Figure 1) and someday it will be greater or equal to the server processors performance.

There is a research project ongoing in order to evaluate if these mobile chips are ready for supercomputing. HTC, the well-known smartphone manufacturer launched in 2014 the

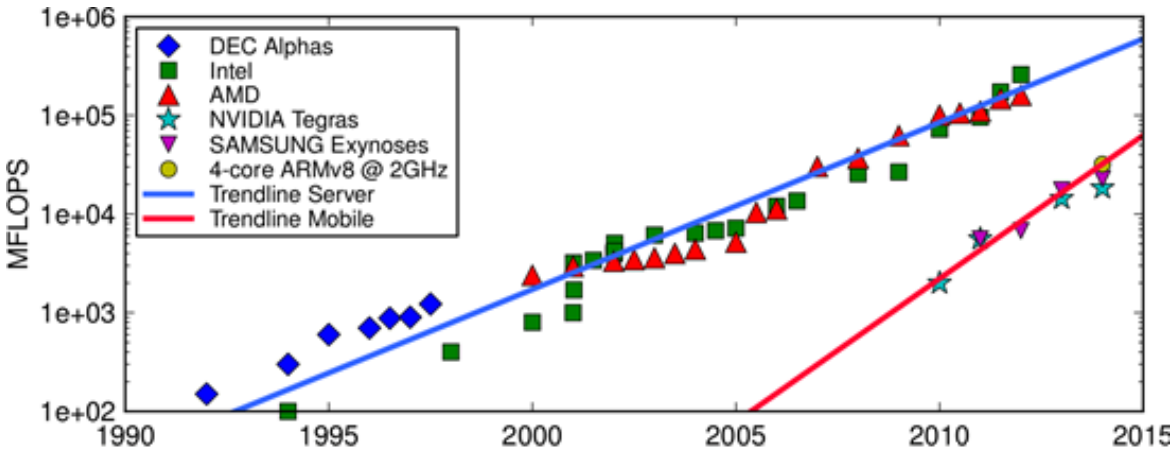


Figure 1 "Server and mobile performance trend"

“Power to give” program which aims to use the processing power of your Android smartphone to compute some of the data necessary to solve world’s deepest challenges [3]. Research problems are broken into many small tasks and distributed across a connected network of smartphones, so they can contribute in the calculation required to process the individual tasks by using the processing power of the built-in ARM processor from the smartphone.

The idea of using a public volunteer computing network where every connected machine computes part of the problem is not new, however these project helps to conclude that ARM processors are not only promising for their performance, but for their fast-growing presence in the market.

2.2 Mont-Blanc European Project

The Mont-Blanc project shares the vision of developing a European Exascale supercomputer focused in the power and cost-efficiency by using embedded technologies [4].

Nowadays, the Mont-Blanc project has developed two prototypes, the Mont-Blanc and the Mont-Blanc 2. This last one represents the first large ARM-based prototype dedicated to HPC, with 1000+ computational nodes which have been operational since May 2015 at Barcelona Supercomputing Center and open to the partner access.

The Mont-Blanc project has also developed a complete HPC (High Performance Computing) system software stack tested on both, prototype and commercial ARM-based platforms.

The idea behind using ARM-based processors for HPC it bases on the fact that producing this kind of chipsets requires less money but also due the fast-growing market of embedded systems (Internet of Things, smartphones, tablets and smart gadgets), a higher demand of this kind of chipsets has reduced significantly the production cost.

2.3 Power-efficient supercomputing

Power consumption has always been a big issue in the computers industry. As joule heating effect states, consuming more power means that more heat has to be dissipated.

Nowadays, there is a big interest on green-computing, a clear example is the green500 list that classifies the supercomputers found in the TOP500[4] list by their performance per watt [5].

The ARM processors are mainly targeted to run on battery-powered devices and because that, they are well-recognized as a very efficient processors. This efficiency is achieved by using smart governors[5] that can manage the resources wisely however, at full load they still are not the best power-to-solution compute units.

The GPU's / accelerators are considered the most efficient performance per watt devices and they are widely used in worldwide supercomputers. Most of the top supercomputers that can be found in the green500 list uses GPU's (Graphics Processor Unit) and accelerators. The idea behind the GPGPU[6] (General Purpose on Graphical Processor Unit) is not new, it started in the early 2000 and has experiment a fast adaptation since NVIDIA launched CUDA (Compute Unified Device Architecture) in 2007 [6].

The GPU's offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the rest of the code still runs on the CPU. This approach simplifies the logic of the GPU architecture resulting into a really power-efficient chips with unmatched parallel performance.

[4]The TOP500 project ranks and details the 500 most powerful supercomputer systems in the world

[5]The governor defines the power characteristics of the CPU, which in turn affects CPU performance. Each governor has its own unique behavior, purpose, and suitability in terms of workload.

[6]The GPGPU (General Purpose on Graphical Processor Unit) approach aims to take advantage of the computational capacity of the GPU (Graphics Processor Unit) by performing computation in applications traditionally handled by the CPU (Central Processing Unit)

2.4 Parallel programming models

In computing, a programming model refers into an abstraction of the parallel computer architecture which usually take the form of a library invoked as an extension to an existing language. The programming models are divided in two main areas: process interaction and problem decomposition.

Process interaction

Process interaction refers into how the parallel processes are able to interact/communicate between them.

Shared memory: Processes share a global address space that can be read and written asynchronously.

Distributed memory: Processes does not share the same address space and requires a message-passing mechanism to exchange data between the other processes.

Problem decomposition

Problem decomposition refers into how the workload is divided between the parallel processes.

Task parallelism: It focuses in the distribution of the work through threads. This approach requires communication between the threads.

Data parallelism: It focuses in doing operations on a data set. Using structures similar to arrays, it performs an operation in each of the array position simultaneously.

2.4.1 OpenMP

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) oriented to the programming of parallel systems with shared memory using threads. It supports C, C++ and Fortran and can run in most of the architectures and operative systems. It also supports MPI (Message Passing Interface) which is a portable message-passing system that permits the communication between processes [7].

Nowadays, OpenMP is managed by a nonprofit technology consortium called OpenMP ARB (OpenMP Architecture Review Board) formed by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle...

2.4.2 OmpSs

OmpSs is a programming model developed exclusively in the BSC which objective is to extend the OpenMP programming model with new directives to support asynchronous parallelism and heterogeneity[7]. The OmpSs environment is built on the top of Mercurium compiler and Nanos++ runtime system [8].

OmpSs has adopted the StarSs execution model. While OpenMP uses a fork-join model, StarSs uses a thread-pool model. In the fork-join model, one thread executes the code and the rest of the threads are created only when a parallel region is found. However, in the thread-pool model, the threads are created at the beginning of the execution, ready for receiving work, and destroyed at the end of the execution [9].

2.4.3 CUDA / OpenCL

CUDA (Compute Unified Device Architecture) is an API developed by NVIDIA oriented to programming of parallel systems using CUDA-enabled GPU's for general purpose processing (GPGPU). OpenCL (Open Computing Language) is very similar to CUDA but not only lets you execute programs in GPU's, but also in CPU's (Central Processing Unit), DSP's (Digital Signal Processor), FPGA's (Field-Programmable Gate Array) and other hardware accelerators. The main benefit of OpenCL is that it is open source and not only limited to NVIDIA devices.

The GPGPU approach bases its execution in a host/device model, where the host sends the necessary data and launches the same kernel function in each thread of the device. Each thread will be responsible of doing the corresponding calculus based on their global thread index.

[7] A Heterogeneous system not only increases its performance by adding more processors of the same type, but by adding dissimilar coprocessors with specialized processing capabilities.

These programming models are great for reducing the execution time of the programs by parallelizing the work, but it will not ensure that the applications will take full-advantage of all the potential that a cluster can offer, so and extensive analysis of the execution results and an optimization of the code will be necessary.

2.5 Similar projects

The benchmarks are widely used in the computer field in order to quantify the performance of a PC. The own TOP500 organization, who makes an annual TOP of the best 500 supercomputers of the world uses the linpack benchmark in order to decide which position of the list corresponds to each supercomputer [10].

In the nowadays context, green computing is becoming more and more important in the supercomputing field and using ARM computers is a way to achieve a more energy-efficient computing model. That's why there are some studies which main objective is to optimize libraries like GCC for ARM architectures in order to achieve a better performance [11] or to develop a more energy-aware application [12]. There is also a similar project that offers a library that includes a set of common and useful functions that have been heavily optimized for the ARM architecture [13]. Although all of them have a similar goal to our project, they are very generic and our project pretends to adapt and tune the execution of a specific set of benchmarks to get the best performance in a specific hardware configuration for a later analysis of the obtained performance.

Chapter 3

Scope

3.1 Objectives and requirements

3.1.1 Objectives

The main idea behind this project is to explore the possibilities that an ARM architecture can offer, this includes to learn how to tune and adapt a parallelized computer program (a benchmark in this case) for a relatively new architecture in the supercomputer field such the ARM architecture. Also, the evolution of the performance obtained after apply each tuning technique will be documented and shown.

3.1.2 Requirements

- The used tools for developing should be present in Merlin, Thunder and Jetson-TX clusters.
- The execution after applying each different tuning technique should improve the performance.
- The tuning techniques should be aware of the architecture where is going to run.
- The applied techniques should not vary the precision of the results compared to the obtained results nor the algorithm nature of the original version.
- The evolution of the performance after applying each technique should be documented in order to have a detailed guide of how to code applications in parallel for an ARM architecture for future users.

3.2 Obstacles and risks of the project

During the development of the project, there are probabilities that we face different problems that are directly or indirectly related with the own project. For each case, we will try to give an efficient solution that minimizes the impact with the development of the project.

3.2.1 Non-availability of the Jetson TX, Merlin or Thunder clusters

TheJetson TX, Merlin and Thunder clusters are three new mini-clusters that are part of the Mont-Blanc 2. This means that they are relatively new and the probabilities of facing problems will be higher than usual.

Solution: This risk is the one that will have the highest negative impact to the project, however, if it occurs, the Mont-Blanc prototype can be used but it will obligate us to redo all techniques in order to be optimized for the Mont-Blanc 1 architecture.

3.2.2 Non-availability of internet

The project is developed through a personal computer connected to clusters remotely. So any internet issue will affect in the development of the project.

Solution: If the interruption of the internet is long term, the development of the project can be done through another site where there is Internet, for example, through the FIB computers or FIB Wi-Fi with a personal laptop.

3.2.3 Source code errors

The benchmarks used in this project are not developed by us. If there is a bug in the original source code, the result after applying the different techniques will give an erroneous result that can confuse us to think that the problem is related to our optimizations.

Solution: In order to minimize this risk, each benchmark will be tested before each optimization and double checked to detect any possible bug in the implementation.

3.2.4 Calendar

The available time to develop this project is very short (8 months), if any task gets harder than expected, it will affect to the deadline of all the other tasks.

Solution: Give a realistic deadline for each task and maintain weekly meetings with the director of the project in order to validate the done work.

3.2.5 Incorrect interpretation of the results

Due it will be the first time that I will work in a supercomputer, the results obtained and the conclusions I extract after analyzing them can be erroneous by lack of knowledge.

Solution: After applying each tuning technique, the results will be checked with the director of the project in order to ensure that the behavior is the expected one.

3.2.6 Execution queue overbook

In both Thunder and Merlin, there are only 4 nodes available per cluster. This machines are shared with other user and sometimes because there aren't a lot of nodes, all the resources are taken by another user and we have to wait.

Solution: In this case there is no other solution than wait until the job queue is empty for executing our job.

3.3 Methodology

Before starting the real development of the project, we have defined the tasks to be done and decided how much development time it will require.

First of all, we will start by configuring and testing if the necessary tools we are going to need like OpenMP or OmpSs are installed and working in the clusters.

After we know for sure that all the necessary tools are installed and working, we will run the sequential version of the benchmark and annotate, the obtained result and the execution time it takes. We will also check the correctness of the program by checking if the result is

the expected one.

After executing the sequential version, we will execute the parallel version (OpenMP and OmpSs) and in the same way as in the sequential version, we will annotate and check the correctness of the obtained results.

Once we know that the parallel version is correct, we will proceed to adapt and apply different tuning techniques. In order to know which technique apply, we have to understand the behavior of the parallel application. To achieve that we will analyze the execution of the program with instrumentation tools like extrae and paraver.

After applying each technique, we will execute the benchmark and evaluate the effect of the optimization. The evolution of the performance after applying each technique will be annotated and analyzed.

To validate the optimizations we have done, we will compare the result with the sequential version and if all has went well we should obtain exactly the same results.

3.3.1 Development methodology

The software part of the project will be developed at home with a personal laptop connected through SSH (Secure Shell) to the clusters, however, the weekly meetings will be in the FIB.

3.3.2 Source code management

In the Mont-Blanc machines, we have limited permissions, this means that we cannot use sudo commands or install third party apps. However, there is git installed in the machines so we will use it in the project in order to have a control of the versions.

3.3.3 Partial evaluation

After applying an optimization to the benchmark, the result obtained will be compared to the result obtained in the execution of the sequential version in order to check it correctness. We will also analyze the behavior of the execution and decide if the optimization is good enough to be implemented in the final version of the code. This last part will be done in cooperation of the director of the project that will help me to understand the obtained results.

3.3.4 Final evaluation

Once all the tuning techniques have been done and we have checked that the execution result still correct, the performance speedup in comparison to the original version will be quantified and the evolution after applying each technique will be explained and shown.

Chapter 4

Planning

This project is rated to be done in 8 months, from February of 2016 to September of 2016. In this section we will show the temporal planning of the project which includes the description of the tasks that are going to be executed, the action plan that will ensure finishing the project within the given time and the resources that will be necessary in order to complete each task successfully. However, the planning described in this document is not rigid and can be subject of changes if the project requires it.

4.1 Tasks description

In this section we are going to explain what will be done in each activity, their length, and the resources (both material and human) needed to complete them. The temporal planning of the tasks can be seen in the Gantt diagram (Figure 2).

4.1.1 Project planning

This task is structured in six different stages:

- Context and scope of the project (24,5 hours)
- Planning (8,25 hours)
- Budget and sustainability (9,25 hours)
- Video presentation (6,25 hours)
- Scope statement (8,5 hours)
- Oral presentation and final document (18,25 hours)

The resources needed are a computer, Microsoft Office, Microsoft Project, a PDF viewer, a camera (for the video presentation) and OneDrive.

4.1.2 Initial system set up

Before starting the development of the project, we need to set up the necessary software and tools that the project itself requires in order to work on it.

To develop the project, we need to have OpenMP and OmpSs installed for compiling and executing the parallel versions of the benchmarks. Extrae is also needed in order to generate a trace of the execution. These traces will be used to understand the behavior of the execution. All these tools are already installed in the Mont-Blanc machines, however, for doing local tests, we have to install them in our personal computer too.

The resources needed are a computer running Linux and an internet connection for downloading the required tools.

4.1.3 Getting used with the development environment

The Mont-Blanc clusters uses an adapted to ARM Linux distribution, this means that we are already familiar with the main environment (Linux), however, we are going to connect to the machines remotely through an encrypted connection (Secure Shell) and we will develop entirely through the terminal. In the case of text editing, we will have to learn to use vim, which is a built in the terminal text editor.

The executions in the Mont-Blanc clusters are done through the SLURM queue system. SLURM (Simple Linux Utility for Resource Management) is an open-source resource manager designed for Linux clusters of all sizes. Learn how to use SLURM will be key for the project because a queue system lets the user to allocate exclusive access to resources during an execution. This is fundamental for obtaining consistent execution results.

We will investigate about OpenMP and OmpSs, reading tutorials and books, and we will also practice with simple programs in order to get used to their runtimes directives.

The resources needed are the Mont-Blanc clusters and a PC with Linux.

4.1.4 Analysis of the sequential execution

A sequential execution is an execution where only one thread will execute the program in the specified order. During this task, the input sets will be evaluated in order to suit each architecture. This is crucial for the correct development of the future tasks. In order to facilitate the execution of all the benchmarks, a simple bash script will be done. This script will execute each benchmark and will collect the results in a file for a later analysis.

The resources needed are the Mont-Blanc clusters, vim and a PC with Linux.

4.1.5 Analysis of the parallel execution

A parallel execution is an execution where multiple threads can execute the same portion of code. This approach helps to reduce drastically the execution time of the program, however, it also requires a special care because inconsistency risks may occur.

In this task, we will execute the parallel version of each benchmark and we will compare the results with the ones obtained in the previous task in order to discard any possible error related to the parallelization of the program. Similarly to the previous task, a simple bash script will be done to automate the process of executing and collecting the results.

The resources needed are the Mont-Blanc clusters, Extrae, Paraver, performance counters, gprof, vim and a PC with Linux.

4.1.6 Preliminary evaluation with different programming models

Due time limitation, in this project, we are only going to port and tune the benchmarks for a specific programming model (OmpSs, OpenMP) and data type (float, double). In order to identify which configuration suits better for our project, we are going to execute all the benchmarks in all the possible configurations in each cluster (thunder, merlin and jetson-tx).

Once we have executed all the benchmarks in all the possible configurations, we will analyze the obtained results and evaluate the performance in each case before starting to adapt and tune the benchmarks.

The resources needed are the Mont-Blanc clusters, vim, Microsoft Office and a PC with Linux.

4.1.7 Adaptation and tuning of the benchmarks

This task is the core task of the project. It covers most of the investigation, analysis and development part of the project. It can be divided in the following stages:

- **Analysis:** The result obtained from the execution will be analyzed using instrumentation tools like Paraver. This tool will help us to understand the behavior of the execution in the running hardware configuration and to decide which optimization techniques apply.
- **Tuning and optimizing the execution:** In this stage, we will apply a specific optimization technique based on the obtained result in the previous stage.
- **Execution of the benchmark:** The benchmark will be executed in each cluster (Merlin, Thunder) using the SLURM queue system. The obtained result will be compared to the previous one to check if it is still correct and the performance improvement will be quantified and analyzed.

These stages will be repeated for each optimization technique that we apply to each benchmark. The optimization techniques will be aware of the architecture where the benchmark is running. The final analysis will be done with the help of the director of the project in order ensure that the behaviors of the applied optimizations techniques are the expected ones.

Because we had access to the Jetson TX1 in early September, we are only analyzing and optimizing the performance for the merlin and thunder clusters, however, we will do a brief architecture exploration in the Jetson TX1 cluster by comparing the performance of the benchmarks running in the CPU and the GPU.

The resources needed are the Mont-Blanc clusters, vim, Paraver, Extrae, performance counters, gprof, git and a PC with Linux.

4.1.8 Final task

In this task there are two main subtasks, the first one is to prepare the delivery of the project, this includes to document clearly the improvements we have seen after applying each optimization technique, the evolution of the performance and the justification of why using each technique, and the second one is to prepare for the final presentation.

The resources needed are a PC, LaTex, Microsoft Office, a PDF viewer and OneDrive.

4.2 Task dependencies

The dependencies between the tasks mentioned above are simple. The first task to be done is the project planning. This task is key for the correct organization and development of the project. The "Initial set up" and "Getting used to the environment" doesn't have dependencies between them, however, this two are precedent tasks for the remaining tasks. The next tasks will follow a linear dependency: Analysis of the sequential execution -> Analysis of the parallel execution -> Preliminary evaluation with different programming models -> Adaptation and tuning of the benchmarks -> Final task. The follow-up stage is independent of the other tasks but must be done once the project is in an advanced stage.

4.3 Time table

Task	**Estimated time (hours)**
Project planning	75
Initial system setup	5
Getting used with the development environment	15
Analysis of the sequential execution	10
Analysis of the parallel execution	50
Preliminary evaluation with different programming models	40
Adaptation and tuning of the benchmarks	200
Follow-up stage	8
Final task	85
Total	**488**

Table 1 Summary of the time spent in each task.

4.4 Resources

Hardware resources

- Laptop PC (with an Intel i7 4720HQ @ 2.6 GHz, 16GB of RAM, Nvidia GTX 860M) running a Linux distribution: used in all the tasks of the project for developing.
- GoPro HERO camera: used for recording the preliminary oral presentation.
- Thunder cluster: Cavium ThunderX with 128GB of RAM and 128GB of SSD. Each node has 2 sockets with 48x ARMv8 cores each.

- Merlin cluster: Applied Micro XGENE2 with 128GB of RAM and 128GB of SSD. Each node has 1 socket with 8 ARMv8 cores.
- Jetson-TX cluster: NVIDIA Jetson TX1 with 4GB of RAM with an 256-core Maxewll GPU.

Software resources

- Gprof: A profiling tool that helps to understand the behavior of the execution.
- Extrae: A tool that generates a trace of an execution for a post-mortem analysis.
- Paraver: A tool for visualizing the Extrae generated traces.
- Vim: A built in the terminal text editor.
- Performance counters: Built-in hardware counters from the CPU.
- OmpSs: A parallel programming model developed in the BSC.
- OpenMP: A parallel programming model managed by OpenMP ARB (Architecture Review Board) consortium.
- Microsoft Office: An office suite of applications and services.
- Microsoft Project: A project management software program.
- OneDrive: A cloud storage service from Microsoft.
- PDF viewer: A tool used for visualizing documents in PDF format.
- LaTex: Is a document preparation system.

Human resources

The development of this project will involve Xavier Martorell, Filippo Mantovani and me as the main developer of the project.

4.5 Action plan

Our objective is to complete the tasks in the same order as described in the previous section, however, during the development of the project, unexpected problems can appear. If this occurs, we will try to analyze and manage them in order to minimize the negative impact over the original task scheduling. Our software development methodology is similar to Agile, with an adaptive planning, evolutionary development and continuous improvement. We will do a weekly sprint which includes a meeting with the director of the project before each sprint. This methodology will help us to offer a rapid and flexible response to change in case of facing any unexpected problem.

Any possible deviation in the task scheduling, will not affect to the assigned resources of each task. If a task requires less time than expected, other tasks can be started earlier, however, if its development is extended enough to compromise the finalization of the project within time, the number of optimization techniques applied in the "Adapting and tuning of the benchmarks" task will be reduced.

The expected needed time to finish the project are 488 hours. Considering that we have approximately 32 weeks, we have to dedicate 15 hours per week, which lets us to finish the project within time without any issue.

4.6 Gantt Chart

Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		**Project planning**	**31 días**	**lun 22/02/16**	**vie 01/04/16**	
2		Context and scope of the project	8 días	lun 22/02/16	mié 02/03/16	
3		Planning	4 días	jue 03/03/16	mar 08/03/16	2
4		Budget and sustainability	5 días	mié 09/03/16	mar 15/03/16	3
5		Video presentation	4 días	mié 16/03/16	dom 20/03/16	4
6		Scope statement	10 días	lun 21/03/16	vie 01/04/16	5
7		Oral presentation and final	10 días	lun 21/03/16	vie 01/04/16	5
8		Initial system setup	2 días	lun 04/04/16	mar 05/04/16	1
9		Getting used with the development environment	15 días	lun 04/04/16	mar 26/04/16	8
10		Analysis of the sequential version	10 días	mié 27/04/16	mar 10/05/16	9
11		Analysis of the parallel version	15 días	mié 11/05/16	mar 31/05/16	10
12		Preliminary evaluation of the programming	12 días	mié 01/06/16	jue 16/06/16	11
13		Adaptation and tuning of the	60 días	vie 17/06/16	jue 08/09/16	12
14		Monitoring report	1 día	jue 01/09/16	jue 01/09/16	
15		Final task	30 días	vie 09/09/16	jue 20/10/16	13

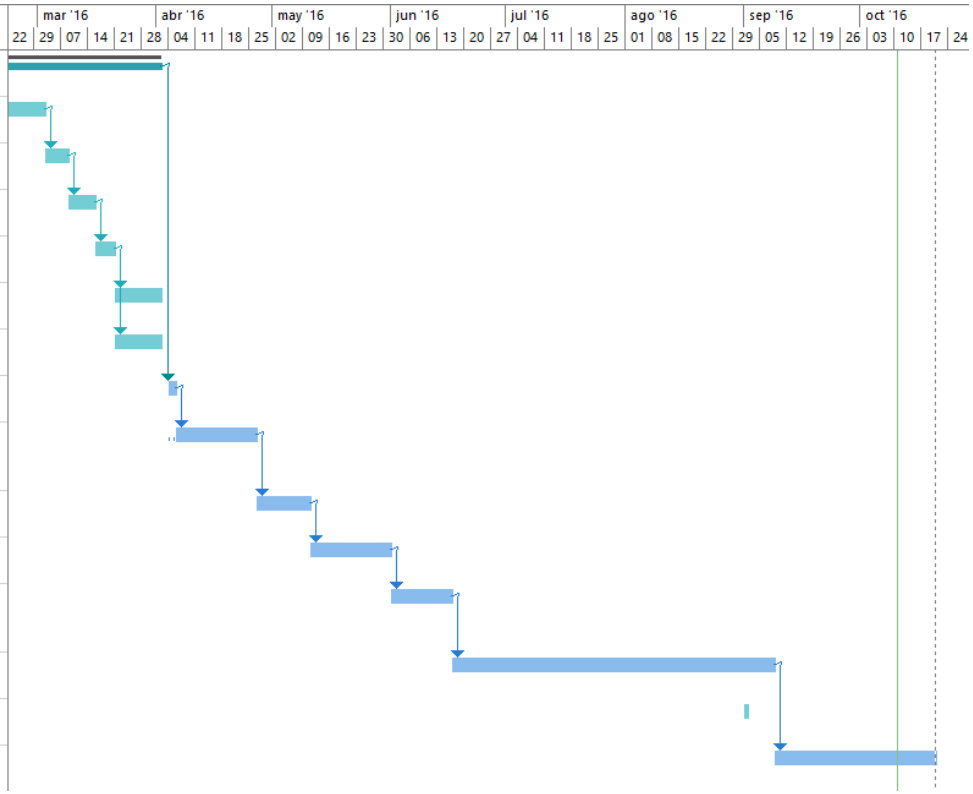


Figure 2 Gantt chart of the project

Chapter 5

Budget and sustainability

This section is about the budget and the sustainability of the project. It will contain a detailed description of both material and human costs, an analysis of how the budget will be affected by any possible deviations and an evaluation of the sustainability of the project. However, the budget described in this document may be subject to changes depending on the development of the project

5.1 Budget estimation

In order to develop the project successfully, we will require all the resources mentioned in the section 3.1 Resources. In this section we will estimate the costs of the human resources, hardware resources, software resources and the indirect costs. The amortization of each product will be calculated through two factors, the first one is the useful life of the product and the second one is the amount of time we will use them (8 months).

5.1.1 Software resources

In this section, we are going to estimate the costs of the software tools that will be used during the development of the project.

Most of the software tools are free because they are open source, however, we are going to need the office suite applications from Microsoft for the project planning and the final task.

Product	**Price**	**Useful life**	**Amortization**
Gprof	0.00€	-	0.00€
Extrae	0.00€	-	0.00€
Paraver	0.00€	-	0.00€
Vim	0.00€	-	0.00€
Performance counters	0.00€	-	0.00€
OmpSs	0.00€	-	0.00€
OpenMP	0.00€	-	0.00€
Microsoft Office	100.00€	4 years	16.66€
Microsoft Project	769.00€	4 years	128.16€
LaTex	0.00€	-	0.00€
OneDrive	0.00€	-	0.00€
PDF viewer	0.00€	-	0.00€
Total	**869.00€**		**144.82€**

Table 2 Software budget

5.1.2 Hardware resources

In this section, we are going to estimate the costs of the hardware resources that are needed to develop the project.

Product	**Price**	**Useful life**	**Amortization**
MSI GS60 2PC	1600.00€	4 years	266.67€
GoPro Hero	130.00€	4 years	21.67€
Thunder cluster	*2 500.00€	4 years	416.67€
Merlin cluster	*2 000.00€	4 years	333.33€
Jetson-TX cluster	~~599.00€~~	4 years	0€
Total	**869.00€**		**144.82€**

*The cost of the merlin and thunder clusters is an estimation because they are not publicly available.

Table 3 Hardware budget

Everything has been purchased at the beginning of the project, however, the Jetson-TX, Merlin and Thunder clusters are property of the BSC (Barcelona Supercomputing Center) and they are part of the Mont-Blanc project. It is important to remember that the access to these machines is not exclusive to the project and the arising indirect costs of using them will be only a small part of the total power consumption of the machine. The Jetson TX1 boards are part of a donation of the CUDA center of excellence located in the Barcelona Supercomputing Center (BSC) in association with Universitat Politècnica de Catalunya (UPC), hence it will not affect to the project budget.

5.1.3 Human resources

In this section, we are going to estimate the costs of the human resources that are needed in this project.

Role	Price per hour	Hours	Cost
Analyst / Advisor	24€	85	2040.00€
Analyst / Programmer	12€	520	6240.00€
Total			**8280.00€**

Table 4 Human resources budget

The programmer will work during all the project development (488 hours). The required analyst / advisor hours are calculated with the weekly meetings in mind and counting the extra meetings for the analysis and interpretation of the benchmarks execution results.

5.1.4 Indirect costs

This section includes all the indirect costs that are related to the development of the project. We consider as an indirect cost, any cost that is not related to a specific resource or task.

Product	Price	Units	Cost
Electricity	10.00€	5	50.00€
Internet	60.00€	5	300.00€
Total			**350.00€**

Table 5 Indirect costs

The electricity cost is estimated through the average power consumption of the PC in a month multiplied by the number of months we are going to use it. The internet cost is the price of the Movistar Fusion rate per month.

5.1.5 Total budget

Once we have calculated all the direct and indirect costs that are related to the project, we are going to describe the total cost of the project:

The contingency has been calculated as the 5% of the total cost (indirect cost, hardware, software and human resources). This will be used for any possible budget deviation that could occur during the project development. The taxes have been calculated as the 21% and they are already included in the final budget estimation.

Task	Hardware	Software	Human	Total
Project planning	973.44€	135.78€	1 293.75€	2 402.97€
Initial system setup	64.90€	9.05€	86.25€	160.20€
Getting used with the development environment	194.69€	27.16€	258.75€	480.59€
Analysis of the sequential execution	129.79€	18.10€	172.50€	320.40€
Analysis of the parallel execution	648.96€	90.52€	862.50€	601.98 €
Preliminary evaluation (programming models)	519.17€	72.42€	690.00€	1 281.58€
Adaptation and tuning of the benchmarks	2 595.83€	362.08€	3 450.00€	6 407.92€
Final task	1 103.23€	153.89€	1 466.25€	2 723.36€
Total	**6 230.00€**	**869.00€**	**8 280.00€**	**15 379.00€**

Table 6 Cost of each task

Concept	Cost
Direct cost	15.379.00€
Indirect cost	350.00€
Contingency (5%)	786.45€
Total (21% TAX incl.)	**19 983.69€**

Table 7 Total budget

5.2 Budget control

In this section we will explain the control mechanism we are going to follow in order to prevent the deviations. Although we have already explained how the deviations are going to be solved (section 3.5 Action plan), we also have to design a mechanism that lets us to quantify the budget deviation and its real cost.

Our budget control mechanism have been designed with the Gantt chart in mind. The potential tasks that may suffer a deviation are: “Preliminary evaluation with different programming models”, “Analysis of the parallel execution” and “Adaptation and tuning of the benchmarks”. After completing each of these tasks, the real number of hours will be quantified and compared to the expected hours. If the difference is noticeable, before starting the next task, we will analyze the situation in order to find the cause of it. This will help us to face better the next tasks and prevent future deviations.

5.3 Sustainability

The sustainability of this project will be discussed through three points of view: economic, social and environmental.

5.3.1 Environmental sustainability

Our project requires the use of a computer during all the development, however, the Mont-Blanc mini clusters will be running 24/7 (24 hours a day, 7 days a week).

The main resource that has a negative impact to the environment is the electricity. The laptop, the jetson-tx cluster, the merlin cluster and the thunder cluster will affect to the environment by just using them. However, it will also depend in how the used electricity is generated (renewable energy).

The power consumption of our laptop computer will be negligible compared to the merlin or thunder cluster, however, we have to keep in mind that these machines are used by other developers too, so only a part of the consumed energy by the clusters will be related to our project.

Although the clusters are running 24/7 even when there is no work to be done (like usually supercomputers do), the Mont-Blanc project will provide Exascale performance using up to 10 times less energy than using the well-known x86 architecture, found in most of the nowadays supercomputers.

The idea behind developing an architecture-aware software limits our possibilities of reuse parts of existing projects.

Although the resulting source code of our project will not be reusable at 100% for other projects, the different optimization techniques could be used as a reference point for optimizing applications to the ARMv8 architecture.

This project is going to be awarded an 8 in the environmental sustainability area, despite the ARM based machines has a negative impact to the environment, they can potentially consume less energy than the average supercomputers.

5.3.2 Economic sustainability

The cost of our project is basically the one stated in the section 4.1 Budget estimation, since we aim to develop an optimization for a program, the maintenance costs are zero. However, any future update with improved performance will increase the development cost.

The only way to do a similar project with a lower cost would be reducing the dedicated hours to develop, however, this will require a more experimented developer and its salary will be also higher.

Both hardware and software resources are clearly justified in the planning section, so the money spent in this resources cannot be less. Most of the core software used in this project are free and open source, however, the cost of the hardware cannot be less because it is fixed by the manufacturer.

Nowadays, the average cost for building a supercomputer is 100 – 250 million of dollars excluding the maintenance, energy and cooling costs. However, the price of an ARM processor is 50 times cheaper than the processors used in the above mentioned supercomputers. This price makes them more accessible to anyone, and because their power-efficiency, the energy and cooling cost are also lower.

This project is going to be awarded a 7 in the economic sustainability area, since most of the software tools are free and there is no maintenance, and despite the Mont-Blanc clusters are expensive, its use will not be exclusive to this project.

5.3.3 Social sustainability

This project aims to support the development of a high performance computing model using mobile technology. The supercomputers play an important role in the field of computational science. A more affordable and energy-aware supercomputer will improve our lives by contributing to new discoveries in the medicine field, science field, environmental causes, etc.

Although the main beneficiary will be the Mont-Blanc project, the analysis of the executions and the applied techniques may be useful for other developers who also want to develop for the ARM 64 bits architecture.

Working in this project is going to help me to learn more about this new type of architectures, which presence is growing up day to day in the everyday things (IoT, smartphones, tablets, smartwatches ...).

The development of a supercomputer that uses an ARM architecture will not have a negative impact in the research field, the opposite, will push further the nowadays research

limits and it also will make the supercomputers a more accessible resource.

This project is going to be awarded a 7 in the social sustainability area, since it will benefit the scientist community by reducing the computational costs related to the research, however, it will not improve the people life quality directly but indirectly.

5.3.4 Sustainability matrix

	Production project	Useful life	Risks
Environmental	Resources analysis	Ecological footprint	Environmental risks
	8:10	15:20	0:0
Economical	Resources cost	Economic viability	Economical risks
	7:10	15:20	0:0
Social	Scientist community	Social impact	Social risks
	7:10	12:20	0:0
Partial assessment	22:30	42:60	0:0
Total	**64:90**		

Table 8 Analysis of the sustainability of the project through the sustainability matrix

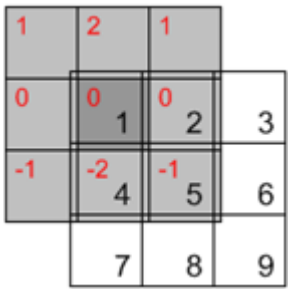
Chapter 6

Mont-Blanc Benchmarks

6.1 2D convolution

The convolution is commonly used in signal processing techniques. It is usually defined as a mathematical operation that combines two signals to produce a third one defined by an impulse response system.

The 2D convolution is similar to the convolution but also convolves both horizontal and vertical directions in the 2D spatial domain.



$$y[0,0] = x[-1,-1]\cdot h[1,1] + x[0,-1]\cdot h[0,1] + x[1,-1]\cdot h[-1,1]$$
$$+ x[-1,0]\cdot h[1,0] + x[0,0]\cdot h[0,0] + x[1,0]\cdot h[-1,0]$$
$$+ x[-1,1]\cdot h[1,-1] + x[0,1]\cdot h[0,-1] + x[1,1]\cdot h[-1,-1]$$
$$= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 0\cdot 0 + 1\cdot 0 + 2\cdot 0 + 0\cdot (-1) + 4\cdot (-2) + 5\cdot (-1) = -13$$

Figure 3 2D convolution calculation for the coordinate (0,0) with a 3x3 kernel

The benchmark takes an input matrix and generates and output matrix of the same dimension. Each point of the output matrix is calculated as a linear combination of the neighboring points to the point (and itself) with the same coordinates in the input matrix.

6.2 3D Stencil

The stencil codes are commonly found in the context of scientific and engineering applications for computer simulations like computation fluid dynamics. They iterate through 2 or 3-dimensional regular grids applying a kernel to each cell.

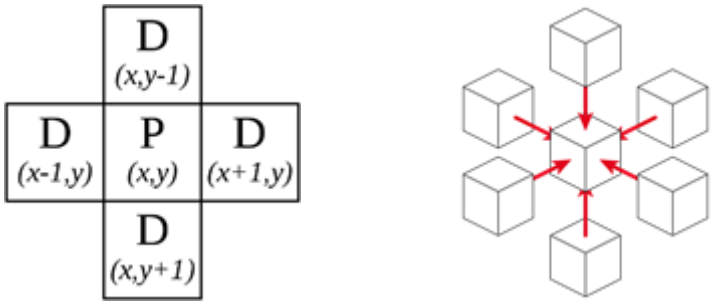


Figure 4 Example of a 5-point 2D stencil (left) and a 6-point 3D stencil (right)

The benchmark takes a 3D volume and produces a 3D volume output of the same dimension. Each output point is a linear combination of the point with the same coordinates in the input volume and neighboring points to this same coordinates but in an N plus/minus 1 dimension, where N is the actual dimension of the input point.

6.3 Atomic Monte Carlo

Markov chain Monte Carlo (MCMC) methods are a class of algorithms commonly used for sampling from a probability distribution. This method it is based in constructing a Markov chain (Figure 5) which is a random process where the probability that an event will occur depends only of the current state [14].

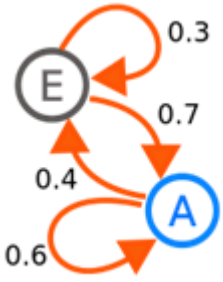


Figure 5 A two–state Markov chain

The benchmark performs a number of independent simulations using the Markov chain Monte Carlo method. The application will apply a random chosen displacements to a ran-

domly selected atoms which are accepted or rejected. This benchmark has been implemented following an embarrassingly parallel[8] strategy.

6.4 Dense matrix multiplication

The matrix multiplication is a very common operation in many numerical algorithms. The matrix multiplication has multiple applications, for example in the scientific computing field, pattern recognition or counting the paths through a graph.

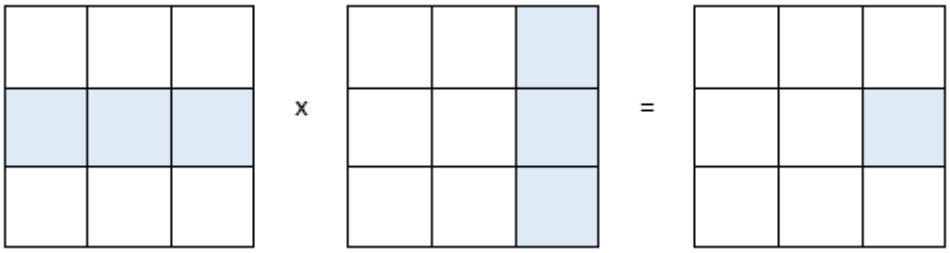


Figure 6 Basic matrix multiplication algorithm

A matrix is called dense when most of its elements are non-zero. The benchmark takes two square dense matrices and generates a third one. Each of the points from the output matrix is the dot product between every row of the first matrix against every column of the second matrix (Figure 6).

6.5 Fast Fourier Transform

In mathematics, the discrete Fourier transform (DFT) converts a signal from its original domain (time or space) into a representation of that same signal but in the frequency domain and vice versa. The fast Fourier transform (FFT) rapidly solves the DFT by factorizing the matrix into a product of sparse factors (mostly zeros). This algorithm reduces the original complexity from $O(n^2)$ to $O(nlogn)$, where n is the data size.

The benchmark takes one input vector and computes its one dimensional Fast Fourier Transform. The output is a vector with the same dimension as the input one.

[8] In parallel computing, an embarrassingly parallel workload or problem (also called perfectly parallel or pleasingly parallel) is one where little or no effort is needed to separate the problem into a number of parallel tasks. This is often the case when there is little or no dependency or need for communication between those parallel tasks. - *Wikipedia*

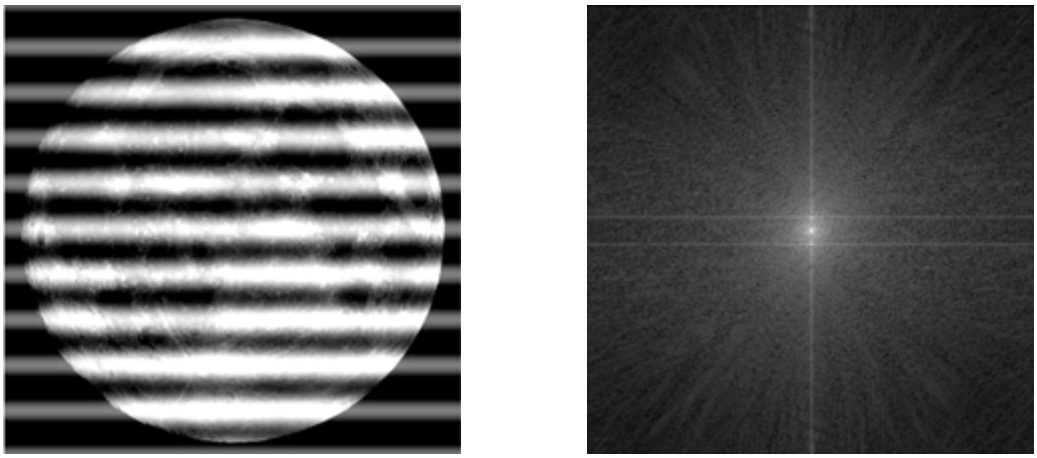


Figure 7 Image with a periodic noise (left) and its frequency domain representation in a spectrum (right)

6.6 Histogram

A histogram is the graphical representation of numerical data classified in bins (intervals). The bins are defined as consecutive and non-overlapping intervals of a variable that also must be adjacent and usually equal size.

The histogram has multiple applications in image processing for adjusting the brightness, the contrast, extract information or equalize an image. This techniques are widely used in the computer vision field.

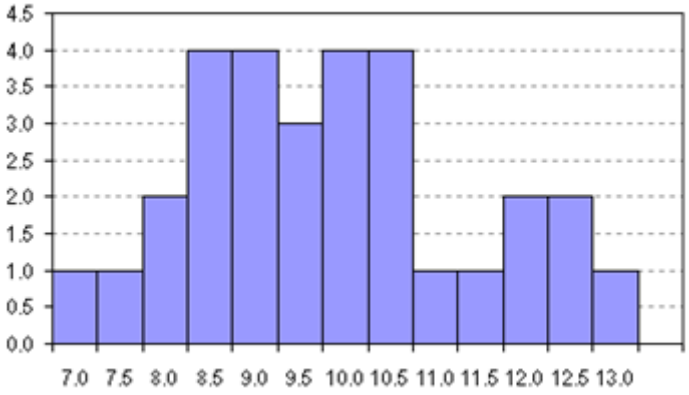


Figure 8 Graphical representation of a histogram

The benchmark takes an input vector and divides the entire range of values into a series of bins and then count how many values from the input vector fall into each bin.

6.7 Merge sort

The merge sort is a sorting algorithm known by its general-purpose and efficiency. The merge sort is considered a “divide and conquer” algorithm due his strategy of recursively breaking the problem into two or more sub-problems of the same type.

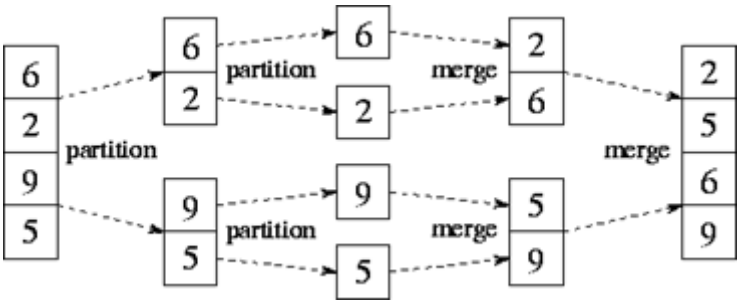


Figure 9 Graphical representation of a histogram

The benchmark takes an input vector and produces a sorted output vector using a recursive divide and conquer strategy.

6.8 N-body

The N-body is usually used in physics and astronomy to simulate the behavior of a set of particles when a physical force is applied on them, for example the gravity.

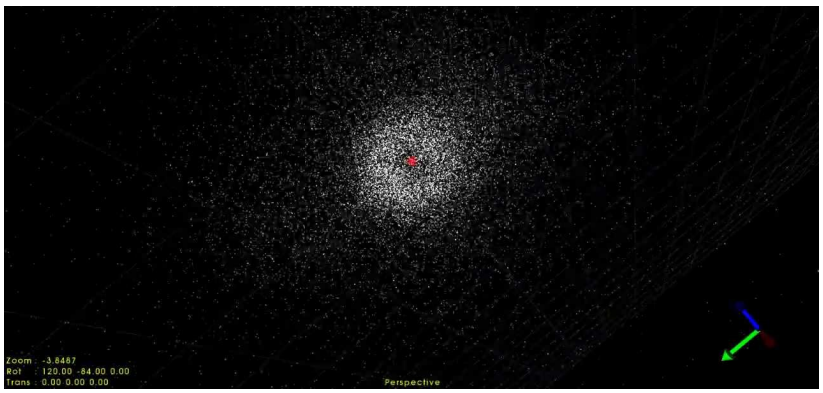


Figure 10 An N-Body simulation of two globular clusters in the Milky Way

These simulations are essential for many investigations that aims to understand the evolution of large-scale structures from the universe or the dynamical evolution of star clusters[9].

The benchmark takes a list of bodies with its characteristics like the position, mass and initial velocity and returns the new values of these parameters after simulating the gravitational interference between each body in a period of time.

6.9 Reduction

The reduction is a simple operation that can be found in image processing techniques related to computer vision. As the name states, it reduces a set of values into a single value of the same type by applying a specific operation between all the values.

The benchmark takes an input vector and applies the addition operator to produce a single output value of the same type.

6.10 Vector operation

The vectors operations are very common in a wide range of applications, for example in the multimedia field for applying a filter in the images, processing audio, etc. Most of the modern processors have dedicated vector units specialized for this kind of computation.

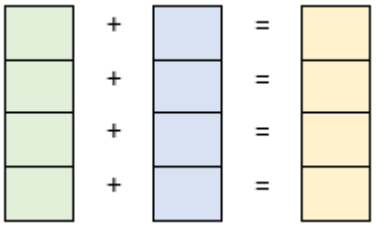


Figure 11 Addition of two vectors

The benchmark takes two vectors of the same size and produces an output vector of the same dimension by performing the addition of each i-element from the first vector with the i-element of the second vector.

[9] A star cluster is a group of stars that remains together due the gravitational force or because they appeared near, however these last ones are disrupted over time by the influence of external forces. - *Wikipedia*

Chapter 7

Preliminary evaluation with different programming models

The Mont-Blanc benchmarks are available in different variants (OpenMP, OmpSs, CUDA, OpenCL...). In this project, we are going to execute them in an intra-node configuration, using a single (X-Gene and Jetson-TX1) and dual socket (ThunderX) configuration and focusing in the shared memory implementations (OpenMP or OmpSs).

Before starting the optimization, we have decided that it will be interesting to compare the default performance of the benchmarks using OpenMP or OmpSs. Also, because the architecture we are working on supports 64 bits, we will also compare the performance between using single precision (float) and double precision (double) values.

7.1 Thunder

The Figure 12 uses the Serial-Double performance as the starting point and compares the speedup with all the other configurations. We can see that the single precision version doubles the performance of the double precision version using the same runtime. This behavior is related to the fact that using double precision variables increases by two the number of bits to process and also increases the chances to get a miss in the cache.

The overall performance is better when we use OmpSs instead of OpenMP. Even though the two programming models uses a similar parallel approach, OmpSs has been more optimized and takes more advantage of the architecture where is running.

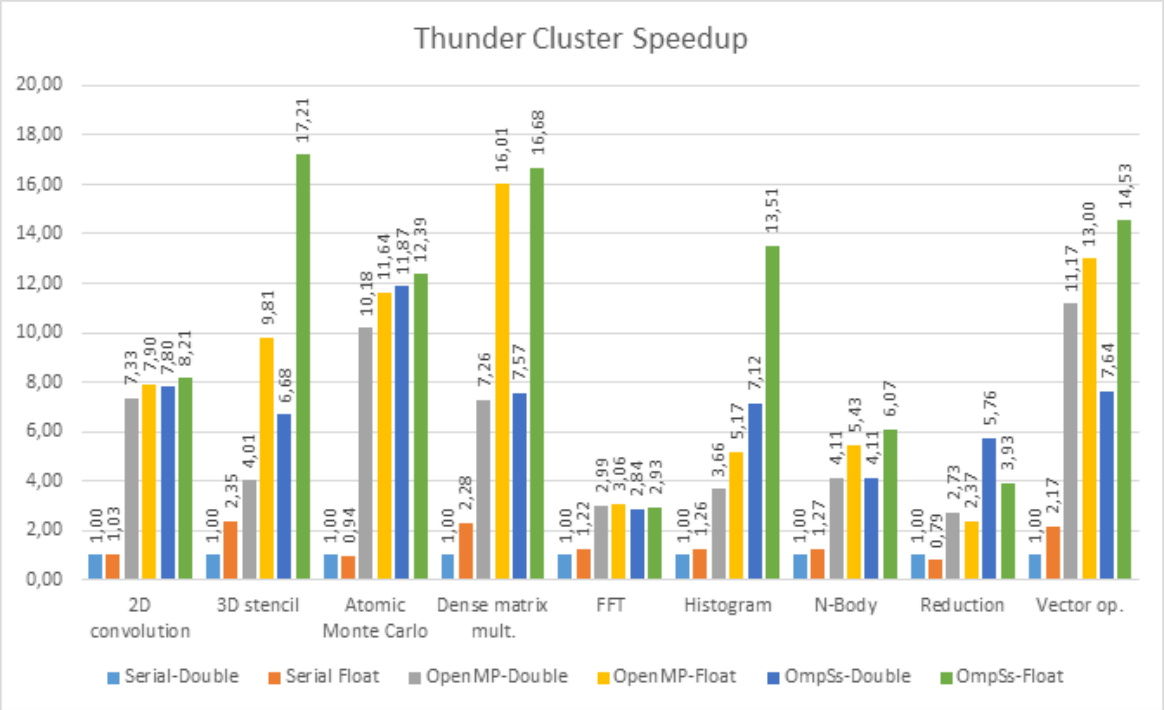


Figure 12 Speed-up chart using double/float and comparing serial versus CUDA, OpenMP and OmpSs configurations in a Jetson TX1 board

7.2 Merlin

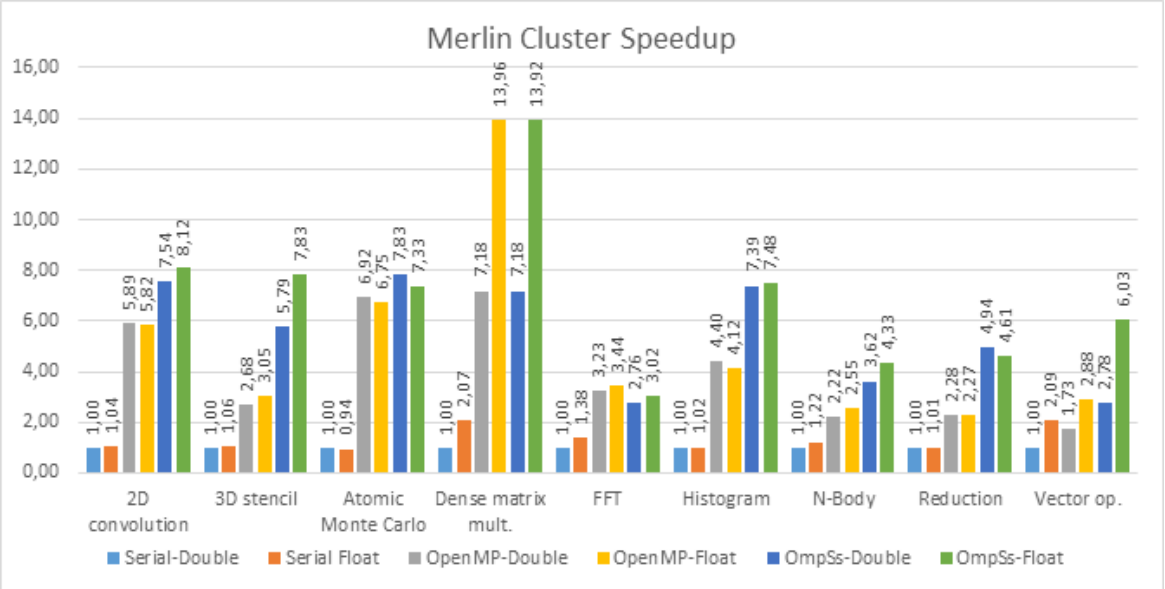


Figure 13 Speed-up chart using double/float and comparing serial versus OpenMP and OmpSs configurations in thunder cluster using 8 cores

The Figure 13 uses the Serial-Double performance as the starting point and compares the speedup with all the other configurations. Similarly to the thunder speed-up chart (Figure 12), the performance using single precision is better than using double precision, however, the difference is not such high as in the thunder machines.

7.3 Jetson-TX

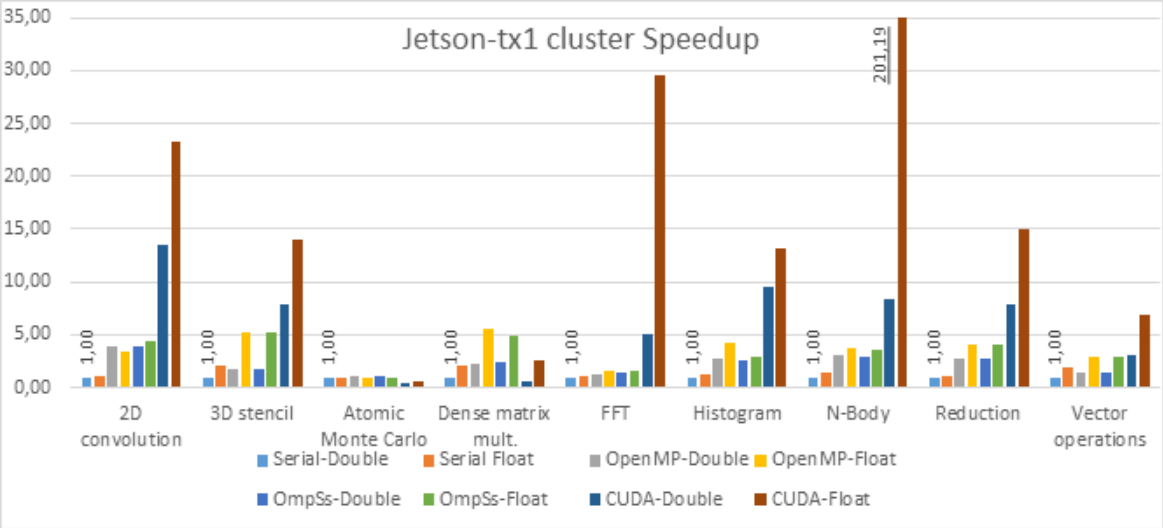


Figure 14 Speed-up chart using double/float and comparing serial versus CUDA, OpenMP and OmpSs configurations in a Jetson TX1 boards

The Figure 14 shows a speed-up chart using serial-double as the starting point, but in this case, we are also comparing the performance of the embedded Maxwell GPU (CUDA). Similarly to the merlin speed-up chart (Figure 13), the single precision version performs slightly better than the double precision (both CPU and GPU), however, the double versus single precision difference is much more accentuated in the CUDA version due the own GPU architecture (Maxwell) limitations. The ratio of a single precision versus double precision operation in the GPU architecture built in the Jetson TX1 GPU is about 1:32, this means that a double precision operation takes the same time as executing 32 operations in single precision [15].

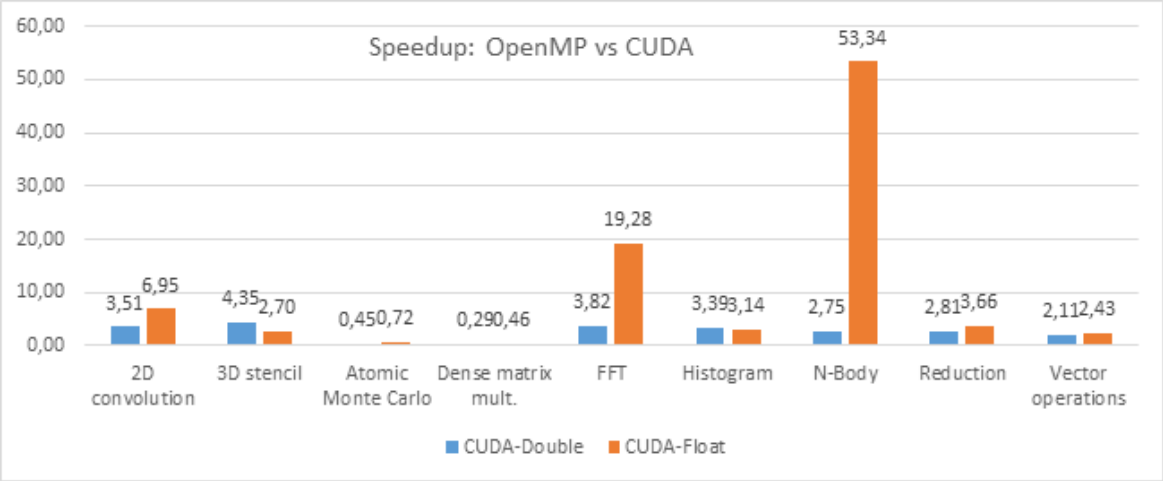


Figure 15 Speedup comparing the performance of the OpenMP version executed in the cortex-A57 versus the CUDA version executed in the embedded GPU

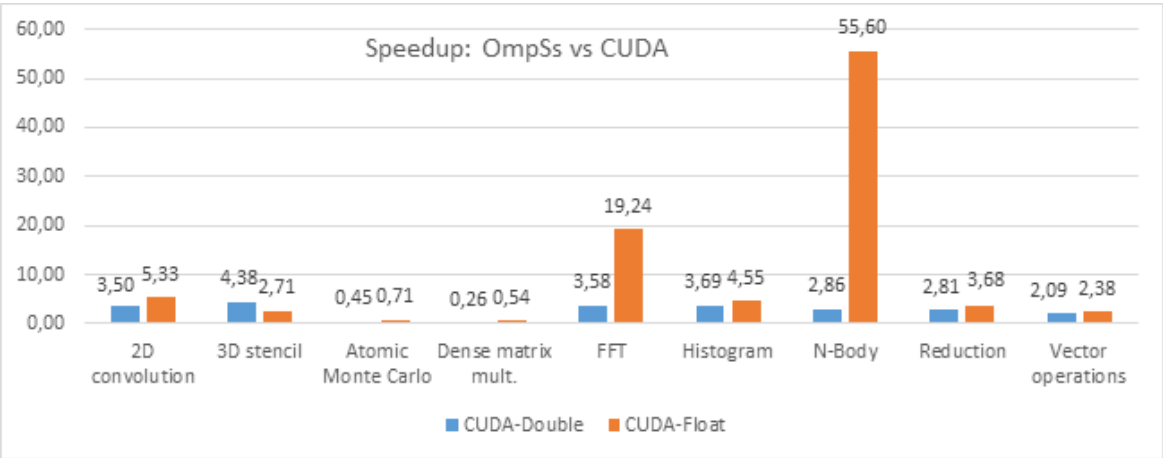


Figure 16 Speedup comparing the performance of the OmpSs version executed in the cortex-A57 versus the CUDA version executed in the embedded GPU

Chapter 8

Trace analysis and evaluation

In high performance computing, the "trace" term refers to a file that it is generated during the execution and which contains information regarding the application performance. The gathered information is used in a post-mortem analysis to find bottlenecks and areas that could be potentially improved.

8.1 Extrae and Paraver

In order to generate the traces in our benchmarks, we are going to use the Extrae tool. Extrae not only gathers information about when and which calls to the runtime are done but also the communication events between threads.

Paraver is an Extrae trace visualizer that offers a great flexibility to explore the gathered information during the execution. This tool helps not only to detect performance problems but to understand the applications' behavior.

Both applications have been developed inside the BSC and they are open-source tools that are widely used in order to support developers on the evaluation, tuning and optimization of their applications.

8.2 Setting up benchmarks for Extrae

In our development environment, in order to generate a trace using OmpSs is necessary to include a special flag (- -instrumentation) during the compilation and when linking the

libraries. This flag will tell the compiler to compile our benchmark using Extrae.

Extrae is a highly customizable profiler tool. Through an XML file, it lets us to configure in a fine-grained way, all the events and system parameters that we require.

Before executing the benchmark, it is necessary to define the NX_INSTRUMENTATION environment variable as “extrae” so the runtime knows whether needs to run Extrae or not and the EXTRAE_CONFIG_FILE environment variable with the path to the XML configuration file.

8.3 Base benchmarks traces

In order to try to achieve the best performance, we will base our study in the best performing configuration, in our case, it has been the OmpSs with float variables configuration (chapter 7).

Also due time and resources limitations, only the thunder traces will be analyzed deeply, however the speed up obtained in the merlin cluster will be shown in the performance evaluation.

2D convolution

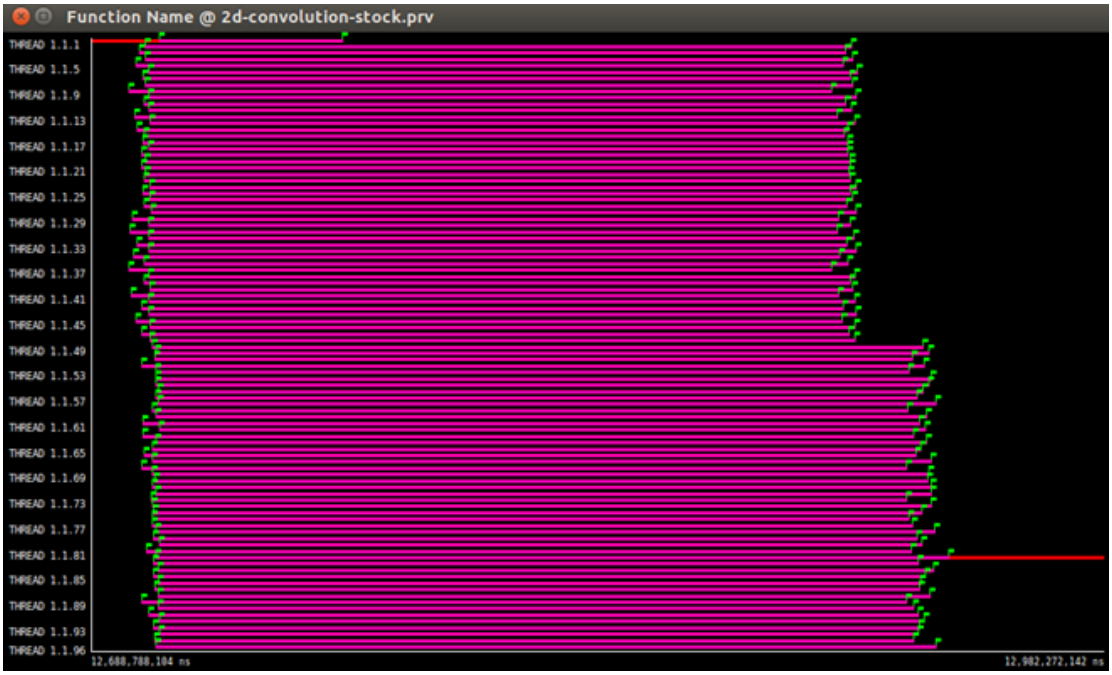


Figure 17 2D convolution base benchmark trace

3D stencil

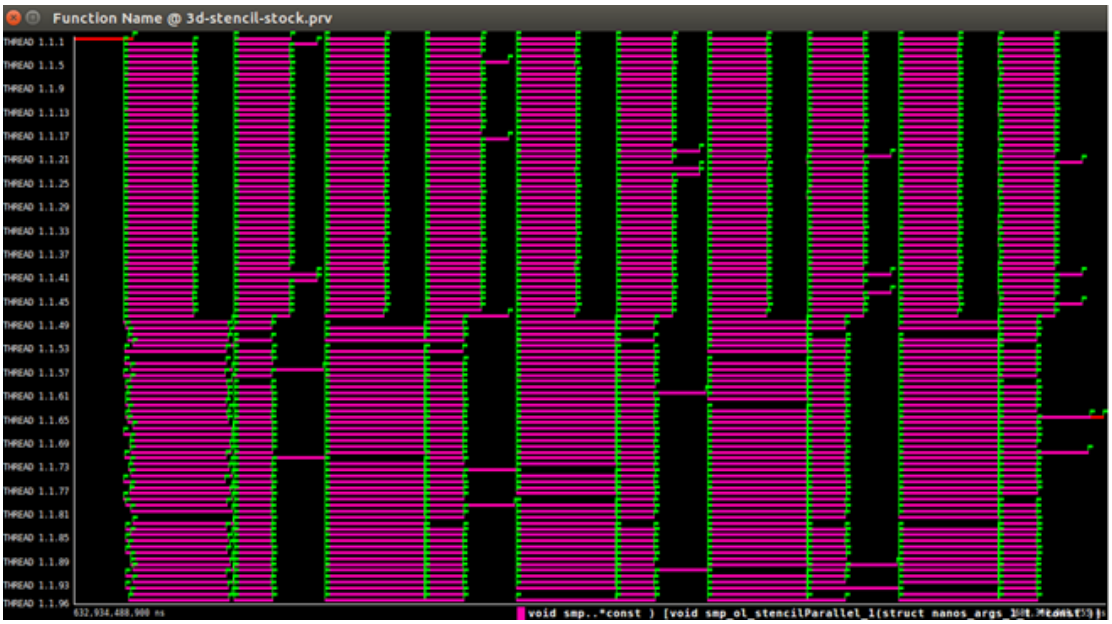


Figure 18 3D stencil base benchmark trace

Atomic Monte Carlo

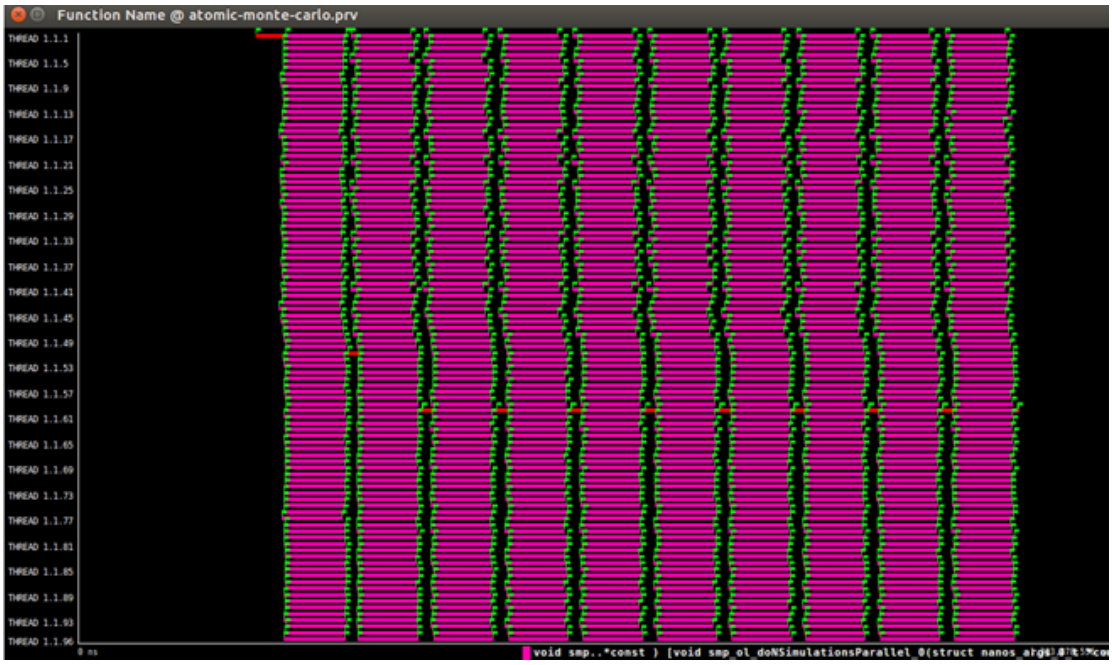


Figure 19 Atomic Monte Carlo base benchmark trace

Dense Matrix Multiplication

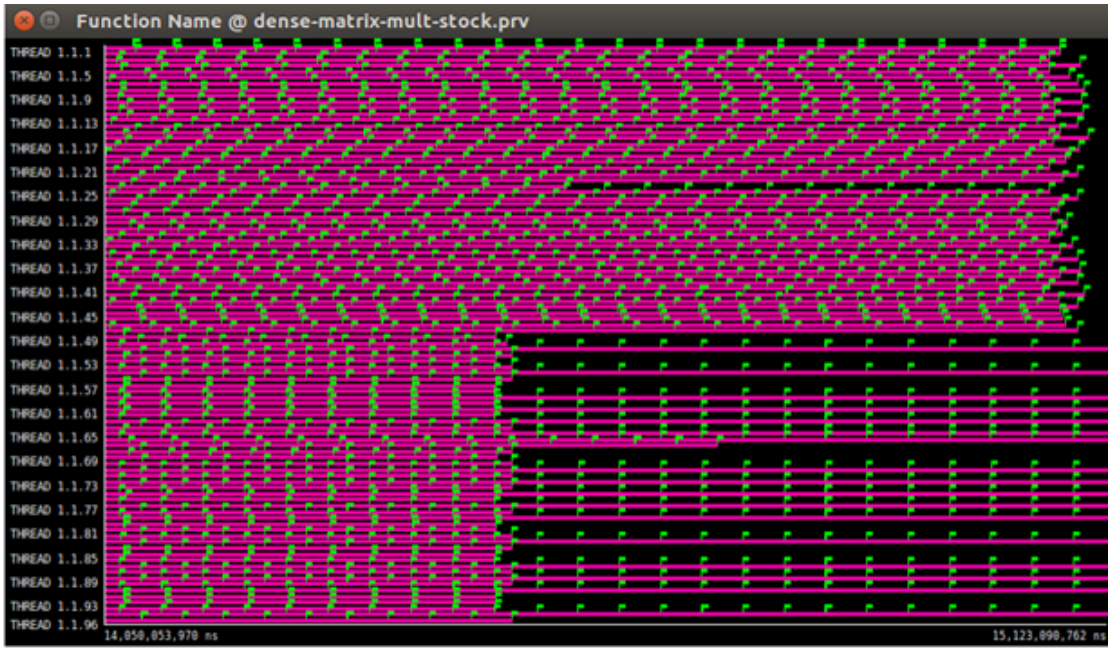


Figure 20 Dense Matrix Multiplication base benchmark trace

Fast Fourier Transform

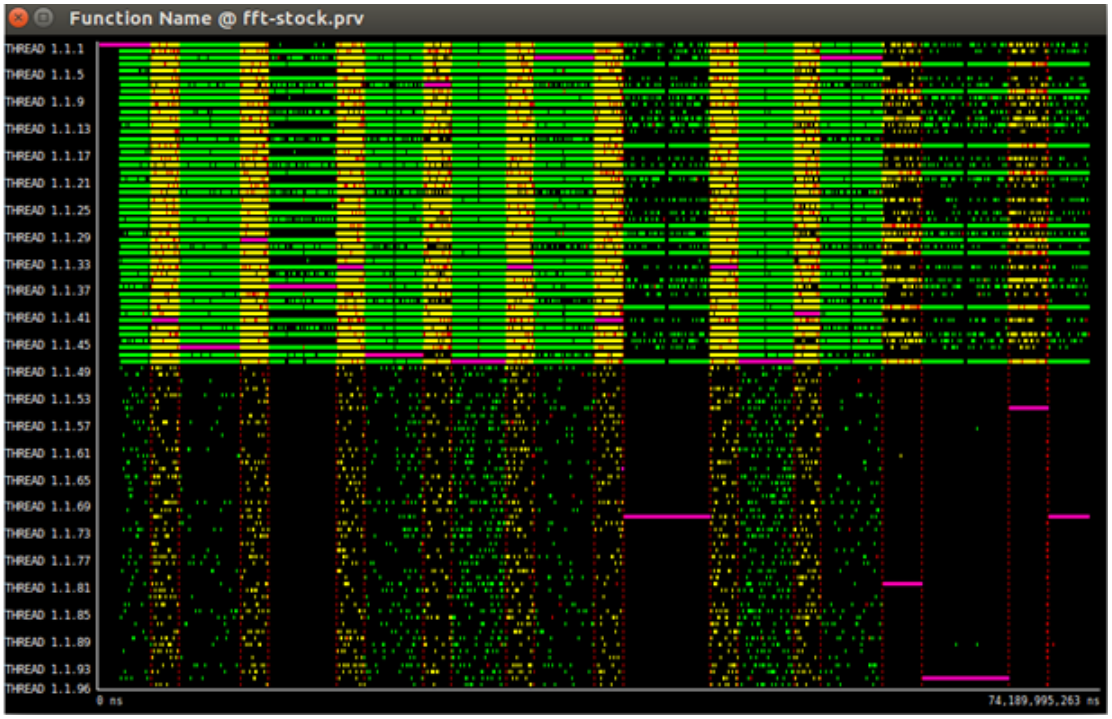


Figure 21 Fast Fourier Transform base benchmark trace

Histogram

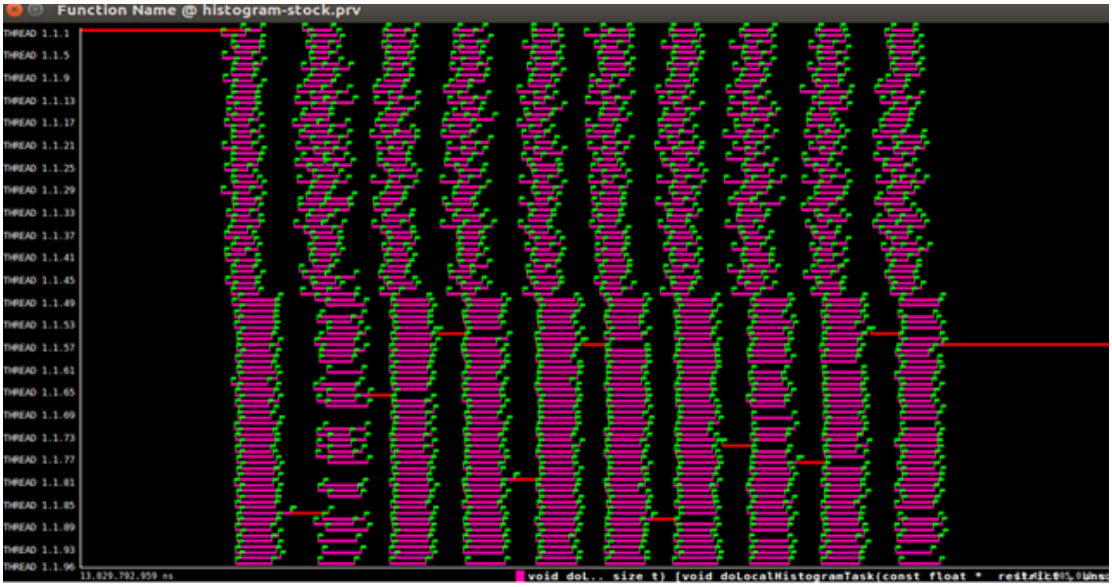


Figure 22 Histogram base benchmark trace

Merge sort

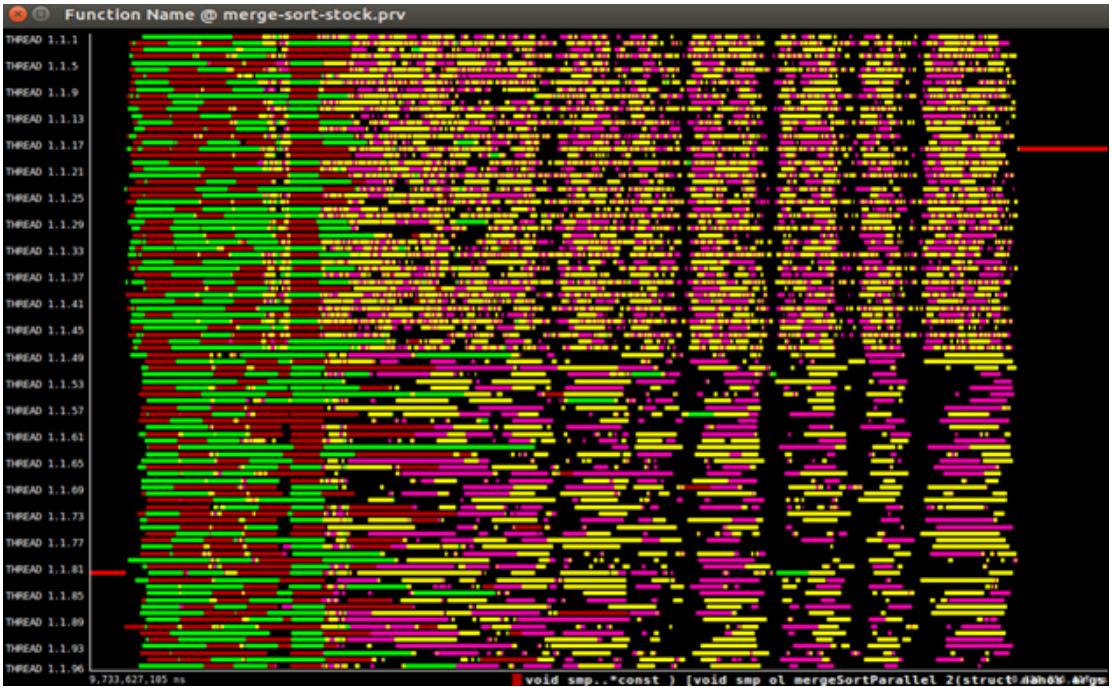


Figure 23 Merge sort base benchmark trace

N-body

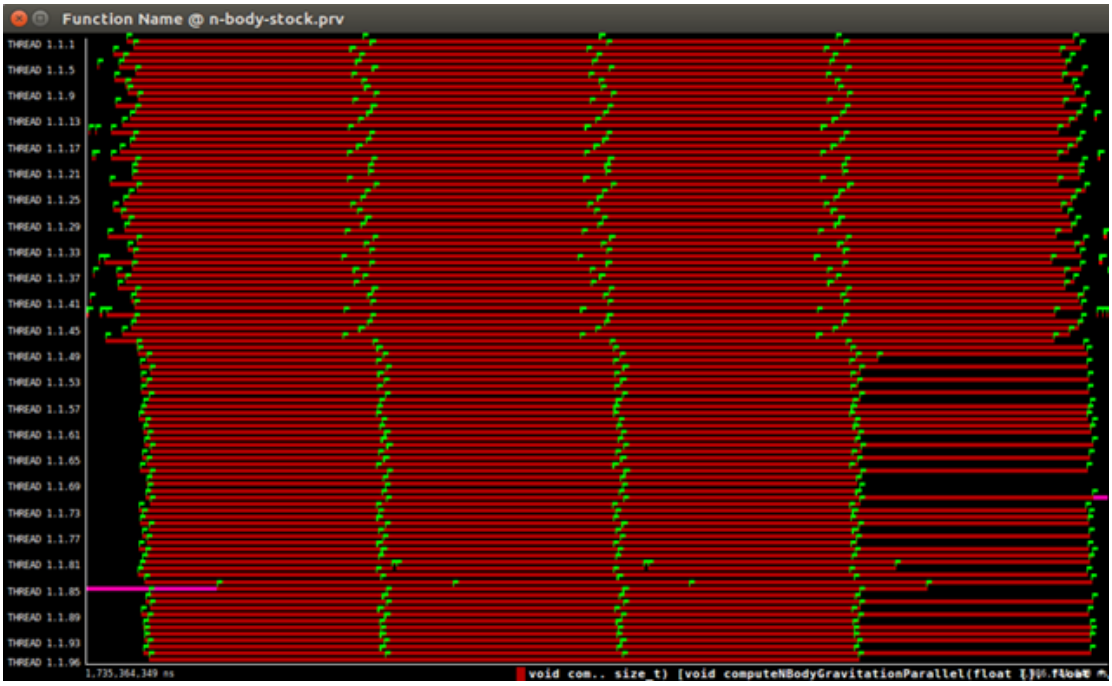


Figure 24 N-body base benchmark trace

Reduction

Figure 25 Reduction base benchmark trace

Vector operation

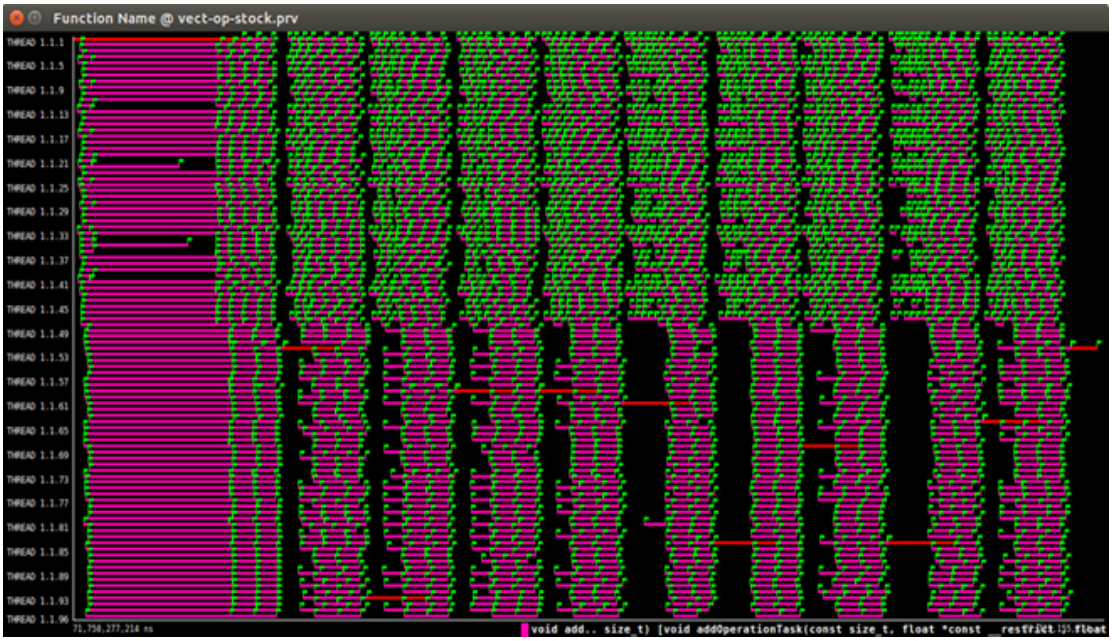


Figure 26 Vector operation base benchmark trace

8.4 Traces analysis

If we analyze the 2D convolution trace (Figure 17) or the one from histogram (Figure 22), we can see a similar behavior that is repeated during all the execution in most of the benchmarks: the tasks executed by the first half of threads always requires less time than the ones from the second half.

The Mont-Blanc benchmarks are parallelized following the task based model, where one thread creates all the tasks and puts them in a pool. The other threads will remain ready for taking tasks from that pool and execute them. At first, seems like the workload is not divided equally between the threads and the first half of threads have executed tasks with less work, but after doing more executions we couldn't see any of the first half of threads taking a larger task.

After a more exhaustive analysis, we found that in OmpSs, the threads are pinned to the physical cores (the thread 1 is executed in the core 1, the thread 2 in the core 2...). With this in mind, we have observed that all the tasks that are executed by threads from 1 to 48 are shorter than the ones that are executed by threads from 49 to 96. This behavior is clearly

related to NUMA (Non-Uniform Memory Access), where the 1-48 threads corresponds to the first socket and the 49-96 to the second socket (our machine is dual socket and uses two processors).

Also by using PAPI hardware counters, we found that some benchmarks are suffering of a high miss branch prediction, which translates into a constant flush of the instructions in the CPU pipeline. This happens because the CPU implements the instruction pipelining[10] optimization. This overhead is related to the fact that the benchmarks are constantly executing a loop.

[10]Instruction pipelining: CPU optimization that allows to process more than one instruction by performing multiple operations at the same time.

Chapter 9

Tuning the benchmarks

9.1 Architecture awareness optimizations

The main goal of this kind of optimizations is to reduce the execution time taking into account our computer architecture knowledge. This optimizations aims to reduce unnecessary overhead, exploit the architecture characteristics, the memory hierarchy, etc. In our case, we will focus on reducing the cost of the branches and using SIMD instructions.

9.1.1 Loop unrolling

Most of the nowadays CPU's, are pipelined superscalar processors. This kind of processors implements an instruction-level parallelism within a single processor and starts a new instruction before ending the previous one. This is achieved by breaking the instructions in stages and dispatching multiple instructions to different execution units on the processor [16].

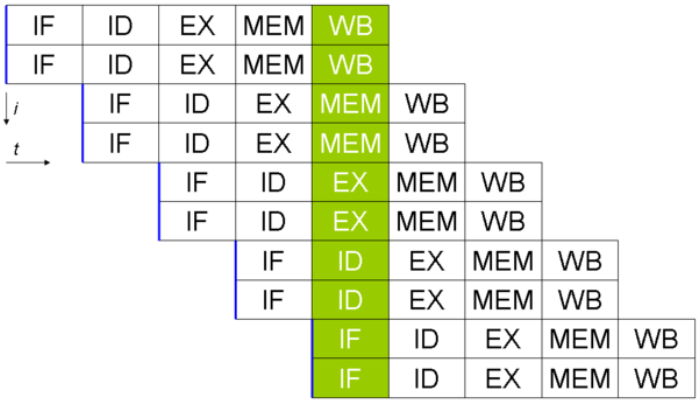


Figure 27 Simple superscalar pipeline

When we execute a program, the expected behavior is that the CPU pipeline is filled with consecutive instructions, but in a loop, the same instructions will be executed multiple times. With this in mind, if we take a branch, all the next consecutive instructions that are already running in the pipeline must be deleted. Fortunately, most modern CPU's implements a branch prediction mechanism that will try to guess if the implicit sequentially execution is going to be broken (execute the loop instructions again) or not (execute the next sequential instructions), however the branch predictor is not working very well in some of our benchmarks, so optimizing this part will give a high improvement.

In order to reduce the overhead related to the branch miss prediction, we will "unroll" de loop. This is done by doing more instructions in a single iteration. Therefore the new loop will execute the equivalent to "X" iterations of the original loop in a single iteration.

```
for(i=0; i<1000; i++) {
        a[i] = b[i] + c[i];
}
```

```
for(i=0; i<1000; i+=4) {
        a[i] = b[i] + c[i];
        a[i+1] = b[i+1] + c[i+1];
        a[i+2] = b[i+2] + c[i+2];
        a[i+3] = b[i+3] + c[i+3];
}
```

Figure 28 Simple superscalar pipeline

The approach of this optimization is to reduce the overhead by decreasing the actual number of branches that will be done. In the case of the Figure 28, the overhead related to the branch miss prediction will be reduced by 4 times.

9.1.2 Vectorization

Vectorization refers to an architecture dependent instruction that applies the same operation to a whole array instead of individual elements. In our case, we will focus on SIMD operations (Single Instruction Multiple Data).

In SIMD instructions, usually there are two input vector registers and one output vector register. Each vector can be interpreted as a set of different number of elements. The size of each element is the size of the vector register divided by the number of elements (1, 2, 4, 8, 16, 32...).

When a CPU executes a SIMD instruction, in parallel, each i-element of the first input register is operated with the i-element of the second input register and the result is saved in the i-element of the output register. This operations are done within the time of a single

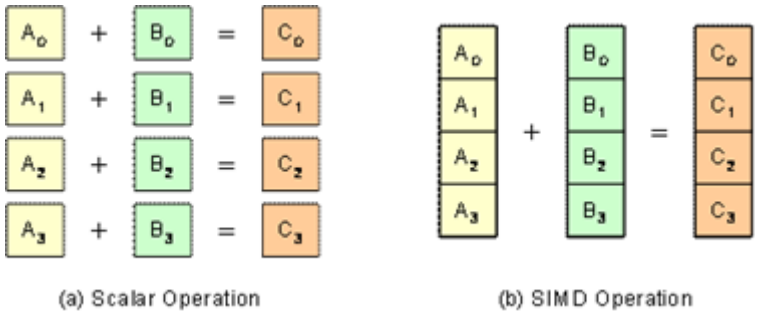


Figure 29 Comparison between a scalar and SIMD operation

operation, however there are always other considerations like the memory throughput that could affect the execution speed.

In the architecture AArch64 (ARM 64bit), there is a SIMD accelerator process integrated inside the chip called NEON (also known as ASIMD). There are instructions sets for both AArch32 (equivalent to the NEON instructions used in ARM 32bit) and AArch64 and both can be used. The AArch64 architecture supports up to 32 x 128 registers which are also shared with the VFP (Vector Floating Point) units. These registers are 128 bits length and supports up to 16 simultaneous operations on 8-bit values [17].

In order to enable NEON instructions there are 3 ways to do it:

- **Compiler flags:** include flags during the compilation letting the compiler to try to auto-vectorize the code.
- **NEON intrinsics:** Use macros that will compile into low level Neon operations.
- **Assembly Code:** Write the program in assembly or link it with highly optimized libraries.

9.1.3 Work done

First of all, before applying any of the above described optimization, we have analyzed the source code of the benchmarks in order understand the potential regions to be optimized.

As stated in the section 8.4 Traces analysis, most of the benchmarks internally executes a loop that can be easily unrolled due the fact that there are not dependencies between iterations. Also, in some benchmarks like the vector operation, we have seen a clear way to apply SIMD

instructions.

Because most of the benchmarks codes are already very clear, free of dependencies between iterations and easy to optimize, he have opted to let the compiler to auto-vectorize and auto-unroll our benchmarks. This has simplified our work, however in some benchmarks, we had apply the optimizations manually because the compiler was not applying the optimizations automatically. The compilations flags that we have used are the following ones:

Thunder	Merlin
-O3	-O3
-march=armv8-a+simd	-march=armv8-a+simd
-mcpu=thunderx+simd	-mcpu=xgene1+simd
-mtune=thunderx	-mtune=xgene1

The "-O3" flag is a general GCC optimization flag. When it is enabled, the compiler will attempt to improve the performance and/or code size at expense of increasing the compilation time. In this case, the "-O3" flag implicitly turns on the "-aggressive-loop-optimizations" (loop unrolling) and the "-ftree-vectorize" (auto-vectorization) [18].

The "-march" flag it is used to specify the target architecture we plan to execute the code and allows GCC to generate code that uses all the features of the specified architecture. In our case, the "armv8-a+simd" value enables support for the ARM 64bit architecture extensions with SIMD [19].

The "-mcpu" and "-mtune" flags specifies to GCC the name of the target ARM processor for which should tune the performance of the code.

9.2 NUMA (Non-Uniform Memory Access)

The Non-Uniform Memory Access is a computer memory design used in multiprocessor systems. In this kind of systems, the operative system applies an abstraction layer and the multiple processors are seen as a single processor unit by the end user, however, each processor haves their own local memory that it is also shared between the other processors. This approach implies that the memory access time depends on the memory location relative to the processor (the access to the local memory is faster than non-local memory).

For the programmer, the non-uniformity only increases the access time when accessing to a certain memory addresses since the multiple memories (of each NUMA node) are also seen as a single unit. The allocation of the physical memory is totally transparent to the programmer and is usually handled by the operative system.

9.2.1 Processor interconnection

In a multi-socket configuration, the processors are interconnected through a bus. When a processor requires data that it is mapped in a certain memory address, it will first look at L1 cache level, then in the L2 and above cache levels and local memory, and then in the non-local memory near the other processors. Each of these groups (processor, memory and possibly its own I/O channels) are called a NUMA node [20].

Nowadays, there is not a standard of how the processors should communicate between them. The communication protocol and the implementation of it differs between processor manufacturers.

In our cluster, we are running a dual socket configuration of the Cavium ThunderX processors. Each processor has 48 cores and has 64GB of local memory. Due the fact that ARM is targeted for running on embedded systems (where only one processor is needed) there is not an ARM inter-processor connection protocol but Cavium has developed and implemented the CCPI (Cavium Coherent Processor Interconnect) in their ThunderX processors family (Figure 30).

The CCPI interconnection protocol is also cache coherent, this means that all the processors will return the latest updated data from any cache in the system automatically. However, the Cavium implementation of their interconnection bus/protocol doesn't performs as good as we could expect.

In the benchmarks that are more memory bound[11], we can observe a performance difference up to a 30% between the threads that are accessing to their local memory and the threads that requires data allocated in another NUMA node. This behavior can be clearly seen in some of the previous traces (section 8.3), for example in the 2D-convolution trace (Figure 17) or the histogram trace (Figure 22).

[11] Memory bound: In computer science, memory bound refers to a situation in which the execution time of an application is primarily limited by the characteristics of the memory instead of the CPU performance.

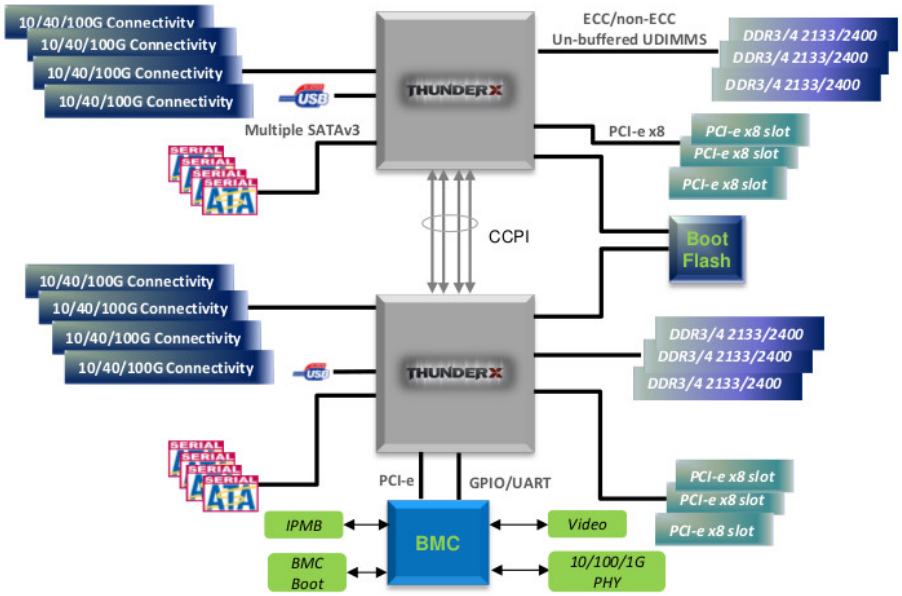


Figure 30 Cavium ThunderX dual socket configuration scheme using CCPI

9.2.2 Reducing the NUMA effect

In order to reduce the NUMA effect, there exists some general solutions, like OS kernels that includes features for optimizing allocation of memory or runtimes that include NUMA aware schedulers, however, the obtained performance will be potentially limited by the decisions they take during the execution time.

We have decided to optimize the benchmarks by taking advantage of some of the OmpSs features but without modifying the nature of the benchmark algorithm. The applied modifications will be explained in the next sections.

9.2.2.1 Input parallelization

As mentioned in the introduction of this section (8.2 NUMA), the allocation of the data in the memory is a task handled by the operative system. The default behavior when there is a page fault in our system, is to allocate memory in the NUMA node containing the page-faulting CPU in favor of reducing the memory access latency. Because the first CPU to “touch” (read / write from) a page of memory is also the CPU that faults the page, this memory allocation policy is called “first touch” [21].

In the Mont-Blanc benchmarks, we have found that the initialization of the data it is done in serial (only one thread initializes all the input data) and because the default policy is

first touch, all the input data will be allocated in the first NUMA node. Because this, when we execute the benchmarks using two processors (96 threads), the threads from the second NUMA node will always take more time to complete the work because they will always require to read/write data from non-local memory, which it is allocated in the first NUMA node.

To improve the performance in our benchmarks, due the fact that the memory allocation is handled automatically by the operative system and the default policy is first touch, we have decided to parallelize the initialization of the input data. By doing that, threads from the first and the second NUMA node will simultaneously initialize the input variables in order to not only allocate memory in the first NUMA node but also in the second NUMA node. This approach will help to reduce the memory access latency, however, the improvement will be as good as the distribution of the tasks. In the best case, the threads will take tasks that only require data allocated in their local memory while in the worst case, the performance will remain equal.

Figure 31 Traces of the vector operation benchmark. In the left, the default benchmark. In the right, the benchmark with parallel initialization.

If we analyze the Figure 31, this first step has not improved the performance that much. We still can see tasks that takes more time than others, but now, they are not only in the second half of threads but also in the first 48 threads (first half). The Figure 31 only shows the trace of the vector operation benchmark, however, the same behavior after parallelizing

the initialization can be seen in the other Mont-Blanc benchmarks.

The followed strategy to parallelize the initialization of the input has been to divide the input set in intervals and create 96 tasks that will initialize each interval. If the input set does not divide to 96, we have prioritized the work load balance and instead of assigning much more work to the last task, we have divided the remainder between the first tasks, so each thread will have to initialize only one more variable in the worst case.

	1-48 threads (NUMA node 1)	**49 – 96 threads (NUMA node 2)**
Short duration task	10,796,714 ns	10,651,019 ns
Long duration task	27,635,309 ns	24,030,352 ns

Table 9 Tasks duration of each case in the vector operation benchmark

In the Table 9, we can observe in a more precise way the tasks behavior after applying this technique in the vector operation benchmark (Figure 31). The short duration task field, represents the duration of the tasks that accesses to local memory, while the long duration task field is the duration when the tasks requires data located in another NUMA node. The difference between the short and the long tasks is about 20 milliseconds, which is 2.7 times slower.

The overall performance in the default version of the vector operation benchmark was 2.15 GFLOPS. After parallelize the initialization and distribute the allocation of the data between the NUMA nodes, the performance has increased to 2.7 GFLOPS. It is a good improvement however we are still suffering of non-local memory accesses in the threads.

9.2.2.2 NUMA node optimization

In the previous section, we have explained how we tried to improve the performance of the benchmarks by parallelizing the initialization of the input in order to increase the probabilities that a thread does not require to access to non-local memory for completing the task.

In this section, we are going to explain an optimization technique that goes one step further than the previous one by forcing the thread to only access data allocated in local memory. To achieve that, we are going to start from the already modified code of the previous section.

This optimization starts from the premise that OmpSs by default pines the threads to a physical core. This means that the thread 0 will be executing in the core 0, the thread 1 in the core 1 and so. Also in case of executing with more threads than physical cores, the threads will be distributed through the physical cores following a round-robin fashion[12].

Awareness of memory allocation

In order to ensure that each thread will only take tasks that requires data allocated in their local memory, we have to be aware about where the data is allocated during the initialization of the input. To achieve that, we started from the same objective presented in the previous section but with a different approach.

Usually when we parallelize a code with OmpSs or OpenMP, the task workload is assigned statically. This means that when the master-thread creates the tasks, they will already have assigned the number of iterations (in case of a loop) or the amount of code they will execute. Also the generated tasks are putted in a "pool" of tasks where the threads will pick them. In the previous section, the applied optimization technique follows the above explained strategy, however this approach makes impossible to know where the data is allocated because any thread can take any task from the pool, becoming impossible a priori to have a knowledge of where the data is allocated.

A potential solution could be to maintain a data structure with the mapping of memory address / NUMA node, but due the fact that we cannot decide which task will be picked by a thread, this workaround is not feasible.

In our benchmarks, we have created N tasks, where N is the number of available threads. The first instruction that each thread will execute it will be a barrier, in order to ensure that all the threads takes only one task. After the barrier, each thread will execute the getInterval() function (Figure 32) which returns an interval of data that has been calculated relative to the thread-id of the executing thread. This interval will be the one that the thread will initialize and in fact, because the thread runs pinned to a physical core it is easy to know which interval of data has been allocated in the first or in the second NUMA node.

[12]Round-robin: algorithm employed for distributing work between resources. In round-robin, the tasks are divided in equal slices of time and assigned to a resource who will execute them in circular order.

```
void getInterval(unsigned int numTasks, size_t n, size_t *startIndex, size_t *endIndex) {
  int id = omp_get_thread_num();          //get the thread-id
  int bsize = n/numTasks;                 //base block size
  int rest = n - (numTasks*bsize);        //rest of the division
  int z = id * bsize + ((id < rest) ? id : rest);  //start index
  *startIndex = z;
  *endIndex = z + bsize + (id < rest);    //end index
}
```

Figure 32 Code example of the followed strategy for assigning the work based in the thread-id

The speedup we obtained after applying this technique is similar to the one from the section 8.2.2.1 Input parallelization, however these modifications are necessary for the next step.

Tasks distribution in the NUMA nodes:

In this step, we are going to modify the body of the benchmark, which is all the code related to the calculus. Even though we are modifying the code, we will ensure that the nature of the benchmark algorithm will remain the same. This is important because we are working with floating point variables and the order of how the operations are done is important to avoid precision loss.

The followed strategy when modifying the benchmark body is similar to CUDA or OpenCL. In these programming models, all the running threads execute the same function (kernel) but the data interval that each thread will calculate is relative to their own global thread index. In our benchmarks, we have modified the body code to be like a kernel, where all the threads make the same calculus but the interval to be calculated by the thread is relative to their thread-id.

Parallel initialization	
0 : 31	#0
32 : 63	#1
64 : 95	#2
96 : 127	#3
128 : 159	#4
160 : 192	#5

Figure 33 Initialization of the data

Benchmark body	
0 : 31	#0
32 : 63	#1
64 : 95	#2
96 : 127	#3
128 : 159	#4
160 : 192	#5

Figure 34 Tasks distribution between threads

By taking advantage to the fact that each thread has initialized a data interval relative to his own thread-id (Figure 33) and the default memory allocation policy is first touch, if the same thread does the calculus related to this same interval (Figure 34), we can ensure that

the required data will be allocated in the same NUMA node.

This approach is feasible because a thread always has the same thread-id and executes in the same physical core. Also because each thread has already initialized a data interval relative to the thread-id, during the body of the benchmark, it is easy to know which data have been allocated in local memory.

Chapter 10

Performance analysis and evaluation

In this section we will discuss the performance we obtained after applying the optimization techniques explained in the previous section (chapter 9). The evaluation includes a detailed analysis of the performance in each machine configuration and the pros and cons of each modification.

10.1 Architecture awareness optimizations

In the Figure 35 and 36, we can observe the performance improvement after compiling the benchmarks with the optimization flags specified in the subsection 9.1.3 Work done. We can observe a great improvement in some benchmarks (reduction, 2D convolution...) and a very poor improvement in other ones (merge sort, Fast Fourier Transform or histogram). We can see a similar trend in both machines which demonstrates that some benchmarks are more benefited by the unrolling and general optimizations while the others are already limited by the own machine resources.

The main benefit of using this kind of optimization is how easy it is for a non-advanced user to optimize any application and the performance improvement you can get, however in order to achieve the best performance sometimes it requires to generate assembly code to double check if the compiler is applying the optimizations we desire and if not, then modify the code to help the compiler to see a clear way to apply the optimization or rewrite that part by applying the optimization by ourselves.

Due the fact that we are compiling the code with a target architecture, in order to achieve the best performance in another machine, we should recompile the code for the running

machine architecture, however the same code should not require any modification to run optimally in the new architecture.

Thunder cluster

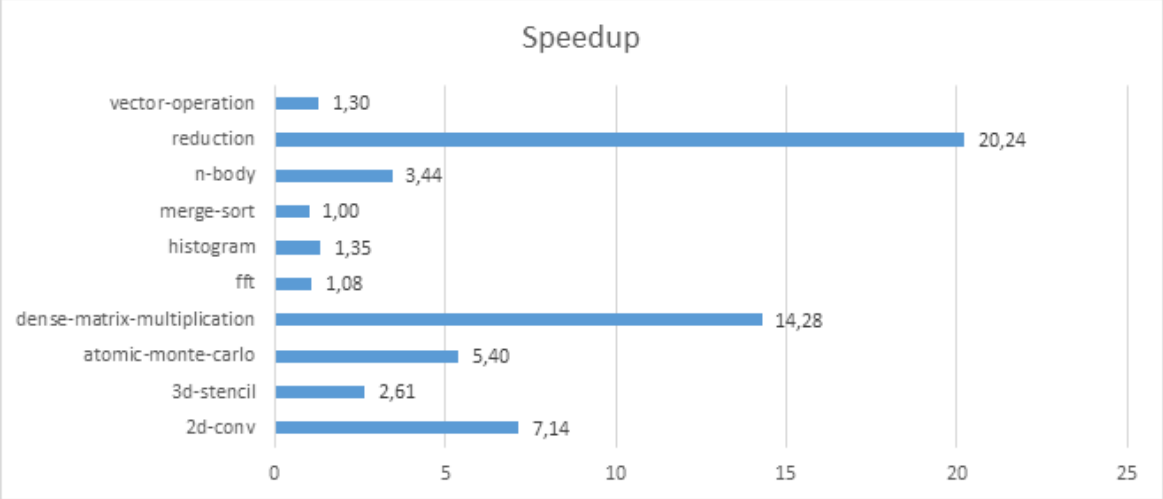


Figure 35 Speedup after applying the optimization flags in thunder machine

Merlin cluster

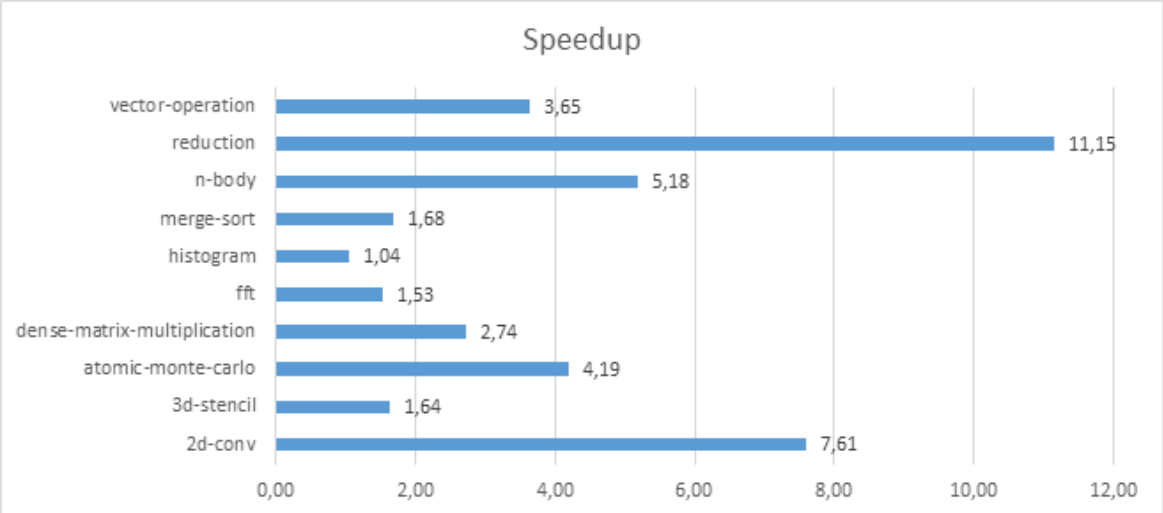


Figure 36 Speedup after applying the optimization flags in the merlin machine

10.2 NUMA optimization

Analyzing the Figure 37 and 38, we can clearly see a difference between the speedup we obtained in both machines, while in the thunder cluster we have improved the performance in almost every benchmark, in the merlin cluster we cannot say the same. This is because the optimization we have applied is oriented to multi-processors systems with a NUMA memory architecture and each merlin node runs a single octa-core CPU configuration.

Despite being an optimization focused for multi-socket systems, we have achieved a slightly improvement in some of the benchmarks, such like the vector operation. This is because with our modifications, the tasks benefit a lot more of the spatial locality in the cache, reducing the cache misses.

On the other hand, in the thunder cluster, even though there are some benchmarks that have been more benefited about this modifications than others, we can conclude that the NUMA optimization has improved the performance the way we wanted to.

Thunder cluster

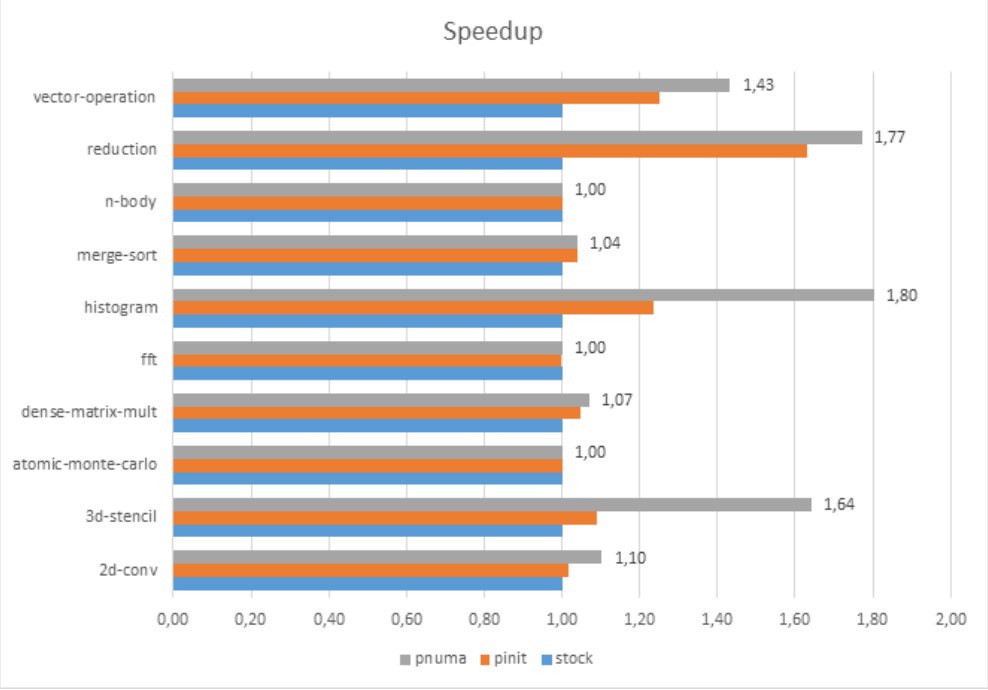


Figure 37 Speedup of the benchmarks after applying the NUMA optimizations on thunder cluster

Merlin cluster

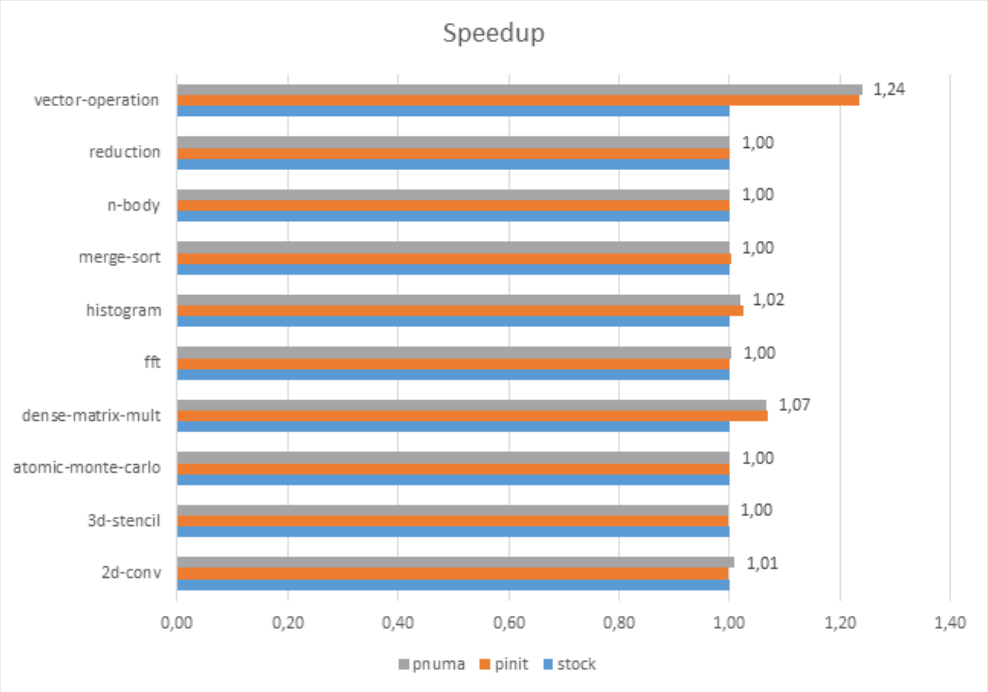


Figure 38 Speedup of the benchmarks after applying the NUMA optimizations on merlin cluster

If we analyze the execution trace for example of the 3D stencil (Figure 39) or the 2D convolution (Figure 40), we can observe the execution behavior before and after applying the NUMA optimizations. In the 3D convolution trace, we can see a larger task duration in the first tasks, but that's intended because in order to prioritize the tasks workload, if the problem size does not divide the number of threads, the first tasks will take the rest instead of assigning much more work to the last task.

Although the global speedup we obtained in the 3D stencil (1.64) and in the 2D convolution (1.10) is not very high, analyzing the traces we have reduced the overhead related to access non-local memory without including too much overhead in the code. Also in the Table 10 and 11, we can observe that after the NUMA optimization, the duration of the tasks that executes in the second NUMA node have been reduced to a similar duration as the first NUMA node tasks.

This behavior after applying the NUMA optimization can also be seen in the other benchmarks, but we are only showing the traces from the 3D stencil and the 2D convolution as a reference for the performance analysis.

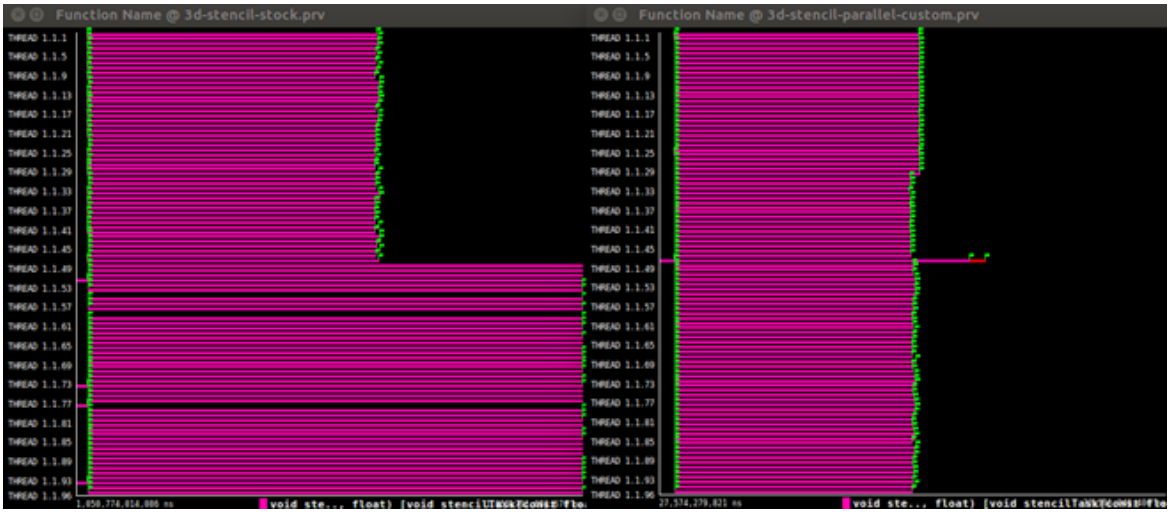


Figure 39 3D stencil execution traces. Stock code (left) versus NUMA optimized code (right)

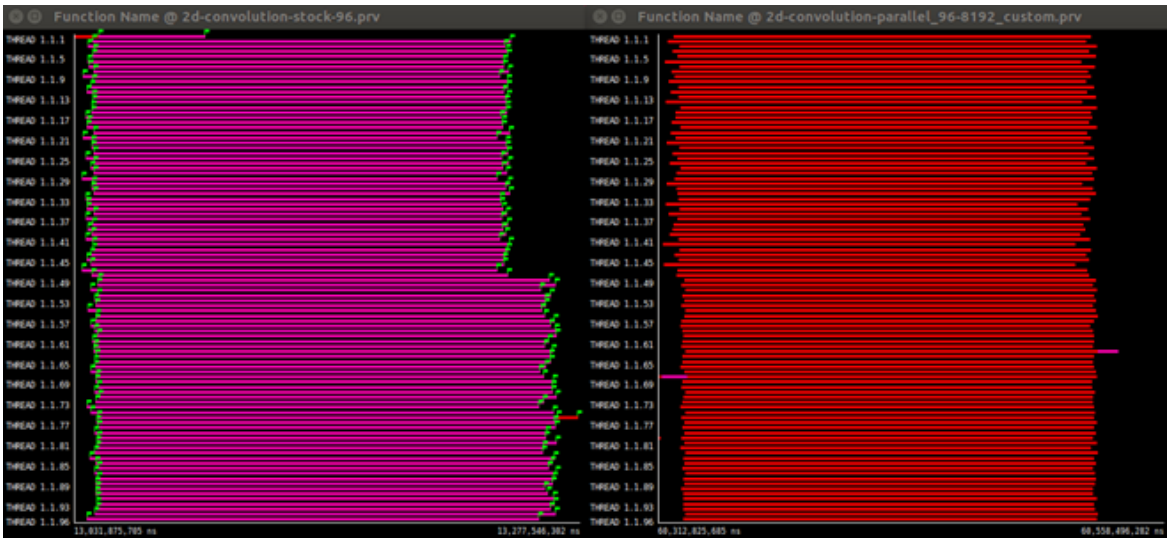


Figure 40 2D convolution traces. Stock code (left) versus NUMA optimized code (right)

2D convolution	**1-48 average tasks duration**	**48-96 average tasks duration**
Default	204,641,839 ns	223,067,128 ns
NUMA optimized	205,952,745 ns	206,803,958 ns

Table 10 2D convolution tasks durations before and after the NUMA optimization

3D Stencil	**1-48 average tasks duration**	**48-96 average tasks duration**
Default	3,364,418,245 ns	5,292,431,421 ns
NUMA optimized	2,344,731,526 ns	2,367,133,144 ns

Table 11 3D stencil tasks duration before and after the NUMA optimization

Also in the Figure 37 we can observe some benchmarks that after parallelizing the input, the performance improves a little and after applying the NUMA technique the performance improves a lot, but there are also some benchmarks (e.g. reduction) which improves a lot after parallelizing the input but not so much after the NUMA technique.

The expected behavior is to achieve a higher speedup when applying the NUMA optimization, but after analyzing the reduction benchmark trace (one of the benchmarks with this strange behavior), we have found that this is because during the task distribution, the tasks have been distributed in a very ideal way (Figure 41), where most of the first generated tasks are taken by threads of the first NUMA node while the second half of tasks are taken by threads of the second NUMA node. This distribution has a very positive impact in the performance because most of the threads are only going to access to local memory.

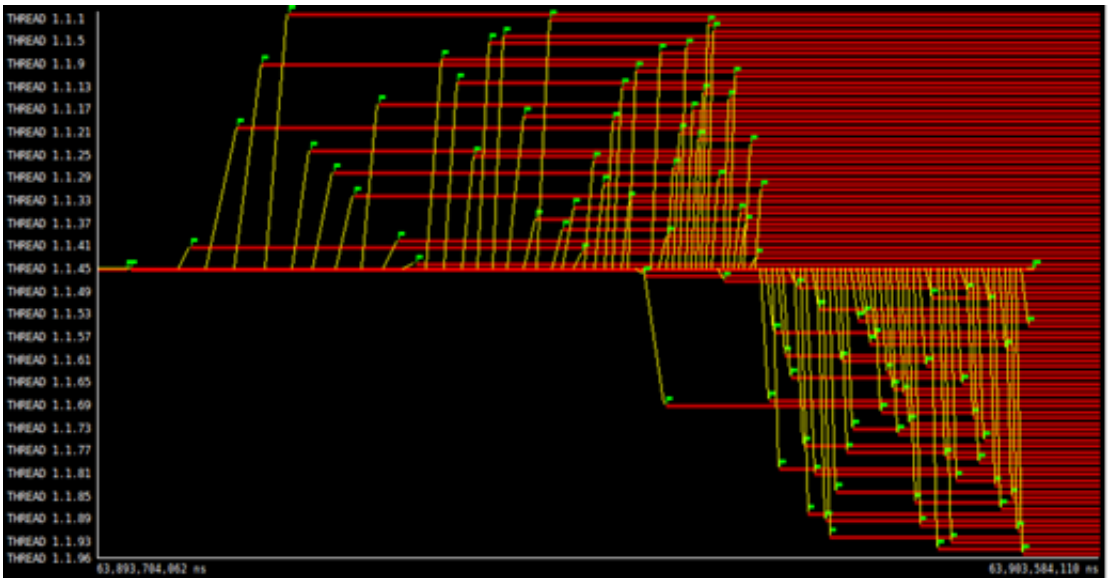


Figure 41 Reduction benchmark trace (after parallelizing the input) with communication lines (tasks distribution)

The main benefit of this optimization is the effectiveness in reducing the non-local memory accesses without adding too much overhead while maintaining the same algorithm behavior of the benchmark. Also due the fact that we have implemented the optimization using the *pragma omp_num_threads()*[13] call, the optimized application is scalable to any number of threads and does not require any modification nor recompile when executing with less or more than 96 threads.

[13]The *pragma omp_num_threads()* is a runtime (OmpSs / OpenMP) related call that returns the number of available threads that are executing the application.

However, because this optimization requires modifying the code, it is less desirable if our goal is to build a very portable application due the extra efforts that are going to be necessary to port the application between different operating systems or machines, but that is not the case.

10.3 Benchmarks scalability

After applying the optimizations, we have done a scalability analysis of the benchmarks in order to evaluate the core performance of our optimizations with different machine configurations.

The Figure 43 and 42 shows the scalability of each benchmark using different threads configuration. In order to have a clear view of how the benchmarks scales, the results shown in the charts are the speedup relative to a serial execution (1 thread). In each case, the maximum ideal performance is 8 and 96 for the merlin and thunder clusters respectively.

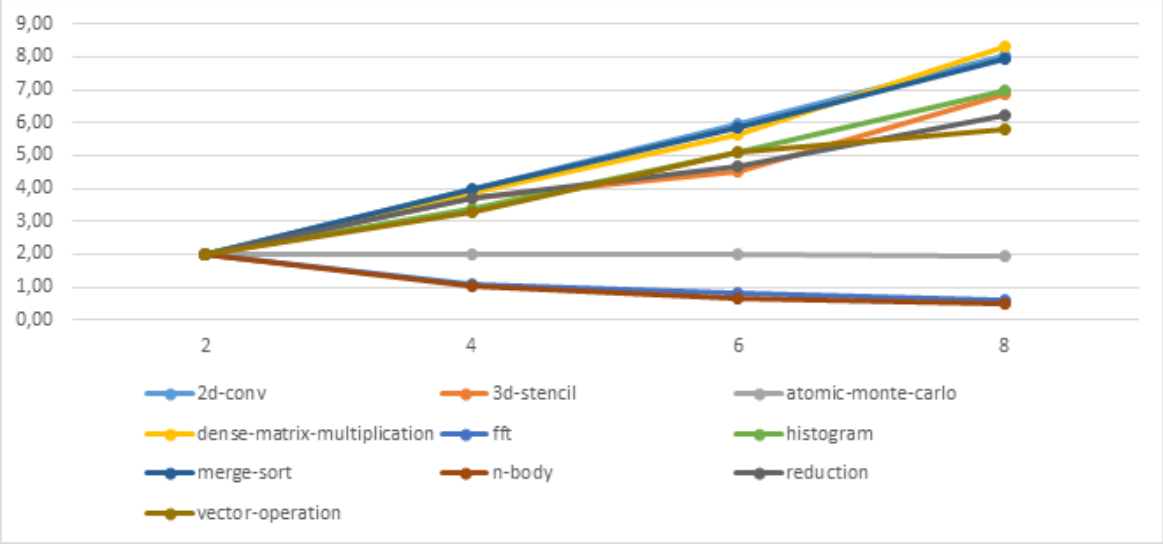


Figure 42 Mont-Blanc benchmarks scalability chart in merlin cluster after all the optimizations

The benchmarks in the merlin chart (Figure 42) scales very well. Most of the benchmarks achieves a speedup near the ideal one. On the other hand, overall, in the thunder chart (Figure 43) we can observe a good scalability using any thread configuration, however, the performance drops a little when the number of threads is higher to 48. This is due

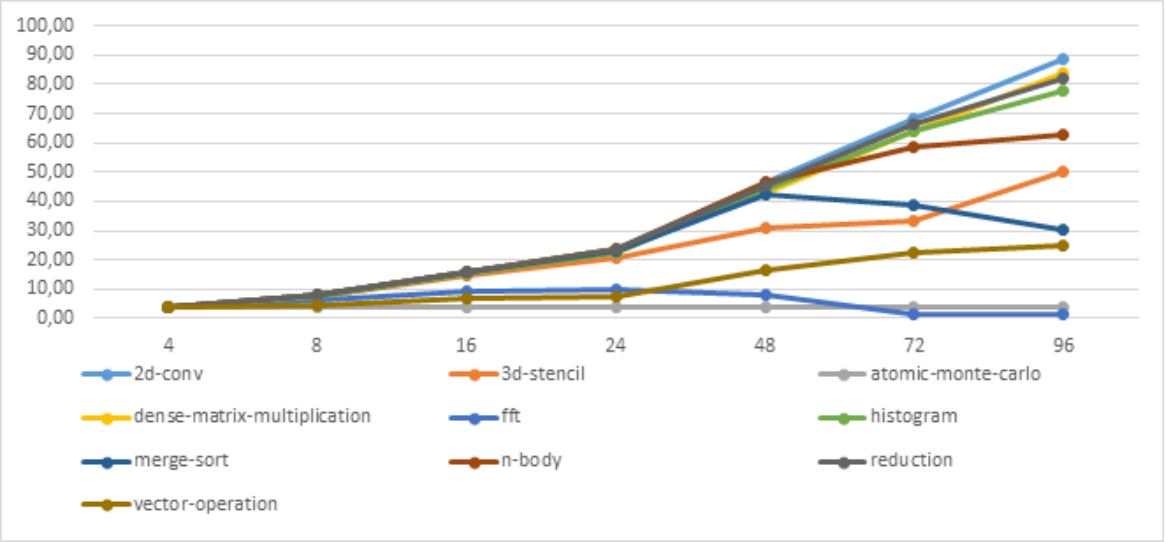


Figure 43 Mont-Blanc benchmarks scalability chart in thunder cluster after all the optimizations

the own algorithm limit and some overhead caused by the NUMA optimization however, if we compare it with the scalability before applying the NUMA optimization (Figure 44), the minimal overhead that our optimization creates it is justified for the performance improvement we achieved.

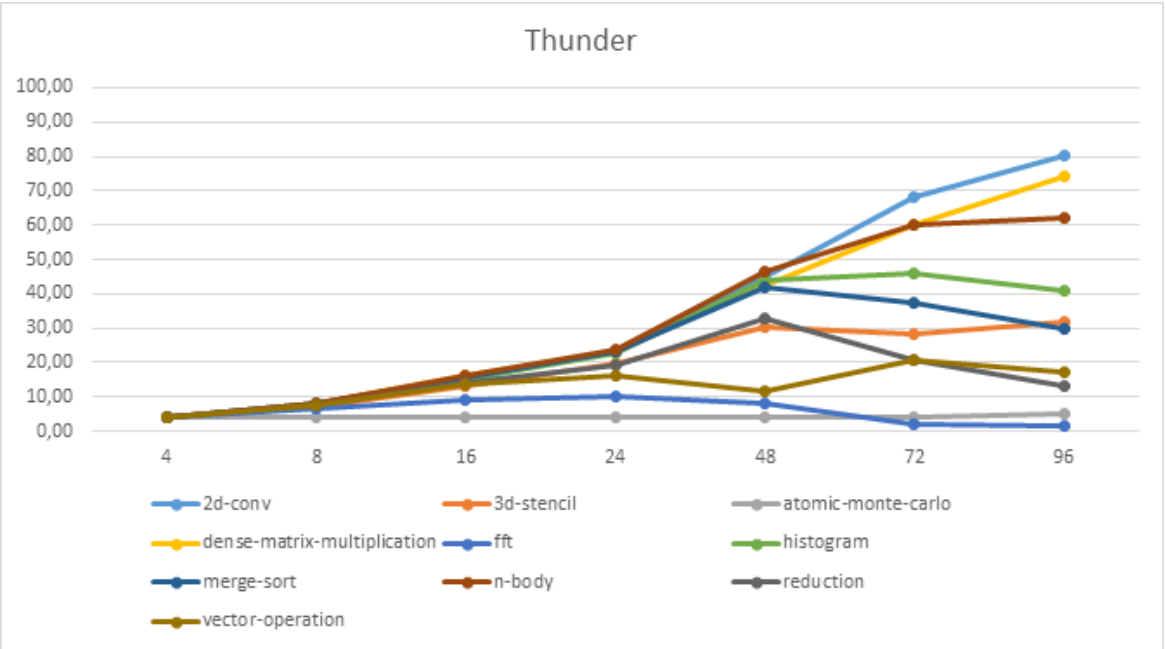


Figure 44 Mont-Blanc benchmarks scalability chart in thunder cluster before the NUMA optimization

The NUMA effect is clearly present in the non-optimized benchmarks scalability chart (Figure 44). Compared to the NUMA optimized version, the scalability of the best scaling benchmark only gets a speedup of 80 while in the NUMA optimized version most of the benchmarks gets a speedup near to 90.

Despite having optimized the benchmarks, we still can see a similar trend in some of the benchmarks in both machines. The Fast Fourier Transform, Atomic Monte Carlo and N-Body benchmarks still do not scale as good as the other benchmarks, but this is something directly related to the own benchmark algorithm.

10.4 High density multi-core machines

While executing in the thunder machines we have found a clear behavior in our executions. The thunder nodes are running a dual socket configuration with 48 cores for each processor and compared to the standard HPC processors which use 16 cores in average, the difference is noticeable. This difference implies a higher overhead for the ready queue because 48/96 threads trying to take tasks from the pool at the same time creates a lot of contention. This contention translates into a bad work distribution between the threads.

Also because the thread that initializes the runtime is executed in the first NUMA node, all the runtime related variables are also allocated in the first NUMA node, including the ready-queue[14]. This creates a huge disadvantage for the second NUMA node threads, because taking tasks from the pool will require access to non-local memory.

The Figure 45 clearly shows this disadvantage situation in the reduction benchmark. While the tasks are created by the master thread at the beginning, most of the tasks are taken by threads from the first NUMA node and very few threads from the second NUMA node do work. The yellow lines are communication lines and indicates when the task is created by the master thread (red), and when is taken and executed by the other threads (pink).

In order to achieve the best performance in high density multicore architectures, an execution with less tasks but with higher workload per task will contribute in reducing the contention by reducing the number of times a thread checks for work in the ready queue.

[14] In OpenMP or OmpSs, a master thread creates the tasks and puts them in a pool (a.k.a. ready-queue) while the other threads are constantly checking this queue to take work

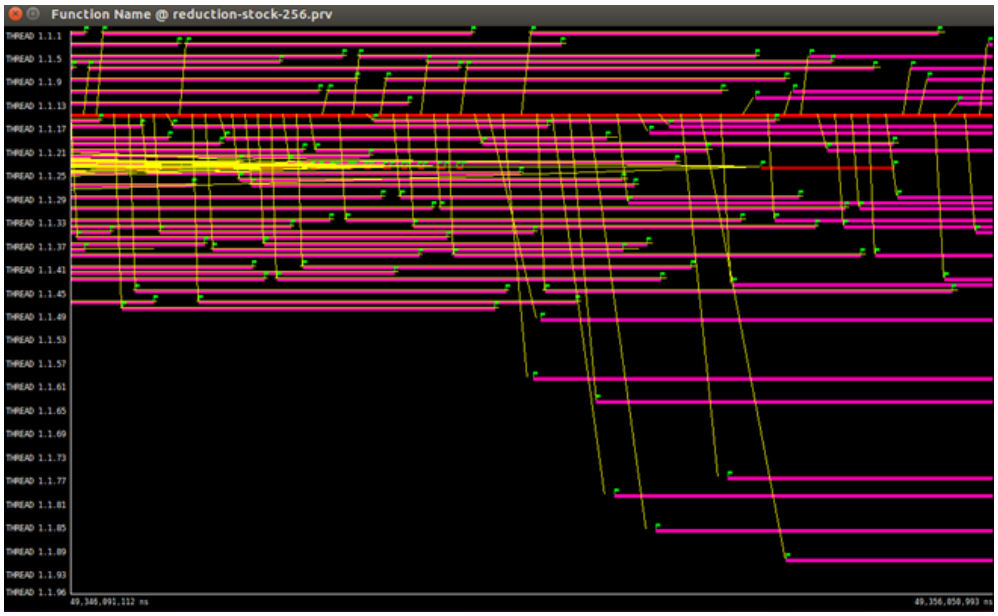


Figure 45 Reduction benchmark trace with 384 tasks and 134217728 elements to reduce

Increasing the tasks workload also increases the chances that a thread from the second NUMA node takes a tasks. Otherwise, with smaller tasks, the threads from the first NUMA node will check the ready queue more often and most of the times they will take the tasks before the petitions of the second NUMA node threads arrive.

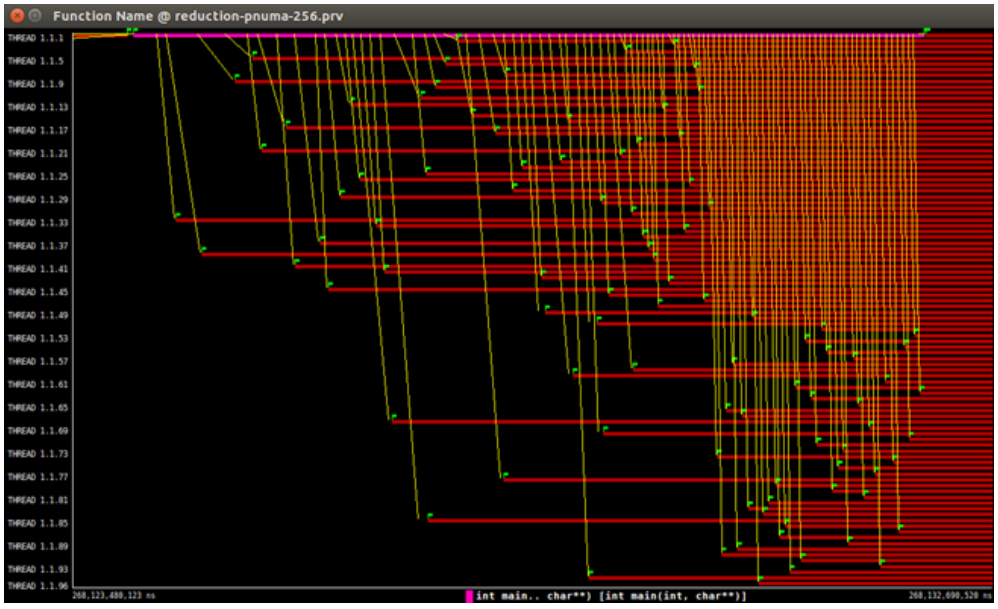


Figure 46 NUMA optimized reduction benchmark trace with 384 tasks and 134217728 elements to reduce

The strategy followed in our NUMA optimization has also taken into account this approach, by dividing all the work in 96 tasks and forcing all the threads to take a task and do work.

In the Figure 46 we can observe a trace of the reduction benchmark executed with the same input values of the Figure 45 but this time we have applied the NUMA optimization. The thread master (pink) creates the tasks and the other threads (red) take them. Similarly to the previous trace, the yellow lines indicate when the task is created and when is executed by a thread. In this case, unlike the Figure 45, the workload is uniformly distributed between all the threads, taking advantage of all the available resources.

Chapter 11

Conclusions and Future Work

11.1 Conclusions

This project has allowed me to explore a completely unknown field in the computer engineering world. I always have been curious about the ARM architecture and its fast growing implementation in most of the nowadays portable devices and since I heard about the Mont-Blanc project during the AC subject, I always wanted learn more about them.

This project not only have brought me the opportunity to make my "little dream" come true but to apply all what I have learned during the career, expand my knowledge about computer architectures and the best of all, it has given me the opportunity to meet many in-house talented researchers that are specialized in mobile and embedded-based HPC systems.

As a collateral effect, the development of this project has introduced me to the HPC world. On one hand, thanks to my director (Xavier Martorell), my codirector (Filippo Mantovani) and due my deep involvement with the ARM architecture, I had the opportunity to be part of the UPC team at the Student Cluster Competition held in the ISC16 (International Supercomputing Conference) in Frankfurt, Germany. On the other hand, this project has helped me to finally decide to continue my studies in the UPC FIB, doing the Master's Degree in High Performance Computing.

Given the results we have obtained, we can conclude that ARM is a powerful architecture with a lot of future. With an OS that applies an abstraction layer over the hardware, programming for ARM does not differ too much to the traditional x86 architecture. If we analyze the market trend and the performance improvement that ARM chipsets have suffered in the past

years, it is easy to conclude that what we are seeing nowadays is only the tip of the iceberg.

11.2 Future work: ARM + CUDA

Even though ARM is an architecture that targets for mobile devices, the presence of this kind of architecture in other fields is increasing. An example is this project, where we have executed a set of benchmarks, analyzed its execution behavior and optimized its performance in a HPC environment.

The Jetson TX1 is a very powerful and power-efficient solution. On one hand it uses ARM processors with all the benefits that this implies and on the other hand it offers full support for GPGPU (General Purpose on Graphical Processor Unit) through the CUDA environment.

As part of the future work, it will be interesting to not only optimize the Mont-Blanc Benchmarks to the ARM cortex-A57 architecture but also tune the benchmarks to run in the embedded NVIDIA GPU of the SoC (System on Chip) and learn more about this kind of configurations

Bibliography

[1] BSC (Barcelona Supercomputing Center). Paraver, parallel program visualization and analysis tool., 2001.

[2] Wikipedia. Arm architecture. https://en.wikipedia.org/wiki/ARM_architecture, 2016.

[3] HTC Corporation. Power to give. http://www.htc.com/us/go/power-to-give/, 2015.

[4] Mont-Blanc Project. Introduction. https://www.montblanc-project.eu/project/objectives, 2015.

[5] Green500. The green lists. http://www.green500.org/greenlists, 2015.

[6] NVIDIA Corp. What is gpu computing? http://www.nvidia.com/object/what-is-gpu-computing.html, 2016.

[7] OpenMP ARB. Openmp 4.0/4.5 specifications. http://openmp.org/wp/openmp-specifications, 2015.

[8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. pages 173–193, 2011. http://doi.org/10.1142/S0129626411000151.

[9] Teruel, X., BSC (Barcelona Supercomputing Center). Ompss quick overview, a practical approach. http://www.training.prace-ri.eu/uploads/tx_pracetmo/OmpSsQuickOverviewXT.pdf, 2013.

[10] TOP500. The linpack benchmark. http://www.top500.org/project/linpack/, 2015.

[11] D. Melnik, A. Belevantsev, D. Plotnikov, and S. Lee. A case study: optimizing gcc on arm for performance of libevas rasterization library.

[12] P. R. Mahalingam and Asokan Shimmi. Chapter 1: Energy-aware mobile application development by optimizing gcc for the arm architecture. in eco-friendly computing and communication systems, 2012.

[13] S. Banniste, M. Dupuy, and F. Bao. Project ne10: An open optimized software library project for the arm architecture. http://projectne10.github.io/Ne10/, 2015.

[14] Wikipedia. Markov chain. https://en.wikipedia.org/wiki/Markov_chain, August 2016.

[15] Shehzan. Explaining fp64 performance on gpus. http://arrayfire.com/explaining-fp64-performance-on-gpus/, 2015.

[16] Wikipedia. Superscalar processor. https://en.wikipedia.org/wiki/Superscalar_processor, August 2016.

[17] ARM Ltd. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*, 2015.

[18] GCC GNU. 3.10 options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2016.

[19] GCC GNU. 3.18.4 arm options. https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html, 2016.

[20] Rouse, M. Numa (non-uniform memory access). http://whatis.techtarget.com/definition/NUMA-non-uniform-memory-access, 2011.

[21] Sourceforge. Numa status: Item definition. http://lse.sourceforge.net/numa/status/description.html.