Universitat Politècnica de Catalunya

Final year project

Parallelization techniques of the x264 video encoder

Author:
Daniel Ruiz

Advisors:
Marc Casas
Miquel Moretó
Mateo Valero

June 20th, 2014

Acknowledgements

First and foremost, I would like to thank to my advisors, Miquel Moretó, Marc Casas and Mateo Valero for the priceless guidance and advice they offered to me during the whole project execution.

Besides, I would like to thank both Tools and Programming Models groups from Barcelona Supercomputing Center for the support offered to me when anything was working.

Also I would to take this opportunity to thank Dimitrios Chasapis for the advices he gave to me with bash scripts related stuff.

Finally, I would like to thank my friends, specially my girl, for all the moral support, without this I don't think I would had been able to complete this project.

Abstract

Higher video quality is demanded by the users of any kind of video stream service, including web applications, High Definition broadcast terrestrial services, etc. All of those video streams are encoded first using a compression format, one of them is H.264/MPEG-4 AVC. The main issue is that the better the quality of the video the larger the encoding time will be, so it is very important to be able to improve the performance of video encoders. In order to do it, those video encoders must be executed in a parallel way, trying to reduce the encoding time as many times as the number of cores available on the computer which performs the encoding of the video stream.

One of these video encoders is x264, which already implements a parallel version using POSIX threads. But this version has some scalability issues we want to solve using a different programming model. The chosen one is OmpSs, a task-based parallel programming model.

During this document, we are going to explain the process of porting x264 application to OmpSs programming model. Such a task requires doing a planning and monitoring the resources we will need. All the planning and the resources used within the project will be explained within several sections. Afterwards, a bit information about the H.264 compression format and the x264 application which encodes using this format will be provided, as well as information about the working environment set up needed for developing the OmpSs version of the x264 application.

Of course, a section of how the porting has been done, which includes the design, the implementation and the evaluation of the OmpSs version, will be provided as well. The obtained version will be compared against current parallel version of the x264 video encoder, in order to prove that OmpSs is also valid for non high performance computing workloads. This comparison will allow us to know if the port we made performs better or worse than the actual implementations of the x264 application.

Abstract - Spanish

A día de hoy, podría decirse que todos los usuarios de cualquier tipo de servicio de reproducción de vídeos demandan una mejor calidad en éstos. Estos servicios incluyen desde aplicaciones web hasta servicios de difusión de televisión en alta definición. El reto está en que todos los vídeos retransmitidos deben ser codificados primero utilizando alguno de los formatos de compresión disponibles, siendo uno de estos H.264/MPEG-4 AVC. El principal problema es que a mayor calidad de vídeo más tiempo requerirá la codificación, por lo que es de suma importancia ser capaces de mejor el rendimiento de los codificadores de vídeo. Para ello, estas aplicaciones deben ser ejecutadas en paralelo, intentando reducir el tiempo de codificación en un orden igual al número de hilos de ejecución disponibles en el computador donde se ejecute la codificación del flujo de vídeo.

Uno de estos codificadores es la aplicación x264, la cuál dispone de una implementación paralela utilizando el estándar *POSIX threads*. Pero esta aplicación tiene un problema, y es que su escalabilidad se puede ver reducida en ciertos casos. Estos problemas son los que queremos resolver mediante el diseño e implementación de una nueva versión la cuál utilizará el modelo de programación OmpSs.

A lo largo de este documento se explicará el proceso de portar la aplicación x264 a OmpSs. Esta tarea requiere de una cuidada planificación, así como de la monitorización de recursos que se emplearán en el proyecto. De hecho, la planificación realizada, y estrictamente necesaria para llevar a cabo el proyecto, se explicará en diferentes secciones de este mismo documento. Por otra parte, también se va a proporcionar información sobre el formato de compresión H.264 y la aplicación x264, está última codificando al formato de compresión previamente citado. También se dará información relacionada con la plataforma hardware sobre la que se ha desarrollado el proyecto, así como el conjunto de aplicaciones de desarrollo utilizadas, incluyendo el model de programación OmpSs y sus componentes.

Por último, pero no menos importante, hablaremos sobre el proceso de portar la aplicación en sí. Este proceso incluye el diseño e implementación de la nueva versión basa en tareas, así como la evaluación de rendimiento de ésta. A partir de esta evaluación y de las realizadas sobre las versiones secuencial y paralela ya implementadas, se realizará una comparación con la idea de descubrir hasta qué punto nuestra aplicación rinde mejor o no que las actuales implementaciones.

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

1.1 Motivation

The main purpose of this project is to design and implement a parallel version of the x264 application. For this task we will use the task-based programming model OmpSs (OpenMP SuperScalar), currently developed at BSC (Barcelona Supercomputing Center).

The x264 application is an H.264/AVC (Advanced Video Coding) video encoder. In the 4th annual video codec comparison it was ranked 2nd best codec for its high encoding quality. It is based on the ITU-T H.264 standard which was completed in May 2003 and which is now also part of ISO/IEC MPEG-4. In that context the standard is also known as MPEG-4 Part 10. H.264 describes the lossy compression of a video stream. It improves over previous encoding standards with new features such as increased sample bit depth precision, higher-resolution color information, variable block-size motion compensation (VB-SMC) or context-adaptive binary arithmetic coding (CABAC). These advancements allow H.264 encoders to achieve a higher output quality with a lower bit-rate at the expense of a significantly increased encoding and decoding time. The flexibility of H.264 allows its use in a wide range of contexts with different requirements from video conferencing solutions to high-definition (HD) film distribution. Next-generation HD DVD or Blu-ray video players already require H.264/AVC encoding. The flexibility and wide range of application of the H.264 standard and its ubiquity in next-generation video systems makes the x264 application an excellent candidate to improve its performance using the task-based programming model OmpSs.

OmpSs extends OpenMP with a new set of directives in order to support asynchronous parallelism and heterogeneity (devices like GPU's). It's composed by Mercurium source-to-source compiler and Nanos++ runtime system, both also developed at BSC. Also, OmpSs provides asynchronous parallelism, so the idea is to exploit it in order to achieve the maximum performance, this increment can be computed beforehand using Amdahl's Law, which will tell us the maximum performance that we could achieve taking into account the portion of code will be parallelized.

In that context, the idea is to improve the current parallel implementation for shared memory systems which it is currently implemented using pthreads. This implies that we are not going to use OmpSs' heterogeneous features.

Finally, the x264 version we will port to OmpSs is the one that can be found at the PARSEC Benchmark Suite. PARSEC is a benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to contain a diverse selection of applications that are representative of next-generation shared-memory programs for chip-multiprocessors.

1.2 Stakeholders

At this section we are going to discuss about who could be interested on the project, this is, its stakeholders [St13]. In this line we are going to split the potential stakeholders in several groups. The following list is a summary of the different parts of this section. A further detailed discussion about each stakeholder will be provided afterwards.

Developers
: In this case I'm the only one developer. The reasons for being interested on the project are more than obvious.

Project directors
: Both directors of the project are interested on it since it takes part of a bigger one which consists on porting the whole PARSEC (Princeton Application Repository for Shared-Memory Computers) Benchmark Suite[PSC08] to OmpSs.

HD video player system developers
: It is mandatory for both HD DVD and Bluray players to be able to reproduce the H.264 video codec since it is one of the codecs that these formats support. [Blu05].

Video streaming system developers
: H.264 is widely used on these kind of systems, so it could be interesting for its developers if we improve the current implementations.

OmpSs developers
: Increasing the performance over the current x264 implementations means that OmpSs is a valid programming model that could provide a greater performance than the actual ones.

Video encoder and decoder software developers
: This kind of software usually use applications like x264 as a video encoder while they only implement the front-end. It could be interesting for them to use our version if it is better than the actual one.

Users
: This includes all the people that will use the encoder and decoder software to convert their, for example, old non-digital videos to a digital ones. Also includes television users since H.264 is already used in some countries as the codec for terrestrial broadcast services.

Developers

As we said before, the one and the only developer of this project it will be only me. The reasons for being interested in are obvious of course. Besides this, I am concerned about the impact of an improvement of the widely used H.264 video codec on both environment and the market could make.

Nowadays the H.264 is likely an standard for High Definition video. But its range of applications is not that closed, actually it can also be used as video codec for HTML 5 embedded videos [htl14]. So, improving the decoding and encoding performance of the H.264 video codec could reduce the cost in terms of energy consumption for servers providing these videos, making web services or also video streaming services more greener. This actually translates into a less costs in environmental fees for the companies.

Project directors

The project is actually a part of a bigger one consisting on porting the whole PARSEC Benchmark Suite to OmpSs. Both project directors are participating on it, so they are interested on designing the task-based version of x264 since it has to be done on its projects frame.

PARSEC is actually composed of several multi threaded applications, being x264 one of them. This benchmark suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors.

For that reasons they are quite interested on knowing if the performance of the PARSEC benchmarks can be improved using the task-based programming model OmpSs. They are also looking forward to trying to port the whole benchmark suite to an MPI plus OmpSs version at second stage, and an MPI plus OmpSs plus CUDA or OpenCL at a third one. Both versions fall on the success of this project, which consists only on porting the x264 to OmpSs.

HD video player system developers

It is mandatory for Bluray and HD DVD video player systems to support the H.264 video codec because it is one of the three codecs supported by these formats. So, improving the encoding performance translates directly into a more compressed file which, potentially, will be burned on a digital optical disc.

On this could focus the interest of the developers since an smaller file size means that they can increase the quantity of data per disc (which translates into a major video duration) or they can improve the bit-rate of the video stream (which means an increased video quality).

In this line developers will be able to achieve a better product than the actual one or maybe to continue offering the same quality and duration but with a lower costs due to the reduced power consumption as the burn process would take less time than before.

Video streaming systems developers

Developers of this kind of systems can be concerned about our version if it is better enough than the current one to justify adopting it.

Here we have mostly the same reasons as for high definition player system developers. If we are able to improve the performance, they will be able to increase the quality or the length of the videos they offer or to reduce the needed storage capacity.

OmpSs developers

In this case the interest is more about bringing to programmers more reasons to use its programming model than others. So, improving the performance of the x264 application over the current one, which don't use a task-based approach, grants to OmpSs a better impression of its programming model rather than the one used at the actual implementation, which it is pthreads.

Video encoder and decoder software developers

An improved version respect to the actual one could be very interesting for video encoder and decoder software developers because it will improve also its application. This software usually implements only a friendly front-end interface to smooth the way for the users since them could avoid concerning about decoding or encoding parameters for the application that actually decodes or encodes [FF]. Furthermore, one of the most used encoding and decoding applications as FFmpeg uses x264 as its video decoder and encoder for the H.264 video codec [FFP].

So, this kind of software, which usually uses an application like x264 to encode and decode video streams, could improve its overall performance using our version as long as our one will be better than the others.

Users

Not only terrestrial broadcast services but also direct broadcast satellite TV ones as well as some IPTV services actually use H.264 video codec. Also Apple (via iTunes) is offering video content that uses H.264 codec [APL05]. Furthermore, Android platform natively supports H.264 [ADR].

In this context the end-user of a wide range of applications will take profit of our version if it is attractive enough to make x264 developers to think about adding it to its application.

1.3 State of the Art

In this section we are going to go deeper into the solutions that are already on the market. This is, the current parallel implementations of the x264 application.

At the time this document is written, there are only two parallel versions of the x264 application implemented. The first one is a shared memory based one that uses pthreads, the second one uses OpenCL [hcl], but this version is not the best one to do a fair comparison with ours since OpenCL it is not a shared memory programming model [OCL13].

At this point one can see that there are not too many parallel versions of the x264 application. Even though some paid H.264 decoders and encoders exist that are implemented using CUDA, OpenCL, etc. [hcu13]. However, those implementations are not free of charge as it will be our version.

Of course those implementations must be considered by us in order to see how our version performs. The idea is to know if there is any reason to propose a new solution to the problem the H.264 video codec presents. In that line, at the eighth MPEG-4 AVC/H.264 video codecs comparison [hcm12], x264 was chosen the best H.264 video encoder. In that comparison the following encoders were tested:

- DivX H.264
- Elecard H.264
- Intel Ivy Bridge QuickSync (GPU encoder)
- MainConcept H.264 (software)
- MainConcept H.264 (CUDA based encoder)
- MainConcept H.264 (OpenCL based encoder)
- DiscretePhoton
- x264

Note that the x264 implementation used was the parallel one that use pthreads.

About the results of the comparison, the worst version was the CUDA based one in terms of video quality. The best version at performance tests was the Intel Ivy Bridge QuickSync (GPU-based). But the one that contributes with the greatest quality is the x264 application. Maybe it is not the fastest one, but it is the one that can achieve the greatest video quality, and this is the main reason because it is actually the most used one.

Then, knowing that the application we are going to port to OmpSs is the best encoder for the H.264 video codec, we think that it won't be a waste of time porting the application.

About using CUDA or OpenCL on the x264 application to offload work to the GPU instead of the CPU, it seems that in our case it is not better than using only the CPU, so it looks feasible that our version could be the chosen one to improve even more the x264 performance. Taking this into account, I think it is not a waste of resources to design and implement a new parallel version for shared memory systems, not only because it has been demonstrated that these

versions are the best ones, but also because we are going to use a new approach to design it, a task-based approach.

In order to be able to really understand why we want to use a task-based approach instead of another one, this is the programming model OmpSs instead of another one, we are going to explain the current state of the art of the parallel programming models also.

In this line we are going to focus only on programming models that support shared memory systems. So we will discuss about the following ones:

- CUDA
- Chapel
- CHARM++
- Liszt
- Loci
- OpenMP
- OmpSs

CUDA

CUDA is a set of C++ language extensions plus an accompanying runtime API for programming NVIDIA GPUs [cu12]. A computational kernel is programmed essentially as a C++ function that is run for every thread. Threads are grouped hierarchically into warps, blocks and grids. The finest group, a warp (currently 32 threads), runs the same set of instructions in SIMD fashion with support for diverging the execution paths. Threads in the same block are active at the same time and have access to fast, on-chip shared memory and local synchronization primitives. Finally, the various thread-blocks in a grid are executed completely independently and in arbitrary order, allowing for execution of problems too large to fit on the hardware simultaneously.

Chapel

Chapel is an emerging parallel language initiated under the DARPA HPCS program with the goal of improving programmer productivity [ch07]. Chapel is designed using a block-imperative syntax with optional support for object-oriented programming, type inference, and other productivity-oriented features. Chapel supports both task and data parallel styles of programming, and permits these styles to be mixed arbitrarily. Task-parallelism is supported by creating abstract concurrent tasks that coordinate through shared synchronization and atomic variables. Data-parallelism is expreseed via loops and operations on data aggregates. Chapel supports reasoning about locality on node via the concept of a *locale*; for example, locales are often used to represent compute nodes on large-scale systems. Domains and arrays can be distributed across sets of locales in a high-level manner using the concept of *user-defined domain maps* [ch11].

CHARM++

CHARM++ is a parallel programming system based on message-driven migratable objects [Ka93, La03]. It is implemented as additions to the C++ language coupled with an adaptive runtime system. Parallelism in CHARM++ is created by over-decomposing an application into its logical work and data units, referred to as *chares*. The number of chares is typically more than the number of processors. The programmer expresses application flow, computation and communication as operations performed by chares. The distribution of chares to processors and scheduling of their execution is handled by the CHARM++ runtime system. Communication between chares is performed through remote method invocations and chares is performeed through remote method invocations and is also handled by the runtime system. Communication is asynchronous with respect to other chares which provides the benefit of adaptive overlap with computation. One optional language feature is that the parallel control flow can be specified by the user through a structured directed acyclic graph (SDAG) which can lead to more elegant code.

Liszt

Liszt is a Scala-based domain-specific language for solving partial-differential equations on meshes [De11]. The language is designed for code portability across heterogeneous platforms. The problem domain is represented as a three-dimensional mesh whose elements can be accessed only through a mesh-based topological functions as immutable first-class values. The mesh is initialized at program start time and its topology does not change over the program's lifetime. Fields are abstracted as unordered maps indexed only using mesh elements. Liszt provides three features for parallelism: a parallel *for-comprehension* on sets of mesh elements, atomic *reduction operators* on field data, and *field phases*, i.e. read/write restrictions on field data inside a for-comprehension. Moreover, Liszt does not support recursion. These semantic constraints ensure that the Liszt compiler can infer data dependencies automatically, enabling it to generate a parallel implementation for code written in a serial style. One drawback is that Liszt provides no high-level abstraction for load balancing and mesh decomposition. Lack of direct programmer control on these aspects has performance implications for certain back ends.

Loci

Loci is a C++ framework that implements a declarative logic-relational programming model [Lu05]. The programming model is implicitly parallel and uses relational abstractions to describe distributed irregular data structures. A logic programming abstraction similar to Datalog [Ul88] is used to facilitate composition of transformation rules. The programming model exploits a notational similarity to mathematical descriptions found in papers and texts of numerical methods for the solution of partial differential equations [Zh09]. In addition, the programming model facilitates partial verification by exploiting the logic programming model to provide runtime detection of inconsistent or incomplete program specification. Parallel execution is achieved using loosely synchronized SPMD approach that exploits the data-parallelism that naturally

emerges from the distribution of relations to processors. Communication costs in the generated parallel schedule are controlled through message vectorization and work replication optimizations [So08].

OpenMP

OpenMP uses pragma directives that are added to C, C++ and Fortran programs [Da98]. These directives can specify regions and loops to be parallelized by the OpenMP compiler using threads. Further, directives can be used to mark critical or atomic sections within the parallel regions. Through information added to the compiler directives, a programmer can specify which variables are shared or private in order to prevent false sharing and to isolate effects from multiple threads. Additionallym the pragmas allow specification of the number of threads per loop as well as reductions. Finally, OpenMP allows nested parallelism with each thread capable of spawning child threads.

OmpSs

The OmpSs programming model is an effort to integrate features from the StarSs programming model developed by BSC into a single programming model. In particular, the objective is to extend OpenMP with new directives to support asynchronous data-flow parallelism and heterogeneity (as in GPU-like devices). The most prominent feature of OmpSs programming model is the extension of OpenMP tasks with dependences. Dependences let the user annotate the data flow of the program, this way at runtime this information can be used to determine if the parallel execution of two tasks may cause data races. Asynchronous parallelism is enabled in OmpSs by the use of data-dependencies between the different tasks of the program. The OpenMP task construct is extended with the in (standing for input), out (standing for output) and inout (standing for input/output) clauses to this end. They allow to specify for each task in the program what data a task is waiting for and signalling its readiness. Note, that whether the task really uses that data in the specified way its the programmer responsibility.

In our case, the chosen programming model was OmpSs. The main reason for chosen this programming model instead of another one is because we want to prove that OmpSs can be used also for non high performance computing workloads.

At second place, we really believe that the characteristics of the x264 application made feasible to design a better parallel version of it using a task-based approach than the one used at the current parallel version which it is pthreads.

We think also that OmpSs can achieve a greater levels of performance than the programming models we discussed before. The last reason is that we believe that OmpSs is the programming model that mix in a better way all the characteristics that a parallel programming model must have.

1.4 Objectives

Our objectives can be split in main objectives and secondary objectives as following:

- Main objectives:
  - Design a parallel version of the x264 application using a task-based approach.
  - Implement the parallel version we designed using the task-based programming model OmpSs.
  - Comparison with current parallel version, emphasizing on differences between programming model used on the current parallel version and OmpSs programming model.
- Secondary objectives:
  - Characterization of the x264 application.
  - Compare performance analysis of the current parallel version in front of our parallel version.
  - Prove that OmpSs programming model is also valid for non high performance computing workloads.

1.5 Project Scope

The main purpose of the project is to design and implement an OmpSs version of the x264 video encoder, so it is needed to specify what will be done and what not. This is very important to avoid wasting time on tasks that are not supposed to be done by us.

The following lists show what will be done and what not under the project's frame that there are directly related to the porting itself.

- What will be done
  - Evaluation of the serial version of the application
  - Evaluation of the current parallel version of the application
  - Characterization of the application
  - Design of the task-based algorithm
  - Implementation of the task-based algorithm using OmpSs
  - Evaluation of the OmpSs version of the application
- What will not be done
  - A version for distributed memory systems
  - A study of application's impact on the market
  - Comparison between different programming models
  - Evaluation of the final video quality

– Comparison in terms of energy efficiency

Of course the tasks we just listed require time in order to be completed. Time planning will be discussed in a deeper way later on a different section, but the following list could bring to you a better understanding about how the tasks we are going to do along the project will be scheduled. Also there is included an estimation of the time the task will take long to be completed.

Documentation - 10 days
This includes the study of the algorithm.

Performance Analysis - 15 days
We must know which are the portions of the code that needs more time to be executed than others in order to apply Amdahl's Law in a proper way.

Porting x264 application to OmpSs - 40 days
This is the main task of the project. Includes the design and the implementation of the application using the task-based programming model OmpSs.

OmpSs version evaluation - 10 days
The implemented version must be evaluated and compared with the current ones.

Write memory - 30 days
A memory containing all the results as well as the methodology, risk management, etc. must be written.

Chapter 2

Planning, budget and sustainability

2.1 Gantt chart

Lets begin explaining the Gantt chart. Firstly, I am going to show the chart (see Figure 2) and then I will start explaining the main tasks. Subtasks are going to be explained afterwards. The project was started at 21st of March.

The tasks are distributed in time respecting both working and resting days. The working days of the week are Monday, Tuesday, Wednesday, Thursday and Friday; the resting are Saturday and Sunday. The idea is to work six hours every day. So, the theoretical starting day is the 21st of March and it is planned to be finished at the 20th of June, we had sixty-six working days that made a total of 396 hours and 34 resting days. These resting days can be used to work too in case some of the tasks requires more time than the estimated one, adding 204 hours more than the estimated to finish the project.

Note that at Figure 2 it is indicated the working days of each task, but it is not reflected that some of the tasks can be done at the same time. This is the reason why the sum of the working days showed at Figure 2 is not the same as the one explained before. Note also the red tasks are the ones that compose the critical path of the project.

As we can see at *Figure 2*, the main tasks are the following:

Project Management
: This task is the first one of the project and it consists on designing the project itself.

Documentation
: This task is mainly focus on searching the information we need to be able to complete the project.

Get familiar with working platform
: In this task we will get familiar with the working environment. This includes tools, compilers and Mare Nostrum the supercomputer itself.

Performance Analysis
The intention of this tasks is to analyse the original application in order to obtain the needed information that will allow us to a better job in the design and the implementation of the OmpSs version.

Porting x264 application to OmpSs
This task is composed by designing and implementing our application. In order to do it, we need to study all the code dependencies, both data and task dependencies.

OmpSs version evaluation
At this point, we will have 3 versions of the application in which we are interest. First one is the sequential version, second one the current parallel version implemented at x264 application and the third one is the OmpSs application we are going to implement. So, we are going to compare all three in order to know the real performance of our application in front of the current implementations.

Write memory
This task is actually split in 3 tasks along the project. The idea is to start writing the memory as soon as possible to avoid writing about sections that we finished months ago.

Critical Path

As said before, the red nodes that one can see at *Figure 2.1* on page 14 are the ones that compose the critical path. This path is the one that define the duration of your project in terms of time.

The tasks included at the critical path are also the more critical ones because a delay on one of them could provoke not to be able to finish the project within the scheduled time. So it is really needed to be able to manage possible risks to avoid delays.

In that line the planning has been made making room to this possible delays on each task. This means that all the estimated time for each plan includes more time that the one would be needed firstly.

Table 2.1 on page 13 references each task with a code in order to make easier the identification on *Figure 2.2* showed on page 15.

Task Name	Task Code
Project Management	T1
Project Scope	T1.1
Time planning	T1.2
Budget and Sustainability	T1.3
Context and bibliography	T1.4
Spec sheet	T1.5
Documentation	T2
Study of the Algorithm	T2.1
State of the Art	T2.2
Write documentation memory	T3
Get familiar with working platform	T4
Get familiar with evaluation tools	T4.1
Get familiar with Extre	T4.1.1
Get familiar with Paraver	T4.1.2
Get familiar with OmpSs	T4.2
Get familiar with Mercurium	T4.2.1
Get familiar with Nanos++	T4.2.2
Performance Analysis	T5
Profiling	T5.1
Timing	T5.2
Characterization	T5.3
Write Performance Analysis memory	T6
Porting x264 application to OmpSs	T7
Study dependencies	T7.1
Desing OmpSs version	T7.2
Implement OmpSs vesrion	T7.3
OmpSs version evaluation	T8
Sequential version evaluation	T8.1
Current parallel version evaluation	T8.2
OmpSs version evaluation	T8.3
Comparison between sequential, parallel and OmpSs version	T8.4
Write remaining memory	T9

Table 2.1: Tasks codification

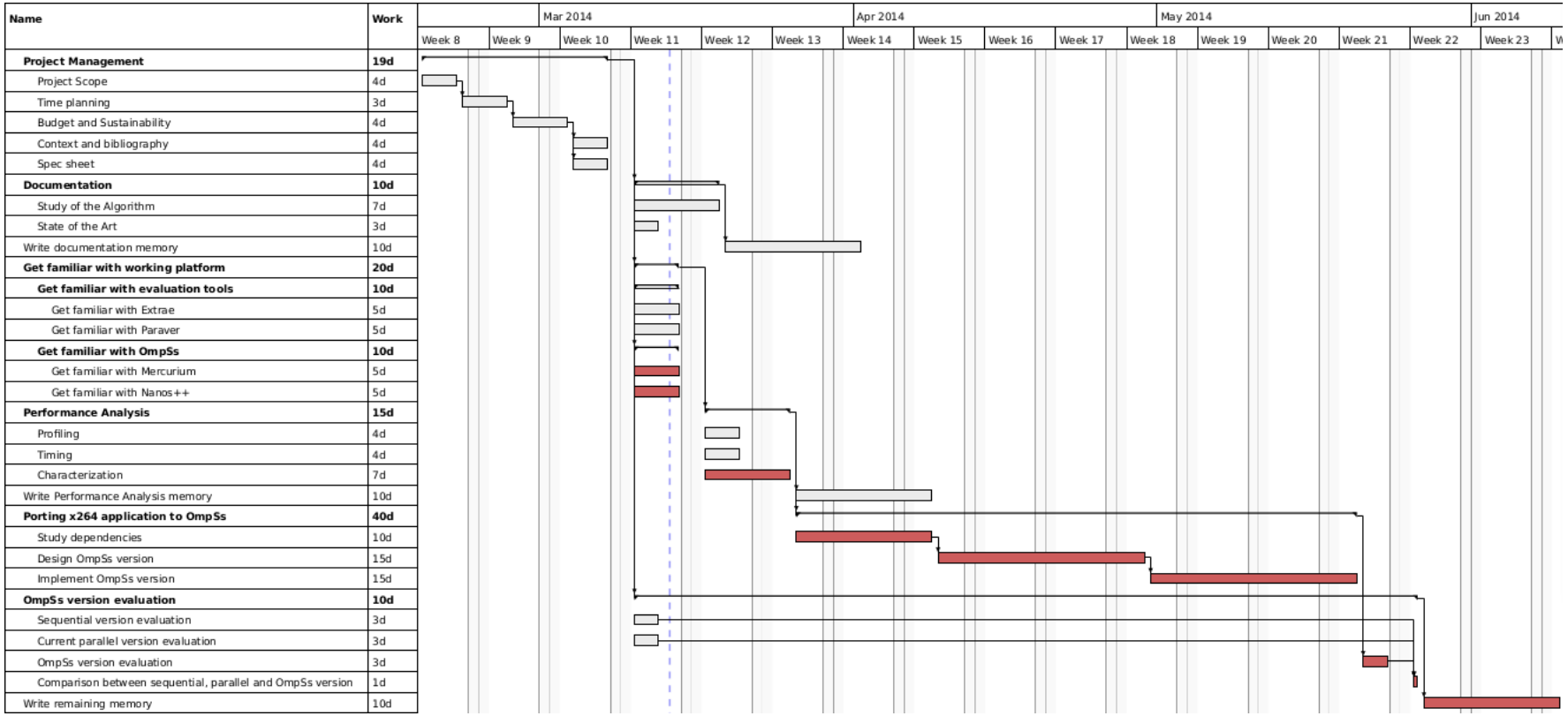

Figure 2.1: Gantt chart

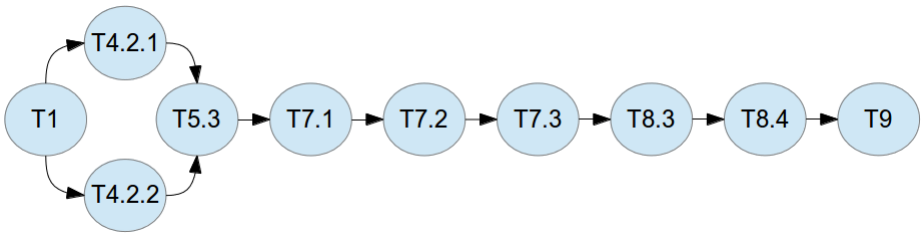

Figure 2.2: Critical Path

2.2 Tasks

We are going to describe in a deeper way all the project tasks. In this line, you will find an explanation for each task of the project.

Project Management

This task is the initial one and it consists on designing the project itself. So it includes time planning, budgets, sustainability, state of the art, etc.

It is also a dependency for every task since one cannot start a project without designing it in a first place.

Resources:

- Laptop Dell Latitude E5420
- Linux Ubuntu 13.10 Gnome

Documentation

This section is split in two minor tasks. The first one is the study of the algorithm and the second one writing the state of the art. Both tasks can be done at the same time. No especial resources are need for this task, we have more than enough with a laptop in order to access papers that located on the Internet.

Resources:

- Laptop Dell Latitude E5420
- Linux Ubuntu 13.10 Gnome

Study of the algorithm - 7 days

The main activity of this task is to read papers, documentation, etc. in order to get as much information as possible about the algorithm. The idea is to become an expert about the algorithm in order to easily find all the data and task dependencies while porting the x264 application to OmpSs.

So a lot of papers, documentation will be read in order to know not only the portions of code that can be potentially improved but also to be able to split in different tasks the algorithm.

As well as this task is the first one, it has only one dependency, this is the project management of the project.

State of the Art - 3 days

We will search about all the current parallel implementations, not only the ones that can be executed on a shared memory system but also implementations for distributed memory systems or accelerators as CUDA and OpenCL.

We will look at different programming models also, comparing it with the one we are going to use to make our version as well as discussing about the advantages and the contras of using the programming model OmpSs.
As same as for the study of the algorithm, this task has the same dependencies as the one before.

Get familiar with working platform

This task mainly consists on experimenting and reading the manuals of the software we are going to use. In this case we would need access to the working environment, this is Mare Nostrum. This means this task has a resource dependency since the required steps to be allowed to use Mare Nostrum can take long.

In our case we already have access to the cluster, so the task could start as soon we start to work on the project.

Resources:

- Laptop Dell Latitude E5420
- Mare Nostrum III
- Linux Ubuntu 13.10 Gnome
- SuSe Distribution 11 SP2
- OmpSs
- Extrae
- Paraver

Get familiar with evaluation tools - 10 days

We are going to work a lot with Extrae and Paraver, so I decided to include a task where to get familiar with these tools. So, at this task the idea is to do some training with Extrae and Paraver in order to avoid wasting time in the future.

About Extrae, we will need to do some training in order to know all the capabilities it offers like which parameters we could gather of the application and how to configure and run the application in order to obtain it.

Once we get traces, then they have be visualized using Paraver, but it offers a lot of configuration that allow the user to see several information of the application. So we will need to do some research in order to know which ones are the most important to see in order to be able to understand in a better way why our application is not giving us the expected performance.

Get familiar with OmpSs - 10 days

This task is almost the same as the one before. We are going to use OmpSs which includes Mercurium and Nanos, so we must be an experts using both compiler and runtime. This includes compiler flags, execution options, etc.

About Mercurium, is very important to know which are the best flags we will need to improve even more our application at compilation time. We will need to integrate our building configuration with the actual one that it is currently implemented. In order to do this we will need to study how to compile the application using mercurium as well as how to compile normal code.

Nanox is a runtime, but it accepts some parameters that could improve the application at execution time. So we need to know which are these parameters and the only way to do this is doing some training using it.

Performance Analysis

In order to be able to compare and also to design the application we are going to implement, we will need first to characterize the application as well measure how time it needs to be executed and how this execution time is distributed into the portions of the code.

This task depends on the last one because we need to get familiar with the tools we are going to use if we want to make a nice performance analysis.

Resources:

- Laptop Dell Latitude E5420
- Mare Nostrum III
- Linux Ubuntu 13.10 Gnome
- SuSe Distribution 11 SP2
- OmpSs
- Extrae
- Paraver
- GNU gprof

Profiling - 4 days

Before we start to design our OmpSs implementation we must need to know where are we spending the CPU time in order to be able to optimize the maximum portion of code as Amdahl's Law suggests. In this line we must do the best profile we can. We will use GNU gprof software to get the application profiles.

These profiles will allow us to focus our efforts on improving only the portions of code that will have a bigger impact on the performance of the application.

Performance evaluation - 4 days

We need to know also the quantity of time the original application needs to execute. In order to do this we will need to run several times the application using real input data. This can be very expensive in time terms, but it is absolutely needed.

This tasks consists on run several times the application at Mare Nostrum in order to obtain different measures that will allow us to approximate in a better way the real execution time calculating the median. Depending on the size of the input data, each execution can take more or less time to be completed.

Characterization - 7 days

At this point, we will use both Extrae and Paraver in order to make a complete characterization of the application to be able to know and understand the factors that affect the performance of our application.

As same as for the timing and the profiling, we will need to run the application in order to obtain traces. And then we will need to study these ones using Paraver, looking for the issues in the execution that makes our application to run slower than we expect.

Porting x264 application to OmpSs

This is the main task of the project. So every task is dependant of this task or its results will be use by this task.

So it is not strange that this task uses the major percentage of time of the whole project.

Resources:

- Laptop Dell Latitude E5420
- Mare Nostrum III
- Linux Ubuntu 13.10 Gnome
- SuSe Distribution 11 SP2
- OmpSs

- Extrae
- Paraver
- GNU gprof

Study dependencies - 10 days

We are going to do a deeper study of the algorithm and the characterization of the application in order to know which are the data and task dependencies. This task has an incredible importance because we will not be able to design our version without knowing which dependencies we really have. Also, it will allow us to exploit OmpSs asynchronous parallelism.

In this line we will need to study the algorithm and try to break all the dependencies we found. This will require time and the knowledge of the algorithm that we learned at the documentation task.

Design OmpSs version - 15 days

At the time we start this task, we already would have finished the study of dependencies, so this task is going to use that knowledge in order to design a solution to overcome dependencies. At the time this task will be finished, we will have a pseudo code implementation of our version. This pseudo code will be the starting point of the next task.

Implement OmpSs version - 15 days

In this task we are going to convert our design, obtained at the previous task (the one where we designed the OmpSs vesrion), to an OmpSs application. So, this task is only about the translation from pseudo code to OmpSs.

This task also consists on debugging the application and running it with a small inputs to know if all the improvements we made are not affecting to the output of the application.

OmpSs version evaluation

In order to know if our implementation is good enough for us, we need to compare it with the current versions of the application. In our case we are going to compare it with the sequential one and the current parallel version that uses pthreads.

Resources:

- Laptop Dell Latitude E5420
- Mare Nostrum III
- Linux Ubuntu 13.10 Gnome
- SuSe Distribution 11 SP2

Sequential version evaluation - 3 days

Here we are going to evaluate sequential version of the x264 application in order to be able to compare it with our OmpSs implementation. This task can be started as soon as we start the project.

Current parallel version evaluation - 3 days

Same as before, we need to evaluate the current parallel version of the application to be able to compare it with our implementation. As before, we can start the task at the earliest stage of the project.

OmpSs version evaluation - 3 days

We also need an evaluation of the OmpSs version to compare with the sequential and the parallel versions. At the time all three evaluations are done we would be able to do the comparison between each. This task cannot be started until we have finished the implementation of the OmpSs version.

Comparison between sequential, parallel and OmpSs verion - 1 day

At this task we are going to compare all three versions of the x264 we have evaluated. The result of this task will be the one that will allow us to write the remaining memory with the results of our work.

Write memory - 30 days

This task is actually split in several ones. The idea is to be able to write down all the details as soon as possible. So to accomplish that we are going to write a portion of the memory after we finish the study of the algorithm and the state of the art, another one after we finish the performance analysis and the remaining after finishing the evaluation of the application.

Resources:

- Laptop Dell Latitude E5420
- Linux Ubuntu 13.10 Gnome
- LaTex

2.2.1 Resources

In this section we are going to list all the resources that appeared before. These resources are the ones needed to develop this project:

- Hardware
  - Laptop Dell Latitude E5420
  - Mare Nostrum III
- Software
  - Linux distributions

    - Ubuntu 13.10 Gnome at Laptop
    - SuSe Distribution 11 SP2 at Mare Nostrum III
  - LaTex
  - OmpSs
    - Mercurium compiler
    - Nanos++ runtime
  - Extrae
  - Paraver
  - GNU gprof

2.3 Budget

Budget monitoring

First of all, we compulsory need to have a method to control project budget in order to avoid a startling rise of it. In consequence, the budget would be updated at the end of each main task of the project.

This monitoring technique will allow us to maintain a real budget that will consider not only the real time we spend on each task but also the real indirect costs.

Human resources budget

Despite the fact that the project will be developed by only one person, this one will will be forced to do several roles along it, like a project manager, a programmer, etc. Also, Mare Nostrum requires a strict maintenance. This will be handled by BSC support team, it will be reflected as if we are paying them this service. The reason behind is to reflect the cost as if we are renting the working platform. In *Table 2.2*, an estimation of cost is provided.

Role	**Estimated hours**	**Estimated price per hour**	**Total estimated cost**
Project Manager	75	40,00 €	3.000,00 €
Programmer	396	25,00 €	9.900,00 €
Technical Support	50	25,00 €	1.250,00 €
TOTAL	N/A	N/A	**11.450,00 €**

Table 2.2: Human resources budget

Hardware budget

In order to study, design, implement and check our application as well as to execute it in order to obtain traces, we will need a set of hardware. This includes a laptop, Mare Nostrum computing nodes and some extra equipment. You will find an estimation cost of the different hardware equipment at *Table 2.3*, that table includes their service life as well as their depreciations[1].

Two Mare Nostrum nodes have been added to the hardware budget as we will need at least that number of nodes to be able to compile and execute the application in an isolated way. This means that one node will be used to edit and compile the application and other one just to execute the application using the job scheduler that Mare Nostrum offers. Each node is composed by the following hardware:

- Intel Xeon E5-2670 8-core at 2.6 GHz - 1.480,98€
- 8x4GB DDR3-1600 DIMMS - 360,00€
- HDD IBM 500GB SATA 3.5" - 240,00€
- Mellanox ConnectX-3 Dual Port QDR/FDR10 Mezz Card - 950,00€
- 2 Gigabit Ethernet network cards - 100,00€

Product	**Price**	**Units**	**Service life**	**Estimated residual value**	**Total estimated depreciation**
Dell Latitude E5420	1.445,00 €	1	5 years	250,00 €	79,67 €
Mare Nostrum node	3.130,98 €	2	5 years	500,00 €	175,40 €
Dell Monitor P2213	185,00 €	1	8 years	50,00 €	5,29 €
Lenovo Keyboard SK-8825	30,00 €	1	10 years	5,00 €	0,83 €
Logitech Mouse RX-250	8,00 €	1	10 years	2,00 €	0,20 €
TOTAL	**4.798,98 €**	N/A	N/A	N/A	**261,39 €**

Table 2.3: Hardware budget

[1]Depreciation can be obtained by the following formula:
$Depreciation = \frac{Value - ResidualValue}{ServiceLife}$

Software budget

All the software that is going to be used at this project is free of charge at the time this document is written. This includes Mercurium, Nanos++, Linux distributions, Extrae, Paraver, GNU gprof and LaTex.

But, as long as we are going to keep updated the project budget along the different stages of the project, if any of the application starts to require a fee, or we need to use a software that needs it, it will be reflected on the budget.

Indirect costs

In this section we are going to estimate the cost of having Mare Nostrum executing our application and obtaining the traces of it.

To do this we need to take into account the actual electrical fee. But nowadays this fee has a very volatile value, this means we will need to update this cost constantly to make our budget more realistic.

Despite of that, you will find an estimation of the indirect costs at *Table 2.4*. Please note it is also provided Mare Nostrum's power consumption taking into account the whole cluster, not only the nodes that will be used. The reason of this is that Mare Nostrum cannot be only powered on node by node, it must be powered on completely before using it.

Product	**Price**	**Units**	**Total estimated cost**
Mare Nostrum power consumption	0,133295 €/kWh	772.200 kWh	76.537,99 €
Two nodes power consumption	0,133295 €/kWh	505,37 kWh	67,36 €

Table 2.4: Indirect costs

Total budget

By adding all the budgets provided, we are able to calculate the total estimated cost of our project. All this information is gathered at *Table 2.5*.

Concept	Estimated cost
Human resources	11.450,00 €
Hardware	4.798,98 €
Software	0,00 €
Indirect costs	67,36 €
TOTAL	**16.316,34 €**

Table 2.5: Total budget

2.4 Sustainability

In this section we will discuss about the sustainability of the project. The sustainability of a project is the quality which made your project able to last for a long time after it is finished. In this line we are going to discuss about the market needs, quality and investment.

Market need

The x264 video enconder is widely used all over the world in several scenarios as the ones listed below:

- Video streaming services
- HDDVD and Blu-ray players
- Video conferencing systems
- etc.

As we can see, the impact of an improved version of the x264 encoder is likely to be very high. This means that community and companies could be interested in our application if we guarantee a minimum performance and quality.

Quality

As said above, both companies and community can be interested in our application, but this will only be a true statement if it achieves a minimum of quality. In our case quality will be granted via performance.

Taking into account that there are some other versions of the application that can be executed on a shared memory system as ours, quality is mandatory because our application will compete with the current parallel version.

Also, if we are able to ensure the quality, it will be more likely that our application will be used as a starting point to design new versions of the x264 video encoder as, for example, an OmpSs plus CUDA or MPI version, thus improving the performance of the application.

Investment

As well as market needs are strictly dependant on the application's quality, the investment is also dependant on it because nobody is going to invest on its improvement if the application doesn't work as it is expected to.

The investment has an extremely importance because it will lead to many opportunities to improve the application since you will have the resources needed to continue developing the application. As we said before, it is feasible to write a version of the application using OmpSs plus CUDA or MPI, but we need resources for this, and we cannot have resources without investment.

Chapter 3

x264 application

The x264 application is a free software library for encoding video streams into H.264/MPEG-4 Part 10 compression format. It is released under the terms of the GNU General Public License.

During this sections we are going to explain the compression format the application encodes and how is actually doing it. Afterwards, we will talk about PARSEC benchmark suite because the x264 application is one of the benchmarks which PARSEC is composed and because this project is part of a bigger one which consists on porting all the PARSEC benchmark Suite to OmpSs programming model.

3.1 H.264/MPEG-4 Part 10 video compression format

H.264/MPEG-4 Part 10 or AVC (Advanced Video Coding) is a video compression format that is currently one of the most commonly used formats for the recording, compression, and distribution of video content, like for example High Definition films or video streaming services. The final drafting work on the first version of the standard was completed in May 2003, several extensions of its capabilities have been added in new editions.

H.264/MPEG-4 AVC is a block-oriented motion-compensation-based video compression standard developed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC JTC1 Moving Picture Experts Group (MPEG). The project partnership effort is known as the Joint Video Team (JVT). The ITU-T H.264 standard and the ISO/IEC MPEG-4 AVC standard (formally, ISO/IEC 14496-10 – MPEG-4 Part 10, Advanced Video Coding) are maintained at the same time so they have identical technical content.

H.264 is perhaps best known as being one of the video encoding standards for Blu-ray Discs. Actually, all Blu-ray Disc players must be able to decode H.264. It is also widely used by streaming internet sources, such as videos from Vimeo and the iTunes Store. It is also used in web applications like Adobe Flash Player and Microsoft Silverlight as well as in several High Definition Television

broadcasts over terrestrial. For example, DVB-T (Digital Video Broadcasting - Terrestrial), which is used in Europe, Greenland, Russia, Australia and some areas from Africa, Asia and South America uses H.264 video compression format.

H.264 is mainly used for lossy compression, even though the amount of loss may sometimes be imperceptible. Furthermore, it is possible in some exceptional cases to create a truly lossless video file using H.264 video encoder.

Features

H.264/AVC/MPEG-4 Part 10 contains unique features that allow it to compress video in a more effective way than older video compression formats and is also able to provide more flexibility for using it into quite different network environments. Some of those features are the following:

- Multi-picture inter-picture prediction including the following features:
  - Variable block-size motion compensation (VBSMC) with block sizes as large as 16×16 and as small as 4×4, enabling precise segmentation of moving regions.
  - The ability to use multiple motion vectors per macroblock.
  - The ability to use any macroblock type in B-frames, including I-macroblocks, resulting in much more efficient encoding when using B-frames.
  - Weighted prediction, allowing an encoder to specify the use of a scaling and offset when performing motion compensation, and providing a significant benefit in performance in special cases—such as fade-to-black, fade-in, and cross-fade transitions. This includes implicit weighted prediction for B-frames, and explicit weighted prediction for P-frames.
- Spatial prediction from the edges of neighbouring blocks for intra coding.
- Lossless macroblock coding features including:
  - A lossless "PCM macroblock" representation mode in which video data samples are represented directly, allowing perfect representation of specific regions.
  - An enhanced lossless macroblock representation mode allowing perfect representation of specific regions while using fewer bits than the PCM mode.
- Flexible interlaced-scan video coding features, including:
  - Macroblock-adaptive frame-field (MBAFF) coding, using a macroblock pair structure for pictures coded as frames, allowing 16×16 macroblocks in field mode.
- New transform design features, including:

  - Adaptive encoder selection between the 4×4 and 8×8 transform block sizes for the integer transform operation.
  - A secondary Hadamard transform to obtain even more compression in smooth regions.

- A quantization design including:
  - Logarithmic step size control for easier bit rate management by encoder.
  - Frequency-customized quantization scaling matrices selected by the encoder.

- An in-loop de-blocking filter that helps preventing blocking artifacts, resulting in better visual appearance and compression efficiency.

- An entropy coding design including:
  - Context-adaptive binary arithmetic coding (CABAC), an algorithm to losslessly compress syntax elements in the video stream knowing the probabilities of syntax elements in a given context.
  - Context-adaptive variable-length coding (CAVLC), which is a lower-complexity alternative to CABAC.
  - A common simple and highly structured variable length coding (VLC) technique for many of the syntax elements not coded by CABAC or CAVLC.

- Loss resilience features including:
  - A Network Abstraction Layer (NAL) definition allowing the same video syntax to be used in many network environments.
  - Flexible macroblock ordering (FMO), also known as slice groups, and arbitrary slice ordering (ASO), which are techniques for restructuring the ordering of the representation of the fundamental regions (macroblocks) in pictures.
  - Data partitioning (DP), a feature providing the ability to separate more important and less important syntax elements into different packets of data.
  - Redundant slices (RS), an error/loss robustness feature.
  - Frame numbering, a feature that allows the creation of "sub-sequences", enabling temporal scalability by optional inclusion of extra pictures between other pictures.

- Switching slices, called SP and SI slices, allowing an encoder to direct a decoder to jump into an ongoing video stream for such purposes as video streaming bit rate switching and "trick mode" operation.

- Supplemental enhancement information (SEI) and video usability information (VUI), which are extra information that can be inserted into the bitstream to enhance the use of the video for a wide variety of purposes.

- Auxiliary pictures, which can be used for such purposes as alpha compositing.
- Support of monochrome (4:0:0), 4:2:0, 4:2:2, and 4:4:4 chroma sub-sampling (depending on the selected profile).
- Support of sample bit depth precision ranging from 8 to 14 bits per sample.
- The ability to encode individual color planes as distinct pictures with their own slice structures, macroblock modes, motion vectors, etc.
- Picture order count, a feature that serves to keep the ordering of the pictures and the values of samples in the decoded pictures isolated from timing information.

All of these features, along with others, makes H.264 one of the best compression formats in several cases. For example, H.264 can often perform better than MPEG-2 obtaining the same quality at half of the bit rate, especially on high bit rates and high resolution video files.

As other ISO/IEC MPEG video standards, H.264 has a reference software implementation that can be downloaded for free. The main purpose of this implementation is to give examples of H.264 features rather than being used daily.

Finally, H.264 has several profiles. A profile is a set of features which can be implemented or not within the video encoder application. This mean that not all the features must be supported in every video encoder or decoder implementation. In our case, x264 is one the applications that implements most of the features, but not all of them actually.

3.2 x264 video encoder algorithm

x264 application follows the same main algorithm as almost all the H.264 video encoders. Actually, differences between several H.264 encoders are usually number of H.264 features implemented or not.

So, the main idea when encoding H.264 video codec is to avoid encoding the maximum number of frames. Three different type of frames exist in order to do that: I, P and B frames. See Figure 3.1 as a example of this.

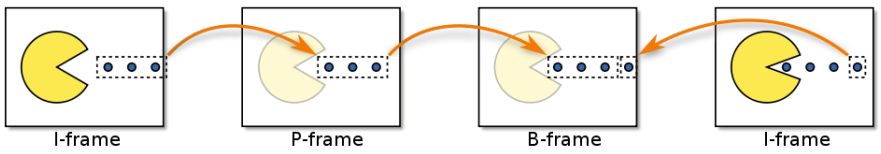

Figure 3.1: I, P and B frames

I frames

An I frame is an Intra-coded picture, in effect a fully specified picture, like a conventional static image file. P frames and B frames hold only part of the image information, so they need less space to store than an I frame and thus improve video compression rates.

P frames

A P frame (Predicted picture) holds only the changes in the image from the previous frame. For example, in a scene where a car moves across a stationary background, only the car's movements need to be encoded. The encoder does not need to store the unchanging background pixels in the P frame, thus saving space. P frames are also known as delta frames.

B frames

A B frame (Bi-predictive picture) saves even more space by using differences between the current frame and both the preceding and following frames to specify its content.

The x264 application works at macroblock level thus. This means each frame is split in several sub-frames, allowing the application to choose which type of frame use for every segment of the frame. This feature increase the compression of the final video stream as well as the complexity of the code, but of course is worth the effort.

Once the macroblock is analysed the application decides the type (I, P or B) of it. Depending on this decision the encoding process will be different. In case the macroblock is P or B type, then the encoding process cannot start until the reference macroblock or macroblocks are encoded. Refer to Figure 3.2 in order to see a diagram of the algorithm.

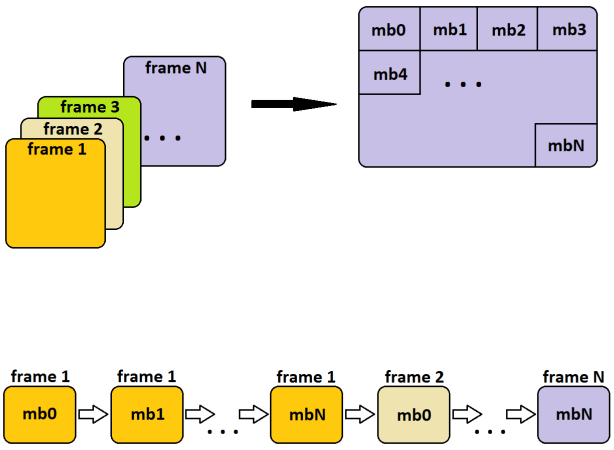

Figure 3.2: x264 algorithm pipeline

3.3 The PARSEC benchmark Suite

The PARSEC benchmark Suite is currently developed at the Princeton University. It aims to be a reference benchmark suite for Chip Multiprocessors (CMPs) since they have become the most widespread kind of general purpose processors.

There a lot of benchmark suites actually, but almost all of them are focused on high performance computing workloads. But of course, evaluating the performance of general purpose processor by using a benchmark designed for high performance computing systems is not a good metric. For this same reason PARSEC Benchmark Suite was developed, to give computer architects and chip designers information about future applications, making tomorrow's applications available today.

Key features

PARSEC differs from other benchmark suites in the following ways:

Multithreaded
: Nowadays serial programs are the most common, but they are not useful to test multiprocessor machines. PARSEC is one of a few benchmark suites that are parallel.

Emerging Workloads
: PARSEC benchmark suite includes emerging workloads which are likely to become widespread application in the near future. The goal of this suite is to provide a set of application that could be typical in a few years.

Diverse
: PARSEC includes a huge variety of different applications. Latest version includes 13 different workloads, trying to be as representative as possible.

Not HPC focused
: Parallel programs are very common in the domain of High Performance Computing, but are just a small subset of the whole parallel applications. So it is very important to be able to apply parallelization techniques into all application domains and the aim of PARSEC is to be prepared for this.

Research
: This suite was mainly developed for research but it can also be used for measuring performance on real machines, but its original purpose is insight, not numbers.

Chapter 4

Working environment set-up

4.1 OmpSs programming model

OmpSs is a programming model currently developed at Barcelona Supercomputing center which aims to extend OpenMP with some new directives in order to add support for asynchronous parallelism and heterogeneity. One of OmpSs capabilities is to execute regions of code in a parallel way even if they are not explicitly declared as parallel with respect other regions. This can be done because OmpSs parallelism declarations are made by using data dependencies for each task. This means, a section of code (i.e. an OmpSs task) will be queued in pool of ready tasks only when its data dependencies are satisfied. Once in that queue the tasks will be executed once a thread will be available.

About heterogeneity, OmpSs provides support for executing parts of the application at the GPU just adding an OpenMP directive to the code. So one can see that one of the most important features is the simpleness. This translates into clear-cut code which is easy to maintain afterwards. It allow the developer to forget about complex functions and only think into the algorithm he want to implement and nothing else.

As mentioned before, OmpSs creates a pool of tasks. Those tasks will be bind to any available thread. In order to know which thread is available a pool of threads is created as well. This means that all threads are created at the initialization of the application, being one of those threads the master one which is responsible of the user serial code execution. The rest of threads are called worker threads. This worker threads can become master ones in case they create more tasks, this means nesting is also possible within OmpSs programming model.

As we already said, each parallel region of code is defined as a task in OmpSs programming model. Each task has its own dependencies, which are declared using *in*, *out* and *inout* directives. These directives are set by the user and using it only serves to help the runtime when constructing the graph dependency in order to be able to execute each task just when the data is available. Use of each

directive is self-evident, *in* directive tells the runtime the task needs to read the data indicated at *in* clause, *out* directive tells the runtime the task will modify the data indicated at *out* clause, finally, *inout* directive tells the runtime the task will need to read and write the data indicated at *inout* directive. There are some other directives like *concurrent* which tells the runtime that the assuring properly access to data will be in charge of the programmer.

Figure 4.1 shows how OmpSs programming model works. We can see that from the source code, compiling it with Mercurium, which can use several native compilers for serial execution, an executable will be obtained. Mercurium also uses Nanos++ runtime libraries at linking time. Finally, the executable will use dynamic libraries from Nanos++ in order to be executed in a parallel way.

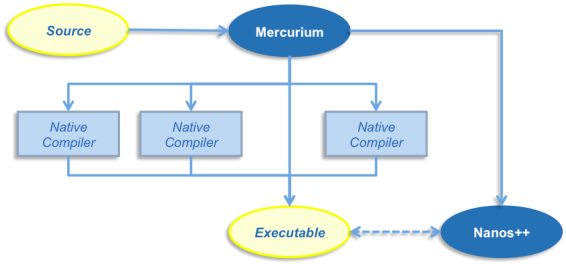

Figure 4.1: OmpSs implementation

4.1.1 Mercurium compiler

Mercurium is a source-to-source compiler which currently supports C and C++ languages. Mercurium is maninly used with Nanos runtime, providing support for OmpSs and OpenMP programming model. Actually, it also supports some other programming models (e.g. Chapel) and compiler transformations (e.g. Cell Superscalar and Distributed Shared Memory). All of this features are included in form of plugins written in C++ and dynamically loaded by the compiler depending on what configuration the programmer chooses.

Mercurium compiler has stable support to compile several programming languages. Table 4.1 shows current supported ones.

Mercurium	**Language**
mcc	C
mcxx	C++
mnvcc	CUDA and C
mnvcxx	CUDA and C++
mfc	Fortran
oclmcc	OpenCL and C
oclmcxx	OpenCL and C++

Table 4.1: Mercurium compilation options for back-end compilation choice

4.1.2 Nanos++ runtime

Nanos++ is a runtime library (RTL) mainly developed in C++ which aims to give support for parallel environment. Its main objective is to be in charge of dependencies generated by OmpSs directives. It has also support for OpenMP and Chapel. OmpSs tasks are implement by Nanos++ with user-level threads when possible, in this line x86, x86.64, ia64, ppc32 and ppc64 are supported.

One of the most important features of Nanos++ runtime is that the programmer does not need to deal with complex usage, he only needs to compile the source code of its applications linking Nanos++ runtime libraries to that executable. This mean the executable can be executed the same way you can execute a serial application for example.

Nanos++ structure is mainly an idle loop in which idle threads are waiting to the be called by the master thread. Once this happen, master thread will assign work to the worker thread, the way this is actually done consist on creating a work descriptor for the task using a thread also represented in Nanos++ runtime library.

Finally, Nanos++ package can be compiled in four versions: performance, debug, instrumentation and instrumentation-debug.

4.2 Paraver

Paraver is a development tool currently developed at Barcelona Supercomputing Center and aims to respond to the need of having a way to visualize in a graphical view the behaviour of an application in order to obtain an analysis of it. One of its features is to be able to read the traces generated via Extrae.

Those traces can be analysed using Paraver. Of course, what is showed and how can be configured by the developer in order to allow him to debug or to know what is really happening inside his application.

Another important feature is that its trace format has no semantics, this translates into support for new programming models with no cost. Furthermore, metrics are programmed within the tool. To compute them, the tool offers large set of time functions, filters and mechanisms to combine two time lines. This means the developer can obtain a huge number of metrics with the available data provided by the trace. Of course, once views are configured, this configuration can be saved for using it later by only loading the configuration file.

Some other Paraver features are the support for:

- Detailed quantitative analysis of program performance
- Concurrent comparative analysis of several traces
- Customizable semantics of the visualized information
- Cooperative work, sharing views of the trace
- Building of derived metrics

4.3 Extrae

Extrae is a development tool currently developed at Barcelona Supercomputing Center which allows the developer to generate traces from the execution of its application. These traces can be analysed later using Paraver. This tool uses different mechanisms to obtain significant information of the execution like hardware counters or function calls from both user or system space. Extrae instrumentation package can instrument the following programming models:

- MPI
- OpenMP
- CUDA
- OpenCL
- pthreads
- OmpSs

All parallel programming models are supported in conjunction with MPI, except MPI of course.

Extrae configuration is made via an XML file, which contain which kind of counters, events or execution states will be gathered by the tool.

4.4 Mare Nostrum III

For this project we are going to use Mare Nostrum III, which is hosted at Capella building located at Campus Nord, for running, debugging and evaluating our application. This supercomputer is based on Intel SandyBridge processors, iDataPlex compute racks interconnected through an Infiniband network and runs a Linux Operating System.

Further information is provided at the following list:

- Peak Performance of 1.1 Petaflops
- 100.8 TB of main memory
- Homogeneous Nodes
  - 3,056 compute nodes
  - 2x Intel SandyBridge-EP E5-2670/1600 20M 8-core at 2.6 GHz
  - 8x4GB DDR3-1600 DIMMS (2GB/core)
- Heterogeneous Nodes
  - 42 heterogeneous compute nodes
  - 2x Intel SandyBridge-EP E5-2670/1600 20M 8-core at 2.6 GHz
  - 2x Xeon Phi 5110 P

  - 8x8GB DDR3-1600 DIMMS (4GB/core)
- 2 PB of disk storage
- Interconnection networks:
  - Infiniband FDR10
  - Gigabit Ethernet
- Operating System: Linux - SuSe Distribution

Chapter 5

x264 application evaluation

The x264 application provided with the PARSEC benchmark suite has two versions implemented. The first one is mean to be executed on a single processor with one thread, this means, in a serial way. The other one uses POSIX threads (i.e. pthreads) in order to reach a new level of performance via encoding more than one frame at a time.

The objective of this project is to design and implement a new parallel version of the x264 video encoder using OmpSs programming model. In order to do it, we need to evaluate the performance of current implementations to know what can be done and what not in order to improve the performance via parallelization.

We are going to start evaluation the serial version of the application, obtaining a complete profile of it as well as computing the execution time and the frame per second processed for several different input files.

The input file is quite important in this application because the main idea, as it was explained in section 3, is to create a virtual pipeline where all the macroblocks of every frame are enqueued and are processed taking into account potential dependencies between each. Is for this reason as well as the resolution of the frame (i.e. the number of the pixels to be encoded) that scalability is not perfect.

The input files we are going to use are the same as PARSEC benchmark Suite provide. All the input files have been derived from the uncompressed version of the short film Elephants Dream. You can see the differences between each different input at Table 5.1.
Each input file is mean to be used in different situation. For example, simtest and simdev are intended to use them as test files to check if the application works properly or not. About simsmall, simmedium and simlarge, are used to be chosen when one wants to check scalability depending on the number of frames, as we are going to do later when evaluating pthreads version. Finally, native version allows us to check the real performance of the application providing a very close real case input.

Name	Resolution	Number of frames
simtest	32x18	1
simdev	64x36	3
simsmall	640x360	8
simmedium	640x360	32
simlarge	640x360	128
native	1920x1080	512

Table 5.1: Different input files for x264 application

5.1 Serial version

The serial version of the application encodes every macroblock in a serial way. This means that one frame cannot be encoded before the encoding process of the previous one is finished, no matter the type of it.

So in this version we will see that the time of the encoding process depends on the resolution of each frame, which affects on how many time the encode process of a frame last long, and the number of the frames to be encoded.

5.1.1 Profiling

During this section we are going to obtain a profile of the serial version of the x264 video encoder when encoding simsmall, simmedium, simlarge and native input files. These profiles will allow us to know which parts of the code are executed more often, this mean which functions are consuming more execution time in our application.

For this we are going to use GNU gprof. GNU gprof is a performance analysis tool for Unix applications. It uses a hybrid of instrumentation and sampling and was created as extended version of the older prof tool. Unlike prof, gprof is capable of limited call graph collecting and printing.

Starting from profile obtained via GNU gprof we will get also a call graph using a script called Gprof2Dot which parses the output of gprof in order to generate a DOT file (i.e. a plain text file describing a graph) which can be processed by Graphviz to obtain an image with a function call graph.

Figure 5.1 shows the function call graph of an execution of the serial version of x264 using the simsmall input set. As one can notice looking at the graph, the application does not call more than one function before it starts x264_slice_write one. From there, it calls three different functions one of them consuming 87% of the execution time. This function is x264_macroblock_analyse, which is called 8280 times. Note also that x264_slice_write is called 9 times when we are only encoding 8 frames. This is because it is possible to call this function before reading the input file, in this case the function will do an early exit in order to try to be executed later.

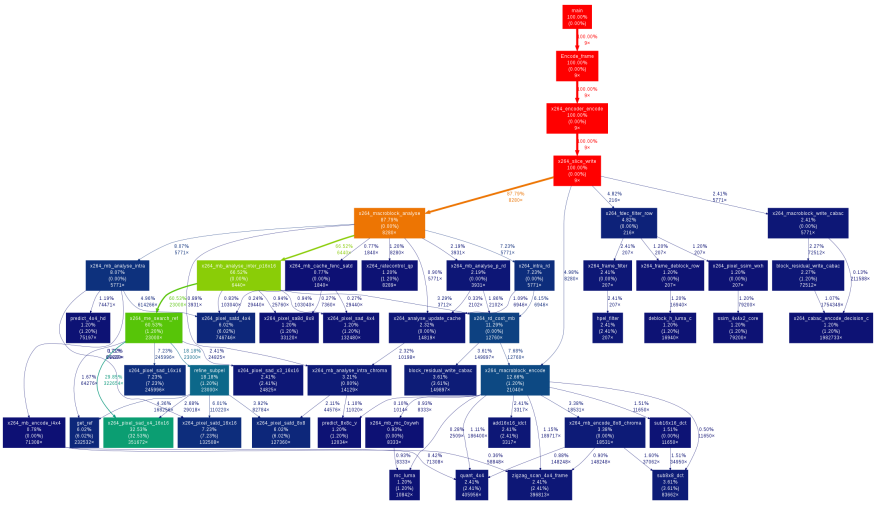

Figure 5.1: Call graph using simsmall input set

Now, we can apply Amdahl's Law in order to know which is the maximum speed up we can achieve if we optimize only x264_macroblock_analyse function. Note the use of infinite in order to compute the maximum reachable speed up.

$$S(B) = \frac{1}{(1-B) + \frac{B}{\infty}}$$

Where S is the speed up achieved and B the portion of the application which is optimized. If we use the percentage of the execution time used by x264_macroblock_analyse, which is a 87.79% we can see the maximum speed up is the following:

$$S(0.8779) = \frac{1}{(1-0.8779)} = 8.19$$

This means that even optimizing the application in a level that the cost of the most execution time consuming function would be zero (i.e. infinite improvement), the speed up we could reach would be around 8 in case of the simsmall input set.

At Figure 5.2 we can see the call graph of an execution using the simmedium input set. It is mostly the same as the one using the simsmall input set. The most important here is to note that the number of the calls to x264_slice_write and x264_macroblock_analyse has increased due to the rising of the number of frames that must be encoded.

In this line we can see that the number of calls to x264_slice_write has increased from 9 to 33 (now we are encoding 32 frames) while the number of calls to x264_macroblock_analyse rises from 8280 to 30360. This means that the calls to the most consuming functions have been increased by a factor of around 4. Note also the percentage of the execution time consumed by x264_macroblock_analyse is virtually the same as before so the theoretical speed up would be nearly the same in this case.

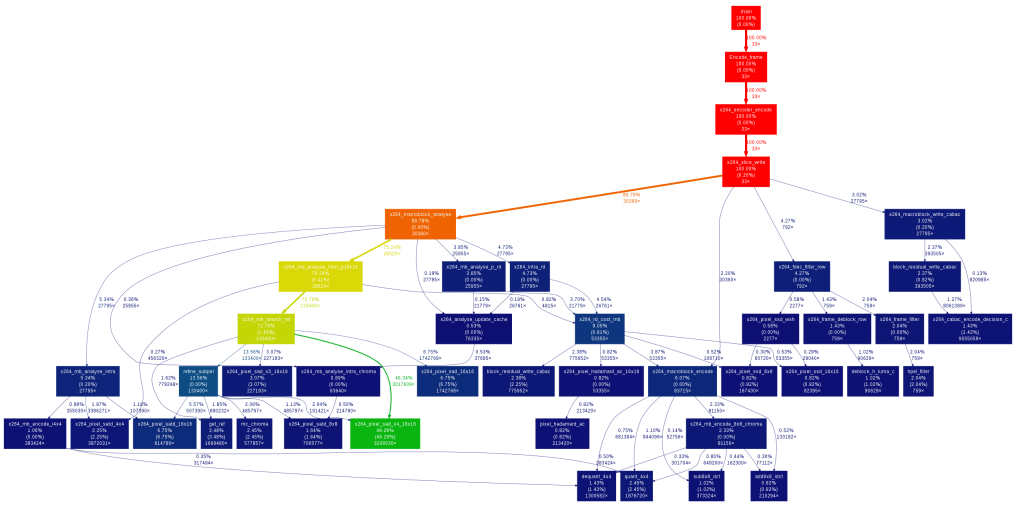

Figure 5.2: Call graph using simmedium input set

For simlarge input set we can see the same behaviour as with simsmall and simmedium, but increasing the number of calls. See Figure 5.3 in order to see it. The percentages are likely the same as well as the number of calls is increased by a factor of around 4 again.

Finally, native input set, which its call graph can be seen at Figure 5.4, repeats the same behaviour. Nevertheless, is important to note how the percentage of time executing analysis (which will decide what kind of macroblock we are working with at that moment) grows up at the same time we increase the number of frames. The reason of that behaviour is that these functions are the ones that need more compute.

5.1.2 Performance evaluation

First, let me explain that we computed both total execution time and frames per second. The difference between each is the frames per second does not take into account the overhead of having a runtime neither synchronization. In the case of the serial version both are very close to each because we don't have overhead caused by runtime nor synchronization.

Table 5.2 shows the execution time and the frames per second for simsmall, simmedium, simlarge and native input set. Note that both measures are obtained computing the median of 5 executions.
We can see execution time rises as we increase the number of frames to be encoded but is not the same for frames per second value. Looking at execution time, we can see the reason for this increase is self-evident. At the same resolution (simsmall, simmedium and simlarge) the increase is more or less lineal. When using native input set, the execution time goes sky high due to the change on the resolution of the video, this can be noticed also at frame per second value, which is the lowest of all the different input set.

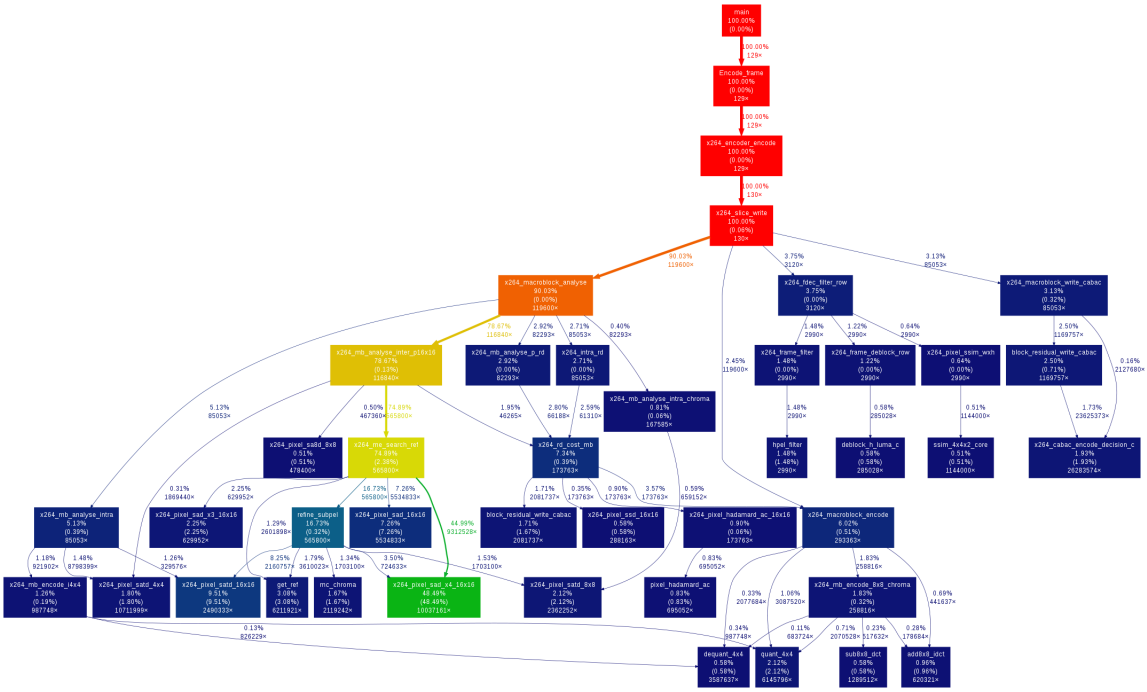

Figure 5.3: Call graph using simlarge input set

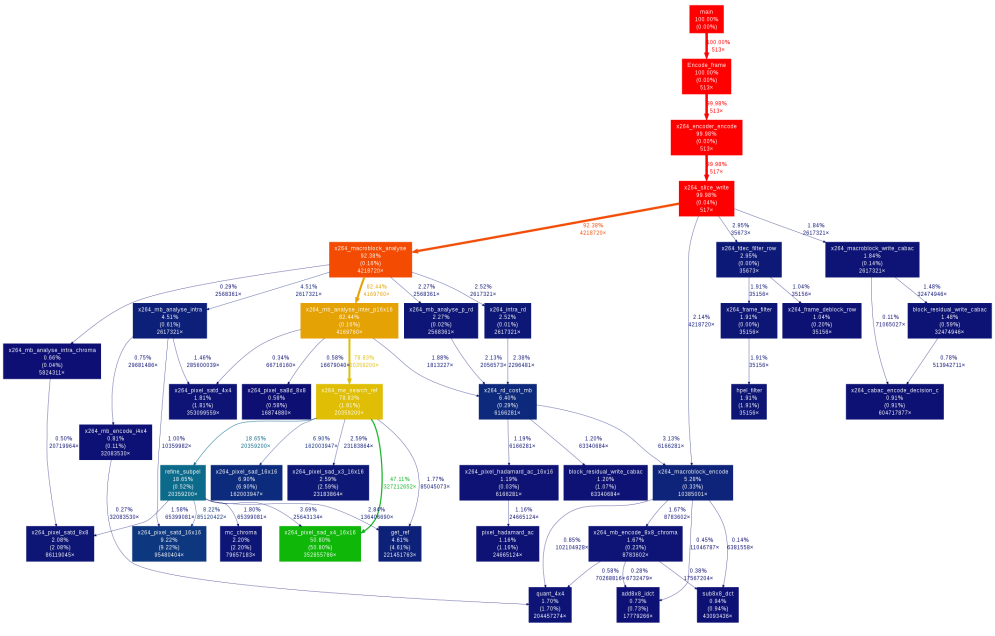

Figure 5.4: Call graph using native input set

Input Set	Execution Time (seconds)	Frames per second (fps)
simsmall	0.8252	9.715
simmedium	5.4505	5.875
simlarge	17.4204	7.345
native	612.8471	0.835

Table 5.2: Execution time and frames per second of serial version using different input set

About frames per second metric, we can see that there is a lot of difference when using different input set. In the case of native, the reason is the resolution of each frame. At HD resolution (i.e. 1920x1080 pixels, a total of 2073600 pixels) the amount of pixels is greater than at a resolution of 640x360 pixels (i.e. 230400), actually, in native we have 9 times more pixels than at simsmall, simmedium or simlarge. This is the reason why the frames per second is too low compared with the other input set. About the rest (i.e. simsmall, simmedium and simlarge) differences are produced because in case of P type frames, it is not mandatory to encode every pixel. So this is the reason to have a different values when encoding frames of equal resolution.

5.2 pthreads version

The x264 application provided with PARSEC benchmark Suite also implements a parallel version using POSIX threads. This version performs very nice but has some scalability issues when encoding video files with low resolutions as we will see during the following sections.

5.2.1 Profiling

As we did before, we are going to start discovering how the applications works. In this case we are going to focus on the usage of the cores. This mean we are going to study how many cores we have running at the same time during the whole execution. This is very important for us since the bigger the number of cores are running at the same time the better will be the final performance. We can check again Amdahl's Law taking into account the parallelization made by using pthreads.

$$S(B,N) = \frac{1}{B + \frac{1}{N} * (1 - B)}$$

Where S is the speed up we achieve, B the fraction of the algorithm that is strictly serial and N the number of threads using at the parallel region.

In our application, pthreads parallelize the x264_slices_write function, so, if we look at native profiling of the serial version, we could see that using 8 and 16 threads the potential speed up is supposed to be the following:

$$S(0.9998, 8) = \frac{1}{1 - 0.9998 + \frac{1}{8} * (0.9998)} = 7.9888$$

$$S(0.9998, 16) = \frac{1}{1 - 0.9998 + \frac{1}{16} * (0.9998)} = 15.9521$$

With these results one can see that the potential speed up we can achieve seems to be approximately the same as the number of threads we will use. But, in order to do a deeper analysis we are going to look at the trace of the application using Extrae to obtain the trace files and Paraver to visualize them.

Looking at Figure 5.5 we can see a trace which shows how many cores are used during the whole execution of the application using simlarge input set and 8 threads. We can see that the 8 threads are not running at the same time in a big portion of the application. This reason of this is the size of each frame. Since it is not as bigger as it is at native input set, computation does not require more time, so we spent more time synchronizing threads because of the potential dependencies between different frames. This is a big issue for performance. See also Table 5.3 to see the same information in a table.

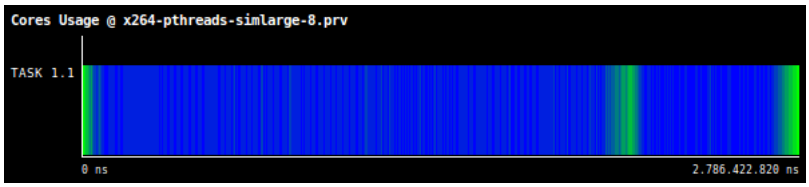

Figure 5.5: Trace showing as a gradient color the thread usage within x264 execution with simlarge input set and 8 threads. Green zones are the ones with only 1 thread, the more blue it comes the more threads are running, with a maximum of 8.

Number of threads	**Execution Time (percentage)**
1	1.05
2	1.29
3	1.97
4	2.20
5	2.63
6	2.21
7	21.02
8	66.45

Table 5.3: Percentage of execution time where a certain number of threads are running at the same time using simlarge input set and 8 threads

So now we are going to look at the same figure and table but using 16 threads. Now we can see the that the synchronizing time is more percentage than before. The reason now is again the resolution. Each thread, if we are encoding a P or B frame, has to wait to the threads which are encoding the frame that is a dependency of the other. This means that we need to wait a lot of times during the encoding of the frame, affecting the final performance because we

are not using the maximum number of threads the maximum possible time. See at Figure 5.6 to see the trace and Table 5.4 to see a table containing the same information Figure 5.6 but in a numerical way.

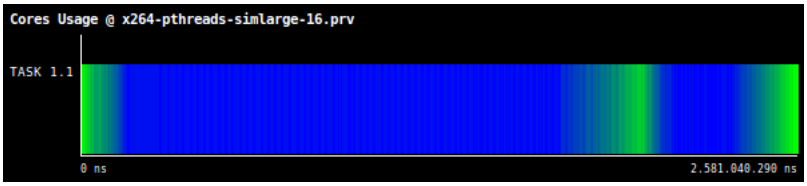

Figure 5.6: Trace showing as a gradient color the thread usage within x264 execution with simlarge input set and 16 threads. Green zones are the ones with only 1 thread, the more blue it comes the more threads are running, with a maximum of 16.

Number of threads	**Execution Time (percentage)**
1	1.18
2	0.99
3	1.28
4	2.13
5	2.13
6	2.35
7	2.72
8	2.36
9	2.47
10	2.32
11	2.64
12	2.44
13	3.06
14	2.32
15	10.87
16	57.99

Table 5.4: Percentage of execution time where a certain number of threads are running at the same time using simlarge input set and 16 threads

Now, we are going to see the same but for native input set. At Figure 5.7 we can see the trace showing how many threads are used by the x264 application in each moment. At Table 5.5 the information is showed in numerical way. We also provide Figure 5.8 and Table 5.6 for the same input set but using 16 threads. We can see that the usage of the threads is not as different as it was when using simlarge input set. The reason is the amount of pixels that need to be encoded. Nevertheless, we are going to spend more time doing computation, which can be done in a parallel way, and less time doing synchronization, which affects the performance.

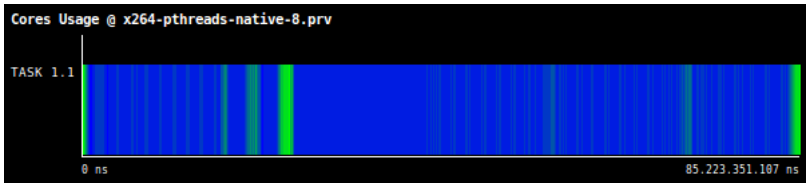

Figure 5.7: Trace showing as a gradient color the thread usage within x264 execution with native input set and 8 threads. Green zones are the ones with only 1 thread, the more blue it comes the more threads are running, with a maximum of 8.

Number of threads	**Execution Time (percentage)**
1	0.92
2	1.50
3	1.03
4	1.61
5	2.07
6	2.84
7	7.23
8	79.54

Table 5.5: Percentage of execution time where a certain number of threads are running at the same time using native input set and 8 threads

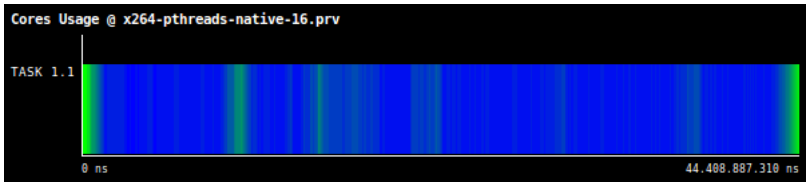

Figure 5.8: Trace showing as a gradient color the thread usage within x264 execution with native input set and 16 threads. Green zones are the ones with only 1 thread, the more blue it comes the more threads are running, with a maximum of 16.

Number of threads	Execution Time (percentage)
1	0.83
2	0.39
3	0.30
4	0.26
5	0.36
6	0.40
7	0.78
8	1.52
9	0.96
10	1.45
11	2.87
12	7.51
13	8.63
14	6.20
15	10.30
16	52.64

Table 5.6: Percentage of execution time where a certain number of threads are running at the same time using native input set and 16 threads

5.2.2 Performance evaluation

As we saw when performing the profile of this version, there are several issues which could potentially affect the final performance of the application. And they do as we can see at Figure 5.9 and Figure 5.10. We can see that for all input set except for native, scalability is not lineal.

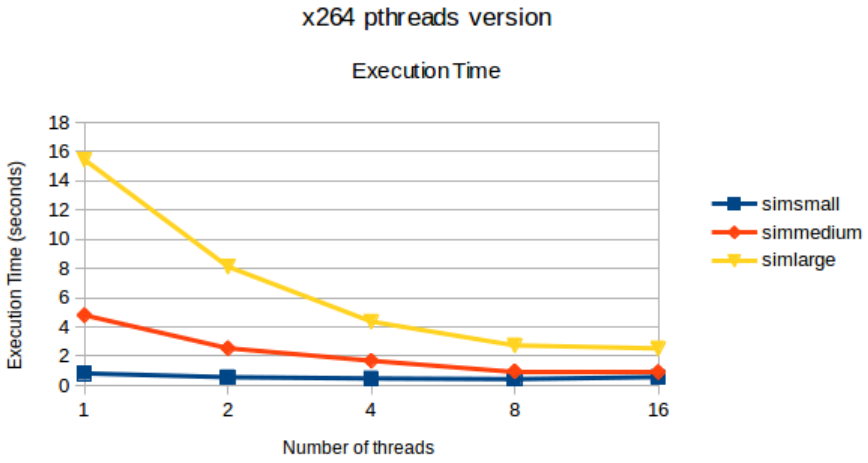

Figure 5.9: Execution time using pthreads version with simsmall, simmedium and simlarge input set and different number of threads

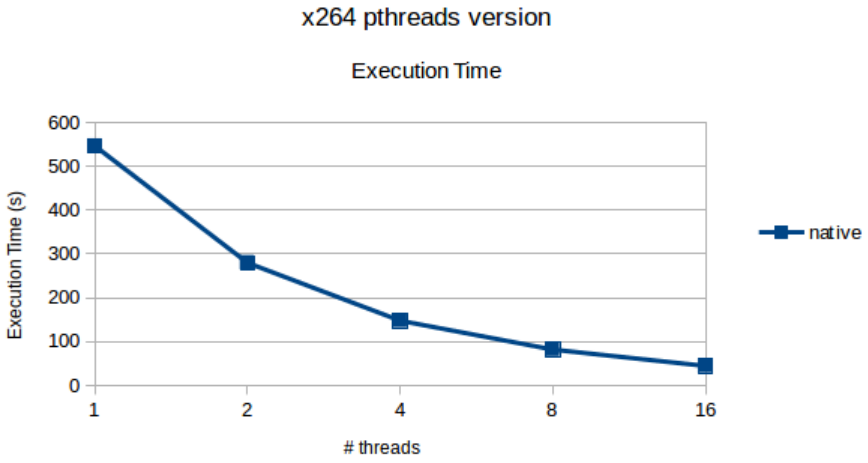

Figure 5.10: Execution time using pthreads version with native input set and different number of threads

At Chart 5.11 we could see a similar figure but using frames encoded per second as performance value. Again, we can see that for all input set except for native, scalability is not lineal.

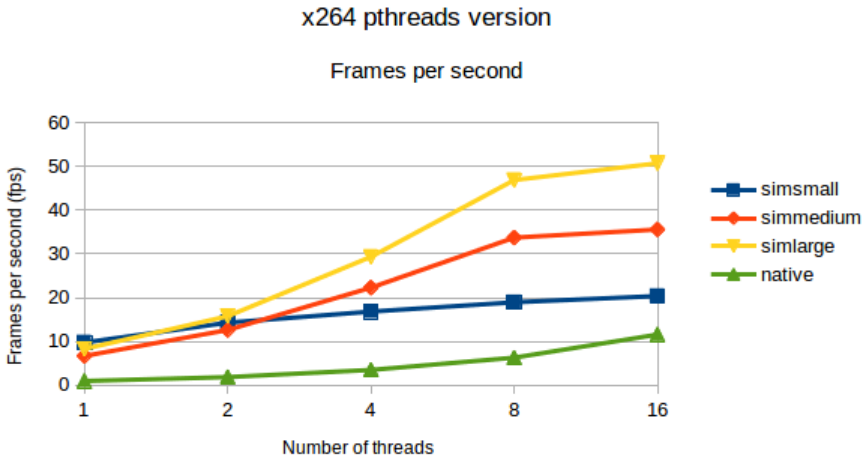

Figure 5.11: Frames per seconds using pthreads version with every input set and different number of threads

The performance we obtained is the expected for all the reasons we mention when studying the traces of the application. The main issue with the application is when we are not able to split each frame in order to be able to do the encoding

with all the threads at the same time. When this happens, some threads needs to wait to have its dependencies satisfied, affecting the final performance. So a better performance is expected when using native input set just because the amount of computation that has to be done is huge comparing with, for example, simlarge input set.

5.2.3 Scalability

Chart 5.12 shows the speed up achieved for several input sets using different number of threads. As it shows, the speed up when using native input set scales properly due to the percentage of the time spent in computation and not doing synchronization. For other input sets we can see the speed up stuck when reaching certain number of threads.

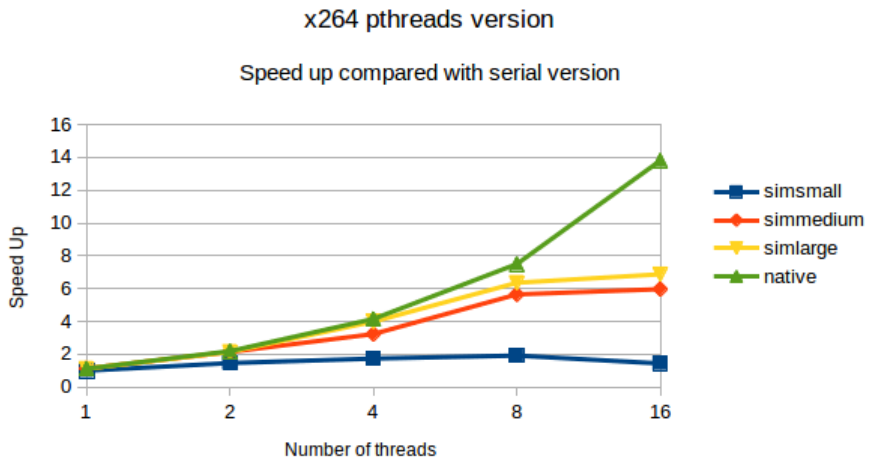

Figure 5.12: Speed up against serial version using pthreads with native input set and different number of threads

In some cases the problem is the time spent at doing synchronization between threads, at others the reason is more self-evident. If we are using 8 threads to encode a video with 3 frames and each frame is encoded only by one thread, then we are not able to use the 8 threads at all. This is the case of simsmall input set for example, and for simmedium when using 16 threads (remember simmedium is a video composed by 32 frames only). Even though, the main reason is the size of the frames for all of them.

Chapter 6

Porting the x264 application to OmpSs

During this chapter we are going to study which are the dependencies we have at the x264 application that can be a problem for reaching the degree of parallelization we want to achieve, which is the same as the pthreads version at least.

Afterwards, we will discuss about how we are going to use OmpSs and its capabilities trying to avoid those dependencies. The idea is to try to use these OmpSs features to design a better parallel version than pthreads one.

Finally, we will show how this design is implemented at the code of the application. The code is a bit large, so we are not going to show the whole source code during this section. Only the parts of the code that need to be modified will be showed.

6.1 Dependencies

As we already explained, the x264 video encoder creates a virtual pipeline of frames to be encoded. The idea is to be able to encode the highest number of frames at the same time, but there is not that easy due to the different types of frames, more exactly P and B frames. These kind of frames are the ones which take profit of the pixels that are already encoded in another frame or frames in order to avoid re-encoding them again. This translates directly into dependencies.

Figure 6.1 shows an example of the virtual pipeline the application makes. There are also marked dependencies between frames. Actually, a P frame doesn't need to wait until the whole previous frame is encoded. One idea is to wait until a part of the frame is already computed and then start the encoding process of the frame. Potentially, we will not have to wait again, but it will not be the common case, so when reaching some point of the encoding process of the frame, it will be mandatory to check if the next portion of the previous frame is already

computed in order to continue. With B frames it will be worst, just because we need to wait also for the following frame.

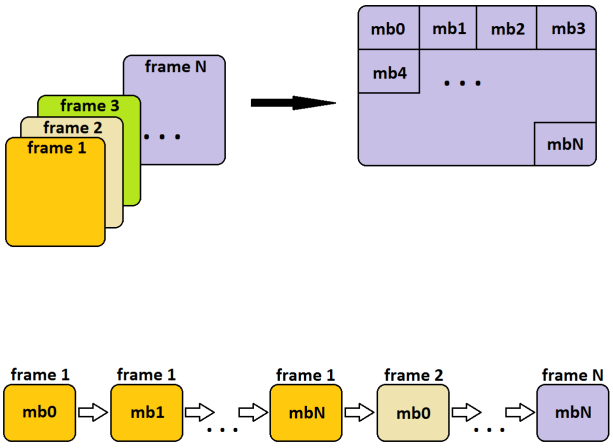

Figure 6.1: x264 algorithm pipeline

This is the main dependency of the application. As we mentioned at pthreads version performance evaluation section, a lack of scalability is expected when encoding videos with a low resolution, but only if there are a lot of P or B frames. If not (i.e. all frames are I type) then scalability is expected to be more or less lineal, but in that case the frames per second metric will be worse since it will be mandatory to encode all the pixels, work that is not done when encoding a P or B frame, in those cases, only some pixels are really encoded. Of course a perfect scalability is not possible just because we are not executing in a parallel way the whole application, only a portion of it.

We have another dependency which is not that critical. As one can guess, input video file must be read before start encoding it. Actually, the read process is split in many parts as the amount of frames has the video file, this means we will read the video file frame per frame. This means we cannot start encoding a frame before reading it. This dependency is not a big issue just because the time the application spends reading the frame is negligible compared to the encoding time of each frame. Figure 6.2 shows the same pipeline as before but adding the read process for each frame.

6.2 Design of the OmpSs version

At first we are going to start summarizing the last section. We must deal with two dependencies which we need to break them with our design. These are:

P frame depends on I or P frame
: One P frame cannot be encoded before the preceding I or P frame is not encoded.

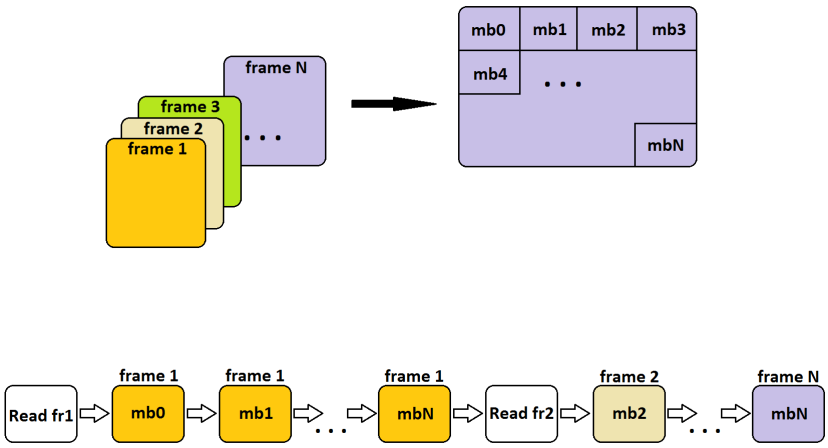

Figure 6.2: x264 algorithm pipeline including reads

B frame depends on previous I or P frame and following I frame
One B frame cannot be encoded before the preceding I or P frame and the following I frame are not encoded.

Frame depends on the read of the frame
One frame cannot be encoded before it has been read from the input file.

The last one is quite self-evident actually, we depend on the input file to be read before start processing (i.e. encoding) the data stream. About the first and the second kind of dependency, this inter-frame dependency is likely the same, we need to wait until some data is processed before starting to process the one we have.

It is mandatory to break those dependencies because is the only way to achieve a degree of performance we want, at least equal to the current implementation which uses pthreads. At least equal just because one of the objectives of the project is to prove that OmpSs is also valid for non high performance computing workloads. This can be proven improving the performance of this application over the current parallel implementation or achieving the same performance but reducing the complexity of the code.

The following sections will explain how each dependency is broken.

A frame depends on the read of the the frame itself

This dependency is not very hard to break actually. The main reason is that there are a lot of work to do between reading of the frame and encoding it like preparing data structures for example. Taking this into account, the idea is to read the frame and continue executing the application until we do not reach the start of the encoding process at the same time. Figure 6.3 shows this idea.

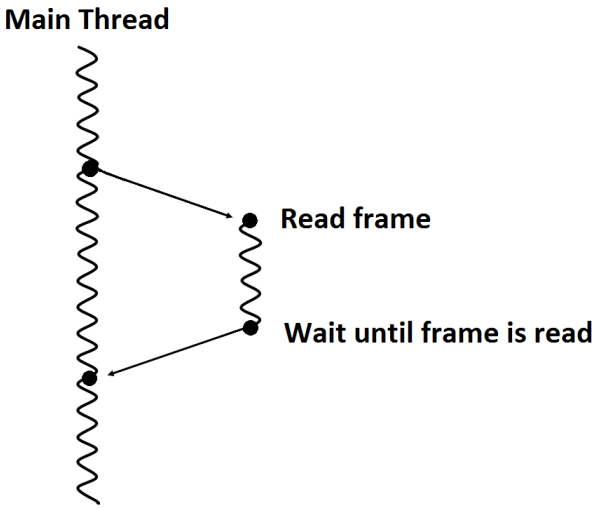

Figure 6.3: Scheme of how reads are designed to be performed

P frame depends on I or P frame

The following dependency relation is about two frames. We are going to suppose that both frames are already read and analysed. We are not working with frames actually but with macroblocks, which are a portion of the frame, but we are going to imagine that each frame is composed by macroblocks of the same type.

At this point we have an I or P frame and another P frame which depends on the first one. Now we are going to design what happens if there is an I frame and a P frame just to avoid complexity.

Both frames cannot be encoded at the same time, so we need to encode the I frame first, which is the one that entered first into the virtual pipeline. The idea is to avoid waiting for the whole encoding process of the I frame to avoid having threads without any work to do. So we are going to think how this dependency work.

Looking at Figure 6.4 we can see not all the pixels of the P frame depend on all the pixels of the I frame, there are only a few of them that actually do. So is worth to start the encoding process of the P frame just when some point of the I frame is already encoded. The idea is split the frame into several portions in order to do this synchronization within different frames several times in order to avoid reaching pixels of the frame that have dependencies.

One could think about marking which pixels are the ones that depend on others. The problem with this design is that those dependencies are movement of the pixels, not change on them, so one would need to store transformations and displacements for every dependent pixel, which can be potentially more than a three quarts of the total pixels in the frame. Another issue is the need of checking every pixel, which can affect the final performance.

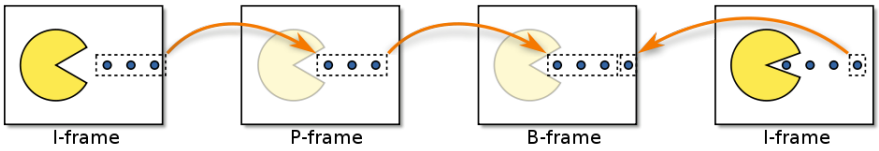

Figure 6.4: I, P and B frames

So the idea is to design a sort of waterfall algorithm. I mean, imagine we have a succession of frames starting from one I frame followed by a finite number of P frames. Then, the idea is to encode, for example, a 10% of the I frame until start encoding the first P frame we will have on the virtual pipeline.

The second frame, which is the first P frame, will eventually encode a 10% of its picture. At this point, this encoding process will wait until the I frame will reach a 20% of the encoding process, but the third frame queued (i.e. the second P frame which depends on the first P frame) will start its encoding process.

After encoding again, we will have 3 frames with a 30, 20 and 10% of the total of the encoding process done. Again, a fourth frame could start because depends on the one which is currently on a 10% of the encoding process.

This behaviour could continue forever, but one can see that if we use a 10% of the image as synchronization value then we will be able to have only ten frames being encoding at the same time. If we think about parallelism, the idea is to set this percentage taking into account how many threads we are using. This way is not worth to set 10% if we have only 2 threads for example, the best choose would be 50% in order to avoid the maximum synchronization possible. Figure 6.5 shows how the virtual pipeline will be executed when using the explained algorithm.

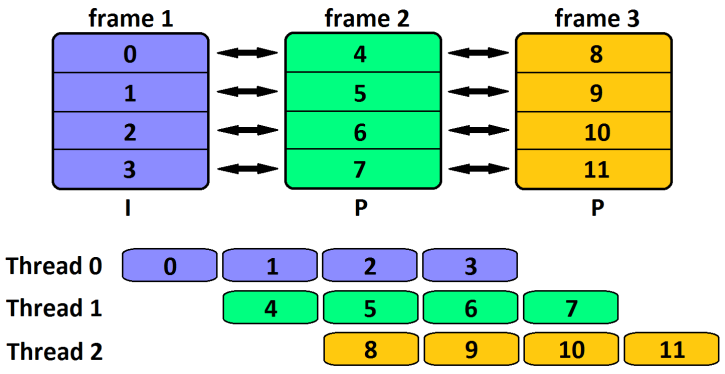

Figure 6.5: x264 parallelism algorithm example

B frame depends on previous I or P frame and the following I frame

This dependency is quite similar as the one before, but the frame depends not only on the previous frame but also on the following one. The nice part is that we can be sure that the following frame will be and I one, i.e. a frame with no dependencies.

So the idea is to start encoding the previous I or P frame and the following I frame at the same time. The B frame will not start its encoding process until a percentage of the other two frames is achieved. As before, we will have a sort of checkpoints where the B frame will must wait until the other two frames get that percentage of encoded pixels.

The only modification is the order in which are encoded the frames actually. This translates into a little modification in the virtual pipeline. Now, the frames that must be encoded before the B frame must be encoded first, so we need to move them within the virtual pipeline. Figure 6.6 shows how the actual pipeline is modified.

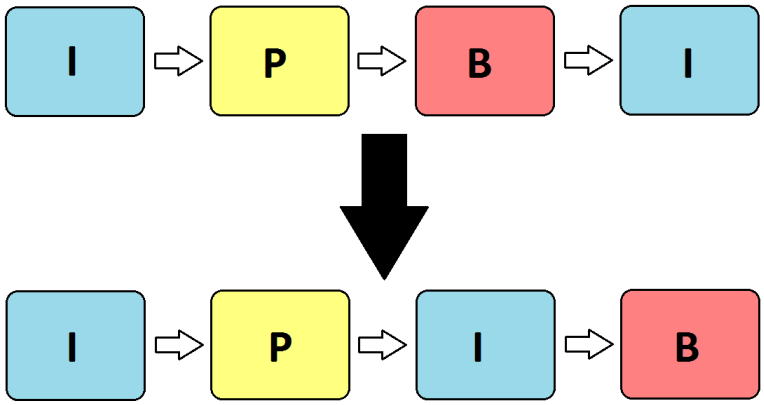

Figure 6.6: Virtual pipeline obtained after reordering frames when encoding a B frame

After this modification of the pipeline the dependency turns into something very similar to the dependency between a P frame and its predecessor. The only difference between that P frame and the B frame is that when checking if we can continue encoding it is mandatory to check both frames which the frame is dependent and not start the encoding process until both dependencies are satisfied.

6.3 Implementation of the OmpSs version

In this section we are going to explain how the design we already explained has been implemented. As there are already implemented a parallel version which takes care about the creation of several data structures to isolate each encoding process properly, some of the code of the current pthread version was reused. Of course, any call to any pthreads function will not be executed never within our OmpSs implementation, we just reused data structures actually.

Modified source code will be provided for each modification. Of course not the whole code of the application will be showed here, so only affected ones will be provided. This section will be split into several ones as well in order to isolate properly each part of the design that was implemented.

In this line, the main parts will be the ones for the reading and the one for the encoding process itself. Another section will explain how the application installation scripts are modified in order to allow the user to generate the OmpSs version implemented. First and second parts are dependent between, but we thought that splitting them will give a more understandable approach to the reader.

During the first section we are going to explain how it was implemented the process of reading each frame. For this read we are creating a task for reading each frame. Furthermore, more child tasks can be created from this one taking into account some special cases that will be explained later.

The second section will explain how the encoding process was parallelized. As before, a task is created for each encoding process, this actually means for each frame. Since each frame could have dependencies with others, we need to synchronize each task with the ones which are encoding the frame which the first one is dependent. How this synchronization is done will be explained in a more detailed way at the specific section for this dependency case.

The last section will explain how configuration and installation scripts were modified in order to use Mercurium compiler to generate the OmpSs version executable.

6.3.1 Reading the frame

Reading a frame needs to be done before starting the encoding process of the frame. In order to read the frame, we create a new task each time a frame needs to be read, this is once per frame.

The processing of each frame, which includes the read of it is included at the following code:

Listing 6.1: Encoding loop

```
1 for( i_frame = 0, i_file = 0; b_ctrl_c == 0 && (i_frame <
      i_frame_total || i_frame_total == 0); )
2 {
```

```
3       int res = 0;
4   #ifndef HAVE_OMPSS
5       res = p_read_frame( &pic, opt->hin, i_frame + opt->i_seek );
6   #else
7       #pragma omp task out(pic) label(p_read_frame)
8       res = p_read_frame( &pic, opt->hin, i_frame + opt->i_seek );
9       #pragma omp taskwait on(pic)
10  #endif
11      if(res)
12          break;  //frame not readed properly, exiting
13
14      pic.i_pts = (int64_t)i_frame * param->i_fps_den;
15
16      if( opt->qpfile )
17          parse_qpfile( opt, &pic, i_frame + opt->i_seek );
18      else
19      {
20          /* Do not force any parameters */
21          pic.i_type = X264_TYPE_AUTO;
22          pic.i_qpplus1 = 0;
23      }
24
25      i_file += Encode_frame( h, opt->hout, &pic );
26
27      i_frame++;
28
29      /* update status line (up to 1000 times per input file) */
30      if( opt->b_progress && i_frame % i_update_interval == 0 )
31      {
32          int64_t i_elapsed = x264_mdate() - i_start;
33          double fps = i_elapsed > 0 ? i_frame * 1000000. / i_elapsed
                : 0;
34          double bitrate = (double) i_file * 8 * param->i_fps_num / (
                (double) param->i_fps_den * i_frame * 1000 );
35          if( i_frame_total )
36          {
37              int eta = i_elapsed * (i_frame_total - i_frame) / ((
                    int64_t)i_frame * 1000000);
38              sprintf( buf, "x264 [%.1f%%] %d/%d frames, %.2f fps,
                    %.2f kb/s, eta %d:%02d:%02d",
39                       100. * i_frame / i_frame_total, i_frame,
                             i_frame_total, fps, bitrate,
40                       eta/3600, (eta/60)%60, eta%60 );
41          }
42          else
43          {
44              sprintf( buf, "x264 %d frames: %.2f fps, %.2f kb/s",
                    i_frame, fps, bitrate );
45          }
46          fprintf( stderr, "%s  \r", buf+5 );
47          SetConsoleTitle( buf );
48          fflush( stderr ); // needed in windows
49      }
50  }
```

We can see that for each frame of the video file the application reads a frame (line 8), performs several checks and set up the frame to encode it and finally calls Encode_frame function at line 25 which actually performs the encoding of the frame. Afterwards there is a conditional statement at line 30 which outputs information about the encoding process of the frame if the user has set the verbose flag when launching the application. The initialization of the data

structure which represents the frame (i.e pic variable) is not showed because is out of the loop.

About the modifications that have been done within the loop are all contained within line 4 to 10. As one could see, we used a conditional group in order to check if HAVE_OMPSS is defined or not. This macro is set up at compilation time in case we want to compile OmpSs version of the application. If it is defined, code from lines 7 to 9 will be executed and line 5 if not. Line 5 was the original code.

At line 7 the task is declared using OmpSs syntax. We are declaring one output and setting the label of the task in order to improve traces visualizations in near future. After declaring the task, we wait until this task is finished.

It is not a lack of performance waiting at this moment just because the time we spend on reading from the input file is negligible. About why we are creating the task when we are waiting until is finished is just because we need to assure some conditions within the task and creating it at this moment will reduce complexity afterwards. Furthermore, more threads can be created within the tasks, so it is not serial execution after all.

Now, let me explain something about p_read_frame function. This function is actually a variable that contains a pointer to the function which is called. This was made that way because we have several implementations of the same function depending on which version is compiled. So for example, the function which reads a frame is different if we are using serial version of the application or pthreads version. In case of the OmpSs, we are using the same as used in pthreads version but modifying it. The following code shows the function used to read a frame.

Listing 6.2: Read frame function

```
1  static void read_frame_thread_int( thread_input_arg_t *i )
2  {
3      i->status = i->h->p_read_frame( i->pic, i->h->p_handle, i->
           i_frame );
4  }
5
6  int read_frame_thread( x264_picture_t *p_pic, hnd_t handle, int
       i_frame )
7  {
8      thread_input_t *h = handle;
9      UNUSED void *stuff;
10     int ret = 0;
11
12     if( h->next_frame >= 0 )
13     {
14 #ifndef HAVE_OMPSS
15         x264_pthread_join( h->tid, &stuff );
16 #else
17         #pragma omp taskwait
18 #endif
19         ret |= h->next_args->status;
20         h->in_progress = 0;
21     }
```

```
22
23      if( h->next_frame == i_frame )
24      {
25          XCHG( x264_picture_t, *p_pic, h->pic );
26      }
27      else
28      {
29          ret |= h->p_read_frame( p_pic, h->p_handle, i_frame );
30      }
31
32      if( !h->frame_total || i_frame+1 < h->frame_total )
33      {
34          h->next_frame =
35          h->next_args->i_frame = i_frame+1;
36          h->next_args->pic = &h->pic;
37  #ifndef HAVE_OMPSS
38          x264_pthread_create( &h->tid, NULL, (void*)
                read_frame_thread_int, h->next_args );
39  #else
40          #pragma omp task label(read_frame_thread_int)
41          read_frame_thread_int(h->next_args);
42  #endif
43          h->in_progress = 1;
44      }
45      else
46          h->next_frame = -1;
47
48      return ret;
49  }
```

Looking at the code you can see there are actually two functions. The first one (i.e. read_frame_thread_int actually call the function which it is called. This means that function only serves for calling recursively your own function in a clear way.

The other function called read_frame_thread is the one that actually performs the read of the frame. This function tries to read all the frames actually. The idea is to start reading all the frame and afterwards we continue the execution of the application. In order to do it we create a task for each frame, doing a parallel read of the input file in order to skip a lack of performance.

The interesting portion of the code is compressed between lines 37 and 42. As you can see we are using again a conditional group in order to split the original implementation which uses pthreads from ours which uses OmpSs. In case HAVE_OMPSS macro is defined, we create a new task which will read the next frame.

6.3.2 Encoding the frame

This is the biggest modification that has been done in the code in order to create our OmpSs version. First we added the creation of the main task which actually encodes the frame. The function which actually do that is called x264_slices_write. This function is called for another one called x264_encoder_encode which prepares a lot of data structures that are mandatory to encode the frame.

About x264_slices_write function, is the one who calls another ones to encode the frame. Those functions are the ones that split the frame into macroblocks,

analyse them in order to choose the best type (i.e. I, P or B type) for each of them and perform the encode of each macroblock of the frame. The following code snippet shows the creation of the task and the code which finalizes the encoding process of one frame.

Listing 6.3: Creation of the task which encodes a frame

```
1  int     x264_encoder_encode( x264_t *h,
2                               x264_nal_t **pp_nal, int *pi_nal,
3                               x264_picture_t *pic_in,
4                               x264_picture_t *pic_out )
5  {
6
7      ...
8
9      /* Write frame */
10     if( h->param.i_threads > 1 )
11     {
12 #ifndef HAVE_OMPSS
13         x264_pthread_create( &h->thread_handle, NULL, (void*)
               x264_slices_write, h );
14 #else
15         #pragma omp task out(*h) label(x264_slices_write)
16         x264_slices_write(h);
17 #endif
18         h->b_thread_active = 1;
19     }
20     else {
21         x264_slices_write( h );
22     }
23
24     ...
25
26     x264_encoder_frame_end( thread_oldest, thread_current, pp_nal,
           pi_nal, pic_out );
27     return 0;
28 }
```

We can see the task is created from a function called x264_encoder_encode as it was explained before. The modified part is the enclosed in lines from 12 to 17. We use a conditional group to check if we are using OmpSs or not and if so, we create a task which will execute x264_slices_write. This task will have an output value and a custom label which will allow us to improve the trace files we will generate later.

Finally, within the same function, we will close the frame encoder, this means we will clean up all the data that is not going to be used again to avoid memory leaks. The code of the function x264_encoder_frame_end is the following.

Listing 6.4: Close the frame encoder

```
1  static void x264_encoder_frame_end( x264_t *h, x264_t *
       thread_current,
2                                      x264_nal_t **pp_nal, int *
                                           pi_nal,
3                                      x264_picture_t *pic_out )
4  {
5      int i, i_list;
```

```
6       char psz_message[80];
7
8       if( h->b_thread_active )
9       {
10  #ifndef HAVE_OMPSS
11          x264_pthread_join( h->thread_handle, NULL );
12  #else
13          #pragma omp taskwait on(*h)
14  #endif
15          h->b_thread_active = 0;
16      }
17      if( !h->out.i_nal )
18      {
19          pic_out->i_type = X264_TYPE_AUTO;
20          return;
21      }
22
23      ...
24
25  }
```

This function waits until the task just created to encode a frame is already finish. This is useful because we are constantly checking if a frame is already encoded in order to do some operations that needs this condition. So it is the only way to be sure that one frame is already encoded. Allowing us clean up data structures or reusing them again.

Now we are going to take a look inside x264_slices_write function. The following code shows the content of it and important code of x264_slice_write function, which is called from the other one:

Listing 6.5: x264_slices_write function

```
1   static int x264_slices_write( x264_t *h )
2   {
3       int i_frame_size;
4
5       ...
6
7       x264_stack_align( x264_slice_write, h );
8       i_frame_size = h->out.nal[h->out.i_nal-1].i_payload;
9
10      ...
11
12      h->out.i_frame_size = i_frame_size;
13      return 0;
14  }
15
16  static void x264_slice_write( x264_t *h )
17  {
18      int i_skip;
19      int mb_xy, i_mb_x, i_mb_y;
20      int i, i_list, i_ref;
21
22      ...
23
24      while( (mb_xy = i_mb_x + i_mb_y * h->sps->i_mb_width) < h->sh.
            i_last_mb )
25      {
```

```
26          int mb_spos = bs_pos(&h->out.bs) + x264_cabac_pos(&h->cabac
                );
27
28          if( i_mb_x == 0 ) {
29              x264_fdec_filter_row( h, i_mb_y );
30          }
31
32          /* load cache */
33          x264_macroblock_cache_load( h, i_mb_x, i_mb_y );
34
35          /* analyse parameters
36           * Slice I: choose I_4x4 or I_16x16 mode
37           * Slice P: choose between using P mode or intra (4x4 or 16
                 x16)
38           * */
39          x264_macroblock_analyse( h );
40
41          /* encode this macroblock -> be careful it can change the
                mb type to P_SKIP if needed */
42          x264_macroblock_encode( h );
43
44          x264_bitstream_check_buffer( h );
45
46          ...
47
48          /* save cache */
49          x264_macroblock_cache_save( h );
50
51          ...
52
53          x264_ratecontrol_mb( h, bs_pos(&h->out.bs) + x264_cabac_pos
                (&h->cabac) - mb_spos );
54
55          ...
56
57      }
58
59      ...
60
61      x264_fdec_filter_row( h, h->sps->i_mb_height );
62
63      ...
64
65  }
```

At line 7 you can see the function x264_slice_write is called. Actually this call is the most important part of the function x264_slices_write for us because the rest of the code is for updating the display in case we were using on fly visualization feature of the application.

So looking at the x264_slice_write function, we can see that after doing other stuff like preparing variables, at some point the execution will enter in a loop statement. This loop is the one that allows to do the same operations for each macroblock of the frame. Looking into the loop statement we will see some calls to functions that are important to explain how the video encoder works.

The very first one is x264_fdec_filter_row. This function basically updates the status of the encoding process for this frame in order to allow other tasks (which will be encoding other frames) to be able to check if they can continue its

execution (i.e. synchronization). So the first thing we need to do is update the current status of the encoding process, afterwards we will load the macroblock and then we will analyse this macroblock via x264_macroblock_analyse function. It is during the execution of this function where we will check the status of the frame or frames which the current frame depends. In case the status of those frames are not enough to start analysing the macroblock, we will wait until those frames encode enough macroblocks and then we will continue analysing the frame.

There are some other functions which are the ones that encode the macroblock and saving it and updating some data that will be used afterwards. Of course, before exiting x264_slice_write it is mandatory to update the status of the encoding process of this frame. This is done by calling again the x264_fdec_filter_row function.

Now we are going to look into x264_fdec_filter_row function in order to see how the status is updated. It is important because there are some code in there which has been modified in order to make version OmpSs work. The imporant portion of function are the following.

Listing 6.6: x264_fdec_filter_row function

```
1  static void x264_fdec_filter_row( x264_t *h, int mb_y )
2  {
3
4      ...
5
6      if( h->param.i_threads > 1 && h->fdec->b_kept_as_ref )
7      {
8  #ifndef HAVE_OMPSS
9          x264_frame_cond_broadcast( h->fdec, mb_y*16 + (b_end ?
               10000 : -(X264_THREAD_HEIGHT << h->sh.b_mbaff)) );
10 #else
11         #pragma omp atomic
12         h->fdec->i_lines_completed = mb_y*16 + (b_end ? 10000 : -(
               X264_THREAD_HEIGHT << h->sh.b_mbaff));
13 #endif
14     }
15
16     ...
17
18 }
```

As one can see, this update is only done when the number of threads is more than one. This means that if we are running the application only with one thread, we will not do any kind of synchronization just because is not needed at all.

Besides this, a conditional group has been used in order to be able to obtain our OmpSs version. We can see that the update of the status consists only on updating one variable which will be accessed afterwards by other tasks. Because of that, this access must be protected in order to avoid race conditions. We are using an OmpSs atomic statement to protect the access to the variable.

And now, we are going to see how this variable is consulted at x264_macroblock_analyse function. The access to that variable is actually performed in a function called

from x264_macroblock_analyse which name is x264_mb_analyse_init. Both functions are showed at the following code snippet. Only important portions of code are provided.

Listing 6.7: x264_macroblock_analyse and x264_mb_analyse_init functions

```
1  void x264_macroblock_analyse( x264_t *h )
2  {
3      x264_mb_analysis_t analysis;
4      int i_cost = COST_MAX;
5      int i;
6
7      ...
8
9      x264_mb_analyse_init( h, &analysis, h->mb.i_qp );
10
11     /*---------------------------- Do the analysis
           ----------------------------*/
12
13     ...
14
15 }
16
17 static void x264_mb_analyse_init( x264_t *h, x264_mb_analysis_t *a,
       int i_qp )
18 {
19     int i = h->param.analyse.i_subpel_refine - (h->sh.i_type ==
           SLICE_TYPE_B);
20
21     ...
22
23     /* I: Intra part */
24
25     ...
26
27     /* II: Inter part P/B frame */
28     if( h->sh.i_type != SLICE_TYPE_I )
29     {
30         int i, j;
31         int i_fmv_range = 4 * h->param.analyse.i_mv_range;
32         int i_fpel_border = 5; // umh unconditional radius
33         int i_spel_border = 8; // 1.5 for subpel_satd, 1.5 for
               subpel_rd, 2 for bime, round up
34
35         /* Calculate max allowed MV range */
36
37         ...
38
39         if( h->mb.i_mb_x == 0)
40         {
41             int mb_y = h->mb.i_mb_y >> h->sh.b_mbaff;
42             int mb_height = h->sps->i_mb_height >> h->sh.b_mbaff;
43             int thread_mvy_range = i_fmv_range;
44
45             if( h->param.i_threads > 1 )
46             {
47                 int pix_y = (h->mb.i_mb_y | h->mb.b_interlaced) *
                       16;
48                 int thresh = pix_y + h->param.analyse.
                       i_mv_range_thread;
49
50                 for( i = (h->sh.i_type == SLICE_TYPE_B); i >= 0; i
```

```
                    -- )
51                  {
52                      x264_frame_t **fref = i ? h->fref1 : h->fref0;
53                      int i_ref = i ? h->i_ref1 : h->i_ref0;
54                      for( j=0; j<i_ref; j++ )
55                      {
56  #ifndef HAVE_OMPSS
57                          x264_frame_cond_wait( fref[j], thresh );
58  #else
59                          while(fref[j]->i_lines_completed < thresh)
                            {
60                          #pragma omp taskwait
61                          }
62  #endif
63                          thread_mvy_range = X264_MIN(
                                thread_mvy_range, fref[j]->
                                i_lines_completed - pix_y );
64                      }
65                  }
66
67                  ...
68
69              }
70
71              ...
72
73          }
74  #undef CLIP_FMV
75
76              ...
77
78          }
79
80          ...
81
82      }
83  }
```

At line 9 we can see that x264_mb_analyse_init is invoked and it will start executing. Then, the important part is contained within lines 56 and 62. We can see in there another conditional group, and in case HAVE_OMPSS is defined, we will check if the status of the frame which we have a dependency is enough to allows us to continue. If not, we will perform a taskwait in order to check it later. Once the check will be positive we will continue the analysis of the macroblock. This check will be performed for every frame which we will have a dependency.

I want to say that this is not the best option to implement this synchronization within OmpSs programming model. The main problem here is the need to synchronize different tasks which cannot be seen by each other. OmpSs programming model is focused on establishing input and output dependencies, but in our application we will need to synchronize with an state of one variable, so it is not that easy to create this dependency using OmpSs statements.

One possible solution will be using a sentinel (i.e. a variable that does nothing but creating fake dependency relationships) but we decide not doing it. The reason is that using sentinels in OmpSs is not suggested since you could face issues if an improvement of the runtime is done.

Finally, the last modification into the application was adding a taskwait when finalizing the encoding process for each frame. This is mandatory since you need to update some parameters and free some resources in order to use it afterwards. The code of the function that performs that is the following.

Listing 6.8: x264_encoder_frame_end function

```
1  static void x264_encoder_frame_end( x264_t *h, x264_t *
       thread_current,
2                                      x264_nal_t **pp_nal, int *
                                           pi_nal,
3                                      x264_picture_t *pic_out )
4  {
5      int i, i_list;
6      char psz_message[80];
7
8      if( h->b_thread_active )
9      {
10 #ifndef HAVE_OMPSS
11         x264_pthread_join( h->thread_handle, NULL );
12 #else
13         #pragma omp taskwait on(*h)
14 #endif
15         h->b_thread_active = 0;
16     }
17     if( !h->out.i_nal )
18     {
19         pic_out->i_type = X264_TYPE_AUTO;
20         return;
21     }
22
23     x264_frame_push_unused( thread_current, h->fenc );
24
25     ...
26
27     /* ---------------------- Update encoder state
           ------------------------- */
28
29     ...
30
31     /* ---------------------- Compute/Print statistics
           --------------------- */
32
33     ...
34
35 }
```

As one can see in lines from 10 to 14, a conditional group is declared and, in case the macro HAVE_OMPSS is defined, we just execute a taskwait on the **h** variable. This means that we will only wait for the task x264_slices_write of this encoding process to be finished. More work is performed within the function, but the only important part for us is the one we already mentioned.

6.3.3 Configuring and installing OmpSs version

The x264 video encoder provides a configuration script which configures all the needed variables and programs in order to compile the application. This allows the user to choose which version of the application wants to compile as well as enabling debugging information or not for example.

This script must be modified in order to use Mercurium compiler to obtain an OmpSs version of the application. Available options can be obtained using the command *./configure –help*, obtaining the following output. Note that the output showed here already contains our modifications.

Listing 6.9: configure_help

```
Usage: ./configure [options]

available options:

  --help                print this message
  --disable-avis-input      disables avisynth input (win32 only)
  --disable-mp4-output      disables mp4 output (using gpac)
  --disable-pthread     disables multithreaded encoding
  --enable-ompss            enable OmpSs version
  --enable-ompss-instrumentation    adds --instrumentation (--
      enable-ompss must be set)
  --disable-asm         disables assembly optimizations on x86
  --enable-debug            adds -g, doesn't strip
  --enable-gprof            adds -pg, doesn't strip
  --enable-visualize        enables visualization (X11 only)
  --enable-pic          build position-independent code
  --enable-shared           build libx264.so
  --extra-asflags=EASFLAGS  add EASFLAGS to ASFLAGS
  --extra-cflags=ECFLAGS        add ECFLAGS to CFLAGS
  --extra-ldflags=ELDFLAGS  add ELDFLAGS to LDFLAGS
  --host=HOST           build programs to run on HOST
```

You can see there are two options OmpSs related. One is –**enable-ompss** which will configure the application to be compiled using OmpSs, this is compiling the port we made during this project. The second one is –**enable-ompss-instrumentation**, setting this option will compile the application using OmpSs instrumented libraries. This is useful if one wants to get traces using Extrae or a dependency graph afterwards.

Now we will see which modifications were performed at *configure* script in order to implement the options showed. The following code snippet will show the modified portions of code.

Listing 6.10: configure

```

...

MCFLAGS=""
if test "$ompss_instr" = "yes"
then
    if test "$ompss" = "no"
    then
        ompss_instr="no"
    fi
fi
if test "$ompss" = "yes"
then
    CC="mcc"
    if test "$ompss_instr" = "yes"
    then
```

```
17          CFLAGS="--ompss --instrument $CFLAGS -DHAVE_OMPSS"
18          MCFLAGS="$CFLAGS"
19          LDFLAGS="$LDFLAGS"
20      else
21          CFLAGS="--ompss $CFLAGS -DHAVE_OMPSS"
22          MCFLAGS="$CFLAGS"
23          LDFLAGS="$LDFLAGS"
24      fi
25      if test "$gprof" = "yes"
26      then
27          echo "--enable-gprof incompatible with --enable-ompss"
28          exit 1
29      fi
30  fi
31
32  ...
33
34  echo "Platform:       $ARCH"
35  echo "System:         $SYS"
36  echo "asm:            $asm"
37  echo "avis input:     $avis_input"
38  echo "mp4 output:     $mp4_output"
39  echo "pthread:        $pthread"
40  echo "debug:          $debug"
41  echo "gprof:          $gprof"
42  echo "PIC:            $pic"
43  echo "shared:         $shared"
44  echo "visualize:      $vis"
45  echo "ompss:          $ompss"
46  echo "ompss instr:    $ompss_instr"
47  echo
48  echo "You can run 'make' or 'make fprofiled' now."
```

At the code we can see how we check if OmpSs related options are enabled and if so we set some variables like the compiler and some flags. Note also that in case we try to set both –**enable-ompss** and –**enable-gprof** the configuration will fail just because it will not be possible to compile the application afterwards.

Finally, information about what has been configured or not is printed out. This lines will provide the user information regarding the executable he will obtain if he executes make command afterwards.

Chapter 7

OmpSs version evaluation

During the following sections we will do an evaluation of the OmpSs version we already implemented. This evaluation will be done through profiles, trace files obtained from the execution of the application and measuring the execution time as well as the frames per second achieved. We will do also a study of the scalability of the application, discussing which are the scalability issues we faced.

Finally, we will compare our version which the ones that are already evaluated within this project (i.e. serial and pthreads version).

7.1 Profiling

First we are going to take a look into a trace file obtained by executing OmpSs version using simlarge input set with 8 and 16 threads. Figure 7.1 shows which task was executed in which moment. Color red is for x264_slices_write task and pink for p_read_frame. Color brown is for read_frame_thread_int. We can see that parallel part is only a small part of the whole execution. This is the reason why the scalability is not lineal when encoding a video file with a small resolution.

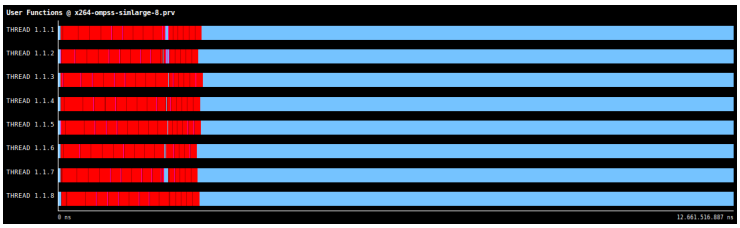

Figure 7.1: Trace file showing Tasks executed within OmpSs version with simlarge as input set and running on 8 threads

Now, looking at Figure 7.2 we can see the same as before but using 16 threads instead of 8. Colours used are the same as before. We can see that, now, the

encoding part (i.e. x264_slices_write) spends less time than before, but not the same for read_frame_thread_int function. The reason of this behaviour is that there are more synchronization than before.

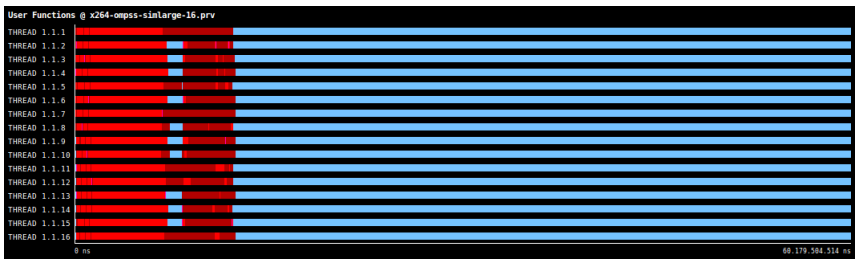

Figure 7.2: Trace file showing Tasks executed within OmpSs version with simlarge as input set and running on 16 threads

But looking at the traces generated when using native input set, we can see a quite different behaviour. So if we look at Figure 7.3 we can see such a difference comparing the trace with the one generated using simlarge input set. Here we can see that x264_slices_write (red color) spends a lot more time than before. We can see also that reading the frame (brown color) is also expensive, just because of the size of the input video file and the size of each frame.

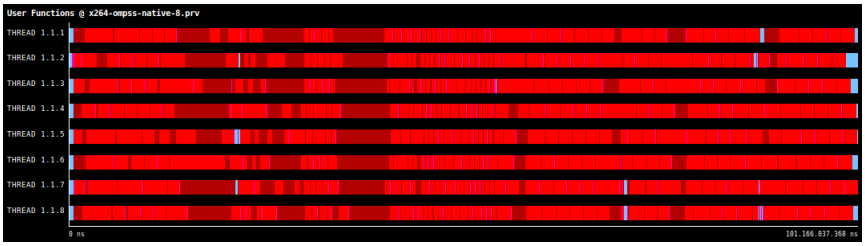

Figure 7.3: Trace file showing Tasks executed within OmpSs version with native as input set and running on 8 threads

Figure 7.4 shows a trace generated by the execution of the OmpSs version using native input set and 16 threads. It is straightforward to see that the behaviour is likely the same as with 8 threads. As computation is heavier than using simlarge, we can split better the encoding process of each frame in order to achieve a better parallelism degree which translates directly into more scalability.

7.2 Performance evaluation

This version has the same performance issues that pthreads one, but in some cases could perform a bit better though. Actually, when using native input set we could see a slightly better performance than in pthreads version.

In Figure 7.5 and Figure 7.6 we can see the execution time of OmpSs version for each input set (i.e. simsmall, simmedium, simlarge and native). We can

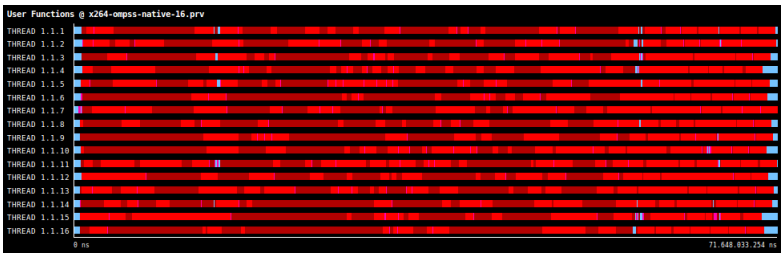

Figure 7.4: Trace file showing Tasks executed within OmpSs version with native as input set and running on 16 threads

see a similar behaviour as in pthreads version evaluation. For a small number of frames the execution time reduction is near to zero. The reason is that we cannot take profit of executing in more than one core. But for simlarge and native, we can see a bigger reduction in execution time. The reason is that simlarge and native have a higher number of frames, 128 frames for simlarge input set and 512 in native input set. Furthermore, for native input set one could appreciate the reduction in terms of execution time is more lineal than for simlarge. This is caused by the difference resolutions of each input set. A more resolution more computing which translates into more parallelism degree.

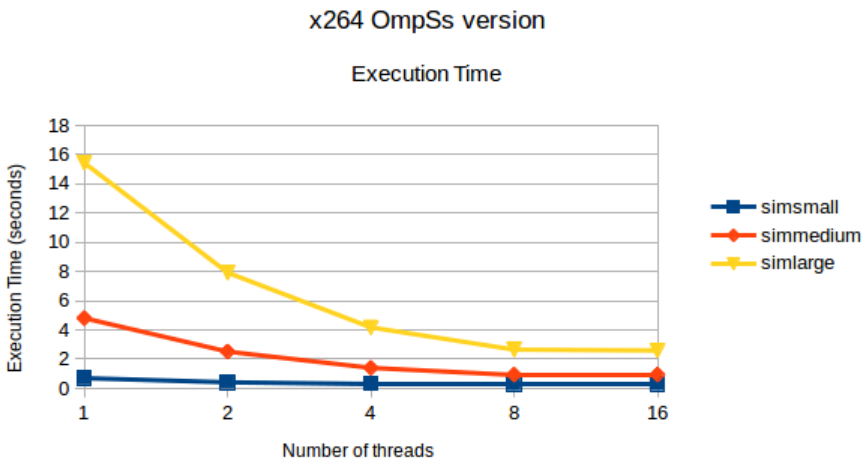

Figure 7.5: Execution time using OmpSs version with simsmall, simmedium and simlarge input set and different number of threads

Now, we are going to take a look into another metric, frames per second. This is another important metric since is the one which tells the user how fast is encoding the application each frame. At Figure 7.7 we can see that chart.

Note that the frames per seconds depend basically on two factors, the first one is the size of the frame (i.e. the resolution) and the other one is the quantity

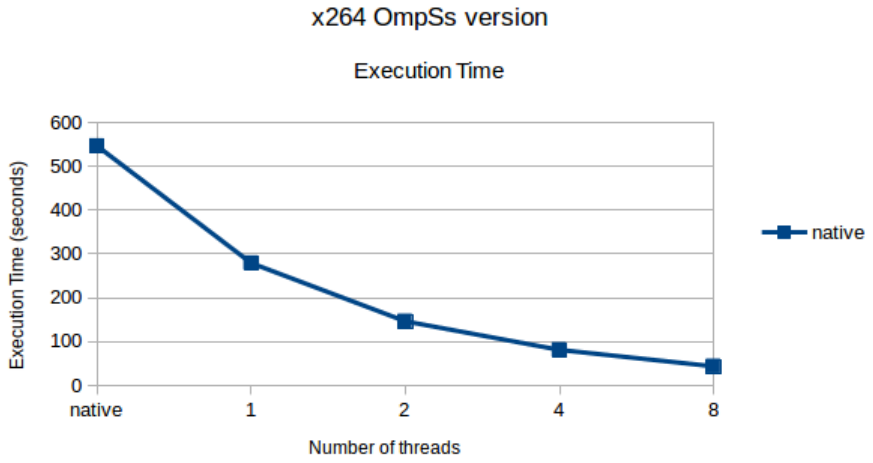

Figure 7.6: Execution time using OmpSs version with native input set and different number of threads

of compression (i.e. number of P or B frames) has been applied. This means is not fair comparing different input set. Even tough one can notice that for all input set except for native, after 4 or 8 cores, frames per second remains quite constant. As it is already said, this is due to the need of spend too many time doing synchronization instead of doing computation, this is the opposite when using native input set. As the resolution of video file increase, the computation also does, so it is easier to hide the synchronization overhead.

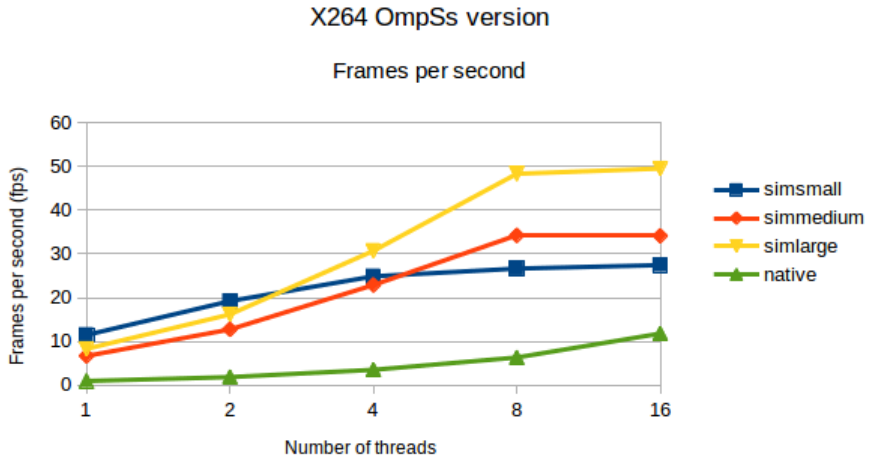

Figure 7.7: Frames per second using OmpSs version with simsmall, simmedium, simlarge and native input set and different number of threads

7.3 Scalability

As we already saw, scalability is not lineal when encoding video files with a small resolution like it does when using simsmall, simmedium or simlarge input set. Actually the reduction in terms of execution time starts to decrease when we are executing with more than 8 threads. One of the reasons is that the application we made achieves better degree of parallelism when encoding bigger frames due to the relation between computation and synchronization. The other reason is the overhead of the Nanos runtime. So lets see how this can affect the final performance and the scalability.

Figure 7.8 shows the percentage of the time spend in each one of the different states of the Nanos runtime in an execution of OmpSs version using simlarge input set and 8 threads. You can see that almost all the time is spend on the runtime. The reason is that not too many computation is done actually, so the overhead of the runtime is very high. This affects the final performance and the scalability as well.

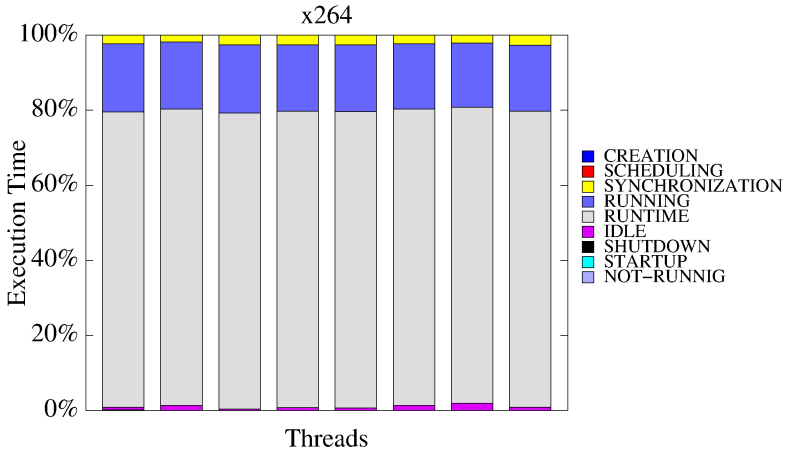

Figure 7.8: Percentage of the Execution time spend in Nanos runtime using simlarge input set

Now at Figure 7.9 we can see a similar figure but now it shows how many time is spent in each task when executing the OmpSs version using simlarge input set and 8 threads. Not too many time is spend on x264_slices_write actually. The reason is that the size of the frame does not require too much computation, which translates into a short encoding process per frame. This is reflected also at execution time in form of poor scalability when simsmall, simmedium and simlarge input set.

In summary, the higher the resolution of the frames the better the scalability will be as we can see at Figure 7.10. This figure shows the percentage of the execution time spend in each OmpSs task of the application. We can see that now we are most of the time encoding frames, not doing other work as it happens

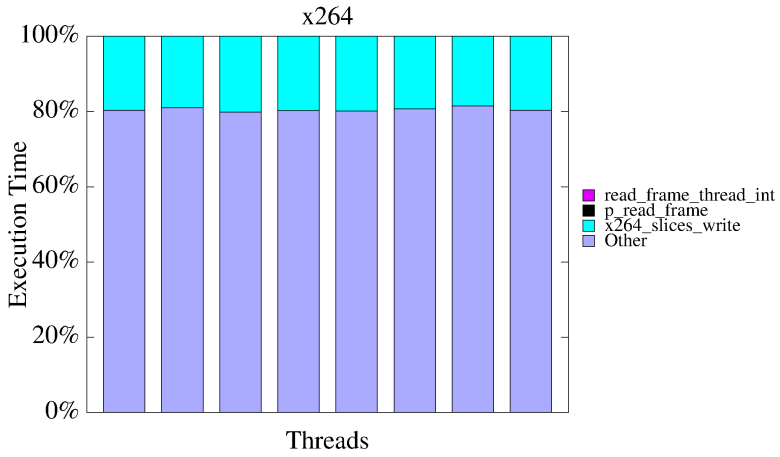

Figure 7.9: Percentage of the Execution time spend in each task using simlarge input set

when using simlarge input set. This behaviour translates directly into a better scalability of the application.

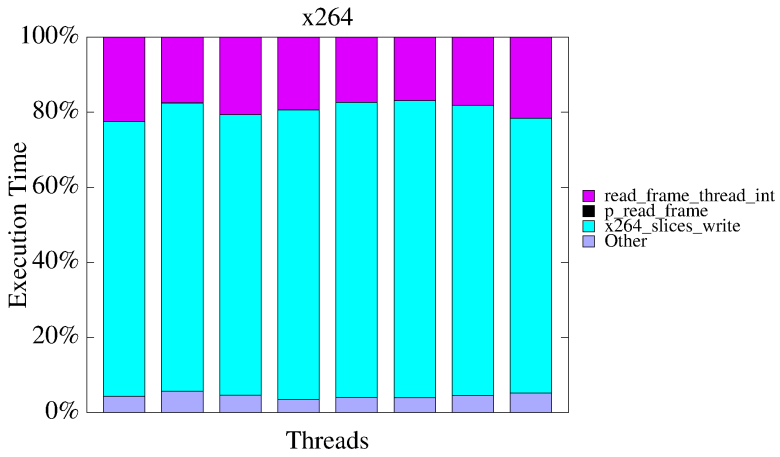

Figure 7.10: Percentage of the Execution time spend in each task using native input set

7.4 Comparison with serial and pthreads version

Now we are going to compare our application with serial and pthreads versions. The idea is to know if our application performs better or not against pthreads version and also discover the speed up achieved against serial one.

In this line, Figure 7.11 shows the speed up of both pthreads and OmpSs version against serial one when using simlarge input set. We can see that speed up is better in OmpSs application for 1, 2, 4 and 8 threads, but not 16. The main problem here is the overhead of the Nanos runtime and the lack of heavy computation.

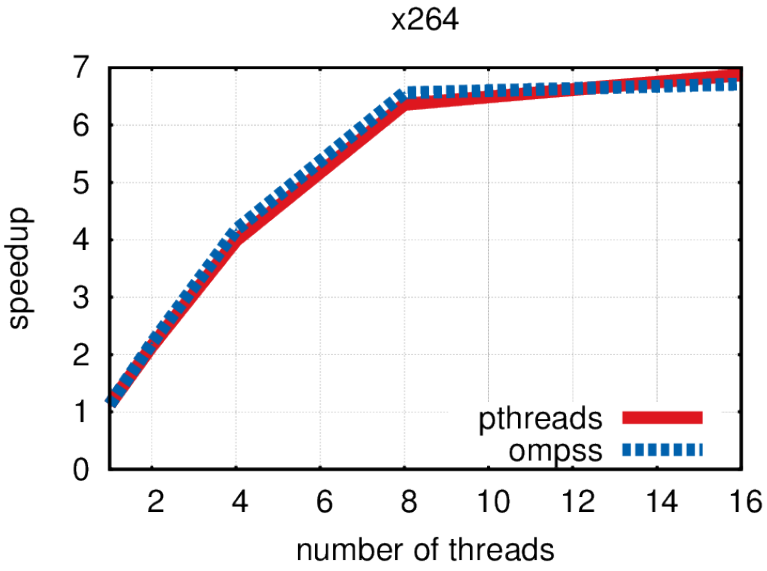

Figure 7.11: Speed up comparison of pthreads and OmpSs version against serial version when using simlarge input set

Now, Figure 7.12 shows the same chart but using native input set. As one can see, the performance is slightly better when executing OmpSs version with 16 threads. This is due to the heavy computation have to be done with high resolutions. Now the Nanos runtime overhead is hidden by computation and, since we can avoid the synchronization that pthreads version has, a better performance and scalability than in pthreads version can be noticed.

In summary, our application scales slightly better when using high resolution video files but scales slightly worst when using low resolution video files. The cause of this is the relation between synchronization and computation that have to be done.

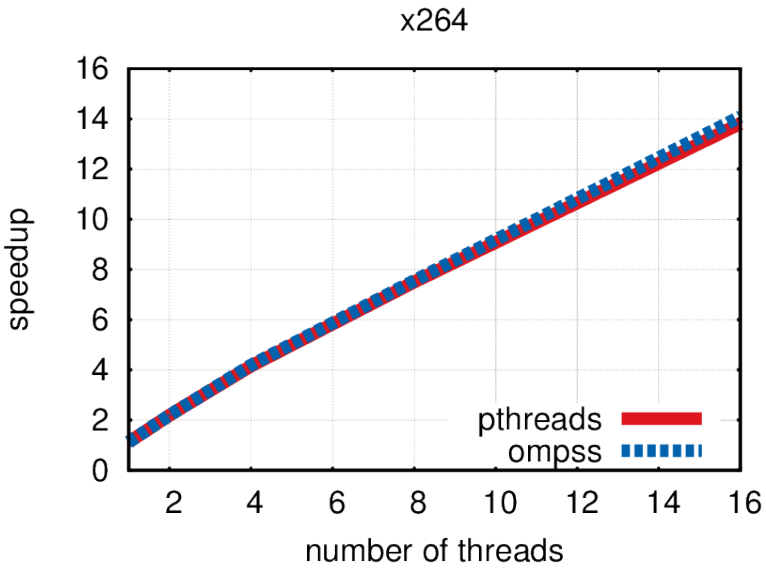

Figure 7.12: Speed up comparison of pthreads and OmpSs version against serial version when using native input set

Chapter 8

Conclusions

Now that the application is already ported to OmpSs and an evaluation has been done, is time to think about the results we have obtained. In this line we will talk not only about performance results but also about what we have learned from this project.

In this line, I will run through objectives that were set in order to know which ones have been achieved and which ones not. So lets start remembering which were those objectives.

- Main objectives:
  - Design a parallel version of the x264 application using a task-based approach.
  - Implement the parallel version we designed using the task-based programming model OmpSs.
  - Comparison with current parallel version, emphasizing on differences between programming model used on the current parallel version and OmpSs programming model.
- Secondary objectives:
  - Characterization of the x264 application.
  - Compare performance analysis of the current parallel version in front of our parallel version.
  - Prove that OmpSs programming model is also valid for non high performance computing workloads.

In summary, we could say that both main and secondary objectives are accomplished. We designed a parallel version which was implemented using OmpSs programming model afterwards. We compared also this version against the current parallel version which uses pthreads. About the secondary objectives, a characterization and a comparison in terms of performance has been done against both serial and pthreads version of the x264 video encoder, with nice results.

About if OmpSs programming model was the best choose or not, I personally think that has been proved that OmpSs programming model is also valid for non high performance computing workloads. I mean, maybe we are not that faster compared to pthreads version, but we are not worst. Actually, we proved that for high resolution video streams our application scales a bit better than pthreads version does.

About the design and the implementation, I think we are not taking profit of all the features that OmpSs provides just because the way the dependencies are made are not the best ones when using OmpSs programming model. I mean, OmpSs is just perfect when we have a lot of data dependency, but not when we need to synchronize tasks between each using a certain value contained in a certain variable. Besides this issue, the performance was more than enough to perform as the pthreads version.

Furthermore, I think that our version is more understandable than pthreads one. In this line, I have to say that many of the things that are performed by pthreads version were translated into OmpSs version using less lines of code, reducing complexity of the code. This is very important because it reduces the cost of maintaining the application or maybe adding modifications in the future. This, in addition of the fact that OmpSs provides the user with features like obtaining trace files very easily as well as a graph dependency, provides the developer with a lot of tools to debug or maintain its application.

In summary, I think it has been worth the time and the effort spent on this project, not only because was an amazing journey (this does not mean it was easy) but also because I think that all the objectives were accomplished in one way or another.

Finally, the following list will show a summary of what I have already mention within this section.

- Using OmpSs programming model for a non data dependency algorithm is a bit difficult.
- Even though it can be used also in this kind of applications.
- Implementing an algorithm using OmpSs is easier than doing the same with pthreads.
- OmpSs provides the developer with very useful development tools which allow to debug or improve the application.
- Using OmpSs translates easily into less lines of code.
- Using development tools like Extrae, Paraver or gprof are the only way to know what is happening within your application in order to correct bugs or improve the performance.

Glossary

B | C | E | F | G | L | M | O | P | R | T

B

benchmark

The result of running a computer program to assess performance. 2, 3, 27, 32, 39, 44

C

codec

Computer program capable of encoding or decoding a digital data stream or signal. 1–5, 30

compiler

Computer program that transforms source code written in a programming language into another computer language that can be executed by the computer. 1, 7, 8, 11, 17, 21, 34, 57, 68, 69

CPU

Acronym for Central Processing Unit. 5, 18

CUDA

Acronym for Compute Unified Device Architecture. A programming model mean to be executed at a GPU. 3, 5, 6, 16, 24, 25, 34, 36

E

encoder

Software program that converts information from one format or code to another, for the purposes of standardization, speed, secrecy, security or compressions. 1, 2, 4, 5, 9, 28–31, 39, 40, 51, 61, 63, 67, 79, I

F

frame

One of the many single photographic images in a motion picture. 3, 9, 28–31, 39–42, 44, 45, 49–64, 66, 67, 71–75

G

GPU

Acronym for Graphics Processing Unit. 1, 5, 6, 8, 33

L

LaTex

Is a mark up language specially suited for scientific documents. 20, 21, 23

Linux

Is a generic term referring to the family of Unix-like computer operating systems that use the Linux kernel. 15–20, 23

M

MPEG

Acronym for Moving Pictures Experts Group. 1, 5, 27, 28, 30, I

MPI

Acronym for Message Passing Interface. Is a standardized and portable message-passing system designed and implemented for distributed memory systems. 3, 24, 25, 36

O

OmpSs

OpenMP Super-scalar programming model. 1–6, 8–10, 12, 13, 15–21, 24, 25, 27, 33–36, 39, 51, 53, 57, 59–61, 64, 66, 68, 69, 72, 75, 77, 79, 80

OpenCL

Open Computing Language. A programming model mean to be executed across heterogeneous platforms. 3, 5, 16, 34, 36

P

PARSEC

Acronym for Princeton Application Repository for Shared-memory Computers. 2, 3, 27, 32, 39, 44

pixel

Physical point in a raster image, or the smallest addressable element in an all points addressable display device. 31, 39, 44, 46, 51, 52, 54, 56

POSIX

Acronym for Portable Operating System Interface for Unix-like operating system. 39, 44, I

pthreads

Is a POSIX standard for threads. 2, 4, 5, 8, 19, 36, 39, 44, 51–53, 57, 59, 60, 71–73, 76, 77, 79, 80

R

runtime

Software designed to support the execution of computer programs. 1, 6–8, 17, 21, 33–35, 42, 66, 75, 77

T

thread

The smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. 6, 8, 33, 35, 39, 44–48, 50, 54, 55, 59, 64, 71, 72, 75, 77, I

trace

A list of a computer program's past execution steps. 17, 18, 22, 23, 35, 36, 45, 46, 49, 59, 61, 68, 71, 72, 80

Bibliography

[St13] Edward Freeman and Alexander Moutchnik, *Stakeholder management and CSR: questions and answers*, Springer Berlin Heidelberg, 2013

[PSC08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li, *The PARSEC Benchmark Suite: Characterization and Architectural Implications* , Princeton University Technical Report TR-811-08, January 2008

[Blu05] *Application Definition Blu-ray Disc Format*, `http://www.blu-raydisc.com`, March, 2005

[htl14] *HTML 5.1 A vocabulary and associated APIs for HTML and XHTML*, `http://www.w3.org/TR/html51/`, W3C, February 2014

[Kar13] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz and Charles H. Still, *Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application*, IEE 27th International Symposium on Parallel and Distributed Processing, 2013

[FF] *FFmpeg-Based Projects*, `http://www.ffmpeg.org/projects.html`

[FFP] *FFmpeg Codecs Documentation*, `http://www.ffmpeg.org/ffmpeg-codecs.html#Video-Decoders`, March 2014

[APL05] *Apple Announces iTunes 6 With 2,000 Music Videos, Pixar Short Films and Hit TV Shows*, `http://www.apple.com/pr/library/2005/`, October 2005

[ADR] *Supported Media Formats*, `http://developer.android.com/guide/appendix/media-formats.html`

[OCL13] *The OpenCL Specification*, `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`, November 2013

[hcu13] *CUDA H.264/AVC ENCODER SDK 2.0*, `http://www.mainconcept.com/products/sdks/gpu-acceleration.html`, 2013

[hcm12] *MPEG-4 AVC/H.264 Video Codecs Comparison*, `http://compression.ru/video/codec_comparison/h264_2012/mpeg4_avc_h264_video_codecs_comparison.pdf`, May 2012

[hcl] *x264 Main Development tree*, `http://git.videolan.org/gitweb.cgi/x264.git/`

[cu12] *NVIDIA CUDA C Programming Guide Version 4.2*, Technical Report, April 2012

[ch07] B. Chamberlain, D. Callahan and H. Zima, *Parallel Programmability and the Chapel Language*, In. J. High Perform. Comput. Appl, vol. 21, no. 3, pp. 291-312, August 2007

[ch11] B. L. Chamberlain, S. E. Choi, S. J. Deitz, D. Iten and V. Litvinov, *Authoring User-defined Domain Maps in Chapel*, Cray, Inc., May 2011

[Ka93] L. Kalé and S. Krishnan, *CHARM++: A portable Concurrent Object Oriented System Based on C++*, in Proceedings of OOPSLA'93, A. Paepcke, Ed. ACM Press, September 1993

[La03] O. S. Lawlor and L. V. Kalé, *Supporting dynamic parallel object arrays*, Concurrency and Computation: Practice and Experience, vol. 15, pp. 371-393, 2003

[De11] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso and P. Hanrahan, *Liszt: a domain specific language for building portable mesh-based PDE solvers*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC'11. ACM, 2011

[Lu05] E. A. Luke and T. George, *Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis*, Journal of Functional Programmin, Special Issue on Functional Approaches to High Performance Parallel Programming, vol. 15, no. 93, pp. 477-502, 2005

[Ul88] J. Ullman, *Principles of Database and Knowledgebase Systems*, Computer Science Press, 1988

[Zh09] Y. Zhang and E. A. Luke, *Concurrent composition using Loci*, IEEE/AIP Computing in Science and Engineering, vol. 11, no. 3, pp. 27-35, May/June 2009

[So08] K. Soni, N. Cain and E. A. Luke, *Work replication: A communication optimization in Loci*, in Proceedings of the ISCA 21nd International Conference on Parallel and Distributed Computing and Communication Systems, New Orleans, LA, September 2008

[Da98] L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, Computational Science Engineering, IEEE, vol. 5, no. 1, pp. 46-55, January-March 1998