PARALLELIZATION TECHNIQUES OF THE FACESIM SIMULATOR

Author: Raul Vidal\textsuperscript{1,2}

Degree in Informatics Engineering
Specialization in Information Technologies
Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Director: Marc Casas\textsuperscript{2}
Co-director: Miquel Moretò\textsuperscript{1,2}

Deponent: Mateo Valero\textsuperscript{1,2}
Department: Computer Architecture

\textsuperscript{1}UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) - BarcelonaTech (Spain)
\textsuperscript{2}Barcelona Supercomputing Center (Spain)

January 26, 2015
Acknowledgements

This project has been a good opportunity for improving myself as individual and meet interesting people who I consider great persons both in their professional career and when in need of their support and help.

I would like to thank the people of Barcelona Super Computing Center for their help. Concretely, the teams from Programming Models, BSC Tools, BSC MareNostrum support and BSC Help Desk. I thank my mates from RoMoL too, specially Dimitrios Chasapis for his help and support.

Of course, the advisors of this project have had a very important role. Without them I could not have achieved the obtained results during this project. Thanks to Marc Casas, Miquel Moretó and Mateo Valero for their assistance and guidance.

I do not forget the people from Facultat d’Informàtica de Barcelona who transmits knowledge to their student body, helping in the process of training new professionals in the field of computer science.

At last, I would like to mention my family because they have always been there and I could not have finalized my Degree in Informatics Engineering without their support during all this long time.

Agradecimientos

Este proyecto ha sido una buena oportunidad para mejorar como individuo y conocer gente interesante que considero grandes personas tanto en su carrera profesional como cuando he necesitado su ayuda.

Querría agradecer a la gente del Barcelona Supercomputing Center su ayuda. Concretamente, a los equipos de Programming Models, BSC Tools, BSC MareNostrum support y BSC Help Desk. Gracias también a los compañeros de RoMoL, especialmente a Dimitrios Chasapis por la ayuda prestada.

Por supuesto, los tutores de este proyecto han tenido un rol muy importante. Sin ellos no podría haber conseguido los resultados obtenidos durante este proyecto. Gracias a Marc Casas, Miquel Moretó y a Mateo Valero por su asistencia y guía.

Tampoco olvido al profesorado de la Facultat d’Informàtica de Barcelona que transmite conocimiento a su alumnado, ayudando en el proceso de formación de nuevos profesionales.

Finalmente, quiero mencionar a mi familia dado que ha estado siempre ahí y no podría haber acabado mi Grado en Ingeniería Informática sin su apoyo durante todo este largo tiempo.

Agraïments

Aquest projecte ha estat una bona oportunitat per millorar com a individu i conèixer gent interessant que considero grans persones tant en la seva carrera professional com quan he necessitat la seva ajuda.

Voldria agraïr a la gent del Barcelona Supercomputing Center la seva ajuda. Concretament, als equips de Programming Models, BSC Tools, BSC MareNostrum support i BSC Help Desk. Gràcies també als companys de RoMoL, especialment a Dimitrios Chasapis per la ajuda prestada.

Per suposat, els tutors d’aquest projecte han tingut un rol molt important. Sense ells no podríria haver aconseguit els resultats obtinguts durant aquest projecte. Gràcies a Marc Casas, Miquel Moretó i a Mateo Valero per la seva asistència i guia.

Tampoc m’oblido del professorat de la Facultat d’Informàtica de Barcelona que transmet coneixement al seu alumnat, ajudant en el procés de formació de nous professionals en el camp de les ciències de computació.

Finalment, null mencionar a la meva família donat que ha estat sempre allà i no podríria haver acabat el meu Grau en Enginyeria Informàtica sense el seu suport durant aquest llarg temps.
Contents

Abstract

1 Project introduction, state of the art and planning
  1.1 Introduction and contextualization ......................................................... 5
    1.1.1 Stakeholders ................................................................. 6
    RoMoL project ................................................................. 6
    Project developers ............................................................. 6
    OmpSs developers ............................................................... 6
    Project directors ................................................................. 6
    Computer architecture researchers .............................................. 6
    Physics engines developers ....................................................... 6
    Visual effects studios ............................................................. 7
    Videogame developers ............................................................. 7
  1.2 State of the art ................................................................................. 7
    1.2.1 Physics engines ................................................................. 8
    Bullet ................................................................. 8
    Havok Physics .............................................................. 8
    Open Dynamics Engine ........................................................... 8
    PhysBAM ................................................................. 8
    PhysX ................................................................. 8
    1.2.2 Physics engines summary .......................................................... 9
    1.2.3 Parallel programming models ................................................... 9
    OpenACC ................................................................. 9
    Microsoft C++ AMP ........................................................... 10
    CUDA ................................................................. 10
    OpenMP ................................................................. 10
    1.2.4 Final thoughts on state of the art .................................................. 10
  1.3 Project Scope .................................................................................... 10
    1.3.1 Risks ................................................................. 12
    1.3.2 Requirements ................................................................. 12
  1.4 Methodology ...................................................................................... 13
    1.4.1 Development method ............................................................ 13
  1.5 Tools ................................................................................................. 13
  1.6 Validation .......................................................................................... 14
  1.7 Project schedule ................................................................................ 14
    1.7.1 Task description ........................................................................ 14
    Project Planning ................................................................. 16
    Read documentation related to Facesim ........................................ 16
    Prepare development environment .............................................. 17
    Evaluate current implementations of Facesim ................................... 17
    Adapt and port current Facesim code base to OmpSs ...................... 18
    Evaluate the OmpSs version of Facesim ....................................... 18
    Write final report with all the collected data ................................... 19
    Weekly meetings ................................................................. 19
5 Design and implementation of Facesim adaptation to OmpSs

5.1 Introduction .................................................. 67
5.2 Facesim preparation .......................................... 67
5.3 OmpSs adaptation design ..................................... 70
5.4 OmpSs adaptation implementation .......................... 70
5.4.1 Array Parallel Operations class .......................... 72
5.4.2 Face Example class ........................................ 74
5.4.3 Deformable Object 3D class ............................... 75
5.4.4 Deformable Object class .................................... 76
5.4.5 Collision Penalty Forces class .............................. 80
5.4.6 Diagonalized Finite Volume 3D class .................... 80
5.4.7 Other work and additional notes ......................... 80
5.4.8 Final thoughts on OmpSs implementation ............... 82
5.5 OmpSs performance evaluation ............................... 82
5.6 Validation ..................................................... 84

6 OmpSs Programmability ........................................ 89

7 Conclusions ..................................................... 91

List of terms ...................................................... 95
List of figures ..................................................... 97
List of tables ..................................................... 99
Code listings .................................................... 101
References ....................................................... 103
Abstract

Since the creation of the first supercomputers, the efficient execution of software on multiple processors has been both a necessity and a challenge. A necessity because everyday the problems that human society needs to solve are growing, and as they grow so do the computing needs. A challenge because the rise of computing performance needs also brings new restrictions in terms of energy consumption. Today, the deployment of datacenters in companies is common. They can be seen as an specialization of the clusters employed in supercomputers[1]. Society is putting more pressure on companies and their datacenters and clusters. On today’s world this is even more relevant due to the new wave of mobile devices which connect to them. Having the proper software which allows to adapt these new times in order to offer the needed quality of service is important.

Today, things like smartphones are common and new devices like smartwatches and smartglasses are starting to emerge. All of these devices are ubiquitous and adding software support to exploit their potential and offer an optimal user experience is a very important topic on today’s and future information technologies. Mobile devices of today also use multiple processors. Lately, support for executing applications in a Graphics Processing Unit (GPU)s has also been added. These devices usually connect to servers located in big datacenters and use the resources they make available to their users.

With all that, this project explores the maintenance needed for the application Facesim in order to adapt it to the OmpSs programming model. OmpSs is being developed in Barcelona Supercomputing Center and can be used in multiple types of systems, including datacenters and clusters or devices with mobile processors. One of the objectives of OmpSs is to make easier the creation and maintenance of concurrent or parallel applications. Application development like web servers can benefit from this type of programming models[2], as productivity and degree of concurrence increase. With the work made in Facesim it has been proven that OmpSs can facilitate the development of big applications using less code. The resulting code is more readable and programmability is improved. An application like Facesim is used in datacenters like the ones employed by Industrial Light and Magic to render film special effects[3]. Having a programming model like OmpSs which allows a cleaner maintenance and development is something that has always been sought after in the world of information technologies. Evidently, the efficiency must not be forgotten. A promising technology may fall into disgrace if its performance is too low. That is why this project apart from showing improvements of Facesim from the point of view of programmability, it also shows improvements in execution time. Thus, this project shows that OmpSs is a promising technology which can offer better maintainability and on top of that better performance.

In this project an example of an information technologies infrastructure for an hypothetical company searching for a cluster to be at its disposal where executing an application or applications based in OmpSs is also presented. This leads to an analysis of the MareNostrum computer and the capacities of the administration software it has. MareNostrum uses Extreme Cloud Administration Toolkit[4] for maintenance tasks. This software has a big quantity of characteristics which make it a great system to manage big datacenters and clusters. With this study, a global view of a full software and hardware stack is completed, from the service platform (MareNostrum) up to the applications and its service libraries (OmpSs).

Resumen

Desde la creación de los primeros superordenadores, la ejecución de software en múltiples procesadores ha sido una necesidad y un reto. Una necesidad porque los problemas diarios que la sociedad necesita resolver están creciendo, y a medida que lo hacen también lo hacen las necesidades de cómputo. Un reto porque la necesidad de más rendimiento del hardware trae consigo nuevas restricciones en términos de consumo energético. Hoy día se ha hecho común el despliegue de centros de datos en empresas, que se pueden ver como una especialización de los clusters utilizados en los superordenadores[1]. La sociedad pone cada vez más presión en las empresas y sus clusters o centros de datos. En el mundo de hoy esto es aún más relevante debido a la hornada de dispositivos móviles que se conectan a ellos. Tener el software adecuado que permita adaptarse a estos nuevos tiempos para ofrecer la debida calidad de servicio es crucial.
Hoy, cosas como los smartphones son comunes y nuevos dispositivos como los smartwatches y smartglasses están comenzando a tomar importancia. Todos estos dispositivos son ubicuos y tener el soporte software necesario para explotar su potencial y ofrecer una experiencia de uso óptima es un tema importante para las tecnologías de la información de hoy y del futuro. Los dispositivos móviles disponen de procesadores múltiples. Además, recientemente se les ha añadido soporte para ejecutar aplicaciones en una GPU. Estos dispositivos suelen conectarse a servidores ubicados en grandes centros de datos y usan los recursos que éstos ponen a disposición de sus usuarios.

Con todo ello, este proyecto explora el mantenimiento necesario para la aplicación Facesim para adaptarla al modelo de programación OmpSs. OmpSs está siendo desarrollado en el Barcelona Supercomputing Center y puede ser usado en múltiples tipos de sistemas, incluyendo centros de datos y clusters o dispositivos con procesadores móviles. Uno de los objetivos de OmpSs es conseguir que la creación y mantenimiento de aplicaciones concurrentes o paralelas sea más sencillo. El desarrollo de aplicaciones como servidores web se puede ver beneficiado por este tipo de modelos de programación[2], al aumentar la productividad del programador y el grado de concurrencia de la aplicación. Con el trabajo realizado en Facesim se ha probado que es posible desarrollar aplicaciones grandes empleando menos cantidad de código. El código resultante es más legible. Una aplicación como Facesim se utiliza en centros de datos como los que tiene la empresa Industrial Light and Magic para renderizar efectos especiales del cine[3]. Disponer de un modelo de programación como OmpSs que permita un mantenimiento y desarrollo más claro es algo que siempre se busca en el mundo de las tecnologías de la información. Obviamente no hay que olvidar la vertiente de la eficiencia. Una tecnología prometedora como OmpSs puede caer en el olvido si su rendimiento es demasiado bajo. Es por ello que este proyecto además de presentar mejoras en Facesim desde el punto de vista de la facilidad de desarrollo, también presenta mejoras en los tiempos de ejecución. Así, este proyecto muestra que OmpSs es una tecnología prometedora que puede ofrecer una mejor mantenibilidad y además un rendimiento mejor.

En este proyecto también se muestra un ejemplo de infraestructura de tecnologías de la información para una hipotética empresa que busque tener a su disposición un cluster en el que ejecutar una aplicación o aplicaciones basadas en OmpSs. Esto lleva a un análisis del ordenador MareNostrum y las capacidades del software de administración que éste dispone. MareNostrum emplea Extreme Cloud Administration Toolkit[4] para las tareas de mantenimiento. Este software tiene una gran cantidad de características que hacen de él un buen sistema para manejar grandes centros de datos y clusters. Con este estudio se completa una visión global de una pila de software y hardware completa, desde la plataforma de servicios (MareNostrum) hasta las aplicaciones y sus bibliotecas de servicios (OmpSs).

Resum

Des de la creació dels primers superordinadors, l’execució de software en múltiples processadors ha estat una necessitat i un repte. Una necessitat perquè els problemes diaris que la societat necessita resoldre estan creixent, i a mesura que ho fan, també ho fan les necessitats de computació. Un repte perquè la necessitat de més rendiment del hardware porta amb s’ì mateix noves restriccions en termes de consum energètic. Avui dia s’ha fet comú el desplegament de centres de dades en empreses, que es poden veure com una especialització dels clusters emprats en els superordinadors[1]. La societat posa cada vegada més pressió a les empreses i els seus clusters o centres de dades. En el món d’avui dia això és encara més rellevant donada la fornada de dispositius mòbils que es connecten a ells. Tenir el software adequat que permeti adaptar-se a aquests nous temps i oferir la deguda qualitat de servei és molt important.

Avui, coses com els smartphones son comuns i nous dispositius com els smartwatches i smartglasses estan començant a tenir importància. Tots aquests dispositius son ubicus i tenir el suport software necessari per explotar el seu potencial i oferir una experiència de us óptima és un tema important per a les tecnologies de la informació d’avui i del futur. Els dispositius mòbils disposen de processadors múltiples. A més, recentment se’ls hi ha afegit el suport per executar aplicacions en una GPU. Aquests dispositius solen connectar-se a servidors ubicats en grans centres de dades i utilitzen els recursos que aquests posen a disposició dels seus usuaris.

Amb tot això, aquest projecte explora el manteniment necessari per a l’aplicació Facesim per adaptarla al model de programació OmpSs. OmpSs està sent desenvolupat en el Barcelona Supercomputing Center
i pot ser utilitzat en múltiples tipus de sistemes, incloent centres de dades i clusters o dispositius amb
processadors mòbils. Un dels objectius d’OmpSs es aconseguir que la creació i manteniment d’aplicacions
concurrents o paral·leles sigui més senzill. El desenvolupament d’aplicacions com servidors web es pot
veure beneficiat per aquest tipus de models de programació[2], al augmentar la productivitat del pro-
gramador i el grau de concurrencia de l’aplicació. Amb el treball realitzat en Facesim s’ha provat que
es possible desenvolupar aplicacions grans emprant menys quantitat de codi. El codi resultant es més
ilegible. Una aplicació com Facesim s’utilitza en centres de dades com els que té l’empresa Industrial
Light and Magic per renderitzar efectes especials del cinema[3]. Disposar d’un model de programació
com OmpSs que permeti un manteniment i desenvolupament més clar és quelcom que sempre es busca
en el món de les tecnologies de la informació. Obviament no s’ha d’oblidar la vertent de l’eficiència. Una
tecnologia prometedora com OmpSs pot caure en l’oblit si el seu rendiment es massa baix. És per això
que aquest projecte a més de presentar millores en Facesim des de el punt de vista de la facilitat en
el desenvolupament, també presenta millores en els temps d’execució. Així, aquest projecte mostra que
OmpSs és una tecnologia prometedora que pot oferir una millora en la mantenibilitat i a més un millor
rendiment.

En aquest projecte també es mostra un exemple d’infraestructura de tecnologies de la informació per a
una hipotètica empresa que busqui tenir a la seva disposició un cluster en el que executar una aplicació o
aplicacions basades en OmpSs. Això porta a fer un anàlisi de l’ordinador MareNostrum i les capacitats
del software d’administració que aquest disposa. MareNostrum utilitza Extreme Cloud Administration
Toolkit[4] per a les tasques de manteniment. Aquest software té una gran quantitat de característiques
que fan d’ell un bon sistema per administrar grans centres de dades i clusters. Amb aquest estudi es
completa una visió global d’una pila de software y hardware completa, des de la plataforma de serveis
(MareNostrum) fins a les aplicacions i les seves biblioteques de serveis (OmpSs).

3
Chapter 1

Project introduction, state of the art and planning

1.1 Introduction and contextualization

This document will present the needed contextualization of the project. That means explaining the theme, terms and actors involved. Explanations of the current state of the art are presented too, documenting previous developments or implementations also in use in the field of the project.

Contextualization

This project involves the evaluation of a programming model in terms of ease of programming and execution efficiency. A programming model can be seen as a software library that allows to program software with the functionality it provides, that is, create code with less effort. The programming model in question is named OmpSs, an acronym for OpenMP Super-scalar. OmpSs will be used for adapting Facesim’s current code base to OmpSs, effectively doing a maintenance on the application. That serves the purpose of showing whether OmpSs is a promising technology and how does it improve the ease of programming.

Facesim uses a particular version of PhysBAM library for its physics computations [5]. That library contains a number of algorithms in order to achieve realistic animations of varying types of objects, which range from fluids to deformable objects going through rigid bodies. There are other physics libraries in the market, being PhysBAM one of the most important and most developed software application libraries [3]. There are a lot of interesting physics engines and while they are also valid for simulation, Facesim already includes PhysBAM in its software distribution, as such, this project will work with PhysBAM.

Physics simulation involves computationally intensive calculus to solve a system of equations which determines the position of the objects in a point of time [6, 7, 8]. The objects to simulate have some properties associated to them in order to properly mimic the materials the object is composed of. To represent those objects and to be able to do a realistic simulation, normally they are discretized as grids of triangles or tetrahedra [8, 6]. This document will keep away from details on implementations and algorithms available although the reader might find the provided documentation profitable if it is from his field of study. There are different methods for doing physics simulation and they depend on the type of body to simulate, that is, if it is a rigid, fluid or soft body. Facesim employs the PhysBAM functions related with soft body simulation. Given the complexity of a physics simulation application like Facesim, it poses as a great test bed for doing an adaptation to OmpSs programming model and show how does it behave in such type of applications.

Finally, this project does a study of the service platform which MareNostrum provides. MareNostrum is a cluster, which is formed by a number of computers acting as one. To manage clusters there exists special software. MareNostrum uses the Extreme Cloud Administration Toolkit. The findings from the study will serve to show a selection of software and hardware platforms for any company wishing to build their own cluster or datacenter.
All this project is motivated by the fact that the future of the software goes through the ability of organizations to be able to harness the potential of their IT infrastructure. Going in that direction, OmpSs is evaluated and put into test by adapting Facesim. In order to better complement it, knowing what type of infrastructure can be used for the demands of organizations is also relevant. That leads to the study of MareNostrum as an example, giving a view of a possible full stack for executing applications in a datacenter.

The next section of the document discusses the main actors in this project.

1.1.1 Stakeholders

This project is mainly directed towards computer architecture researchers and the RoMoL project. In any case, although this project is mainly a research project, there are more actors other than the project investors which might be interested in the results obtained. They will also be commented.

RoMoL project

This project forms part of the RoMoL project, a research group in Barcelona Supercomputing Center dedicated to drive new computer architectures and parallel software [9]. The interest RoMoL has in Facesim is the use of OmpSs in several applications. Concretely, Facesim belongs to the PARSEC Benchmark Suite [10], an important collection of applications widely used for evaluation of new computer architectures. Having success in porting the whole PARSEC suite to OmpSs will allow RoMoL to have an important contribution in computer architecture and runtime development. It lays a good base for continuing doing more research with the OmpSs runtime as new ideas or ways of improving OmpSs might be explored or revealed.

Project developers

There is only one developer in this project. His implication in the project clearly states special interest in it and its results. Improving physics simulation performance has an important role. Demonstrating that easier to use programming models are also valid for achieving good performance in non high performance workloads is also relevant for the developer. Other benefits are the improved knowledge in the related fields, like parallel programming, computer architecture and physics simulation.

OmpSs developers

OmpSs developers might benefit from the development of this project as they might find cases where the runtime does not work as expected. It is an opportunity for improving the runtime.

Project directors

They are the supervisors of the adaptation of Facesim to OmpSs. Their interest is then highly justified as they also have an important role in RoMoL, managing other related work. If the porting of Facesim gives good performance results and it can show the effectiveness of OmpSs in non high performance computing benchmarks, then they can drive RoMoL to pursue new objectives related with OmpSs development and its utilization.

Computer architecture researchers

Facesim belongs to the PARSEC Benchmark Suite. Benchmarks ported to OmpSs allow to have new test cases for task based and heterogeneous programming models that can drive computer architecture research and new processor architectures. Researchers from computer architecture would have more software available for doing computer architecture research.

Physics engines developers

Facesim uses PhysBAM [3] as the library for its physical computations. PhysBAM is a physics engine and has been employed in important projects involving visual effects and research in physics simulation.
A new parallelization made with a programming model said to be easier to use is welcome by physics
engine developers.

**Visual effects studios**

Movie studios which employ a lot of money in creating visual effects for their films are in need of ever
increasing performance in physics simulation libraries. Giving them the opportunity to have a new
version of PhysBAM is of their interest.

**Videogame developers**

Videogames physics engines are going towards the methods employed by PhysBAM. Recent videogames
have already added basic support for the same type of simulations that Facesim employs. A code
adaptation made with a programming model said to be easier to use is welcome by physics engine
developers. And if it has better performance then it is even more relevant for them. A good physics engine
with great accuracy and high performance is important for real-time physics in interactive applications
like videogames.

The next section will talk about current state of the art in both physics engines and parallel programming
models topics.

### 1.2 State of the art

The first thing to ask yourself is if there is someone that already has done something similar. The current
Facesim code base uses Pthreads. There is no other implementation available. That leaves an open path
for adapting the current code base to the the programming model the developer might find adequate,
and in the case of this project OmpSs was chosen.

In the field of physics simulation, there are other libraries employed for doing physics simulations. Some
libraries are more videogame oriented while others offer more accurate simulations, useful for simulation
engineers or any other user that can benefit from accurate results [6]. This section will do a brief
description of some of them as there are so many that it is impossible to cite them all.

As for parallel programming models, there are also some other alternatives to OmpSs that could have
been used for doing the maintenance done for Facesim. A description of various programming models
follows the physics engines.

On the topic of datacenters and clusters, this project focuses on the application software side of things.
There are a lot of hardware vendors and each one offers their own suite of administration software.
As Facesim’s adaptation is going to be developed on MareNostrum supercomputer, the study has been
done on MareNostrum infrastructure. MareNostrum is an IBM machine which uses Extreme Cloud
Administration Toolkit (also known as xCAT), an open source tool licensed under the Eclipse Public
License. Other vendors like BuLL also offer their own IT equipment, including their own software
suites for managing the cluster. In reference [11] one can find a PDF showing the portfolio of products
offered by Bull. It includes services for cloud computing, diverse software supported by them like OSes,
distributed/networked filesystems (GPFS, NFS), tools for managing their clusters and a utility for users
named extreme factory which allows to access HPC applications and use the cluster as an IaaS or SaaS.
This means to be able to manage data generated, and even accessing 2D or 3D visualizations from
extreme factory secure portal. Their webpage offers more explanations.

Some OS vendors like Red Hat and Oracle also have their own solutions. Red Hat offers the Red Hat
Cluster Suite and Oracle offers the Solaris Cluster software. The size of the clusters supported by these
tools is smaller than the size supported by xCAT.

Next section explain some of the most common physics engines and also some programming models
competing with OmpSs.

---

1.2.1 Physics engines

Bullet

Bullet is a physics engine that can do simulations on rigid bodies and deformable objects in real-time. It also supports the simulation of collisions between objects. It has been used for doing visual effects for movies and also in videogames. This library is licensed under the terms of the zlib license, making it free software although not copyleft. Not only can it run on a normal CPU but it also can be executed under GPUs thanks to being also programmed in OpenCL and CUDA parallel programming models. It is also compatible with the console hardware platforms PlayStation 3, Xbox 360 and Nintendo Wii. It also has support for mobile devices running Android and iOS. The general purpose code (the one not programmed for specific accelerators like graphics chips) is programmed in C++ and employs Pthreads or Win32 threads [12, 13, 14].

Havok Physics

It is a commercial physics engine and has been used in more than 150 videogames [14]. Havok is available for multiple platforms including Xbox 360, PlayStation 4, PlayStation 3, Windows, PlayStation Vita, Wii, Wii U, Android, iOS, Mac OS and GNU/Linux. The main use of Havok is in videogames. As such, it offers plugins for 3D modeling software like 3DS Max, Maya and XSI. [15].

Open Dynamics Engine

ODE is a very popular engine among robotics simulation applications and currently it is not parallelized. It is programmed in C/C++ and offers simulation of rigid bodies. [16, 14, 17].

PhysBAM

Developed by the Stanford University, PhysBAM ended being one of the applications used by Intel RMS suite. PhysBAM has been used for developing visual effects for films and its main contributor, Ronald Fedkiw, has worked for different companies in the visual effects industry like Industrial Light and Magic or Pixar. PhysBAM supports rigid bodies, fluids, smoke, cloth, fire and deformable objects. It is programmed using the C++ language and for its parallelization employs Pthreads along a task queue library. Currently only works on CPUs, there is no accelerator support like GPUs [18, 3].

PhysX

PhysX is a library initially developed by Ageia. Originally thought for use in Ageia’s PPUs (Physics Processing Units), finally ended being a purely software library compatible with GPU execution. NVIDIA acquired Ageia in 2008 and added the GPU support for their own GPUs with a CUDA framework. PhysX supports real-time simulation of rigid bodies, soft bodies, cloth, joints, fluids and others. PhysX is more oriented towards videogames although it can be used for more realistic simulations. The range of supported hardware platforms includes GPUs as previously stated, CPUs, and consoles like PlayStation 3, Xbox 360 and PlayStation 4 [19, 20].

Contrary to most of the presented engines, if not all, PhysBAM has not been designed for use in real-time with current hardware technology. PhysBAM can work with really complex models that even might include muscles, as in the case of Facesim [7].

By the previous engines, real-time engines seem to be mostly focused or limited to the simulation of rigid bodies. As the hardware has progressed some of them have started to add soft body support, like PhysX or Havok. Solutions like the ones employed in the usage that Facesim does for PhysBAM should become more prevalent. One example of that is the videogame Star Wars: The Force Unleashed which employed finite element models for deformation and destruction effects of wood, steel, flesh and plants [21, 22]. With sufficiently detailed models, ie, high resolution meshes, the quality of simulations can be real-life accurate.
1.2.2 Physics engines summary

In table 1.1 the reader will find a table summarizing the relevant aspects of the different physics engines.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bullet</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, No, No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Windows, GNU/Linux, Xbox 360, PlayStation 3, Wii</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Havok</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, yes, No, No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PS3, Windows, PS Vita, Wii, Wii U, Xbox 360, Android, iOS, Mac OS, GNU/Linux</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Open Dynamics Engine</td>
<td>No</td>
<td>No</td>
<td>No, no, no, no</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Platform independent</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PhysBAM</td>
<td>Yes</td>
<td>No</td>
<td>Yes, yes, yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Solaris, GNU/linux</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Physics engines properties comparison

PhysBAM is the most complete physics engine, is free of charge and it is already included with Facesim.

1.2.3 Parallel programming models

The previous section tried to note if the most used engines are currently parallelized or not. The research done showed that most of them are not prepared for executing physics computations in parallel except if they use accelerator-enabled code. The accelerators currently supported are the GPUs as the PPUs (Physics Processing Units) disappeared. The only PPU that ever became commercially available was Ageia’s PPU. Current trends like the work done in the videogame Star Wars: The Force Unleashed showed a multicore aware solver. Their finite element approach is more amenable to parallelization [21] and that is also the case of Facesim. It is to be expected that in the near future engines will be catching up.

Given the evolution of physics engines towards more parallelizable methods, it is important to have a good programming model to manage incoming workloads. In following pages, this document will explain some of the current parallel programming models that may be considered to compete with OmpSs.

OpenACC

OpenACC is an acronym for Open Accelerators and is a standard for parallel computing on heterogeneous CPU/GPU systems. Uses special C/C++/Fortran compiler directives which indicate which code will
be compiled specifically for GPUs. Current implementation of OpenACC only supports acceleration on NVIDIA GPUs. This programming model assumes that the execution is directed by the host and the compute-intensive regions are offloaded to the attached accelerators. The memory of the host and the accelerators is separated and they are not updated automatically.

**Microsoft C++ AMP**

C++ AMP, acronym for C++ Accelerated Massive Parallelism is a programming model with an open specification made by Microsoft for accelerating C++ applications by running code on GPUs. Microsoft has implemented C++ AMP on top of DirectX although other implementations of C++ AMP are not needed to be based on DirectX [23]. Instead of using compiler directives like OpenACC, OpenMP and OmpSs, C++ AMP uses C++ classes as if they belonged to the C++ Standard Template Library.

**CUDA**

CUDA, acronym for Compute Unified Device Architecture is a programming model created by NVIDIA to program its GPUs. It is made of a set of C++ language extensions accompanied by a runtime library. The programmer using this model develops CUDA (called kernels) like if it was programming C++ functions. The code is run in parallel by the multiple threads executing in the GPU. [24]

**OpenMP**

OpenMP, acronym for Open Multi-Processing, uses compiler directives for C, C++ and Fortran. These directives can specify regions and loops to be parallelized by the OpenMP-aware compiler. OpenMP employs what is known as a fork-join model. A fork-join model creates and destroys a team of threads every time a parallel region is reached and exited, respectively. [25]

1.2.4 Final thoughts on state of the art

Currently OmpSs and its asynchronous task-based approach with no fork-join model can be considered a good balance between programmability and performance. The programmer does not have to manage threads explicitly, he only has to specify tasks and he has the additional possibility of creating tasks targeting specific accelerator hardware. OmpSs builds on current programmer language knowledge and adds tasks to it so generally it should not pose a lot more effort to create parallel or concurrent code from sequential code. Maintainability of software on new devices is an objective of development and use of these new programming models.

In respect to physics engines, PhysBAM is included in Facesim. In any case, it is always a good thing to see what other physics engines are available and where they are going to. Basically they are going towards how PhysBAM works, making PhysBAM a good representation of what is to come in the near future.

Finally, as one of the objectives of this project is to show that OmpSs can be useful for applications used (or will be used) in desktop and workstation environments, the use of OmpSs is mandatory. Facesim is a good application for that, as it represents a trend in videogames that is going to be common.

For preparing this review of state of the art, the most important articles have been the ones made by Ron Fedkiw in order to get some knowledge on physics simulation of deformable bodies. The articles comparing engines were also relevant and "Real-time deformation and fracture in a game environment" [21] which shows recent developments on game industry. As for programming models, the given references are all equally important.

1.3 Project Scope

This project aims at analyzing an already deployed cluster/data center stack which includes the OmpSs programming model as the top layer. The cluster/data center is a service platform for users to execute their applications. With that in mind, this project will collect information on a typical cluster software
deployment and will adapt an application to the OmpSs programming model which takes advantage of the cluster deployment. That serves the purpose of showing how effective can be the proposed stack, both at a cluster administration level and at a cluster application maintenance and development level. Companies are in need of having their own clusters and having the proper software and tools to better use them is important. This project can be considered a proposal of a full software stack for that kind of environments, showing the strengths and weaknesses of each component. Next paragraph lists the main tasks this project involves.

Tasks to be completed:

• Study current cluster/datacenter administration stack of MareNostrum
• Evaluation of the current code base of the user software application running on top of the cluster/data center
• Design and implement an adaptation to OmpSs of the current code base of the user application
• Evaluation of the OmpSs version of the application

This project will compare both current code base and the OmpSs adaptation in terms of ease of programming (and thus ease of maintenance) and performance. Some other ways of doing an OmpSs adaptation might be studied and compared with the chosen one. This should not consume too much time because the baseline design would be the same, just changing some OmpSs constructs by other ones so they can be compared and determine what they are better suited for in terms of speedup/execution time or portability to other types of systems (ease of programming). The analysis of the administration stack will show how the proposed cluster works and how it is deployed. Currently this project is using MareNostrum supercomputer for development. MareNostrum uses common software for that type of environments. A cluster needs to allow system administrators to manage and maintain an installation with thousands of computers which work as if they were only one. This project will do an study of the cluster architecture. Sometimes it might be called data center as a data center is based on the same idea as a cluster.

The completion of these tasks is aligned with meeting some desirable objectives, directly or indirectly related with the tasks specified above. Those objectives are as follows:

• Primary objectives:
  – Study the deployment of MareNostrum infrastructure
  – Know what software is employed in a cluster and how is it used
  – Design and implement an OmpSs adaptation for an application which needs a cluster or data center for execution (Facesim)
  – Comparison with current code base of the application, including both ease of programming and performance differences between programming models being used.
  – Show whether OmpSs programming model can be a valid environment for developing software needed for the new trend of devices and services

• Secondary objectives:
  – Improve knowledge on systems administration and networked environments
  – Get better knowledge of the C++ programming language.
  – Get better knowledge of scripting languages.
  – Have a broader knowledge of parallel and concurrent programming, and in turn OmpSs.
  – Improve knowledge of tools for analyzing and evaluating an application’s behaviour and performance in order to get better understanding of it.
  – Improve high level knowledge of physics simulation.

The secondary objectives are justified by the need of some requirements in order to be able to complete the proposed tasks. For example, if you want to use MareNostrum you need to do some scripts. Also, the application is written in C++ language so it’s important to have a good base of object oriented programming knowledge and some C++ idioms.
1.3.1 Risks

Possible obstacles or risks which can surface during the development of this project are mainly related with difficulties during the porting process of the application. This involves problems with using some of the needed tools and modification of the original source code of Facesim.

Obstacles:

- **Application code structure**: Due to application code size and how it’s modularized, it may need more modifications than expected. The implication of this is a longer development time. It’s unavoidable as it’s a natural risk of the application. The needed time is already included in the schedule.

- **Application bugs**: Before starting the OmpSs implementation, which will reuse a lot of code if not all, some bugs in Facesim might be discovered and they’ll have to be corrected. This risk is also unavoidable due to the very nature of the application. If it has bugs they have to be solved. The developer can choose to ask for help if it has any doubts with it. This risk can add more development time. The development should proceed with care in order to not mix application bugs with bugs related with the OmpSs adaptation.

- **OmpSs programming model**: Current knowledge of OmpSs is limited to maintaining small snippets of C code. Maybe there are difficulties while using it with C++ code that uses classes and parametric templates. These difficulties can include bugs in OmpSs itself or limitations of C++ knowledge. In order to avoid the risk a simplistic approach should be taken. Also, the OmpSs development team can be asked for help. The implications of this risk is that not all what the developer might want to implement can be implemented, or the developer has to choose other ways to do the OmpSs adaptation.

- **Facesim algorithms and data structures**: Facesim uses PhysBAM physics library. That library has different algorithms and data structures to model objects and simulate force interactions. During development some complications may arise due to the need of some understanding of how those algorithms work together and what data do they access. Going further, once the data accesses have been explored, discovering they don’t allow for an optimal task implementation is also a risk. This risk is inherent to the nature of the application, and it might not allow for an optimal port resulting in more complicated code or less desirable performance. Special attention to the evaluation of the original code must be made.

- **Experiment design and automation**: Although in the case of our objectives this shouldn’t be a problem, during development some difficulties might arise in relation with testing correctness of the output or taking different metrics of the program. Also, experiment automation might pose some problems until having a deeper understanding of development tools usage and cluster structure.

Generally speaking, during development a minimalistic approach should be taken. There is the need to avoid too much modifications in order to minimize possible obstacles. Continuous testing of the application will be made in order to keep problems at a minimum and solve them promptly.

1.3.2 Requirements

After some thoughts, an approximation of requirements for the life cycle of the project can be found in the following list:

- **Time**: Estimated time for planning is around 4-6 weeks. Estimated time to finish development of the OmpSs adaptation is around 5 months.

- **Economical resources**: The estimated cost is expected to be similar to a job with 8h per labour day and probably some extra days in weekends due to unexpected problems. MareNostrum computations and other hardware are to be considered too.

- **Environmental impact**: If the application improves its performance, the energy employed in each simulation would be lower, resulting in a reduction of its environmental impact.
1.4 Methodology

The project methodology is based upon a development method of the OmpSs adaptation and validation practices. Both are described in subsequent sections and subsections.

1.4.1 Development method

First and foremost, knowing at a high level how does OmpSs work is very important. Having a minimum dexterity in using the programming model is mandatory for developing the porting of Facesim to OmpSs. Same thing applies to the use of the different development tools. The training on these tools is part of the application development and skills on using them should improve during the process of adapting Facesim to OmpSs.

The OmpSs adaptation development will follow an iterative method. Starting by analyzing where the application spends its parallel execution time, then taking the direction to where most of the efforts should be put during the development of the adaptation. Mainly, it should go around the idea of creating OmpSs tasks from the found parallel sections. And from those parallel sections, after making a first approximation by adapting all of them to OmpSs, the next step would be to optimize the hot spots found during the analysis phase to obtain better performance. This obeys to Amdhal’s law, which specifies that the maximum speed up (decrease in execution time to the serial version) depends on how large is the parallel code section so if we then optimize the bigger ones, the reward on speedup can be higher.

Weekly meetings with project’s tutors will provide more guidance and objectives to accomplish during the development cycle. The discussions with tutors should provide relative short stages. The need of understanding how does the original code work can be quite a consuming task, making the bulk of the adaptation to have longer iterations.

To summarize, the iterative method is a loop of analyze, understand original code, port code and obtain performance metrics if needed, for example execution times at each iteration if new code has been written, all accompanied by the weekly guidance of the tutors. At each iteration new issues may be discovered or/and the performance may have changed.

1.5 Tools

In order to achieve the project objectives there is a selection of tools which will be part of the development and validation processes:

- **GCC**: A compiler, it is used to generate the executable.
- **OmpSs**: The programming model used for doing the OmpSs adaptation, usually makes maintenance of parallel software an easier task with less lines of code for the same or better performance than pthreads.
- **Extrae**: It is a tracing system. Allows to obtain execution traces. An execution trace contains performance related information of the execution of the program. Traces also show the behaviour of the application, giving comprehension on what it does and doesn’t do or whether if it behaves as expected. It’s also integrated within OmpSs.
- **Paraver**: It is a viewer for Extrae’s traces. Reads the files generated by the Extrae system and shows the performance metrics represented in different types of figures as requested by the user.
- **GNU Gprof**: It is a tracing system. Included with GCC compiler, much like Extrae, it allows to get information of what functions are called during the execution of the application and tells you how much time it spends at each function.
- **MareNostrum**: The supercomputer where the program will be compiled, executed and measured.
- **Subversion**: It is a revision control system. For the project an SVN repository will be used to maintain a change history and data backup.
Now, this document will proceed to explaining the validation caveats and how it will be performed.

1.6 Validation

As the main objective of the project is to evaluate OmpSs based on the work done with Facesim, there’s a need of knowing if during development our program still outputs correct values, so it is critical to check and compare them with results from the original code. For each frame of the animation simulation the output of both implementations (the original parallel implementation and the new OmpSs implementation) will be compared. Facesim has different input sizes. The input size determines the number of frames to simulate. The computations for each frame are always the same. With only one frame is enough to check correctness, although bigger input sizes might be used for seeing how does the residual of the equation system evolve.

One also has to take into account the fact that Facesim employs floating point calculations and the order of the code can influence in the output values. If the output of the OmpSs adaptation is very different then it is invalid and should be inspected to see where’s the bug in the implementation. The final 3D face model state can be printed and compared to the original version to properly see if there are big differences. The validation is qualitative. Continuous testing will be employed for validation.

As for the evaluation of OmpSs, ease of programming will be measured by the reduction in lines of code (if any) and the performance improvements in terms of execution time. Evolution of these parameters during the project serve as indicators of progress. Results will be discussed with tutors.

Next sections present the schedule of the project. That means describing tasks and their dependencies. It also means showing a temporal figure displaying the task schedule. And at the end, the action plan is defined.

1.7 Project schedule

This project has an estimated duration of six months and should be finalized by 19th January 2015 with 30th January 2015 being the deadline for reading the TFG. Research on this project started during summer 2014, concretely, 23rd June 2014. August is not taken into account.

1.7.1 Task description

This project is composed of different stages or tasks. Each stage is divided in different substages. A detailed list of them to be completed would be as follows:

1. Project Planning
   I Define project scope
   II Define temporal schedule and resources
   III Estimate needed budget for the project
   IV Contextualize and search for bibliography
   V Define scope statement

2. Read documentation related to Facesim
   I Read information on how to use Facesim
   II Read papers related to Facesim and its methods

3. Write project planning and documentation sections of the thesis

4. Prepare development environment
   I Read Mare Nostrum documentation
   II Prototype scripts for Mare Nostrum development environment
5. Evaluate current implementations of Facesim
   I Prepare Facesim for evaluation of its current implementations.
   II Evaluate the serial version of Facesim
   III Evaluate the current parallel implementation of Facesim

6. Write evaluation of Facesim sections part of the thesis

7. Adapt Facesim to OpenMP Super-scalar (OmpSs)
   I Prepare Facesim for the OmpSs implementation
   II Design a first task-based implementation using OmpSs
   III Implement the task-based design using OmpSs

8. Evaluate the OmpSs version of Facesim
   I Improve OmpSs code until getting a satisfactory enough implementation, if needed

9. Write remaining text of the thesis

Note that each task is also codified in order to be able to easily formalize their dependencies, formalization which can be seen in the following figure:

![Figure 1.1: Chain of dependencies](image)

Respect to possible deviations, a plus of 50% of hours can be assumed, going up to an equivalent of 9 months instead of six, so the number of hours could be in the range of 100% - 150% of the estimated time. A medium deviation is thus a possibility. These deviations can occur due to the risks of the project, needing additional time for understanding and correcting issues found during development. All of these considerations are already taken into account as mentioned early, weekends and extra hours can be used for the possible deviations.

With the list of tasks already detailed, this document will present in following pages what work do they involve and what resources will they need. Only the main tasks will be specified as each subtask can be understood with its parent task description. the distribution of employed hours might not be uniform. Also, please note that the sum of days is greater than 90 days, that’s because some of the tasks are overlapped. Although there are weekly meetings, those are at most only half hour length and they’re non blocking, as they act more as control sessions than anything else. A total of 1 day can be attributed to weekly meetings. These were not codified before due to their nature, although they consume some resources, which are described bellow.

All time estimations are based on previous experience.
Project Planning

At this stage, the procedures involved are related with the definition of the project in all the possible aspects: scope, time, economics, related work.

Estimated time: 15 days

Resources:

- Human resources
  - Main developer

- Hardware
  - A laptop from ASUS, model N53SN (main developer’s property)
  - A computer screen, Lenovo LT2252p
  - A computer mouse, Lenovo M-U0025-O
  - A computer keyboard, Lenovo SK-8825

- Software
  - GNU/Linux based operating systems, including:
    - Ubuntu 14.04.1 LTS (on the developer laptop)
    - \LaTeX
    - LibreOffice office automation suite.

Read documentation related to Facesim

This task is destined to read documentation talking about Facesim, PhysBAM and its numerical methods. This information should give an idea of what to expect during development. During development of the Facesim adaptation to OmpSs this information might be revisited or some extra documentation might be needed, but at least with the documentation found at this time, it should be good enough to get a first contact with Facesim design. This task is overlapped with the preceding task of contextualizing this project, during the project planning.

Estimated time: 5 days

Resources:

- Human resources
  - Main developer

- Hardware
  - A laptop from ASUS, model N53SN (main developer’s property)
  - A computer screen, Lenovo LT2252p
  - A computer mouse, Lenovo M-U0025-O
  - A computer keyboard, Lenovo SK-8825

- Software
  - GNU/Linux based operating systems, including:
    - Ubuntu 14.04.1 LTS (on the developer laptop)
    - \LaTeX
    - LibreOffice office automation suite.
Prepare development environment

Preparing the development environment includes reading documentation on how Mare Nostrum works, how to execute programs in it and how to load the needed tools. Doing or at least prototyping some executable scripts of commands is also included at this stage. Of course, obtaining Facesim source code is mandatory. The work done during this task can be refined later to fit particular needs not predicted before. This task is overlapped with the preceding task, reading documentation.

**Estimated time: 5 days**

**Resources:**

- **Human resources**
  - Main developer
  - Mare Nostrum Support

- **Hardware**
  - A laptop from ASUS, model N53SN (main developer's property)
  - A computer screen, Lenovo LT2252p
  - A computer mouse, Lenovo M-U0025-O
  - A computer keyboard, Lenovo SK-8825
  - The supercomputer Mare Nostrum III
  - SUSE Linux Enterprise Server 11 for x86_64, patch level 2 (on Mare Nostrum III)

- **Software**
  - GNU/Linux based operating systems, including:
    * Ubuntu 14.04.1 LTS (on the developer laptop)

Evaluate current implementations of Facesim

The evaluation of current implementations of Facesim analyzes its code and behaviour with the correct tools. This will allow to obtain information on Facesim current performance, its parallel sections and their data accesses. GNU Gprof will be an important tool for this step as it allows to get a profile of the application, that is, to know where it spends its time. Total execution time will also be measured with Facesim's own timing system.

To do the evaluation first Facesim must be able to compile with the wanted compiler flags, separated from the PARSEC Benchmark Suite, so some parts of its compilation toolchain should be modified, like environment variables and Makefiles. This means to prepare Facesim for its evaluation.

**Estimated time: 15 days**

**Resources:**

- **Human resources**
  - Main developer

- **Hardware**
  - A laptop from ASUS, model N53SN (main developer’s property)
  - A computer screen, Lenovo LT2252p
  - A computer mouse, Lenovo M-U0025-O
  - A computer keyboard, Lenovo SK-8825
  - The supercomputer Mare Nostrum III
  - SUSE Linux Enterprise Server 11 for x86_64, patch level 2 (on Mare Nostrum III)

- **Software**
  - GNU/Linux based operating systems, including:
Adapt and port current Facesim code base to OmpSs

Once Facesim has been studied enough, current implementations have to be compiled with the OmpSs programming model to see if current code needs some modifications for the Mercurium compiler to compile with the desired options. Once Facesim is able to compile with OmpSs, the original code can start to be modified. During this phase the original code will be analyzed further in order to see what it has and design a fitting OmpSs adaptation. The design might be done progressively as the implementation progresses through the different code sections. Some of the previously read documentation might be useful for this task too. Knowledge on OmpSs and its tools is crucial to know how to better use them. So does knowing how to use Mare Nostrum in a proper manner and having the automation scripts. Refining of these should be done at this stage.

Estimated time: 107 days

Resources:

- Human resources
  - Main developer
  - OmpSs developers

- Hardware
  - A laptop from ASUS, model N53SN (main developer’s property)
  - A computer screen, Lenovo LT2252p
  - A computer mouse, Lenovo M-U0025-O
  - A computer keyboard, Lenovo SK-8825
  - The supercomputer Mare Nostrum III
  - SUSE Linux Enterprise Server 11 for x86_64, patch level 2 (on Mare Nostrum III)

- Software
  - GNU/Linux based operating systems, including:
    - Ubuntu 14.04.1 LTS (on the developer laptop)
    - GNU gprof

Evaluate the OmpSs version of Facesim

At this time, an adaptation to OmpSs should be ready and so the project can proceed to its evaluation. Execution time and traces with Extrae will be taken. For a better comparison an execution trace of the POSIX threads (Pthreads) implementation will also be generated to see particularities in their characteristics and compare them. At this stage, it’s the time to decide whether Facesim OmpSs implementation needs to be improved or if it has finished. Things to improve can be redefinition of data dependency for better global performance or further modifications in code’s modular design.

Estimated time: 5 days

Resources:

- Human resources
  - Main developer

- Hardware
  - A laptop from ASUS, model N53SN (main developer’s property)
  - A computer screen, Lenovo LT2252p
Write final report with all the collected data

This task spans the whole project and is done in three stages. The first one during the project planning at the beginning, the second one after having evaluated the current implementations of Facesim and the last one after finalizing the OmpSs porting. For each stage, an estimation of 5 days is done.

Estimated time: 25 days

Resources:

- Human resources
  - Main developer

- Hardware
  - A laptop from ASUS, model N53SN (main developer’s property)
  - A computer screen, Lenovo LT2252p
  - A computer mouse, Lenovo M-U0025-O
  - A computer keyboard, Lenovo SK-8825
  - The supercomputer Mare Nostrum III
  - SUSE Linux Enterprise Server 11 for x86_64, patch level 2 (on Mare Nostrum III)

- Software
  - GNU/Linux based operating systems, including:
    - Ubuntu 14.04.1 LTS (on the developer laptop)
    - Extrae
    - Paraver
    - LibreOffice office automation suite

Weekly meetings

This task is distributed along the whole duration of the project. In total 2 days will be used. They don’t block any current process.

Estimated time: 2 days

Resources:

- Human resources
  - Main developer
  - Project directors

Given this list of work units and their predicted invested time, in subsequent sections the corresponding Gantt chart and action plan can be found.
Changes

Due to the study of MareNostrum infrastructure in order to give a full stack proposal, more days have been added to the writing of the final report or memoir of the project. A total of additional 15 days have been added. These days have been overlapped with the evaluation of OmpSs adaptation and the writing of the data of OmpSs adaptation. They required less hours than expected, so each working day has been split between both. The corresponding memoir writing task has been added to the gantt diagram, putting it concurrently with the OmpSs evaluation and final report writing. Its dependency comes from the MareNostrum documentation read.

1.7.2 Time management

Given the previous list of work units and their predicted invested time, the resulting Gantt chart can be seen in the next page. Please note that the Gantt chart is including weekends in the tasks durations, so the number of days in the Gantt is higher than those previously specified:
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Project Planning</td>
<td>10d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>2. Define project scope</td>
<td>1d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>3. Define temporal schedule and resources</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>4. Evaluate needed budget for the project</td>
<td>4d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>5. Conduct a last search for bibliography</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>7. Read documentation related to Focasim</td>
<td>5d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>8. Read information on how to use Focasim</td>
<td>1d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>10. Write project planning and documentation sections of the thesis</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>12. Read flow sheet documentation</td>
<td>1d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>13. Prototype scripts for a browser development environment</td>
<td>4d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>15. Prepare Focasim for evaluation of current implementations</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>16. Evaluate the current version of Focasim</td>
<td>4d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>17. Evaluate the current parallel implementation of Focasim</td>
<td>7d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>18. Write evaluation of Focasim sections part of the thesis</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>19. Adapt Focasim to Omepsis</td>
<td>100.4d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>20. Prepare Focasim for the Omepsis implementation</td>
<td>2d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>22. Prepare Focasim for the Omepsis implementation</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>23. Design a task-based implementation using Omepsis</td>
<td>15d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>24. Implement the task-based design using Omepsis</td>
<td>40.75d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>25. Implement the Omepsis version of Focasim</td>
<td>3d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>26. Final report with all collected data</td>
<td>1d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>27. Study of cluster infrastructure (and with report)</td>
<td>1.5d</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

Figure 1.2: Gantt chart
Having defined all tasks and their respective resources and defined the Gantt chart, in next section you will find the action plan and possible alternatives in order to handle deviations.

### 1.8 Alternatives and action plan

Weekly meetings with project’s directors are scheduled. With these meetings, deviations should be addressed promptly by pointing out what can be more useful to do next based on the current project’s state. The most important stage of this project, the implementation of an adaptation to OmpSs programming model, takes a lot of time. It’s important to do it without delays. If cumulative delays occur at each iteration of this iterative phase a lot more time might be spent, implicating a delay in the whole project. Other ways of implementing the same functionality should be taken to avoid spending more time than necessary. This is already expected and the length of that task takes this into account.

To cope with this impediments the iterative methodology should provide ways for overcoming them. For example, avoiding any additional complications to minimize development risks. Continuous testing of the program will be used to find bugs as quicker as possible.

This project employs an iterative process and there are no blocking meetings. This allows to start new tasks as soon as the ones in execution are already finished. With the extra provision of time already allocated, the planning of this project can be accomplished. Extra time allocated for tasks finished earlier than expected can be used for incoming tasks if current march of the project needs it. This ensures the project can be finished in time.

Next sections present all items related with project’s budget and its sustainability. First of all there’s the list of tasks followed by a description of resources and their costs. Then this document proceeds to specify the costs per task and finally presents the total budget. After that, an analysis of viability is done.

### 1.9 Task decomposition

In this section this document presents the list of tasks. Their code is directly derived from their enumeration. For example, *Project Planning* is T1 and *Estimate needed budget for the project* is T1.III.

1. **Project Planning**
   - I Define project scope
   - II Define temporal schedule and resources
   - III Estimate needed budget for the project
   - IV Contextualize and search for bibliography
   - V Define scope statement

2. **Read documentation related to Facesim**
   - I Read information on how to use Facesim
   - II Read papers related to Facesim and its parallel methods

3. **Write project planning and documentation sections of the thesis (5 days)**

4. **Prepare development environment**
   - I Read Mare Nostrum documentation
   - II Prototype scripts for Mare Nostrum development environment

5. **Evaluate current implementations of Facesim**
   - I Prepare Facesim for evaluation of its current implementations.
   - II Evaluate the serial version of Facesim
   - III Evaluate the current parallel implementation of Facesim
6. Write evaluation of Facesim sections part of the thesis

7. Adapt and port current Facesim code base to OmpSs
   
   I Prepare Facesim for the OmpSs implementation
   II Design a first task-based implementation using OmpSs
   III Implement the task-based design using OmpSs

8. Evaluate the OmpSs version of Facesim
   
   I Compare OmpSs and Pthreads

9. Write final report with all the collected data
   
   I Study of cluster infrastructure

The resulting Gantt chart from the list of tasks and their defined precedences can be seen in Figure 1.3.
Figure 1.3: Gantt chart
Labour days are 8 hours each one at a maximum. At first, weekends are not taken into account. They are in the Gantt chart just for deviation purposes. Holidays are not paid in this project, so that pseudo-task is also out of any budget specification. Tasks sharing days have the hours of those days equally shared. In next section this document discusses unit costs for all resources to be used and afterwards it describes the costs for each task, counting the number of hours each task is predicted to use and their direct costs which are the human resources. Finally, the total budget is explained along the contingency budget.

1.10 Costs identification and estimation

In order to do an estimation on tasks expenditures the unitary prices for each resource needs to be calculated. So the first thing to do is describe the unitary cost per resource. Costs come from four main categories: human costs, hardware costs, software costs and indirect costs. All these costs can be found in the next subsection.

By unitary prices this document means the price without taking into account usage during this project, the number of hours each resource will be used. That is specified in a later section where each task is given its respective share of resource usage, and consequently the budget dedicated to it.

1.10.1 Unitary costs

The following tables show in a lean manner how much does each resource cost. Four categories are presented: human costs (personnel budget), hardware costs, software costs and indirect costs.

Personnel budget

Personnel budget estimations can be seen in Table 1.2. Wages have been obtained via [26]. The salary divided by twelve months and 160h per month (four weeks with five days of 8h each) gives the cost per hour. Note that these numbers are not net salaries.

<table>
<thead>
<tr>
<th>Role</th>
<th>Annual wage</th>
<th>Unitary price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Developer</td>
<td>34.975,77 €</td>
<td>18,20 €/h</td>
</tr>
<tr>
<td>Project Directors</td>
<td>53.165,69 €</td>
<td>27,69 €/h</td>
</tr>
<tr>
<td>Barcelona Supercomputing Center (BSC) Support (Mare Nostrum)</td>
<td>28.360,42 €</td>
<td>14,77 €/h</td>
</tr>
<tr>
<td>OmpSs developers</td>
<td>34.975,77 €</td>
<td>18,20 €/h</td>
</tr>
<tr>
<td>Research personnel</td>
<td>34.975,77 €</td>
<td>18,20 €/h</td>
</tr>
</tbody>
</table>

Table 1.2: Human costs

The main developer is the person in charge of doing this project. This project is a one man endeavour. Project directors are responsible of aiding the main developer and those with who the main developer will meet every week for half an hour. Mare Nostrum support are the ones who will allow the main developer to access Mare Nostrum, giving him a user and password and an e-mail. OmpSs developers are the ones who develop the OmpSs programming model. This team of developers will be asked for help about OmpSs use when having problems with it in situations where it should work. The research personnel are people related with the use of OmpSs programming model and can give some tips and other help during the project.

Hardware budget

The materials or devices needed for this project and their unitary costs are related in Table 1.3. In order to calculate depreciation value, the following formula has been used:
Depreciation = \frac{Unitary \ Price - Residual \ Value}{Years \ of \ Service}

<table>
<thead>
<tr>
<th>Asset</th>
<th>Unitary price</th>
<th>Service life (years)</th>
<th>Estimated residual value</th>
<th>Total estimated depreciation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop ASUS N53SN</td>
<td>1100 €</td>
<td>4</td>
<td>50 €</td>
<td>262.5 €/year</td>
</tr>
<tr>
<td>Computer screen, Lenovo LT2252p</td>
<td>325.68 €</td>
<td>4</td>
<td>50 €</td>
<td>68.92 €/year</td>
</tr>
<tr>
<td>Computer mouse, Lenovo M-U0025-O</td>
<td>10 €</td>
<td>4</td>
<td>3 €</td>
<td>1.75 €/year</td>
</tr>
<tr>
<td>Computer keyboard, Lenovo SK-8825</td>
<td>25 €</td>
<td>4</td>
<td>7 €</td>
<td>4.5 €/year</td>
</tr>
<tr>
<td>Mare Nostrum node</td>
<td>5312.60 €</td>
<td>5</td>
<td>250 €</td>
<td>1012.52 €/year</td>
</tr>
</tbody>
</table>

Table 1.3: Hardware costs

The Mare Nostrum node is a computer in a rack. Concretely, it is a blade from IBM, model dx360 M4. See reference [27]. Prices for the configuration used by Mare Nostrum can be found at [28]. For the hardware resources, knowing that this project spans a total of six months, the depreciation to be considered for this project is

Depreciation quantification = \left( \frac{6}{12} \right) \times (262.5 + 68.92 + 1.75 + 4.5 + 1012.52) = 675.095€

Software budget

This project is based on 100% free software which is also free of charge. Thus, no budget is needed for it. If during the curse of development money has to be invested in software, it will be reflected in an updated budget specification. For informational purposes, a table listing the software used is also attached, see Table 1.4 below.

<table>
<thead>
<tr>
<th>Asset</th>
<th>Unitary price</th>
<th>Service life</th>
<th>Estimated residual value</th>
<th>Total estimated depreciation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU/Linux Ubuntu 14.04.1 LTS</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
<td>0</td>
</tr>
<tr>
<td>LibreOffice office automation suite</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
<td>0</td>
</tr>
<tr>
<td>Ganttter</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
<td>0</td>
</tr>
<tr>
<td>ECLEx</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
<td>0</td>
</tr>
<tr>
<td>OmpSs - Mercurium Compiler</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
<td>0</td>
</tr>
<tr>
<td>OmpSs - Nanos++ runtime library</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1.4: Software costs

Other indirect costs budget

Indirect costs include electricity bill and Internet monthly subscription. See Table 1.5 below.
Internet subscription price comes from a Spanish ISP ADSL connection and the electricity cost is an estimate based on current prices. The indirect costs come from electricity use and Internet. The Internet costs $42.2 \times 6\text{months} = 253.2\,\text{€}$. Electricity accounting has two parts: laptop and screen power consumption and Mare Nostrum node power consumption. The Mare Nostrum node has an estimated power consumption of 200 Watts. Mare Nostrum is used by the main developer in tasks 4, 5, 7 and 8. That accounts for a total of 542.5 hours. The laptop and monitor is used for the whole duration of the project: 960 hours. Combined power consumption is at 100W. Electricity costs net $(802.5 \times 0.2 + 960 \times 0.1) \times 0.133 = 34.11\,\text{€}$. In table 1.8 all of these costs are also included.

In next section this document discusses the costs for each task.

### 1.10.2 Task costs

This section presents the list of tasks or work units and their associated direct costs. With unitary prices already established the table of costs for each task is easily derived from the number of days (and consequently, hours). For an easier visualization of the tasks, you can see the Gantt chart (Figure 1.3). The tasks in it also span weekends, but those weekends are reserved for any possible deviation.

The Main Developer is codified as MD, Project Directors as PD, OmpSs developers as OD, Mare Nostrum support as MS and Research Personnel as RP. Detailed costs for each task can be seen in tables 1.6 and 1.7.

### 1.10.3 Final budget

This project uses a series of resources. Each task only takes a share of them but the global picture of their usage is useful too. Table 1.8 reflects the total budget. The human resources budget is obtained directly from the per task specification of direct costs. Next section relates the budget for unforeseen expenses and contingencies. The budget dedicated to that should be added to the final costs.

**Contingencies and unforeseen expenses**

The current planification has 54 additional days of 8 hours each one for the Main Developer to correct any possible delay. Depreciation of hardware and indirect costs is already included as those are accounted for the whole project, so doesn’t matter if during those months they have to be used more days. Due to the highly detailed budget, this project considers a budget for contingencies of around 5% the predicted budget. Risks have to be considered too, and they appear in the table 1.9 Next section discusses how the control of expenditures is going to be done. Afterwards, the only remaining section is the explanations of project sustainability and its viability, which will serve as motivation for the proper valuation of the sustainability matrix.

Changes: Thanks to the contingency budget, the MareNostrum study has been made with the available resources. 15 days at 4 hours per day of the Main Developer amounts a total of 1092 euros, which is within the contingency of 1182.72 euros.

### 1.11 Control of expenditures

For controlling expenditures this project will employ a table which will allow to annotate any possible deviation for each task. Table 1.10 is an example.

In that table the developer will annotate his deviations on tasks done. For each task the human resources employed will be computed. Depreciation of hardware is not added because it is computed for the global
<table>
<thead>
<tr>
<th>Task</th>
<th>Assets used</th>
<th>Usage time (hours)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Project Planning</td>
<td></td>
<td></td>
<td>1701.7 €</td>
</tr>
<tr>
<td>1.I Define project scope</td>
<td>MD</td>
<td>24</td>
<td>436.8 €</td>
</tr>
<tr>
<td>1.II Define temporal schedule and resources</td>
<td>MD</td>
<td>24</td>
<td>436.8 €</td>
</tr>
<tr>
<td>1.III Estimate needed budget for the project</td>
<td>MD</td>
<td>32</td>
<td>582.4 €</td>
</tr>
<tr>
<td>1.IV Contextualize and search for bibliography</td>
<td>MD</td>
<td>8</td>
<td>145.6 €</td>
</tr>
<tr>
<td>1.V Define scope statement</td>
<td></td>
<td>5.5</td>
<td>100.1 €</td>
</tr>
<tr>
<td>2. Read documentation related to Facesim</td>
<td></td>
<td></td>
<td>600.6 €</td>
</tr>
<tr>
<td>2.I Read information on how to use Facesim</td>
<td>MD</td>
<td>2.5</td>
<td>45.5 €</td>
</tr>
<tr>
<td>2.II Read papers related to Facesim and its parallel methods</td>
<td>MD</td>
<td>10.5</td>
<td>191.1 €</td>
</tr>
<tr>
<td>3. Write project planning and documentation sections of the thesis with the collected data</td>
<td>MD</td>
<td>20</td>
<td>364 €</td>
</tr>
<tr>
<td>4. Prepare development environment</td>
<td></td>
<td></td>
<td>251,37 €</td>
</tr>
<tr>
<td>4.I Read Mare Nostrum documentation</td>
<td>MD, MS</td>
<td>2.5, 1</td>
<td>45.5 € + 14.77 €</td>
</tr>
<tr>
<td>4.II Prototype scripts for Mare Nostrum development environment</td>
<td>MD</td>
<td>10.5</td>
<td>191.1 €</td>
</tr>
</tbody>
</table>

Table 1.6: Per task costs (1). The second half is in table 1.7

schedule instead of task by task. Weekly control of time employed will be done. With the real cost one can get the deviation by subtracting to it the estimated cost.

1.12 Project sustainability and viability

The dimensions for sustainability can be explained in terms of performance (power consumption reduction, economical viability), code maintainability (social impact) and investment needed (economical viability). Following subsections try to correctly justify them. There is a last section showing the evaluation based on previous explanations.

1.12.1 Economical viability

This project does not sell any product. At most it sells the development work of a new version of a physics simulation library. This project has been already assigned so its viability is guaranteed as the costs are acceptable for the contractors. Being it a research project it does not come with any direct benefits for the investor. That would only be the case if the investor was not interested only in analyzing OmpSs performance exemplifying it with the port of Facesim and the libraries it uses, but also if it wanted to exploit the port in commercial environments. But that is out of the reach of this project.

In respect to the duration of the project, no project could be done in less time than this one. The main developer has no previous knowledge of the application domain so that has to be considered.
<table>
<thead>
<tr>
<th>Task</th>
<th>Assets used</th>
<th>Usage time (hours)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. Evaluate current implementations of Facesim</td>
<td></td>
<td></td>
<td>1809.95 €</td>
</tr>
<tr>
<td>5.I Prepare Facesim for evaluation of its current implementations</td>
<td>MD, PD, RP</td>
<td>12, 0.5, 1</td>
<td>218.4 € + 14.85 € + 18.20 €</td>
</tr>
<tr>
<td>5.II Evaluate the serial version of Facesim</td>
<td>MD, PD</td>
<td>28, 0.5</td>
<td>509.6 € + 14.85 €</td>
</tr>
<tr>
<td>5.III Evaluate the current parallel implementation of Facesim</td>
<td>MD, PD</td>
<td>56, 0.5</td>
<td>1019.2 € + 14.85 €</td>
</tr>
<tr>
<td>6. Write evaluation of Facesim sections part of the thesis with the collected data</td>
<td>MD</td>
<td>24</td>
<td>436.8 €</td>
</tr>
<tr>
<td>7. Adapt and port current Facesim code base to OmpSs OmpSs</td>
<td></td>
<td></td>
<td>12742.41 €</td>
</tr>
<tr>
<td>7.I Prepare Facesim for the OmpSs implementation</td>
<td>MD, PD, OD, RP</td>
<td>82, 1, 1, 1</td>
<td>1492.4 € + 27.69 € + 18.20 € + 18.20 € + 18.20 €</td>
</tr>
<tr>
<td>7.II Design a first task-based implementation using OmpSs</td>
<td>MD, PD</td>
<td>280, 3.5</td>
<td>5096 € + 96.20 €</td>
</tr>
<tr>
<td>7.III Implement the task-based design using OmpSs</td>
<td>MD, PD, OD, RP</td>
<td>280, 3.5, 2, 2</td>
<td>5824 € + 96.92 € + 36.4 € + 36.4 € + 36.4 €</td>
</tr>
<tr>
<td>8. Evaluate the OmpSs version of Facesim</td>
<td></td>
<td></td>
<td>582.4 €</td>
</tr>
<tr>
<td>8.I Evaluate the OmpSs version of Facesim</td>
<td>MD</td>
<td>40</td>
<td>145.6 €</td>
</tr>
<tr>
<td>9. Write final report with all the remaining collected data</td>
<td>MD</td>
<td>40</td>
<td>728 €</td>
</tr>
<tr>
<td>TOTAL:</td>
<td></td>
<td></td>
<td>22726.05 €</td>
</tr>
</tbody>
</table>

Table 1.7: Per task costs (2). The first half is in table 1.6

**Final cost versus prediction**

This project has met the predicted costs, there have not been any deviations. Thus the maximum score is given.

**Adaptation to changes in scenario**

Even with the small change in the objectives, this project has ended in time. The score does not achieve the maximum puntuation due to not having predicted the need for doing also an analysis of the underlying service platform (datacenters or clusters) on top of which an OmpSs application would run.
<table>
<thead>
<tr>
<th>Asset</th>
<th>Unitary price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human resources</td>
<td>22726.05 €</td>
</tr>
<tr>
<td>Hardware resources</td>
<td>675.10 €</td>
</tr>
<tr>
<td>Software resources</td>
<td>0 €</td>
</tr>
<tr>
<td>Indirect resources</td>
<td>253.2 €</td>
</tr>
<tr>
<td><strong>TOTAL:</strong></td>
<td><strong>23654.35 €</strong></td>
</tr>
</tbody>
</table>

Table 1.8: Final budget

<table>
<thead>
<tr>
<th>Asset</th>
<th>Unitary price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final budget</td>
<td>23654.35 €</td>
</tr>
<tr>
<td>Contingency of 5%</td>
<td>1182.72 €</td>
</tr>
<tr>
<td>Impediment bugs OmpSs (10%)</td>
<td>0.10*5824=582.4 €</td>
</tr>
<tr>
<td>Impediment implementation (30%)</td>
<td>0.3*5824=1747.2 €</td>
</tr>
<tr>
<td>Impediment using Mare Nostrum (10%)</td>
<td>0.10*191.1=19.11 €</td>
</tr>
<tr>
<td><strong>TOTAL with impediments:</strong></td>
<td><strong>27185.78 €</strong></td>
</tr>
</tbody>
</table>

Table 1.9: Budget considering also risks

1.12.2 Social viability

This project is developed in Spain for the innovation center Barcelona Supercomputing Center. This project can improve the expectations of BSC to get more traction in research, obtaining more projects and opportunities for research.

Individually speaking, using OmpSs makes code more maintainable. When adapting current software to OmpSs, it allows to focus more on the algorithms than on how to implement the algorithm. That is a factor for improving life quality of developers. Final users would also receive benefits in form of more parallel and concurrent programs being developed so they spend less time waiting for results of their programs. More useful work per time unit.

Social environment impact

The environment impact has been in the expected range or even better due to the improvements accomplished with the OmpSs adaptation. There is a sizable reduction in code size, benefitting greatly the software development thanks to OmpSs. Facesim’s OmpSs adaptation code is more maintainable than the original code base.

Social damage

No social damage has been produced.

1.12.3 Environmental viability

The main point to be considered for the environmental viability is the power consumption. Power consumption is directly related with execution time. There’s no doubt that the OmpSs adaptation will have better performance than the serial version. As one of the objectives is to show that OmpSs is able to achieve the same performance as Pthreads or better it, the performance gain to take into account should be at least the same as with Pthreads. From [10] the theoretical maximum speedup (gain in performance in respect to the serial version) is 15x. Speedup is calculated as follows:

\[
\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]
Table 1.10: Table for controlling expenditures. For each task the estimated hours and real hours are annotated. Then, with the resources of each task and their costs, both the estimated and real costs can be obtained.

<table>
<thead>
<tr>
<th>Task</th>
<th>Estimated hours</th>
<th>Estimated cost</th>
<th>Real cost</th>
<th>Real hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Task N</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

That means that a parallel version has the potential to be fifteen times quicker than the serial code. That is the theoretical maximum speedup. As previously said, real speedup should be in the same league as pthreads or better. That means an speedup of $8 - 12x$.

For a physics simulation, the performance gain will have a direct effect in power consumption. Using this software in multiple computers for multiple visual effects renders can mean the usage of multiple MW (Mega Watts) of power. So if the adaptation to OmpSs does have better performance than the current one, there is the possibility of saving even more energy when executing in parallel if the implementation was to be commercially exploited.

**Resource usage**

The assigned resources have been used no more than the amount predicted at planning stage.

**Environmental damage**

No environmental damage other than the predicted during the planning stage has been produced. In fact, the improvement in the efficiency of Facesim benefits the environment. Up to 20% of improvement in execution time.

### 1.12.4 Sustainability evaluation

Given the previous explanations, the sustainability matrix is evaluated in table 1.11:

<table>
<thead>
<tr>
<th>¿Sustainable?</th>
<th>Economics</th>
<th>Social</th>
<th>Environmental</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planification</td>
<td>Economical viability</td>
<td>Improvement in life quality</td>
<td>Resource analysis</td>
</tr>
<tr>
<td>Valuation</td>
<td>7</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Results</td>
<td>Final cost versus prediction</td>
<td>Social environment impact</td>
<td>Resource usage</td>
</tr>
<tr>
<td>Valuation</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Risks</td>
<td>Adaptation to changes in scenario</td>
<td>Social damage</td>
<td>Environmental damage</td>
</tr>
<tr>
<td>Valuation</td>
<td>-5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total valuation</td>
<td>46</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.11: Sustainability matrix. The valuations can be negative, meaning those items can negatively affect
Chapter 2

Service Platform

This project uses and tests MareNostrum cluster and part of the software stack (OmpSs) which runs on top of the OS and daemons of MareNostrum. This can give a view of a possible system deployment as could be for any other cluster or data center of a company needing one. Following sections study MareNostrum and the administration tools running it. In this chapter OmpSs is also presented, forming a software stack for a cluster.

2.1 MareNostrum

MareNostrum is a supercomputer employed for research in a vast amount of areas. As a cluster, MareNostrum is composed of a number of racks with each one containing a number of nodes, that is, computers. All of those computers are interconnected so they can communicate with each other. Some of those computers have different uses, providing the different services the cluster offers to the customers (research institutions, companies doing research and innovation). The main services the cluster provides are computing resources, storage and backups. Additionally, the computer can be accessed remotely over the Internet via Secure Shell, providing a secure connection. For security reasons, there is no access to the Internet from within the nodes of the cluster. There are different roles for the nodes: storage, compute, backups, login and management nodes. The login nodes provide remote access via Secure Shell. The backup nodes allow to access long term storage from the login and compute nodes. Compute nodes are the ones in which user applications are executed. Storage nodes provide access to the file systems used in daily basis. Management nodes provide access to system administrators. They serve as support for the rest of nodes, including the first installation and configuration of the OS on the different nodes. During cluster operation the management nodes allow administrators to control, monitor and operate the whole cluster.

The whole set of nodes are interconnected through various networks. There are three in total. One is used for the cluster administration. Another one is used to give nodes access to the storage nodes. The last network is employed for fast communication between compute nodes, allowing the execution of distributed applications across different nodes. Each network has different bandwidth and latency requirements, as the characteristics needed to access networked storage are different than the ones needed during software executions and communications between distributed processes, as is typical of HPC applications. That means each network is implemented with the proper network standard.

The main software used to configure and operate the cluster is xCAT (Extreme Cloud Administration Toolkit) and IBM LSF Platform, a batch scheduling system. xCAT allows to control, monitor and administer the cluster services like DNS and DHCP. LSF is the software which allows users send jobs to the cluster, selecting the node or nodes depending on the job specification and the available resources.\footnote{Information on MareNostrum architecture has been obtained during the use of MareNostrum and through documentation available both to the users and system administrators. See \cite{29}.} As per [1, p. 12], MareNostrum can be considered a datacenter following a server cluster model. These type of clusters are now also part of enterprises due to growing needs in diverse areas such as financial trending analysis, film animation, manufacturing or even search engines.
Next two sections talk about the computer itself and the software used for managing it.

### 2.1.1 General scheme of the system

This section explains how is the cluster interconnected and what it is composed of. This section is split in different subsections, talking each one about an aspect of the system, which are mainly two: the nodes and networks it has and the filesystems it uses. After this section there is a final section talking about the administration software used to manage MareNostrum, Extreme Cloud Administration Toolkit.

#### Nodes and networks of the cluster

After the introduction to the system, a good way to expand on it is to show a scheme on how the nodes are distributed in their physical space. The cluster is in a glass jail and it is maintained at a constant temperature. Figure 2.1 presents a diagram of how the cluster is distributed.

![Figure 2.1: MareNostrum site physical distribution. At the left side there are the storage racks (DX), the management racks (MX) and the Infiniband network racks (IBX). On the center-right all the compute racks can be found.](image)

Compute nodes are in racks of 84 nodes each. Additionally, each compute rack has four 36-port Infiniband switches and four Gigabit Ethernet switches. The Infiniband switches are the ones employed to interconnect the different compute racks, offering 40 Gbps of low latency bandwidth. The Infiniband switches of each rack are connected to the Infiniband racks via 18 optical links. The Gigabit Ethernet switches are used to access the networked storage and for administration purposes. The administration switches provide 2 Gbps of uplink bandwidth through two copper Ethernet links while the storage switches provide 40 Gbps of uplink bandwidth through four optical Ethernet links. Infiniband is a standard for low latency and high bandwidth networks which has replaced previous proprietary network standards like Myrinet, a network interconnect developed by Myricom.

MareNostrum has 3028 compute nodes. With the command `lshosts` of the LSF batch scheduler system one can list the current available hosts for application execution. An extract of the output can be seen in listing 2.1.

**Listing 2.1: lshosts output extract**

<table>
<thead>
<tr>
<th>HOST_NAME</th>
<th>type</th>
<th>nodell</th>
<th>cpuf</th>
<th>ncpus</th>
<th>maxmem</th>
<th>maxswp</th>
<th>server</th>
<th>RESOURCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched1</td>
<td>X86_64 Intel_ELK</td>
<td>60.0</td>
<td>8</td>
<td>31.9G</td>
<td>3.9G</td>
<td>Yes</td>
<td>(ng)</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>Type</td>
<td>ID</td>
<td>CPU</td>
<td>Memory</td>
<td>Storage</td>
<td>Available</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>--------</td>
<td>---------</td>
<td>-----------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>login4</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>31.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>login5</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>31.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s01r1b01</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>127.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s01r1b02</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>127.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s01r1b03</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>127.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s01r1b04</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>127.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s01r1b05</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>127.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s01r1b06</td>
<td>X86_64 Intel EM</td>
<td>60.0</td>
<td>16</td>
<td>127.9 G</td>
<td>11.1 G</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some new nodes were added recently, those with the `mic` resource. Those nodes have accelerators for heterogeneous applications. The login nodes from which the users can access the cluster resources are also present along the job scheduler servers and monitoring servers. There are also other special execution nodes like `smp1` and `haswell01` while `cabeza1` purpose is not known to the author of this document. `Mon1`, `Mon2` are used for monitoring the cluster.

There are more nodes than those appearing under the LSF batch system. Those are the management nodes. MareNostrum has multiple nodes dedicated to management activities. All those activities are orchestrated thanks to xCAT, the Extreme Cloud Administration Toolkit[4]. A logical representation of the cluster can be seen in figure 2.2.

The management nodes are the ones which allow system administrators to operate the whole cluster. They are divided in various groups based on their roles. There are four groups: the masters, the GPFS servers and the service nodes. All of these nodes are considered “xCAT nodes” as they have xCAT installed in order to be able to operate the cluster. xCAT is a toolkit prepared for administering clusters and it has several node profiles depending on the role the node will play. Depending on the size of the cluster there might be one or more management nodes. Additionally it also might have an external file system to serve compute nodes. In the case of MareNostrum the external file system is GPFS (thus the servers are “xCAT GPFS servers”) and is mounted via CNFS (Clustered NFS) from the different nodes. CNFS is the method GPFS uses to share the file system over NFS (Network File System). MareNostrum’s compute nodes boot from network and the root file system is mounted via that NFS file system from the xCAT GPFS servers. The xCAT GPFS servers store the OS image of the nodes and also logs and configuration files. The local disk of the nodes is only used for swap and temporary data of job executions.

Next subsection talks about the filesystems used in MareNostrum and how to access them.

### Filesystems

MareNostrum has different filesystems. In the previous section they were introduced briefly. Mainly, MareNostrum has three different filesystems: GPFS filesystem, Active Archive filesystem and the GPFS filesystem mounted only on the xCAT (management) nodes. This section introduces first the GPFS filesystem available to the nodes and users. Afterwards it talks about the Active Archive filesystem and how to access it. The GPFS filesystem mounted on the xCAT nodes is for storing the images of the nodes. Those images are downloaded at node boot time and mounted as the root file system of each node through NFS. More on that is explained in the section corresponding to the Extreme Cloud Administration Toolkit.

In listing 2.2 there is the output of the `mount` command from a login node. Apart from having the GPFS file system used for storing users data, it also has mounted over NFS the root filesystem (`s03-cnfs:`...),
Figure 2.2: MareNostrum system. A user connects via Internet to a login or data transfer node. From login nodes he can send jobs and access some distributed file system directories. From the data transfer nodes the user can interactively access the long term storage, the Active Archive. That allows to do backups from the GPFS file system available to the login and cluster nodes. The management nodes are a group of nodes which include both the xCAT service nodes and the xCAT management nodes.

mounted at boot time. Listing 2.3 presents the /etc/fstab file with the daily usage user GPFS mounts (applications, home, project and scratch directories). The nodes employed for backups (data transfers) have one additional GPFS directory available, the one corresponding to the long term storage, that is, the /gpfs/archive directory. The GPFS mounts for everyday usage are in the storage racks of the cluster, totalling 1.9 PB of space. The xCAT GPFS file system from which the nodes mount their root file system resides in the management racks. It has a size of 9 TB. The xCAT GPFS file system is accessed through the management network while the user GPFS file system is accessed through the GPFS network. The infiniband network is only used for communication between applications executing on nodes.

Listing 2.2: mount output extract. s03-cnfs... file systems are mounted through NFS at boot time. /dev/gpfs devices correspond to the GPFS servers available to login, compute and data transfer nodes.

<table>
<thead>
<tr>
<th>Device</th>
<th>Mount Point</th>
<th>Type</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>s03-cnfs:/install/netboot/sles11.3/x86_64/compute/rootimg</td>
<td>/</td>
<td>rootfs</td>
<td>(rw)</td>
</tr>
<tr>
<td>s03-cnfs:/gpfs/xcat/statelite</td>
<td>/statelite</td>
<td>tmpfs</td>
<td>(rw, relatime, mode=755)</td>
</tr>
<tr>
<td>rw on /statelite</td>
<td>type tmpfs (rw, relatime, node=755)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 2.3: /etc/fstab file content.

```bash
# /etc/fstab

# Rootfilesystem
rofs on / type rootfs (rw)
s03-cnfs:/install/netboot/sles11.3/x86_64/compute/rootimg on / type nfs (ro,relatime, vers=3,rsize=32768, weize=1048576, namlen=255, hard, nolock, proto=tcp, timeo=14, retrans =2, sec=sys, mountaddr=X.X.X.X, mountvers=3, mountproto=tcp, local_lock=all, addr=X.X.X.X)

# User Home
rw on /statelite type tmpfs (rw, relatime, node=755)
s03-cnfs:/gpfs/xcat/statelite on /statelite/persistent type nfs (rw, relatime, vers=3, rsize=1048576, weize=1048576, namlen=255, hard, nolock, proto=tcp, timeo=600, retrans=2, sec=sys, mountaddr=X.X.X.X, mountvers=3, mountproto=tcp, local_lock=all, addr=X.X.X.X)

# Project and Scratch
s03-cnfs:/install/netboot/sles11.3/x86_64/compute/rootimg/.default/var/lib/rpm on /var/lib/rpm (deleted) type nfs (ro,relatime, vers=3,rsize=1048576, weize=1048576, namlen=255, hard, nolock, proto=tcp, timeo=600, retrans=2, sec=sys, mountaddr=X.X.X.X, mountvers=3, mountproto=tcp, local_lock=all, addr=X.X.X.X)
s03-cnfs:/install/netboot/sles11.3/x86_64/compute/rootimg/.default/var/lib/rpm on /var/lib/rpm (deleted) type nfs (ro,relatime, vers=3,rsize=1048576, weize=1048576, namlen=255,
```
As listings 2.2 and 2.3 show, the long term storage in /gpfs/archive cannot be accessed. That is only available to access interactively from the data transfer nodes. Non interactive access is available from any node of the cluster thanks to batch commands. Those commands wrap “cp, mv, tar...” commands and send them to a data transfer batch system. Of course users do not need to use them to have backups of their $HOME directories, as that file system (/gpfs/home) is backed up. But the $SCRATCH directory which is recommended for storing execution results has no backup, so is up to the user where to store the data in the cluster. The batch script for doing a cp can be seen in listing 2.4. All batch data transfer commands have the same name as the commands they wrap, with the dt prefix prepended. The available data transfer commands are: dtcancel, dtchecksum, dtchmod, dtcp, dtcp_md5, dtls, dtmdir, dtmv, dttar. Each one does the equivalent to the not dt command. dtchecksum and dtcp_md5 are wrappers for md5sum. The latest (dtcp_md5) performs an md5sum on the copied files. dtq, dtquota and dtcancel are commands to check pending data transfers, available quota on the Active Archive (/gpfs/archive) and to cancel data transfer jobs, respectively.

Listing 2.4: dtcp command script

```bash
my $CMD="cp -v " . join (" " " , $ARGV) . " "$;
if (! $blocking) {
    print OUT "#!/bin/sh
    # $partition
    # $job_name = dt_copy
    # $initialdir = .
    # $error = dtcp_md5.err
    # $output = dtcp_md5.out
    # $total_tasks = 1
    # $wall_clock_limit = $TIME

    "$ or myerror("Can not write to $SCRIPTNAME. /tmp full? use another login.");
}

"$ or myerror("Can not write to $SCRIPTNAME. /tmp full? use another login.");
close OUT or myerror("Can not write to $SCRIPTNAME. /tmp full? use another login.");
```
The variables are set before sending the job script. If the script is non-blocking, it sends the copy command ($CMD) to the batch system. If it is blocking, then it executes the command in the current node and the user can not do nothing in the terminal until the copy ends. The variable $SUBMIT contains the command and its parameters to execute the data transfer.

The batch data transfer system employs special nodes called amovers. Those nodes have the GPFS and long term storage (Active Archive) mounted, allowing to do transfers between both file systems. These nodes allow the rest of the system to access Active Archive, including compute and login nodes. To access amovers, users must use the batch commands explained previously. In the global picture of the system, the amovers would appear as additional nodes accessible through a batch system from the cluster nodes, login nodes and data transfer nodes. Their only purpose is to allow management of files.

**Job scheduling**

Until now this section has talked about the different types of nodes, the network and file systems. There is an additional component which is the one that actually allows to send jobs to the compute nodes: the batch scheduler system. MareNostrum uses IBM platform LSF (Load Sharing Facility) to manage jobs created by users and send them to the appropriate nodes. Users access the login nodes and with the command `bsub` they can submit jobs to the cluster. A job basically is a script with some directives in its header which indicate the batch system what resources does the user need or want. The batch scheduling system will schedule that job according to those directives. An example of that was already seen in the `dtcp` data transfer script. LSF has multiple queues configured and monitors machine resources for proper resource management.

When a user submits a job with the `bsub` command, the command reads the input script and submits it to the scheduler server. When the job can start execution, the scheduler server sends the job to the appropriate node or nodes and spawns the process that the script contains. Usually a `bash` process that executes the contents of the script. After the job execution finishes, LSF cleans the node of any remaining processes and files with an `epilog` script. Currently that script basically cleans the temporary directory used by the job, that is, the local disk of the compute nodes it has been executed on. That is the only interaction the cluster has with the user. Through the scheduler server. The compute nodes are not connected to the Internet and there is no way they can be accessed from the Internet. All external connections go to the login and data transfer nodes. The scheduler server is configured at LSF installation along all the names of the compute nodes in the cluster.
Figure 2.3: MareNostrum batch scheduler overview. The user sends a job, cancels it or queries the state of sent jobs. Those commands are sent to the master node.

Job Scripts

Using a datacenter or cluster comparable to MareNostrum needs the skill to use the batch queue system. LSF uses the command `bsub` to submit jobs to the computer. That command also accepts scripts as input. Those scripts have a header which are passed to `bsub` as options. Then, `bsub` will select the proper queue and set the adequate execution parameters in order to get LSF satisfy those needs like queue to use, time limit, priority and node resources like number of cpus and number of nodes.

As an example of usage of a cluster, during this project a number of scripts have been developed, ready for anyone to use with their own applications. Script in listing 2.5 presents the code of a script used during development of this project which executes Facesim.

Listing 2.5: Job Script, submitter

```bash
1 BENCHMARK=$1
2 VERSION=$2
3 INPUT=$3
4 OMPSSTHREADS=$4
5 NDIVS=$5
6 ITERS=$6
7 BENCHID=$(BENCHMARK)-$(VERSION)-$(INPUT)-$(OMPSSTHREADS)
8 DATE=$(date +%F_%H.%M.%S)
9 [...]
10 case $VERSION in
11   openmp* | ompss* | pthreads* | serial*)
12     echo -e "\033[0;32mVERSION = $VERSION, OK\033[0m"
13     BEKCHID+="-$(NDIVS)"
14     ;;
15 [...]
16 esac
```
case $INPUT in
  native | simlarge | simmedium | simsmall | simdev | test)
    echo -e \"\033[0;32mINPUT = $INPUT, OK \033[0m\"
    ;;
esac
[...]
#generate script and submit script
[...]}
echo -e \"\033[1;33mGenerating and submitting ${BENCHID} with ${OMPSSTHREADS} threads \033[m\"
#BSUB -n 1
#BSUB -R "$\"affinity\[core(${OMPSSTHREADS})])\""
#BSUB -o ${SCRATCH}/${BENCHID}-${DATE}-run.out
#BSUB -eo ${SCRATCH}/${BENCHID}-${DATE}-run.err
#BSUB -J ${BENCHID}-run
#BSUB -q bsc_cs
#BSUB -W 1:30
#BSUB -x

cd "$TMPDIR"

for (( iter=0; iter<${ITERS}; iter++ )) do
  echo " ITERATION $iter"
  $ROOT/$BENCHMARK/run.sh $VERSION $INPUT $OMPSSTHREADS $NDIVS > $(BENCHID)-$(DATE)-$(iter).dmp
done

" > temp.run
bsub < temp.run
rm temp.run

The script does parameter checking and then generates the corresponding LSF compatible script writing it into a temporary file which is passed to the bsub command. This script allows for multiple executions of an application within the same submitted job. The run.sh script of the application manages the data needed by the application and its output. The generated job script starts with the #BSUB directives. Different options are used: -n to select the number of processors to use, -R to bind the job to as many cores as threads the application will create. -eo, -oo to indicate the name for the standard output and standard error files to which LSF redirects the output of the execution. -J to put a name to the job. -q to select the execution queue. -W to specify the limit of time this job will have before being terminated by LSF. Finally -x to indicate to LSF that this job needs the compute node in exclusive mode. The node to where the script will be send for execution will not receive any other job until this job is terminated. The $SCRATCH variable contains the path to the directory of the user in the /gpfs/scratch file system.

Conclusions

MareNostrum is quite a complex cluster and has a lot of infrastructure to allow management of the cluster and let users use the system and perform backups of their data. It has three different networks to fulfill different purposes and give a good service. MareNostrum has more than 2000 nodes available for compute and has 1.9PB of shared storage accessible. Additionally, there is the active archive with 3.8 PB of available space. Those are not the only file systems available but there are an additional 9TB of space in the management nodes which store the OS images of the different nodes, including login and compute nodes. All of those file systems are GPFS file systems. MareNostrum also has great tools available for administering the cluster and run application executions on it. Those tools are xCAT and IBM LSF Platform.

Next section does a more detailed description of xCAT, which is the critical software which allows to operate a cluster or datacenter.
2.1.2 Extreme Cloud Administration Toolkit

No system exists without it being managed in order to maintain it working for long periods of time. Maintaining a cluster or datacenter in a functional state is no different. Given the high number of computers a machine like that can have, some specialized tools have been developed in order to allow system administrators to properly configure and monitor a cluster which can have a size of thousands of nodes. Checking each node one by one would take too much time. xCAT is one of those tools and is one of the most used. xCAT enables managing large number of servers including HPC clusters, web farms, online gaming infrastructure and datacenters. It has support for different operating systems, hardware and virtualization platforms. xCAT is made by IBM and is free software licensed under the Eclipse Public License. Next sections talk about xCAT architecture and the main options which allow to configure a system in order to have it ready to boot.

xCAT Architecture

xCAT can be used on both small and big systems. Management is accomplished through a management node which is executing xcatd daemon. xCAT also allows to have a hierarchical infrastructure, adding service nodes. Each service node is in charge of managing a group of nodes of the cluster/datacenter. In the case of MareNostrum, it uses a hierarchical architecture, having multiple service nodes and two management nodes.

![xCAT Architecture Diagram]

Figure 2.4: xCAT architecture. Management node and service nodes allow to contact the different configured nodes and remotely administer them. xcatd daemon can be accessed and used through command line interface, web GUI or even a REST API. Think of CUPS, the Common Unix Printing System which can also be managed through both command line and web interfaces. Note that there are two networks for management.

To store its configurations, xCAT uses a database. xCAT provides multiple commands to be able to manipulate the tables of the database. Common tables to be filled by administrators are described in table 2.1. The database can be MySQL, PostgreSQL, SQLite or DB2. If using a hierarchical architec-
ture, default SQLite can not be accessed by remote service nodes. In that case is preferred that the
Management node runs other databases. PostgreSQL or MySQL are good examples of such databases.

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site</td>
<td>It stores global settings of the whole cluster like timezone, master node, install directory, name servers and dhcp interfaces.</td>
</tr>
<tr>
<td>Networks</td>
<td>All the networks in a cluster are defined here. Some xCAT commands access this database to set up services.</td>
</tr>
<tr>
<td>Nodelist</td>
<td>Maps nodes to node groups commonly used to define attributes obtained from the tables. Groups allow simplified control over nodes with a single command line.</td>
</tr>
<tr>
<td>Noderes</td>
<td>Maps nodes or node groups to servers, services and network boot method. It specifies the resources to use. It also defines what NIC to use on the target nodes.</td>
</tr>
<tr>
<td>Nodehm</td>
<td>Maps nodes or node groups to a hardware management method like IPMI.</td>
</tr>
<tr>
<td>Nodetype</td>
<td>Defines for each node or node group the OS version, architecture and “profile”. A profile defines the osimage or software packages to use for the nodes.</td>
</tr>
</tbody>
</table>

Table 2.1: xCAT tables which have to be filled in order to cater an specific cluster configuration. As for management hardware, nodes can have specialized devices for management. Common management devices in IBM machines are IMM (Integrated Management Module). Other vendors like HP and DELL also have their own devices. xCAT can be configured to run with them. All of them are normally connected to the management Ethernet network and can use their own VLAN.

In order to deploy nodes, xCAT provides different provisioning methods. By provisioning xCAT means how is the software installed on the nodes. The provisioning methods are Stateful, Stateless and Statelite.

Node provisioning

There are different ways to load the operating system onto a node. xCAT provides three different ways. The process of creating an OS image is readily available on the xCAT documentation, on the Internet.

The compute nodes are basically computers. The OS image is thus based on a common OS (usually GNU/Linux). All the management commands are executed from the management nodes. The process of generating and OS image starts from copying the contents of an ISO file containing the OS image to the /install directory of the management node with the command copycds which automatically fills the tables (nodetype) corresponding to the definitions of the image parameters of the nodes. Then the administrator can personalize the image on /install directory to fit his needs, like additional software packages. xCAT commands fill the tables accordingly. The nodes will download at boot time the kernel image from the management node and afterwards will load the rest of the OS in the /install directory.

As for the management node, its OS is installed as in any computer, from the installation disk. From there the administrator installs xCAT and configures it. A full procedure for a cluster based on IBM iDataPlex nodes like the ones found in MareNostrum can be found in xCAT Wiki documentation.

It lacks some detail on the configuration of managed switches as different vendor switches can be used, leaving it up to the administrator to know how to configure the needed options on them. With this small primer on how OS images are prepared, this section continues with the different types of provisioning methods for those OS images.

Stateful nodes have the OS installed on their disk like a standard computer. Each node OS has to be configured and managed individually. Stateless provisioned nodes boot from a RAMdisk OS image.
downloaded from the xCAT management node at boot time. This allows to manage only one OS image and the nodes do not end out of service if the management or service nodes go out of service. That is because each node has the OS in their RAM. The last provisioning method, stateelite, means nodes boot from an NFS-root OS image. Because the root filesystem is mounted over NFS, the image can be bigger than with stateless which loads the root filesystem in RAM. The drawback is that the NFS mount is in the management node or an external NFS server, so if the NFS server or management server go down, all the stateelite nodes end out of service. It also requires more bandwidth as the root file system is accessed through network. Depending on the type of load the nodes have to manage, the network bandwidth might not be enough. In order to mitigate the possible bottleneck of the networked access to the root file system, the file system can be configured with a cache.

Stateelite can also has some persistent files and directories. The “persistence” of files is configured in xCAT litefile table of its database. That table defines for each directory specified in it, the type of persistence it should have. The persistent files are stored in ~/.statelite/persistent directory. Doing an ls on that directory from a login node returns a list of directories, each one with the name of the node the directory belongs to. Listing 2.6 shows part of the output.

Listing 2.6: MareNostrum’s persistent directories

<table>
<thead>
<tr>
<th>Directory</th>
<th>Owner</th>
<th>Group</th>
<th>Size</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.0K</td>
<td>2013</td>
</tr>
<tr>
<td>login1</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  9</td>
</tr>
<tr>
<td>login2</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  9</td>
</tr>
<tr>
<td>login3</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  9</td>
</tr>
<tr>
<td>login4</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  9</td>
</tr>
<tr>
<td>login5</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  9</td>
</tr>
<tr>
<td>master</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Nov  20</td>
</tr>
<tr>
<td>s01r1b01</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Dec  19</td>
</tr>
<tr>
<td>s01r1b02</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  8</td>
</tr>
<tr>
<td>s01r1b03</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  8</td>
</tr>
<tr>
<td>s01r1b04</td>
<td>root</td>
<td>root</td>
<td>4.0K</td>
<td>Jul  8</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MareNostrum uses stateelite provisioning system. The nodes have the root filesystem mounted over the network with NFS. Listing 2.7 shows the boot command line of MareNostrum 3.

Listing 2.7: MareNostrum 3 kernel boot command line, from file /var/log/boot.msg

```
Command line: BOOT_IMAGE=net4:/tftpboot/xcat/osimage/sles11.3-x86_64-stateelite-
compute/kernel NFSROOT=s03-cnfs:/install/netboot/sles11.3/x86_64/compute
STATEMENT=s03-cnfs:/gpfs/xcat/statelite XCAT=s03:3001 quiet netdev=eth2 console=
tty0 console=tty50,115200MNTOPTS='rsize=32768,wsize=32768' swapaccount=1 BOOTIF=
x
```

The kernel image is downloaded at boot time via TFTP and then the kernel mounts the root filesystem via NFS with the configuration provided on the boot command line passed to the kernel. MareNostrum uses additional NFS servers external to the management and service nodes. A complete communication process during the boot of a stateelite node can be seen in figure 2.5.

### 2.1.3 Final thoughts on the studied cluster infrastructure

This chapter has done a study on the architecture of a cluster and its administration tools. The administration of a cluster is not an easy task and as such some tools have been developed to make the task easier for system administrators. The use of standard hardware like the one used in MareNostrum (Intel processors, ethernet switches, infiniband switches) allows to lower the cost of the IT equipment and mitigates any barriers in the case a company wants to change to another provider of IT equipment. Software would not need to be rewritten and the network interconnects can be reused.

The Extreme Cloud Administration Toolkit has proven to be a very powerful tool although this chapter has only grasped the surface of that tool. It supports a wide variety of hardware and operating systems. It allows multiple configurations and datacenter architectures, with different servers for different purposes and allow all of them to be managed from a single location, the management nodes. xCAT has an
Figure 2.5: xCAT boot process of a statelite node. At boot time the PXE ROM in the NIC contacts the corresponding management node in order to get the network configuration, TFTP daemon IP address and bootfile name. Then, it requests to the TFTP server the bootloader executable and passes the execution control to the bootloader. The bootloader asks the needed configuration to the HTTP daemon and downloads the Linux kernel and the initial file system (initial RAMdisk, initrd) used by the kernel. Then, the Linux kernel mounts the root file system and persistent directories, including the litetree directories and files. Finally it switches from the initial RAMdisk to the root file system and executes the final configuration scripts downloaded from the xCAT daemon. All daemons belong to the same management node. The management node to answer requests is configured in the xCAT management and service nodes, according to the management group to which the node is assigned.

important fact in favour: open documentation. Different guides with examples are being written by IBM and users of xCAT. Having good documentation is very important to save time.

As a whole, both xCAT and IBM cluster/datacenter products are something to have in consideration for any organization wanting to install such kind IT infrastructure.

Next section talks about OmpSs, the proposed programming model for a cluster/datacenter and a cloud
oriented derivative, COMPS.
2.2 OpenMP Super-scalar (OmpSs)

OpenMP Super-scalar (OmpSs) is a programming model which allows easier threaded software programming as opposed to Pthreads, which is a lower level API. An OmpSs application has a pool of threads which execute code annotated by the user with compiler directives. Other threaded programming models like OpenMP create and destroy threads instead of maintaining a pool of threads during the whole application execution. While OpenMP employs what is named a fork-join model (creating and destroying threads), OmpSs is a task-based model. Threads are created at the start of the application and are destroyed before the application exits\[30\]. During execution each annotated code section is wrapped in a task and put into a queue or multiple queues waiting for a thread to execute that task.

Tasks allow to express data dependencies. If a task writes to a variable named A and another task created afterwards reads or writes to that same variable, then it will not be executed until the task which writes to variable A ends its execution\[30\].

The code annotations in OmpSs are made of compiler directives, which in the case of C and C++ languages means using #pragma directives. Those pragmas allow to specify the variables the tasks read from or write to. A small example can be seen in listing 2.8

```
Listing 2.8: Two tasks example

void main(int argc, char** argv) {
  int A;
  #pragma omp task inout(A) label(functionA)
  {
    functionA(&A);
  }
  #pragma omp task inout(A) label(functionB)
  {
    functionB(&A);
  }
  doOtherThings();
  #pragma omp taskwait
}
```

In the previous example, 2.8, the main thread creates two tasks to be executed by other threads. When a thread executes one of those tasks, it will execute the code inside the #pragma. Note that the curly braces are not needed for only one line of code. Thanks to the dependency specification, the second task will not be executed before the first task ends. Meanwhile, the main thread can doOtherThings() and then #pragma omp taskwait until all the created tasks have finished execution.

All of these newer programming models for creating threaded applications pursue the objective of making the creation of such software more straightforward. Applications like web servers can benefit from better programmability with less lines of code and a cleaner code base. Some work has been done in concurrent software like web servers. In \[2\], Boa server was modified by adding a new Pthreads, OpenMP and dynamic sections implementations. Dynamic sections can be thought of something similar to using tasks but still less flexible than tasks. With dynamic sections a lot of unneeded code was removed and application performance went up providing better concurrency and accepting more clients. Response time diminished. Both OpenMP and dynamic sections allowed quicker development time and simpler code, while Pthreads needed more time and complexity would increase if other type of tasks different than attending requests were needed to be processed.

The directives used by OmpSs are processed by a source to source compiler named Mercurium, which transforms the code with the data annotations into calls to the runtime library actually implementing the services of the programming model: nanox. Thus, OmpSs is composed by the Mercurium compiler and the nanox library. The code generated by Mercurium is passed to a “native” compiler like GCC. Mercurium invokes the native compiler with the proper flags in order to be able to compile against nanox library. Figure 2.6 pictures the transitions.

OmpSs is able to generate code for heterogeneous systems, invoking the required native compiler of the corresponding source code. Data transfers between different address spaces is also managed automati-
cally, including heterogeneous systems. The user only has to properly annotate data inputs and outputs of the tasks.

![Diagram](image)

Figure 2.6: The code written by the user is transformed by Mercurium into code with calls to the nanox runtime library. Then that code is passed to the corresponding native compiler which links the object code against the nanox runtime library and generates an executable which uses the nanox services for creating tasks, synchronizing threads, manage memory consistency and dependencies.

Applications programmed with OmpSs can also generate execution traces when the compiler is set to generate an “instrumented” binary. An instrumented binary has additional calls to the runtime library which collect events like functions being executed or thread states. An instrumented binary also gathers performance metrics like duration of each event (how much time does some code take, how much time does a thread spend doing something). All that information is written into a trace file. The trace file can be written into “Paraver” format. Paraver is an application from Barcelona Supercomputing Center which allows to visualize the events occurred during an application execution. With other programming models the programmer has to manually add function calls (instrument) to generate a trace. OmpSs automatically generates that code for a lot of event types.

Next subsection does a small introduction to COMPSs, an environment for cloud computing which is also task based with dependencies between tasks. Although it is not used in this project, it provides knowledge on other software prepared for other environments derived from clusters and data centers.

### 2.2.1 COMPSs

COMPSs or COMP Superscalar, is a software though for executing applications in cloud environments. In environments like the ones commonly deployed in private companies for their own private clouds or
mixture of public and private clouds, having the proper software to use the installations as PaaS\(^4\) is much needed. COMPSs fulfills that scenario of cloud computing by allowing the execution of scientific applications on top of virtualized infrastructures through adapters for different middlewares. Common industrial applications can now be deployed on cloud environments, not being limited to use only grids or supercomputers.

COMPSs achieves cloud agnostic deployment thanks to the use of VENUS-C platform. VENUS-C employs “enactment services” which use connectors to be able to use different cloud providers like OpenNebula, Amazon EC2 or EMOTIVE Cloud. For Windows Azure it uses a specific service named Generic Worker Enactment Service while the rest are available through and OCCI (Open Cloud Computing Interface). COMPSs can also run on more traditional clusters and grids[31].

In the same way as OmpSs does, COMPSs employs tasks with dependencies for doing the executions. In COMPSs a task is a remote call, like in Java RMI or JAX WS, available to the local application. For each parameter of the remote functions in the corresponding Java interface, there is a Java annotation to tell if that parameter is an input or output of the task. COMPSs runtime establishes the chain of tasks and executes them in the correct order.[31]

\(^4\)Platform as a Service
Chapter 3

Facesim description

Facesim is an Intel RMS [18] application originally developed by the University of Stanford. Its goal is to obtain visually realistic animations by simulating the physics involved in the motion of a 3D model.

Facesim belongs to a suite of benchmarks named PARSEC. Organizations looking for better ways to maintain their infrastructure such as game and movie studios can get an idea of how current programming models or frameworks behave. PARSEC tries to be a representative set of applications for widely used types of workloads. As a consequence, it is a good test bed for programming frameworks and see how they perform. Facesim is an important application of that suite. The future of video games and movies needs a realistic way of animating faces as it is the place where the users focus more when watching a movie or video games. Character faces get a lot of attention and it is important to give them a convincing appearance.

This chapter explains in general terms how does Facesim use the 3D model, what is the input data and the steps the main algorithm takes in order to obtain the final positions of each mesh node of the 3D model. Next sections discusses in general how does Facesim work.

3.1 Facesim simulation setting

Facesim simulation does an animation of a human face. That face is made of a 3D tetrahedra mesh. For each tetrahedron there are associated forces in terms of stiffness, strains and stress of materials. Before starting any computation, there are some parameters to be set for the simulation. Those parameters form a set of constraints which limit the number of computations spent in each frame of the simulation. The constraints also define constants related with physic characteristics of the model.

Physics are computed via the PhysBAM library[3]. That library implements all the functionality related with physics simulation. Facesim loads the 3D model of a human face and sets the needed simulation parameters specified in code by the programmer. Figure 3.1 is an image of the visible parts of the 3D model.

The simulation setting is composed of the simulation parameters and the object or objects to simulate, that is, the face model and its components. Both are described in following sections.

3.1.1 3D model anatomy

The 3D model is a mesh of tetrahedra and their nodes are what is being animated. The mesh represents the flesh of the head. In each frame the position of the nodes has to be computed by solving a system of partial differential equations. The forces affecting tetrahedra determinate the final positions of the nodes found by solving the system.

Forces affecting each node are a mix of activated muscle forces, elastic response of the flesh and passive forces present in muscle regions. Muscle definition is embedded into the 3D face model. For each muscle that overlaps a tetrahedron, the fraction of overlap is stored in the tetrahedron. For each tetrahedron, vectors with the same direction as the muscles overlapping it along the fraction of overlapping is stored.
One vector per muscle. That fraction scales both the active and passive forces derived from the activation of a muscle overlapped with the tetrahedron. All of this data is already included in the anatomical model and Facesim only loads it. [7].

On figure 3.2 there is a simplified and purely informational representation of how muscle forces are applied to the nodes of the mesh. The forces resulting from the muscles end being applied to the tetrahedron vertices. As each vertex can belong to multiple tetrahedra, the final force applied to each node depends on all the tetrahedra having that node. The muscles are not the only source of forces. Edges connecting nodes also contribute forces to each node as they have associated material properties.

Additional parts of the anatomical model are the skeleton bones: cranium and jaw. These parts form special points to which some flesh nodes are attached, adding additional force contributions to those nodes. Cranium and jaw have constraints defining how much they can move.

The model has a total of 30 thousand surface triangles dedicated to the cranium and jaw. The full head has about 850,000 tetrahedra but only the face flesh movements are actually being simulated. That part is composed of 372,126 tetrahedra sharing 80,598 nodes. The number of muscles is 32, going across the whole face (see figure 3.3)[7].

### 3.2 Input data and simulation parameters

The input data is composed of the face model (tetrahedra mesh, triangle surfaces for cranium and jaw) and muscle activation controls for each frame. That data can be found in a single folder inside the
software distribution of Facesim.

The simulation parameters are the number of frames to simulate, the directories where the model is stored, the convergence tolerance for the equation system solver, the maximum number of iterations that the solver will do, whether use non converged results or not and if the simulation will use body collisions for force computation.

All parameters are defined at compilation time except for the number of frames and number of threads to be used during simulation. Those parameters are passed through the command line. For each frame there is a number of muscles active with more or less tension, varying the force being applied by the muscles.

The tetrahedra mesh is split in a number of chunks, depending on the number of threads. For sixteen threads there are sixteen chunks. The data partition is premade. The partitioning consists of dividing the face model and duplicating tetrahedra crossing partitions. Tetrahedra whose nodes are in different partitions are replicated, so each partition can afterwards update their position based state without incurring in any additional synchronization costs[10, 7].

At this point there is enough information about the data Facesim uses. The next section describes the algorithm used for the simulation.

### 3.3 Simulation algorithm

The method employed by Facesim to animate the face is based in a parallel finite element method. That means to operate on a number of elements whose positions are the unknown variables of a system of partial differential equations. In the Facesim case, the elements are the tetrahedra which discretize the face from which the 3D model has been generated. The positions to be found are the positions corresponding to the vertices of each tetrahedron.

The simulation process is done through different stages, executed sequentially. Each stage is parallelized using Pthreads, forming a parallel kernel. Those stages are listed below in the order they are executed.

1. Update Position Based State
2. Add Forces
3. Conjugate Gradient

The Update Position Based State stage updates the current values of the material properties like stiffness and stresses. Being forces, they are stored as matrices and vectors which depend on the current positions of the face mesh in order to get the proper actuation direction. Each thread does its chunk of work which corresponds to a full partition. After UPBS, Add Forces computes force contributions for each node (vertex) of the face mesh, by adding the forces related with the material properties and the ones coming from the bones (jaw, cranium). Finally, Conjugate Gradient finds the solution of the
system of equations, determining the final displacement of the nodes for the current frame. It is an iterative stage, doing a number of iterations defined by the simulation parameters. At each iteration it approximates the solution (positions of the nodes of the mesh), diminishing the residual. The residual is a value obtained during the conjugate gradient and measures how near the current iteration is from the exact solution\[^32\]. Facesim does a total of two hundred iterations for each frame. That number of iterations is not enough for Facesim to find the exact solution. It finds a \textit{good enough} solution but it never converges. Forces are only updated for the nodes belonging to the same partition, although forces from nodes of other partitions are used to calculate the force contribution of the nodes of the current partition\[^33\]. The matrices of the equation system are encoded in arrays to save memory as they are sparse. The two arrays are named \texttt{dX\_Full} and \texttt{R\_Full}\[^10\].

In case that multiple frames are specified to be simulated, for each frame there is a new set of muscle activations. The final state of the face depends only on the muscle activations and skeletal positions\[^7\]. With them the simulation algorithm derives the displacement of the positions of the face model for each frame. Through the successive frames, the positions of the face are updated with the displacements calculated at each frame, forming an animation.

### 3.4 Input sizes

Facesim can be run with different input sizes. Those sizes are classified in five different categories. All PARSEC benchmarks use the same input categories. Those categories serve as guidelines for people who wants to run the benchmarks under a processor simulator. The bigger is the input, more simulation time will need the benchmark. In the case of Facesim, all the inputs use the same exact data, that is, the same 3D model. Thus, the input size does not vary. The only parameter that varies is the number of frames to be computed. At each frame the computations are the same, but the outputs differ due to different values (muscle activations and the updated node positions)\[^33\].

- Test: 1 frame
- Simdev: 1 frame
- Simsmall: 1 frame
- Simlarge: 1 frame
- Native: 100 frames
Chapter 4

Evaluation of current Facesim implementations

4.1 Introduction

Currently Facesim has two implementations: one is serial and the other is parallel, done with Pthreads (POSIX threads). This chapter is focused on evaluating these initial Facesim code bases. This evaluation will be obtained by means of profiling and locating the parallel sections in code in order to do a first analysis of possibilities for OmpSs adaptation. Other information obtained with this process is the potential parallelism that can be obtained based on the parallel sections. This data will determinate the maximum achievable speedup.

Next sections will discuss the process and data obtained during this evaluation.

4.2 Evaluation process

The evaluation consists on doing a profile of the application and obtaining performance data. Doing a profile of an application means getting execution information, where does the application spend its time. The profiles have been obtained with GNU Prof, also called Gprof. Gprof works on data generated by the executable. That data is generated only when the program source code has been compiled with specific compile flags. In the case of GCC, the compiler needs the flag -pg, which adds both profiling and debugging information. After execution, the program leaves a binary file called gmon.out which is converted into text with Gprof. Finally, with a script called gprof2dot.py the output generated by Gprof can be plotted. The resulting figure is a call graph showing the functions called during execution, time spent in them and how many times each function has been called.

Other graphs like time breakdowns have been done with the help of Extrae, Paraver and paramedir. Traces were generated thanks to Extrae instrumentation and a new Paraver configuration file was created to generate the table with the time percentages. Afterwards, two Python scripts have been written to properly format that table to feed it to a Bash script which in turn runs a Perl script called bargraph.pl [34] that uses GNU Plot.

The performance data consists basically of the execution times with different parameters like input size and number of threads. This data allows to measure things like the speedup of the current parallel implementation. For this evaluation process the compiler version used has been GCC 4.3.

4.3 Serial code evaluation

4.3.1 Profiling

The serial code call graph for test input size is shown in figure 4.2. For native input size, the call graph is shown in figure 4.3. Both figures have been edited with Inkscape in order to show only the functions which
occupy most of the execution time, up to 95% of execution time. Depending on the input size, test or native, the functions appearing on each graph can change. That is because Facesim has some initialization code that is run before simulating the frames. As said in the previous chapter, the computations are the same in each frame. The only difference between input sizes is the number of times the computations are done, that is, the number of frames to be computed. The initialization code loads into memory the 3D model. Depending on the number of frames to be computed (determined by the selected input), the fraction of code that the initialization represents can be smaller or bigger. In the case of the native input, the serial code represents less than 1%, with the Update Position Based State representing 55% of the execution time and the Conjugate Gradient 37%. With the test, simdev, simmedium and simlarge inputs, the initialization code represents a lot more time. As per figure 4.2, the serial code occupies around 30% of the execution time. Figure 4.1a shows the time spent in the parallel sections being executed in serial along the rest of the serial code (“Other” category), measured with Extrae. In figure 4.1b the reader can appreciate how much time does represent the non parallelizable code. As the number of frames to render increases, that serial code is being outweighed.

Callgraphs and time decompositions for both test and native inputs can be seen in figures 4.1, 4.2 and 4.3. The “other” category does not include only the initialization but also serial code executed at the beginning and at the end of each frame. The reader can get an idea of what are the main time consuming functions.
Figure 4.2: Call graph of Facesim, serial code with test input (1 frame)
PhysBAM::SOLIDS_FLUIDS_DRIVER::Execute_Main_Program
99.82%
(0.00%)
1×

PhysBAM::SOLIDS_FLUIDS_DRIVER::Simulate_To_Frame
99.63%
(0.00%)
1×

PhysBAM::FACE_DRIVER::Advance_To_Target_Time
99.63%
(0.00%)
100×

PhysBAM::SOLIDS_FLUIDS_DRIVER_3D::Advance_To_Target_Time
99.61%
(0.00%)
100×

PhysBAM::SOLIDS_EVOLUTION_3D::Advance_Deformable_Bodies_To_Target_Time
99.61%
(0.00%)
100×

PhysBAM::SOLIDS_EVOLUTION_3D::Advance_Deformable_Objects_In_Time
81.23%
(0.00%)
100×

PhysBAM::DEFORMABLE_OBJECT::Update_Position_Based_State
55.14%
(0.00%)
300×

PhysBAM::DEFORMABLE_OBJECT_3D::Advance_One_Time_Step_Quasistatic
81.23%
(0.00%)
100×

PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D::Update_Position_Based_State_Parallel
55.14%
(32.41%)
300×

PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D::Add_Force_Differential
33.00%
(33.00%)
20000×

PhysBAM::COLLISION_PENALTY_FORCES::Add_Force_Differential
1.65%
(1.65%)
20000×

Figure 4.3: Call graph of Facesim, serial code with native input (100 frames)
If one looks only into the frame section of the time breakdowns, the fraction of frame time which occupies the different stages is the same for both test and native. Update_{Position\_Based\_State} fills around 60% of the frame time while Conjugate\_Gradients occupies the other 40%. Call graphs show that the different stages use “Helper” functions which are the ones actually doing the computations, those are the functions appearing in the time breakdowns.

Call graphs also give information on how does the number of computations increase. With 1 frame there is only 1 call per function up to the frame computations related functions, which start in function PhysBAM::SOLIDS\_FLUIDS\_DRIVER\_3D::Advance\_To\_Target\_Time, called from class PhysBAM::FACE\_DRIVER. That function marks the Region Of Interest. UPBS is being called three times while CG related functions are being called 200 times, equalling the number of iterations defined in the simulation parameters. The native input does 100 frames. The call graph shows how the number of calls of the test are increased 100 times. UPBS is now called 300 times and the CG functions 20000 times.

In the time decompositions appear more functions than in the call graphs. They are not included in the call graph due to space constraints. Those functions are the ones related with the Add\_Forces module: Add\_Force\_Differential and Add\_Velocity\_Independent\_Forces. Also, if one recalls what are the parts of the face model, there are mainly three: the jaw, the cranium and the flesh. Jaw and cranium are updated and treated with other functions belonging to different C++ classes which inherit from the same parent class as the ones responsible for doing the computations of the flesh. Those functions are mainly Update\_Forces\_And\_Derivatives and Add\_Velocity\_Independent\_Forces (without the Helper tail).

In the serial code the class with those functions is COLLISION\_PENALTY\_FORCES which also does the share of computations related with the jaw and cranium during the Conjugate\_Gradient. The flesh is treated in the class DIAGONALIZED\_FINITE\_VOLUME. Both CPF and DFV are inherited from the class SOLIDS\_FORCES.

4.3.2 Performance evaluation

Execution time has been obtained by doing the median of ten executions, thus avoiding outliers in measures. Standard deviation of the samples was not high, obtaining tight confidence intervals of up to 2% for native input, although with the test input it went wider, up to 10% with Pthreads and sixteen threads. The t-student distribution was used with $\alpha = 0.05$. Table 4.1 shows the execution time for test and native inputs. Initialization time ends being around 2 seconds of the execution time. These times have been obtained with Facesim’s own timing facility.

<table>
<thead>
<tr>
<th>Input</th>
<th>Whole simulation time</th>
<th>Frame computations time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>5.21</td>
<td>3.63</td>
</tr>
<tr>
<td>Native</td>
<td>369.75</td>
<td>368.18</td>
</tr>
</tbody>
</table>

Table 4.1: Facesim serial execution times for both test and native inputs

The call graphs show that the code goes through successive stages as explained previously in the chapter dedicated to Facesim description. Figure 4.4 shows the flow of stages in a frame.

4.4 Parallel code evaluation

As with the serial code in the previous sections, the parallel version will be evaluated by doing a profile of the application. Its performance will also be measured and speedups will be calculated. Call graphs for the parallel code are not available because in this case Gprof does not work well. That is because Gprof only measures user time but not system calls. Its timers only accumulate execution time while on user space. Facesim uses two system calls for managing threads: pthread\_cond\_wait and pthread\_cond\_broadcast. The time spent sleeping is not counted by Gprof, giving unrealistic time decompositions. In the serial code this does not affect measures because there is no scheduling system.

Time breakdowns are shown for eight and sixteen threads with test and native inputs (figure 4.5 and 4.6). They give a clear view of how the fraction of parallel code is reduced and how the initialization code can dominate the execution time as the number of threads increases.
Update Position Based State of the flesh
Update Position Based State of the bones

Update Collision Penalty Forces, only bones

Add Force Differential of the flesh
Add Force Differential of the bones

Add Velocity Independent Forces of the flesh
Add Velocity Independent Forces of the bones

Conjugate Gradient, with three tasks per iteration

Figure 4.4: Frame stages execution flow. At each frame all those stages are executed one after another. Most of them are parallel except the UCPF (Update Collision Penalty Forces). In the case of multiple frames, when the final AVIF of a frame has ended execution, the next frame starts with the first UPBS stage. The computations at each frame are the same. The only things that vary are the muscle activations for each frame and the positions to update the forces. The positions are accumulated through all the frames. When the conjugate gradient computations have ended, the obtained position displacement is added to the current positions and a final pass of UPBS and AVIF is done.
The Pthreads implementation does the same computations as the serial computations and also in the same order but with added barriers. After each stage there is a barrier which will not be passed by until all the threads of the previous stage end doing their computations. Figure 4.4 shows the steps. After each stage there is a barrier and the execution does not continue until all the threads end doing the computations of that stage.

4.4.1 Profiling

The parallel version adds more code, using a sort of library to schedule work, named TaskQ. The thread starting the program does the initialization and creates a pool of threads. Those threads keep waiting in an idle loop for the main thread to add new work to do. When work is available the threads get it and process it. There is only one thread creating work. That thread is the initial thread, always. The work is created by enqueuing function pointers into queues, one queue per thread. The rest of threads dequeue their assigned work from their queue and can even steal work from other threads if they find their queue empty.

Looking time decompositions of test input and comparing it against the serial code, the serial execution (“Other” category of the left most bar, black color) now occupies up to 90%. The rest of threads are waiting for the master thread to reach the parallel region. In the case of the 16 threads execution, there is no much gain as the initialization code represents too much time.

Looking at the native time decompositions, the overheads start to show. Also, they start to matter more as the number of threads increases. In the 8 threads time decomposition (figure 4.6a) those overheads are still not that visible except for the waitForTasks function. But if we look at 16 threads time decomposition in figure 4.6b the overheads of the scheduling system appears in the form of Add_Task and waitForTasks functions. The function waitForTasks is a great source of overhead, reaching 50% of the execution time. That means that during execution threads spend time waiting as much as doing computations. At the same time, the function waitForEnd executed by the master thread grows in time too. After adding tasks with Add_Task function the master thread waits in a barrier until the rest of threads end the execution of the recently created tasks. Once all threads have finished, the master thread continues program execution until the next parallel section, again creating new tasks and waiting in a barrier.

4.4.2 Performance evaluation

As with the serial performance evaluation, the execution times have been measured with Facesim’s own timing facility. Due to the parallel regions being executed with barriers between them, the measures can be taken normally. Table 4.2 shows the measured execution times.

<table>
<thead>
<tr>
<th>Frame computations time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threads</strong></td>
</tr>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td>Test</td>
</tr>
<tr>
<td>Native</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Whole simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threads</strong></td>
</tr>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td>Test</td>
</tr>
<tr>
<td>Native</td>
</tr>
</tbody>
</table>

Table 4.2: Facesim Pthreads execution times for both test and native inputs

For the native input, the execution time is greatly reduced as the number of threads increases until reaching 8 threads. On the other hand, the input test does not show such improvement as the parallel
Figure 4.5: Time decompositions for 8 and 16 threads, test input. With so many threads, the “Other” category which is exclusively serial, clearly dominates the execution time. Meanwhile, threads other than the master are WaitingForTasks.
Figure 4.6: Time decompositions for 8 and 16 threads, native input. This input simulates many more frames, so the frame computations dominate over the “Other” category. Big tasks like Update Position Based State also show some variation in duration while smaller tasks tend to have more similar duration’s among threads. Also, as the number of threads is increased some code sections start to have a higher percentage of execution time, as for example happens with the master thread. Comparing 16 threads plot with 8 threads, it spends a bigger proportion of time waiting the rest of threads. The proportion of time that task creation represents is bigger too. So, as the number of threads increases, the parallelized regions represent a smaller fraction of execution time and the overheads increase.
region represents a smaller fraction of the whole execution time, as the time breakdowns showed in the serial version.

With the measured execution times, the speedup of the Pthreads implementation can be seen in figures 4.7 and 4.8. Speed up is obtained with the following formula:

\[
\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]

\(T_{\text{serial}}\) is the execution time of the serial version. \(T_{\text{parallel}}\) is the execution time of the parallel version for a given number of used processors. Speedup indicates how many times faster does run the parallel version in respect to the serial version of an application. An augmented speedup formula is derived from Amdahl’s Law:

\[
S = \frac{1}{(1 - P) + \frac{P}{N}}
\]

\(S\) is Speedup, \(P\) is the fraction of parallel code, \(N\) is the number of processors or threads. Amdahl’s Law approximates the achievable speedup depending on the size of the parallel code section and the number of processors to be used. Doing the math, the maximum achievable speedup for the native input given the percentages obtained with the serial call graphs would be \(S = \frac{1}{(1 - 0.99)+0} = 100\). For the test input, the maximum achievable speedup results in \(S = \frac{1}{(1 - 0.70)+0} = 3.33\). Those calculations count with an infinite number of processors and with no added overheads to the parallel code. Overheads can include scheduling work in order to give each thread its share of data to work with. Synchronizing threads to avoid incorrect computations due to data races also add overheads. Synchronization to avoid deadlocks or incorrect ordered access to shared data is also another source of overheads.

Native input speedup is practically identical when measuring the whole application and the frame computations alone. Test input measures show the effects of the non parallelized code. The frame computations still scale near the same as with the native input but the speedup for the whole simulation goes only up to 2.5x. This effect is explained by means of Amdahl’s Law. For sixteen threads the achievable speedup is \(S = \frac{1}{(1 - 0.70)+\frac{0.70}{16}} = 2.90\). The added overheads to allow parallel execution of the frame computations do not allow to achieve that value. As for the native input, the maximum achievable speedup for sixteen threads is \(S = \frac{1}{(1 - 0.995)+\frac{0.995}{16}} = 14.88\). Pthreads parallelization does not perform very good in that
sense, as it reaches little over 50% of the maximum achievable speedup with sixteen threads with a value around 7.5x.

The implementation done by Facesim’s current code does not scale well above 8 threads, showing a decline in the slope of the curve when going from eight to sixteen threads. The time decompositions show the culprit. There is too much time spent in the waiting functions, which take up to 60% of the execution time of the threads other than the master. If that time was taken out of the execution time, the speedup would reach the ideal speedup. Additional views of those overheads can be seen in figures 4.9 and 4.10. Figure 4.9 shows the part of a Pthreads trace belonging to all the frame computations. The darker red corresponds to the function `waitForTasks`. At the same time, the green in the master thread (thread 1.1.1) is the function `waitForEnd`, waiting for the last thread to end the execution of tasks. In the middle of the frame there are the `Conjugate_Gradient` functions, executed during 200 iterations. The `Update_Position_Based_State` is colored in a lighter red, the big parts before and after `Conjugate_Gradient`. As shown in the time decompositions and Gprof call graphs, there are other parallel functions, mainly the ones related with `Add_Forces` kernels. During frame computations there are some parts executed in serial by the master thread. Those functions are `Update_Forces_And_Derivatives` and `Add_Velocity_Independent_Forces`. In the serial version of Facesim they do not account too much time but as the number of threads increases they appear more prominent. Previous ideal speedups did not take into account those two functions. They belong to the parallel section and they are supposed to be able to be parallelized as they belong to the parallel region of the program.

Zooming in the `Conjugate_Gradient` computations as shown in figure 4.10 the accumulation of time in `waitForTasks` can be appreciated. The master thread meanwhile is busy adding chunks of work, doing some computations and waiting for threads to end. This behaviour leads to load imbalance as some threads do not get to execute the work in their queue, so other threads compute more than one leading to longer execution times. The waiting generates problems of load balancing in a way that when threads are woken up they might be in the situation where other thread has already taken its share of work, by stealing it (function `stealTasks`, colored in darker green in the figure 4.10). With so many small tasks, the Conjugate Gradient stage adds a lot of scheduling overhead.
Figure 4.9: Pthreads, frame computations trace. Paraver traces show in the Y axis the different threads or processors. The X axis belongs to the time dimension. Thus, a trace shows the events happening in each thread as time goes by. This trace shows the different stages happening during a frame simulation. As explained in figure 4.4, computations start with UPBS function. The master thread is the one at the top. In green appears the time it spends waiting for tasks. After the first wave of UPBS tasks has ended, it executes the corresponding update of the bones which is not appreciated due to its small duration. Then it performs in dark yellow the function Update Forces and Derivatives which is followed by another wave of UPBS. Then the functions Add Forces Differential and Add Velocity Independent Forces are computed. Afterwards, the 200 iterations of conjugate gradient are performed. After doing the conjugate gradient the final computations of the frame are done: UPBS, Add Forces and the computation of the residual obtained by the iterative solver. During the frame appear some other small parallel computations which are the operations done over arrays, like clearing arrays. Also, there are small green lines which belong to the stealTasks function of the task scheduling system (TaskQ) used by PhysBAM. That function is always executed any time threads are signalled to do tasks. When they finish doing their tasks they try to steal tasks from queues of other threads, iterating over threads that are numbered after them (thread 1, 2, 3... if you are thread 2 you will try to steal from 3,4,5...). Some times may happen that when a thread tries to execute the task assigned to its queue, another thread might have stolen it and then will try to steal from the next one.
Figure 4.10: Pthreads, Conjugate_Gradient trace. This trace shows two complete iterations of the conjugate gradient. The chunks in light green belong to the first parallel loop (CG Helper I). The chunks in light red belong to the second parallel loop (CG Helper II). The chunks in blue belong to the third parallel loop (CG Helper III). The chunks in yellow belong to the Add_Task function, so the master thread is adding chunks of work. The green chunks belong to the waitForTasks function executed by the master thread, waiting for the rest of threads to end. The orange chunks belong to the stealTasks function, executed by each thread whenever they have been woken up to execute a task. The Conjugate_Gradient iterations are big source of overhead with so many small tasks, which amount a total of $(200 \text{ iterations}) \times N\text{threads}$ tasks per type of task. That is, with three parallel loops, there are three task types.
Chapter 5

Design and implementation of Facesim adaptation to OmpSs

5.1 Introduction

During this chapter this document will explain what procedures have been employed for adapting Facesim to OmpSs. The given explanations will show how does the use of OmpSs benefit data centers and organizations by removing unneeded code and making code easier to maintain, thus making them more efficient.

5.2 Facesim preparation

The first thing to do is to modify the Makefiles in order to Facesim to be able to compile OmpSs binaries. Current Facesim Makefile structure is composed of a main Makefile and different sub Makefiles for compiling related software, which mainly is the PhysBAM library and the Pthreads scheduling system. The Facesim basic source code tree structure is as follows:

- facesim
  - src
    * Benchmarks
    * Public_Library
    * TaskQ
    * ...
  - ...

There are other directories but they only contain things like inputs for the program or directories where the compiled binaries are stored. Public_Library is the folder containing PhysBAM source code. TaskQ is the directory containing the Pthreads scheduling system and Benchmarks contains Facesim source code which uses PhysBAM library.

The top-level Makefile is in src directory. There are defined the common flags to use for compiling each software component (Benchmarks/facesim, Public Library, TaskQ). The initial Facesim implementation includes flags for both serial and Pthreads, and flags are selected with a conditional construct which sets the needed compiler and its flags depending on which version has been passed as paremeter to make. In order to be able to compile an OmpSs binary, additional cases have been added in order to be able to compile by executing make version=ompss under the src directory (see listing 5.1). This Makefile totals 186 lines of code.

Listing 5.1: Top-level Makefile

1 [...]

67
The --no-copy-deps flag was added during OmpSs implementation development. By default is copy-deps which means that specified task dependencies are copied during heterogeneous execution with separate memory address spaces. --no-copy-deps was added because during execution there were too many messages informing about dependency copying, slowing down the execution of the program. Other flags which were modified and affect the resulting binary are the PORTABILITY_FLAGS. Being Mercurium a source to source compiler, it has to pass flags to the native compiler if needed with the option --Wn. Instrumentation cases and profiling cases for Pthreads and serial have also been added. Instrumentation needs additional flags which have also been added in the Makefile.

This top-level Makefile calls the one inside Benchmarks/facesim directory. This latest Makefile adds facesim source files to the variables which indicate what sources have to be compiled. Then, it includes the Makefile in Public Library, adding roughly 500 additional lines of Makefile. This one sets additional compile and link flags for compiling the PhysBAM Library. In order to be able to compile with OmpSs, some flags had to be added there too (listing 5.2). The TYPE variable in the top-level Makefile (listing 5.1) is used in the Makefile of PhysBAM for determining some flags and to whether include or not any additional libraries like the task system of Pthreads. Listing 5.2 shows the added Make directives and flags. Makefile.common additional cases for Pthreads and serial-i have their counterpart in the top-level Makefile, as explained before.

Listing 5.2: PhysBAM Makefile (Makefile.common)
PhysBAM Makefile then calls the Makefile `Makefile.PhysBAM` which adds the needed source files to the variables containing the names of files to compile. During development it was decided that was better to implement the OmpSs adaptation in different files, maintaining the exact same names for the classes. As adding an OmpSs implementation and using it is not something to be decided at runtime, and due to time constraints, it was not viable to do a proper class hierarchy. Refactoring a lot of code in order to support different implementations would take too much time. And anyways, depending on the compiler being used the added code or classes could generate compilation problems even if they were not to be used so the files should also be indicated selectively. Listing 5.3 shows the added compilation conditional in order to select the proper files for OmpSs compilation.

Listing 5.3: PhysBAM Makefile (Makefile.PhysBAM) source code definition

```
TARGETS = PhysBAM
TARGET_TYPE = STATIC_LIBRARY

ifeq (,, $(findstring ompss,$(version)))
  OMPSS_SRC = \ 
  Arrays/OMPSS_ARRAY_PARALLEL_OPERATIONS.cpp \ 
  Collisions_And_Interactions/OMPSS_BOX_HIERARCHY.cpp \ 
  Collisions_And_Interactions/COLLISION_BODY_LIST_3D.cpp \ 
  Collisions_And_Interactions/TETRAHEDRON_COLLISION_BODY.cpp \ 
  Constitutive_Models/STRAIN_MEASURE_3D.cpp \ 
  Data_Structures/SPLAY_TREE.cpp \ 
  Fracture/EMBEDDED_TETRAHEDRALIZED_VOLUME_BOUNDARY_SURFACE.cpp \ 
else
  LOCAL_SRC = \$(OMPSS_SRC)

endif

LOCAL_SRC = \$(OMPSS_SRC)
```

```
ifeq (,, $(findstring ompss,$(version)))
  LINK_FLAGS += -xC++ -lm -L$(PHYSBAM_LIBDIR) $(LDFLAGS) $(LIBS)
else
  LINK_FLAGS += -xW -lm -L$(PHYSBAM_LIBDIR) $(LDFLAGS) $(LIBS)
endif
```
In order to use Mercurium to compile Facesim some changes were made in the code as it treats C++
code in a different way than GCC. That meant to change order of headers, add type declarations where
needed and avoid circular header dependencies as a result of the reorder and new declarations. Once all
the Makefiles and sources are ready, the adaptation to OmpSs can be added. Next sections discuss the
design and implementation of the OmpSs adaptation.

5.3 OmpSs adaptation design

The evaluation of Facesim showed two main compute kernels which occupied a lot of time: Update-
Position-Based State and Conjugate Gradient. Conjugate Gradient employs three parallel func-
tions (which in Facesim have the tail Helper added to their name). Other significant parallel computa-
tions belonged to Add_Velocity_Independent_Forces and Add_Force_Differential which in conjunc-
tion form the Add Forces computations. All parallel functions work over chunks of arrays, with each
chunk corresponding to one full partition of the data and each thread processing a task which operates
on a partition. The first approach that comes to mind is to use the same strategy with the ability of
OmpSs to avoid barriers by annotating data used by the parallel computations. That should also give
an opportunity to see how well does OmpSs fare against Pthreads.

Given the iterative nature of the steps done by Facesim during the simulation, there is no opportunity
to exploit out of order execution. Each successive step depends on the previous step of one iteration.
On the other hand, additional parallelism may be exploited in some stages. That is because Facesim has
the face model split in two parts: bones (jaw, cranium) and flesh. For example, the Update_Position-
Based_State kernel operates in a generic manner using polymorphism. The function Update_Position-
Based_State does a loop iterating over each element which needs to be updated. All belong to the same
parent class, but the actual code can differ. In the case of Facesim which handles two objects, UPBS
calls the UPBS function of both objects. The flesh has that function parallelized, working on the different
tetrahedra partitions. The bones are always updated serially as they do not represent too much time
until the number of threads increases high enough. Currently Pthreads has a barrier after each parallel
section. That means the update of bones can’t be started until the update of the flesh has ended. By
removing the barrier on the flesh UPBS, the update of the bones can be computed while the flesh is also
being updated by other threads.

To give a better overview of the explanations on general code dependencies and steps, a diagram of
execution sequence can be seen in figure 5.1. The frame is processed by executing successive parallel
sections one after another, with values computed by a previous section being used by the next section.
The proposed OmpSs adaptation follows that same diagram, creating tasks for each stage, processing
data in parallel but eliminating the need for barriers the Pthreads implementation has.

5.4 OmpSs adaptation implementation

Facesim has a great number of C++ classes and does a strong usage of polymorphism and even multiple
inheritance. Current serial and parallel implementations reside in the same files and use conditional
compilation for generating the appropriate binary. The serial implementation is the same as the Pthreads
implementation but without using the scheduling system and limited to only one partition, the whole
3D model.

To avoid further complexities, the OmpSs adaptation has been added in new files. The Makefiles have
been modified accordingly as explained in the previous section, preparation of Facesim. There are
six main classes where the code to be adapted resides: DEFORMABLE_OBJECT, DEFORMABLE_OBJECT_3D,
Figure 5.1: Frame stages execution flow. At each frame all those stages are executed one after another. Most of them are parallel except the UCPF (Update Collision Penalty Forces). In the case of multiple frames, when the final AVIF of a frame has ended execution, the next frame starts with the first UPBS stage. The computations at each frame are the same. The only things that vary are the muscle activations for each frame and the positions to update the forces. The positions are accumulated through all the frames. When the conjugate gradient computations have ended, the obtained position displacement is added to the current positions.
DIAGONALIZED\_FINITE\_VOLUME\_3D, COLLISION\_PENALTY\_FORCES, FACE\_EXAMPLE and ARRAY\_PARALLEL\_OPERATIONS. The parallel kernels involved in Facesim’s simulation algorithm reside in the first four classes while the latest two (FACE\_EXAMPLE and ARRAY\_PARALLEL\_OPERATIONS) provide small functions for doing basic operations in parallel on arrays, like clearing an array or doing a dot product.

As part of the first approach, all the parallel code of these classes will be translated into OmpSs tasks. The most time consuming ones during execution are in DEFORMABLE\_OBJECT and DIAGONALIZED\_FINITE\_VOLUME\_3D (conjugate gradients and update position based state, respectively) but in order to show a more complete behaviour overview there is the need for working on the other parallel sections too. Let’s start with ARRAY\_PARALLEL\_OPERATIONS. It represents an insignificant amount of time but that class is self contained and uses the same pattern on all of its code, so its a good example of how is Pthreads code used and how to replace it with OmpSs compiler directives.

5.4.1 Array Parallel Operations class

This class consists of a function with a big switch construct which executes the appropriate operation over the array passed as paremeter depending on the operation identifier, also passed as parameter. Listing 5.4 shows an extract of how it is used. All parallel sections work in the same way, with a struct defined specifically for the parallel Helper function and a call to the ad-hoc Pthreads scheduling library, TaskQ.

Listing 5.4: Array Parallel Operations code extract

```c++
1 template< class T, class TS , class TV > void ARRAY\_PARALLEL\_OPERATIONS<T, TS , TV>::
2 Array\_Parallel\_Operations\_Helper (long thread_id, void* helper\_raw)
3 {
4   ARRAY\_PARALLEL\_OPERATIONS\_HELPER<T, TS , TV>& helper = *( ARRAY\_PARALLEL\_OPERATIONS\_HELPER<T, TS , TV>* ) helper\_raw;
5   ARRAY\_PARALLEL\_OPERATIONS\_DATA\_HELPER<T, TS , TV>& data\_helper = *helper\_data;
6   VECTOR\_2D<int>& range = helper\_range;
7   switch ( data\_helper\_operation )
8   {
9     case ARRAY\_PARALLEL\_OPERATIONS\_DATA\_HELPER<T, TS , TV>::CLEAR :
10        {
11           ARRAY<T>& array\_output = *data\_helper.array\_output;
12           for ( int i = range.x; i <= range.y; i++ ) array\_output (i) = T();
13           break;
14        }
15     case ARRAY\_PARALLEL\_OPERATIONS\_DATA\_HELPER<T, TS , TV>::COPY\_ARRAY :
16        {
17           const ARRAY<T>& array\_input\_1 = *data\_helper.array\_input\_1;
18           ARRAY<T>& array\_output = *data\_helper.array\_output;
19           for ( int i = range.x; i <= range.y; i++ ) array\_output (i) = array\_input\_1 (i);
20           break;
21        }
22        ...
23   }
24
25 /* Clear\_Parallel function caller*/
26 template< class T, class TS , class TV > void ARRAY\_PARALLEL\_OPERATIONS<T, TS , TV>::
27 Clear\_Parallel (ARRAY<T>& array\_output, const ARRAY<VECTOR\_2D<int>> & ranges)
28 {
29   ARRAY\_PARALLEL\_OPERATIONS\_DATA\_HELPER<T, TS , TV> data\_helper;
30   ARRAY<ARRAY\_PARALLEL\_OPERATIONS\_HELPER<T, TS , TV>> helpers (ranges.m);
31   #ifndef NEW\_SERIAL\_IMPLEMENTATION
32   THREAD\_POOL& pool = (*THREAD\_POOL::Singleton());
33   #endif
34   data\_helper.array\_output = &array\_output;
35   data\_helper.operation = ARRAY\_PARALLEL\_OPERATIONS\_DATA\_HELPER<T, TS , TV>::CLEAR;
36
37
38
39
40
41
```

72
The previous code contains an extract of the `Helper` function being executed by a thread and the caller function executed by the master thread. It generates tasks for each chunk of an array. It creates as many tasks as threads exist. The variable `THREADPOOL& pool` is initialized with a reference to a singleton which is created at the beginning of the program. That singleton initializes the task queuing system and serves as a C++ wrapper to it, as the task queuing system is programmed in C.

The process for creating tasks needs to get the pointer to the singleton containing the number of threads and functions to access the queues. Then, it fills a struct (`ARRAY_PARALLEL_OPERATIONS_HELPER` in this case) with the needed information: the array to operate on, the parallel operation to do and the range of positions that the task will use. There is as many structs as threads and partitions. Finally, there is a call to the `Add_Task` function of the singleton object which enqueues the function pointer with its struct of arguments, leaving it ready for execution. After doing the task creation loop, the master thread waits in a barrier until all the threads have finished doing their computations, calling the function `Wait_For_Completion` which in turn calls the corresponding function of the task system. As seen in Facesim evaluation, the master thread can also do some tasks while there is work to do.

The `ARRAY_PARALLEL_OPERATIONS` class uses the exact same pattern for each operation, amounting a great number of lines just for only creating tasks and selecting the proper case once the thread executes the `Helper` function. There is a total of 12 operations. That means 12 functions for creating tasks and 12 `switch` cases. All that code can be easily removed with OmpSs, leaving only the functions currently used for creating tasks. Some lines of code are also saved due to the removal of unnecessary helper structs along their definitions in the header file. Listing 5.5 shows the OmpSs equivalent function.

```cpp
/* Clear_Parallel function caller */
template<class T, class TS, class TV>
void ARRAY_PARALLEL_OPERATIONS<T, TS, TV>::
    Clear_Parallel ( ARRAY<T>& array_output , const ARRAY<VECTOR_2D<int>>& ranges)
{
    int START = 0; int END = 0;
    for (int i = 1; i <= ranges.m; i++)
    {
        START = ranges(i).x; END = ranges(i).y;
        T& start = array_output(START);
        #pragma omp task inout (start)\% concurrency(array_output) firstprivate(START,END) label(APO-CP)
        for (int j = START; j <= END; j++) array_output (j) = T();
    }
    #pragma omp taskwait
}
```

All the code has been replaced by a single `#pragma` and some additional variables to pass them as dependencies. The `Helper` function passed as pointer to the queue system has also disappeared as there is no queue system. It is all managed by the OmpSs runtime. The code executed in that function has been put inside the `#pragma` directive. In this case, there is a clear and sizable reduction in code complexity. Another advantage of OmpSs is that its usage also serves as serial implementation. If the code is compiled without the OmpSs compiler flag, the pragmas are ignored and the code will be executed
serially. In contrast, the Pthreads implementation needs to use the `#ifdef` compiler directive to compile
serial or parallel code depending on the macros defined by the user at compilation time.
Next section will explain the actions taken in the class `FACE EXAMPLE`.

### 5.4.2 Face Example class

This class contains a small parallel function which clears specific parts of the vector it receives as parameter. In the same way Array Parallel Operations class did, Face Example employs the TaskQ scheduling system to enqueue function pointers to a `Helper` function. Code can be seen in listing 5.6. Listing 5.7 shows the equivalent OmpSs version which again reduces lines of code and facilitates programmability and application maintenance.

```
Listing 5.6: Pthreads Face Example code extract

template<class T2> struct ZERO_OUT_ENSLAVED_POSITION_NODES_HELPER
{
    ARRAY<VECTOR_3D<T2>> * X;
    LIST_ARRAY<int> const* partition_attached_nodes;
};

static void Zero_Out_Enslaved_Position_Nodes_Helper (long thread_id, void* helper_raw)
{
    ZERO_OUT_ENSLAVED_POSITION_NODES_HELPER<T>& helper = *(ZERO_OUT_ENSLAVED_POSITION_NODES_HELPER<T>*) helper_raw;
    ARRAY<VECTOR_3D<T>> &X = *(helper.X);
    LIST_ARRAY<int> const& partition_attached_nodes = *(helper.partition_attached_nodes);

    for (int i = 1; i <= partition_attached_nodes.m; i++)
    { X(partition_attached_nodes(i)) = VECTOR_3D<T>();
    }
}

void Zero_Out_Enslaved_Position_Nodes (ARRAY<VECTOR_3D<T>> &X, const T time, const int id_number)
{
    // LOG::Time("Zero out enslaved position nodes");
    switch (id_number)
    {
    case 1:
        if (PHYSBAM_THREADED_RUN)
        {
            THREAD_POOL& pool = THREAD_POOL::Singleton();
            ARRAY<ZERO_OUT_ENSLAVED_POSITION_NODES_HELPER<T>> helpers (pool.number_of_threads);

            for (int i = 1; i <= helpers.m; i++)
            {
                helpers(i).X = &X;
                helpers(i).partition_attached_nodes = &(*attached_nodes_parallel)(i));
                pool.Add_Task (Zero_Out_Enslaved_Position_Nodes_Helper, (void*) & helpers(i));
            }
            pool.Wait_For_Completion();
        }
    ...
}
```

The Pthreads code extract continues with additional code for the serial implementation. The same code but written with OmpSs can be seen in listing 5.7.

```
Listing 5.7: OmpSs Face Example code extract

void Zero_Out_Enslaved_Position_Nodes (ARRAY<VECTOR_3D<T>> &X, const T time, const int id_number)
{
    //...
```
Additional struct types and code for serial execution is replaced by a single code block.

5.4.3 Deformable Object 3D class

This class is the one responsible of starting the actual algorithm iterations. Function `Advance_One_Time_Step_Quasistatic` calls function `One_Newton_Step_Toward_Steady_State` which does the conjugate gradients iterations. Code listing 5.8 shows the OmpSs version of `Advance_One_Time_Step_Quasistatic`. The Pthreads version is identical except for the omission of the `#pragma omp taskwait` and the need for adding code for serial execution of the operations over arrays (Array Parallel Operations).

Listing 5.8: Deformable Object 3D code extract (function One Time Step Quasistatic)
The different functions called in that loop execute the different parallel computations, including the conjugate gradient iterations. The barriers specified with the `taskwaits` are needed because after them there is serial code which can not be executed until the previous parallel work has finished.

### 5.4.4 Deformable Object class

This class contains the code needed for executing the conjugate gradient algorithm which finds the new positions of the nodes of the face mesh. The Pthreads code is really long. Only the key parts will be shown along the OmpSs counterparts. As always, Pthreads code has additional blocks for serial execution.

Before starting the conjugate gradients iterative solver, there are some calls to the other parallel kernels, as shown in listing 5.9. These calls compute force values, assembling the system of equations which is encoded in the linear vectors $\mathbf{dX}_{\text{full}}, \mathbf{R}_{\text{full}}$ to save memory, as they are sparse matrices[33].

Listing 5.9: Conjugate Gradient code extract, matrix system assembly code

```cpp
LOG::Push_Scope ("NRS", "NRS");
int i, N = particles.number;
LOG::Time ("NRS - Initialize");
dX_full.Resize_Array (N); // an initial guess might be passed in for dX, otherwise it's zero
R_full.Resize_Array (N, false, false);
LOG::Time ("NRS - Boundary conditions 1");
external_forces_and_velocities->Zero_Out_Enslaved_Position_Nodes (dX_full, time, id_number);
if (update_positions_and_state) external_forces_and_velocities->Set_External_Positions (particles.X.array, time, id_number);

LOG::Stop_Time ();
if (update_positions_and_state) Update_Position_Based_State ();
Force_Differential (dX_full, R_full);
if (!balance_external_forces_only) Add_Velocity_Independent_Forces (R_full);
LOG::Time ("NRS - Boundary conditions 2");
external_forces_and_velocities->Add_External_Forces (R_full, time, id_number);
external_forces_and_velocities->Zero_Out_Enslaved_Position_Nodes (R_full, time, id_number);
LOG::Stop_Time ();
LOG::Push_Scope ("NRS - Compute residual", "NRS - Compute residual");
double rho = 0, supnorm = 0;
#ifndef NEW_SERIAL_IMPLEMENTATION
if (PHYSBAM_THREADED_RUN)
{
    rho = ARRAY_PARALLEL_OPERATIONS<TV, T, TV>::Dot_Product_Parallel (R_full, R_full, *particles.particle_ranges);
supnorm = ARRAY_PARALLEL_OPERATIONS<TV, T, TV>::Maximum_Magnitude_Squared_Parallel (R_full, *particles.particle_ranges);
#endif
#else
[
]
#endif
```

76
The OmpSs code is identical except for the addition of appropriate barriers with \#pragma omp taskwait directives and the elimination of serial code. Code is thus reduced along its complexity.

The conjugate gradients is computed in a following loop which does a number of iterations (Facesim specifies 200 iterations). In each iteration it does a series of computations until the residual is low enough or the maximum number of iterations is reached. The conjugate gradients loop is actually split in three loops which are executed at each iteration. Each loop iterates over all the node positions of the face mesh. As each of the three loops is done in parallel, each one has an associated struct type and a proper initialization of that struct in order to pass it as parameter to the TaskQ scheduling system. As an example, listing 5.10 shows the first loop and the needed struct and thread function.

Listing 5.10: Pthreads Conjugate Gradient first loop (Helper I). The invocation of the second loop also appears, but not the corresponding Helper function.

```cpp
template <class T, class TV> struct CONJUGATE_GRADIENTS_HELPER
{
    DEFORMABLE_OBJECT<T, TV>* deformable_object;
    int partition_id;
    T time, dt;
    T alpha, beta;
    ARRAY<TV>* dX_full;
    ARRAY<double>* S_dot_Q_partial, *rho_new_partial, *supnorm_partial;
};

template <class T, class TV> void DEFORMABLE_OBJECT<T, TV>::
One_Newton_Step_Toward_Steady_State_CG_Helper_I (long thread_id, void* helper_raw)
{
    CONJUGATE_GRADIENTS_HELPER<T, TV> const & helper = * (CONJUGATE_GRADIENTS_HELPER<T, TV>*) helper_raw;
    DEFORMABLE_OBJECT<T, TV>& deformable_object = *helper.deformable_object;
    int partition_id = helper.partition_id;
    VECTOR_2D<int> particle_range = (*deformable_object.particles.particle_ranges)(partition_id);
    T beta = helper.beta;
    // ARRAY<TV>& negative_Q_full = deformable_object.F_full;
    ARRAY<TV>& S_full = deformable_object.S_full;
    ARRAY_RANGE<ARRAY<TV>> S(deformable_object.S_full, particle_range);
    ARRAY_RANGE<ARRAY<TV>> R(deformable_object.R_full, particle_range);
    // double& S_dot_Q = (*helper.S_dot_Q_partial)(partition_id);
    for (int i = 1; i <= S.m; i++) S(i) = beta * S(i) + R(i);
}

if (PHYSBAM_THREADED_RUN)
{
    typedef CONJUGATE_GRADIENTS_HELPER<T, TV> T_CG_HELPER;
    ARRAY<double> S_dot_Q_partial (particles.particle_ranges->m);
    THREAD_POOL& pool = *THREAD_POOL::Singleton();
    ARRAY<T_CG_HELPER> helpers (particles.particle_ranges->m);
    for (int p = 1; p <= particles.particle_ranges->m; p++)
    {
        helpers (p).deformable_object = this;
        helpers (p).partition_id = p;
        helpers (p).time = time;
        helpers (p).beta = beta;
        helpers (p).S_dot_Q_partial = &S_dot_Q_partial;
        pool.Add_Task (One_Newton_Step_Toward_Steady_State_CG_Helper_I, &helpers (p));
    }
    pool.Wait_For_Completion();

    for (int p = 1; p <= particles.particle_ranges->m; p++)
    {
        pool.Add_Task (One_Newton_Step_Toward_Steady_State_CG_Helper_II, &helpers (p));
    }
    pool.Wait_For_Completion();

    S_dot_Q = ARRAY<double>::sum (S_dot_Q_partial);
}
Facesim follows the same pattern as with the rest of the code. Enqueues the function pointer to the
thread function in the TaskQ scheduling system and the threads will take those functions as tasks once
they are signaled by the master thread when it enters the Wait_For_Completion function.
In this case there is a lot of overhead for just working over an array in parallel. In the OmpSs version,
this loop has been merged with part of the second loop. Listing 5.11 shows both the first and second
loop of the OmpSs version.

Listing 5.11: OmpSs Conjugate Gradient first and second loops (Helper I and II)

```c
#pragma omp task inout(S_full) shared(F_full, particles) firstprivate(beta) label (CG-HELPER-I)
{
    for (int p = 1; p <= particles.particle_ranges->m; p++)
    {
        #pragma omp task shared(F_full, particles) firstprivate(p, beta) label(CG-HELPER-I-nested)
        {
            int START = (*particles.particle_ranges)(p).x; int END = (*particles.particle_ranges)(p).y;
            for (int i = START; i <= END; i++) S_full(i) = beta * S_full(i) + R_full(i);
            Force_Differential_Internal(S_full, F_full, p);
        }
    }
    #pragma omp taskwait
}
```

```c
#pragma omp task concurrent(S_dot_Q) in(S_full) shared(F_full, particles)
firstprivate(p) label(CG-HELPER-II)
{
    double local_S_dot_Q = 0.;
    ARRAY_RANGE<ARRAY<TV>> S(S_full, (*particles.particle_ranges)(p));
    Force_Differential_Internal(S_full, F_full, p);
    Force_Differential_External(S_full, F_full, p);
    external_forces_and_velocities->Zero_Out_Enslaved_Position_Nodes(F_full, time, id_number, p);
    local_S_dot_Q = ARRAY<TV>::template Vector_Dot_Product<double>(S, F_full.Range((*particles.particle_ranges)(p)));
    #pragma omp atomic
    S_dot_Q += local_S_dot_Q;
}
```

The first thing to note is the great reduction in lines of code. No needed structs, no needed extra
functions. All code can be inlined within the same function and without the need for additional cases
for serial execution. That improves greatly the code maintainability. Talking about the partial merge
of the second loop with the first one, the difference with Pthreads is that in the first loop there is now a
function called Force_Differential_Internal and in the second loop the original Force_Differential
has been substituted by a function called Force_Differential_External. The splitting was necessary
because the first loop had a very short duration and the OmpSs runtime (nanox) did not have enough
time to create tasks. Afterwards, the nested creation of tasks was also added to allow more than one
thread to create tasks. While one thread is creating tasks of the first loop, the other thread generates
tasks of the rest of the other two loops.

Another advantage to see in the OmpSs code is that dependencies allow for the elimination of waits
between loops. The concurrent and in/out, inout clauses allow the tasks to be chained and be created
way before they are going to be executed. Special mention to the concurrent clause which currently is
only available on OmpSs but not on OpenMP 4.0, the standard implementation used in compilers like
GCC. Concurrent allows to specify a dependence on which tasks have to operate simultaneously and
if previous or subsequent tasks need it as input or output they will have their dependencies respected.
Concurrent can be seen as an inout clause but without the concurrent tasks being serialized because of
that dependency.
The original Force Differential operated over all the nodes, computing force contributions to each node. It also accessed positions from partitions other than the own one. Using polymorphism, the Force Differential function is computed in the classes Collision Penalty Forces and Diagonalized Finite Volume 3D. The Force Differential in Collision Penalty Forces only accesses positions of its own partition. The Force Differential in Diagonalized Finite Volume has three loops. The first two compute force contributions by accessing only nodes in the same partition and the third one loop computes node contributions by accessing other partitions. That can be clearly seen in code listing 5.12. The access to nodes from other partitions initially did not allow the merge with the first loop which operates on the same arrays and that is why the split was made before doing the partial merge.

Listing 5.12 shows the three loops. The edges variables contain an array of pairs of node indexes. The internal edges belong to nodes from the same partition while the external edges belong to nodes linked across partitions. The new functions Add Force Internal and Add Force External used in the OmpSs version contain the corresponding loops. The Internal function has the first two loops while the External function computes the third loop. Further assurance of this splitting was made by means of checking the actual indices returned by the Get functions. At each iteration the m,n indices used for accessing and updating the arrays with the forces and positions where checked against the ranges of all the partitions. The partition numbers which they belonged to were added to an std::map, which was printed afterwards to check what partitions did each task access. That effectively showed that the first two loops only access their own partition while the third one also accesses other partitions.

With the conjugate gradients loop merge, the runtime had enough time to generate tasks and overlap task creation with task computation. Note that if something similar had to be done with Pthreads, the code would need more changes as it uses additional functions.
Until now, there has been a more or less detailed description of the maintenance done to Facesim. Next sections will not show more code as it takes too much space and they are not as complex as the ones detailed up until now. The functions will be described textually.

5.4.5 Collision Penalty Forces class

This class updates the state and computes the forces of the rigid bodies; that is, the bones in the case of Facesim. It is used during Update Position Based State and Add Forces computations. Additionally, it has a function called Update Forces and Derivatives in charge of updating the forces directions based on the based position state computed previously. All the code in this class is executed serially and has to respect the order in the original Pthreads implementation. Consequently, functions Add Force Differential and Add Velocity Independent Forces have been declared as tasks with their inputs and outputs being the arrays of Forces. Those functions are not parallel loops, the task is the whole function as they suppose little computations. With those data annotations the tasks are executed after the flesh nodes attached to the bones have been updated with its new force contributions. Looking at the code before doing the iterations of the conjugate gradient, there reside calls to Force Differential and Add Velocity Independent Forces, which iterate over the deformable and rigid bodies. With the dependencies specified in the corresponding functions of Collision Penalty Forces, the forces of each node mesh are correctly updated without both the bones and flesh incurring in data races over the forces of each node. That allows to eliminate barriers and continue generating tasks as dependencies ensure correct data access.

5.4.6 Diagonalized Finite Volume 3D class

The computations related with the flesh are executed in this class. The partitioning of the mesh and data replication is also done here, based on the information read from the input data (number of partitions, node divisions, 3d model). Update Position Based State and Add Forces computations are parallelized. As with the rest of Facesim, each function creates the proper structs and adds the functions into the TaskQ scheduling system. The parallelized code is a parallel loop, with each iteration computing a partition of the data. Each iteration is then made a task. This class is very large and accounts a total of 1228 lines of code. Apart from the Force Differential function explained in previous sections, the conversion to OmpSs involved the removal of all the code related to Pthreads and the serial implementation. It was full of conditional compilation directives.

5.4.7 Other work and additional notes

This chapter has explained the work done on the adaptation of the current code base to OmpSs. Some additional modifications on top of the current classes have been made. Pthreads code used a singleton class, named THREAD_POOL. For the OmpSs adaptation an equivalent singleton class has been implemented in order to store the number of partitions (and thus chunks as the code uses one chunk per partition). Additionally, with the OmpSs code the number of partitions used is independent of the number of threads. In the case of the Pthreads code that is not true. The number of threads is equal to the number of partitions. If eight threads are to be used, then the application will use a model with eight partitions. The OmpSs adaptation can have a number of threads different of the number of partitions to use. That adds the possibility of having better load balance by changing task granularity when changing the number of partitions to use.

PhysBAM also has a lock class named THREAD_LOCK. It wraps lock implementations like pthreads_mutex_lock. The OmpSs adaptation of Facesim has added the needed code in order to also have the OmpSs locking functions also wrapped in the case anyone wants to use locks.

One more thing which needs to be commented is related with data dependence definition. Looking at the OmpSs code and data annotations, one can appreciate the fact that only the variable name is used in OmpSs adaptation. OmpSs annotations allow for indicating also the size of the data to be accessed, allowing to define an array section. In heterogeneous systems with separate address spaces this would allow to have all the data automatically copied without any problems. During this project the developer tried to specify data dependencies with their size but the dependencies finally have been annotated with
only a reference to the variable without specifying any size. Facesim uses templates, using parametric
types on the whole code. The compiler used by OmpSs, Mercurium, does not instantiate types. It can
not generate the code in order to know the size of the data because the parametric type is not known at
compilation time.

In order to be able to use sizes in data annotations with C++ parametric templates the programmer has
to use some tricks. Take as an example the code from the class Array Parallel Operations in listing 5.13.

Listing 5.13: OmpSs Array Parallel Operations without size specification

```cpp
/* Clear_Parallel function caller */
template<class T, class TS, class TV> void ARRAY_PARALLEL_OPERATIONS<T, TS, TV>::
Clear_Parallel (ARRAY<T>& array_output, const ARRAY<VECTOR_2D<int>>& ranges)
{
    int START = 0; int END = 0;
    for (int i = 1; i <= ranges.m; i++)
    {
        START = ranges(i).x; END = ranges(i).y;
        #pragma omp task inout (start)
        concurrent(array_output) firstprivate (START, END) label (APO_CP)
        for (int j = START; j <= END; j++) array_output (j) = T();
    }
    #pragma omp taskwait
}
```

The data being accessed is the variable `start` pointing to the START position of the corresponding
partition of the array `output`. There is also a `concurrent` annotation which serves to have other previous
or following tasks respect the order on the access to `array_output`. Neither the `inout` nor `concurrent`
clauses specify any kind of size. But the task is accessing an array section which goes from START to
END, that is, a section of size `END - START + 1`. Without pointing out the size, if that task was to be
executed on an accelerator, the only data which would be transferred to the accelerator would be the
initial position of the array. To specify the data section used by the task the compiler directive should read
`#pragma omp task inout ([END-START+1]array_output) ...`. The problem is that `array_output` is
of type `T`, so the `sizeof(T)` is not known to Mercurium compiler because it does not instantiate the
C++ template parameters. The programmer has to trick the compiler in order it sees a known type
which indicates the size of it. C++ has an operator to do that: `reinterpret_cast<type>(variable)`. The previous code in listing 5.13 could be rewritten like in listing 5.14.

Listing 5.14: OmpSs Array Parallel Operations with size specification

```cpp
/* Clear_Parallel function caller */
template<class T, class TS, class TV> void ARRAY_PARALLEL_OPERATIONS<T, TS, TV>::
Clear_Parallel (ARRAY<T>& array_output, const ARRAY<VECTOR_2D<int>>& ranges)
{
    int START = 0; int END = 0; int SIZE = 0; VECTOR_2D<int> range;
    for (int i = 1; i <= ranges.m; i++)
    {
        START = ranges(i).x; END = ranges(i).y;
        SIZE = END - START + 1;
        typedef T (&ArraySectionT)[SIZE];
        ArraySectionT bp_output = reinterpret_cast<ArraySectionT> (array_output.
        base_pointer[START]);
        #pragma omp task inout (bp_output)
        shared (array_output) firstprivate (START, END) label (APO_CP)
        for (int j = START; j <= END; j++) array_output (j) = T();
    }
    #pragma omp taskwait
}
```

`array_output` is an object of type `ARRAY` which wraps a dynamically allocated pointer. `ARRAY` class
behaves as a variable length array. The base pointer is a public member. It is used in order to gain
access to the initial address of the section wanted to be expressed.
The trick consists in doing a `typedef` of the template parameter `T` to a reference type of size `SIZE`. Then, the `reinterpret_cast<type>(variable)` operator is used to convert the base address of `array_output` into the newly defined type `ArraySectionT`. When the compiler applies the function `sizeof` to that dependence it can obtain the size of that type, `SIZE`. In that way, the programmer can avoid the strictness of the C++ type system in order to be able to define contiguous sections of memory in a compiler without template instantiation. The `typedef` should have to be applied to every template parametric type, so it would be tedious to do that in every task and for every type, which in the previous case can be up to three types per task (`T`, `TS` and `TV`).

Next subsection gets into final thoughts of the work done during the OmpSs adaptation.

### 5.4.8 Final thoughts on OmpSs implementation

The addition of an OmpSs implementation to Facesim was not very straightforward. The code base of Facesim is huge at around 35,000 lines of code. Additionally, each modified class has a high degree of usage of other classes and the class hierarchy and its relationships are very intrincated. Nevertheless, OmpSs allowed to save a lot of code in the parallel sections and the resulting classes look much cleaner than the original Pthreads plus serial code. That is a good point for using OmpSs in any organization which needs a great runtime in their data center stack. Easy of maintenance is a valuable parameter to measure the suitability of a determined software component for anyone wanting to maximize productivity with minimal cost. The capability for creating networked applications like web servers or CGIs with OmpSs is there. Cloud or distributed services should benefit from current trends which pursue good utilization of the available resources. COMPSs works in a similar way to OmpSs and can have the same impact as OmpSs on the user side of the cloud. Think for example about Amazon Web Services. Users can develop COMPSs applications and port them to other cloud platforms, eliminating possible barriers imposed by AWS to migrate applications from their cloud to another one.

Next section talks about performance evaluation of the OmpSs implementation added to Facesim, further justifying the efficiency of OmpSs and its possibility of being useful for data centers.

### 5.5 OmpSs performance evaluation

Apart from making the creation and maintenance of applications easier, another important aspect of an application is its efficiency. This section will show that the OmpSs code not only needs less lines of code and effort than Pthreads to get an implementation running. OmpSs code can also run with better efficiency, diminishing running costs of the data center of an organization thanks to spending less energy on it and less maintenance time on applications. Speedups and runtime behaviour will serve the purpose of efficiency evaluation. The total size of the code will also be shown in order to expose the reductions in code size that OmpSs can achieve. This evaluation has been made with the stable branch of OmpSs, which currently is being updated in a monthly basis. The native compiler used has been GCC 4.3.

The final speedup numbers obtained with Facesim show better efficiency than Pthreads. Figure 5.2 charts the speedup comparison between Pthreads and OmpSs. The simulation does not scale so much with the test input because the initialization consumes a great part of the execution time. For native input, achieves a speedup of 9 with 16 threads, which is better than Pthreads which reaches 7.5 with 16 threads. The speedups of the frame part show a similar picture of scalability, except for the case of the test input in which it is nearly equal to the native input. They have not been included due to considering them redundant.

Those improvements come mainly from a better scheduling done for Update Position Based State function and a good enough load balancing during the conjugate gradient iterations, although it is still a bit slower than the Pthreads version. That is because although there is more than one thread creating tasks, the tasks are still too much fine grained to be able to compensate the task creation time. Figure 5.3 shows that UPBS tasks start at the same time and all threads get to execute one of those chunks. Also, the overlap of the Update Position Based State (UPBS) of the bones with the UPBS of the flesh allows to compensate for the different durations the later have. That overlap is the explanation to why the first thread starts its flesh UPBS later.

82
(a) Pthreads vs OmpSs, native input, simulation speedup  
(b) Pthreads vs OmpSs, test input, simulation speedup

Figure 5.2: Comparison of OmpSs and Pthreads speedups for the whole execution. OmpSs adaptation performs better than Pthreads, beginning with 8 threads.

Figure 5.3: Ompss, frame computations trace. This trace is the equivalent to the Pthreads trace in figure 4.9. First appear the UPBS tasks in yellow. One thread starts its chunk of work later than the rest. That is because it is executing the UPBS code of the bones and afterwards the corresponding force directions via the Update Forces and Derivatives function. After doing the two UPBS, Add Force Differential and Add Velocity Independent Forces come in red and green colors. These functions form the Add Forces computation kernels. The trace continues with the 200 iterations of the conjugate gradient, with each iteration having three parallel loops (that is why appear various colors). The frame computations end with another pass of UPBS, Add Forces and the computation of the residual of the equation system solver.
Figure 5.4: OmpSs Conjugate Gradient. In these figures appear two and half iterations of the conjugate gradient. An iteration of CG consists of three loops each one being executed in parallel, one by one. These traces show the refactored code in action. The first loop has its tasks with a long enough duration to allow the OmpSs runtime to create tasks for the other two loops, allowing to have the tasks ready for execution when threads exit the first loop. That provides good load balance and allows all threads to get a chunk of work. The refactored code allows two threads to create tasks. While one thread creates tasks for the first loop and also executes a task of it, another thread spends the time creating the tasks for the other two loops. If the nesting was not made, one thread would have to create all the tasks, losing also the opportunity to execute work from the second loop.

In figure 5.4 two iterations of the conjugate gradient can be appreciated, both the task view and the thread state view. The first loop now is the one that has the longer duration due to the made code refactoring. If it was not for that refactor, the tasks would not be executed in all cores, losing efficiency. That loss is because the creator thread can not generate tasks at a high enough pace in order to maintain the queue of tasks filled. In that case, some threads starve while the others always get the few tasks available, figure 5.5 shows that behaviour. To avoid that situation, the refactoring was made, giving the master thread enough time to create tasks for all threads. Later, the creation of tasks was improved by doing the nesting in the first conjugate gradient loop, splitting the amount of time needed for task creation, achieving better efficiency with the execution of the second loop of the conjugate gradient. Figure 5.4b plots how two threads are creating tasks (the blue color). While one thread is creating the tasks of the first loop, another thread is creating tasks for both the second and third loops of the conjugate gradient.

If the task creation presented less time or was eliminated completely, Facesim would achieve a lot greater efficiency. Nonetheless, OmpSs still performs better thanks to better scheduling when new tasks have to be started and they are ready for execution. That allows both UPBS and Conjugate Gradient second and third loops to start earlier or have less delays.

5.6 Validation

Facesim outputs a number of files which are a binary representation of the final positions of the 3D model. Additionally, it prints in a log file the residual of the equation system solver.

Facesim uses double precision to do its computations. Small changes in computed values are expected due to the use of atomic operations which do not guarantee the same order when updating a value. Code refactoring also has its effects. Even different compiler versions can show different results. Those changes do affect the precision and thus the final positions of the nodes of the face model can differ due to round off error. As a result the residual of the iterative solver can be different. Pthreads code also has differences between the different number of partitions. The serial version also has differences with any Pthreads partitioning. The OmpSs code is in the same degree of variability for each partitioning.

By default Facesim does not come with any kind of visualization tool. The 3D model can not be seen
Figure 5.5: OmpSs, conjugate gradient naive adaptation behaviour. This figure shows one iteration of the Conjugate Gradient. Each iteration is composed of three parallel loops. The naive approach follows the Pthreads implementation. The first, second and third loops are executed consecutively, each one in parallel. Due to the small duration of the first loop, OmpSs runtime does not have enough time to create the tasks for the other two loops. Given this behaviour, something had to be improved and as a result the code refactoring was made. Putting part of the second loop into the first one gave enough time for the runtime to create tasks.

and the output does not write any standard files to get the data into a standard visualizer. To do a qualitative analysis, the output routines have been modified in order to write the final mesh into a .obj file. That file contains the vertices and triangles (faces) of the boundary mesh, that is, the tetrahedron mesh whose tetrahedron have triangles on the skin. Those generated files where loaded into a basic visualization tool, g3dviewer.

The code added to Facesim in order to output the .obj file is in class FACE EXAMPLE. First, the boundary mesh had to be initialized. By default Facesim has only the full tetrahedron mesh constructed but without any triangle mesh, faces or boundary mesh initialized. Dumping all the tetrahedron mesh into an .obj would result in a bigger file and depending on the computer to visualize the 3D model, it could also cause performance problems. The code fragment for initializing the boundary mesh is in listing 5.15.

Listing 5.15: Boundary mesh initialization. After the tetrahedron volume has been loaded from disk, the initialization of the boundary mesh is done

```cpp
input = FILE_UTILITIES::Safe_Open_InputStream (input_file);
std::cout << "Reading simulation model : " << input_file << std::endl;
tetrahedralized_volume.template Read<RW> (*input);
delete input;
std::cout << "Total particles = " << particles.number << std::endl;
std::cout << "Total tetrahedra = " << tetrahedron_mesh.tetrahedrons.m << std::endl;
tetrahedron_mesh.Initialize_Boundary_Mesh();
particles.Store_Velocity (false);
particles.Store_Mass (false);
tetrahedralized_volume.Set_Density (1000);
tetrahedralized_volume.Set_Mass_Of_Particles (solids_parameters.use_constant_mass);
tetrahedralized_volume.Update_Bounding_Box();
```

That code is inside the function FACE EXAMPLE::Get Initial Data(). This was possible thanks to the code being already available, so the initialization just needed to call the corresponding function (tetrahedron_mesh.Initialize_Boundary_Mesh()). Other structures can not be initialized with the currently available code. Looking at the function Initialize Boundary Mesh one could already suspect that it had some order for the vertices, but they could be clock-wise or counter clock-wise. It resulted Facesim meshes are already OpenGL compatible in the sense that the order of the vertices are given counter clock-wise.

With the boundary mesh already initialized, only the code for creating the .obj file was left. That code has been put into FACE EXAMPLE::Write Output Files(const int frame). That function is called at

85
the end of the simulation. Depending on a flag, it can also output files at the end of each frame. The way
the benchmark is currently implemented it only outputs data at the end of simulating all the specified
frames. It serves for verification purposes. The code which writes the .obj file is in listing 5.16

Listing 5.16: Code which writes the .obj file

```cpp
// Write triangles of the face
const DEFORMABLE_OBJECT_3D<T>& deformable_object = solids_parameters.
    deformable_body_parameters.list (1);
const ARRAY<VECTOR_3D<T>>& vertices = deformable_object.particles.X.array;
const TETRAHEDRON_MESH& tetrahedron_mesh = deformable_object.tetrahedralized_volume->
tetrahedron_mesh;
/* All triangles have the vertices in the same order, they already go counter clock-wise,
as OpenGL needs for the normals*/
std::ostream* output = std::ofstream(output_directory + "final_boundary_mesh_" + ostrframe.str() + " .obj");
for (int i = 1; i <= vertices.m; i++) { (*output) << "v " << vertices(i) << std::endl;
    (*output) << "usemtl (null)" << std::endl;
    for (int j = 1; j <= 3; j++) { (*output) << tetrahedron_mesh.boundary_mesh->triangles(j, i) << " ";
        }(*output) << std::endl;
    } delete output;
```

The .obj format specifies that the vertices must be printed. Each vertex line is identified by the v letter.
After the vertices have been printed, the next thing to print is the list of faces (triangles). Each face line
is identified by the f line. Each face is composed of 3 or 4 vertex indices. But that was not enough. In
order to be able to see the model illuminated when trying to view it in a 3D model viewer, it also needs
a material to be specified. OpenGL accepts null materials, which means default values color and light
properties, depending on the viewer. The material has to be specified before a group of faces, that is
why before printing the faces there is the line (*output) << "usemtl (null)" << std::endl;.
Screenshots of the boundary mesh show that there are virtually no differences between the Pthreads face
model and the OmpSs face model. Figure 5.6 shows the final boundary mesh of the native input (100
frames) for both Pthreads and OmpSs.
Figure 5.6: Final mesh state of the face model, native input (100th frame). After simulating one hundred frames, the final mesh of OmpSs does not show any appreciable difference in respect to the Pthreads mesh.
Chapter 6

OmpSs Programmability

As explained on previous chapters, programmability is improved by using OmpSs. Less code complexity can be achieved, allowing to focus more on the algorithmic side than on how to actually write the needed code. Compiler directive annotations save the need for creating additional libraries, classes or struct types specific to an application. Genericity of programming and easyness of data center exploitation is a direct benefit of using OmpSs, allowing to do changes without the need to modify too much related code. Figure 6.1 shows a comparison of number of lines between Pthreads and OmpSs. There is a significant reduction of around 1500 lines of code which is an important fraction of the parallel code. During development another thing to note is the possibility of using the instrumentation plugin of OmpSs, allowing to take measurements and trace events in order to assess what is happening in the program. It can serve to track down bugs, faulty behaviour due to incorrect order execution or other incorrect behaviour derived from concurrency or parallelism. With instrumentation, OmpSs software can generate trace files which can be viewed with Paraver. And those files are generated automatically by compiling the source code with the flag \texttt{--instrument} of the OmpSs compiler, Mercurium. With different configuration files already available the same trace can offer different views of the execution events. With other programming APIs like OpenMP or Pthreads such possibility is not available. The programmer must add manually the needed code in order to generate execution events.

All these measurements validate the fact that OmpSs is a good choice for any organization wanting to deploy new applications in its datacenter or wants to install a new data center. OmpSs allows development and deployment of threaded applications, including networked applications like webservers. Network communications can be taskified the same way I/O can be or parallel computations do without the burden of using low level libraries like POSIX threads thanks to the automatic task creation and no need for manual thread management. COMPSs on top of cloud middlewares like Amazon EC2, follows the same approach as OmpSs allowing the creation of remote methods and web services, spawning them as tasks, exposing an application as a web service.

As a result of the research, maintainability of the top level of the software stack of an organization data
center has been tested to show it can be less problematic thanks to OmpSs and related technologies.
Chapter 7

Conclusions

The work done during this project brings some outstanding contributions. It has shown that the maintenance of applications with OmpSs requires less code and that OmpSs can achieve better performance. Looking at figures 7.1 and 7.2 one can appreciate the reduction in code size. The adaptation of OmpSs has a non negligible impact on the total code base. Focusing on the adapted section converted from Pthreads to OmpSs, the impact is bigger, as that is the part where OmpSs has been applied.

If one looks at figure 7.3, the efficiency obtained with OmpSs equals and surpasses the one obtained with Pthreads.

Additionally, OmpSs offers tools to visually assess the behaviour of the application, which serves as a way to find bugs or problems in application execution. This leads us to the conclusion that OmpSs can be a programming environment valid for developing software for the new trend of devices and services, as it allows developers to be more productive and achieve better results.

Finally, this project has also explored the deployment of a cluster infrastructure in order to complete the proposal of a hardware and software stack for a company. The Extreme Cloud Administration Toolkit has been investigated and it has been found to be a powerful tool for anyone wanting to manage a cluster with lots of nodes like MareNostrum.

![LOC (All source files)](image)

Figure 7.1: Lines of code of the whole program
Figure 7.2: Lines of code of the parallel section

Figure 7.3: Pthreads vs OmpSs, native input, simulation speedup. OmpSs adaptation shows better performance for eight threads and above
Given the previous results, this project has achieved the proposed objectives:

- Study the deployment of MareNostrum infrastructure
- Know what software is employed in a cluster and how is it used
- Design and implement an OmpSs adaptation for an application which needs a cluster or data center for execution (Facesim)
- Comparison with current code base of the application, including both ease of programming and performance differences between programming models being used.
- Show whether OmpSs programming model can be a valid environment for developing software needed for the new trend of devices and services
List of terms

API
Specifies a software component in terms of its operations, their inputs and outputs and underlying types. Its main purpose is to define a set of functionalities that are independent of their respective implementation, allowing both definition and implementation to vary without compromising each other. 95

BSC
BSC, acronym for Barcelona Supercomputing Center, is a research and development center in different areas of high performance computing. 25

compiler directive
It’s a programming language construct that tells the compiler to process code specified by the construct, the directive, in a special way. 95

CPU
It’s the computer hardware that executes the instructions a programmer has coded in a computer program. A computer can has more than one CPU. 95

CUDA
Is a parallel computing platform and programming model created by Nvidia and implemented by the GPUs they produce. 95

GNU
GNU, acronym for GNU is Not Unix, is a Unix-like computer operating system developed by the GNU Project. It is composed wholly of free software. It is based on the GNU Hurd kernel and is intended to be a “complete Unix-compatible software system.”. 18

GNU/Linux
GNU operating system whose kernel is based on Linux. 16–19, 26

GPU
Is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose Central Processing Unit (CPU)s for algorithms where processing of large blocks of data is done in parallel. 1, 2

kernel
Part of an operating system in charge of hardware management and exposing it to the user applications which need to use it. 95
OmpSs
It’s a task based programming model which allows to parallelize applications by adding some compiler directives. It’s implemented as a user-space runtime which computer programs can link to. Allows for parallel programming on multiple types of systems and can use other programming models when being used on heterogeneous system in order to take advantage of hardware that needs to be used with other programming models like Compute Unified Device Architecture (CUDA).
15, 18, 23, 29

POSIX
Defines a series of functionalities that a Unix-like operating system must satisfy. 95

Pthreads
Is a Portable Operating System Interface (POSIX) standard for threads. The standard, POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an Application Programming Interface (API) for creating and manipulating threads. 18, 23

runtime
Library designed to change program state dynamically during program execution. For example, it can dynamically schedule some tasks depending on the program’s needs, like freeing non used memory or changing program or hardware behaviour to better exploit computer hardware. 26, 46, 95

smartglass
Device that is both glasses and a ultra portable computer with an operating system that allows to install any software available for that operating system. Usually less capable than a smartphone, tends to be connected to a smartphone for offloading some functionality that can’t be processed in the glasses. Allows for visualizing information instantly. 1, 2

smartphone
Device that is both a phone and an ultra portable computer with an operating system that allows to install any software available for that operating system. 1, 2

smartwatch
Device that is both a watch and an ultra portable computer with an operating system that allows to install any software available for that operating system. Usually less capable than a smartphone, tends to be connected to a smartphone for offloading some functionality that can’t be processed in the watch. 1, 2

thread
The smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. 95
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Chain of project’s task dependencies</td>
<td>15</td>
</tr>
<tr>
<td>1.2</td>
<td>Gantt chart</td>
<td>21</td>
</tr>
<tr>
<td>1.3</td>
<td>Gantt chart</td>
<td>24</td>
</tr>
<tr>
<td>2.1</td>
<td>MareNostrum site physical distribution</td>
<td>34</td>
</tr>
<tr>
<td>2.2</td>
<td>MareNostrum System</td>
<td>36</td>
</tr>
<tr>
<td>2.3</td>
<td>MareNostrum batch scheduler overview</td>
<td>39</td>
</tr>
<tr>
<td>2.4</td>
<td>xCAT architecture</td>
<td>41</td>
</tr>
<tr>
<td>2.5</td>
<td>xCAT boot process of a statelite node</td>
<td>44</td>
</tr>
<tr>
<td>2.6</td>
<td>OmpSs code transformation diagram</td>
<td>47</td>
</tr>
<tr>
<td>3.1</td>
<td>3D face model</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>Representation of a tetrahedron and stored muscle vector directions</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>Face muscles</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Facesim serial implementation time breakdowns</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Call graph of Facesim, serial code with test input (1 frame)</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Call graph of Facesim, serial code with native input (100 frames)</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Facesim’s frame stages execution flow</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Time decompositions for 8 and 16 threads, test input</td>
<td>60</td>
</tr>
<tr>
<td>4.6</td>
<td>Time decompositions for 8 and 16 threads, native input</td>
<td>61</td>
</tr>
<tr>
<td>4.7</td>
<td>Pthreads implementation speedups for the frame code section</td>
<td>62</td>
</tr>
<tr>
<td>4.8</td>
<td>Pthreads implementation speedups for the whole execution</td>
<td>63</td>
</tr>
<tr>
<td>4.9</td>
<td>Pthreads, frame computations trace</td>
<td>64</td>
</tr>
<tr>
<td>4.10</td>
<td>Pthreads, Conjugate Gradient trace</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Facesim’s frame stages execution flow</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of OmpSs and Pthreads speedups for the whole execution</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>OmpSs, frame computations trace</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>OmpSs Conjugate Gradient</td>
<td>84</td>
</tr>
<tr>
<td>5.5</td>
<td>OmpSs, conjugate gradient naive adaptation behaviour</td>
<td>85</td>
</tr>
<tr>
<td>5.6</td>
<td>Final mesh state of the face model, native input (100th frame)</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of the number of lines of code between Pthreads and OmpSs</td>
<td>89</td>
</tr>
<tr>
<td>7.1</td>
<td>Lines of code of the whole program</td>
<td>91</td>
</tr>
<tr>
<td>7.2</td>
<td>Lines of code of the parallel section</td>
<td>92</td>
</tr>
<tr>
<td>7.3</td>
<td>Pthreads vs OmpSs, native input, simulation speedup. OmpSs adaptation shows better performance for eight threads and above</td>
<td>92</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Physics engines properties comparison ........................................ 9
1.2 Human costs ............................................................................. 25
1.3 Hardware costs .......................................................................... 26
1.4 Software costs ........................................................................... 26
1.5 Other indirect costs .................................................................... 27
1.6 Per task costs (1) ....................................................................... 28
1.7 Per task costs (2) ....................................................................... 29
1.8 Final budget ............................................................................... 30
1.9 Budget considering also risks ...................................................... 30
1.10 Table for controlling expenditures .............................................. 31
1.11 Sustainability matrix .................................................................. 31
2.1 xCAT tables specific to a cluster configuration ............................ 42
4.1 Facesim serial execution times for both test and native inputs ....... 57
4.2 Facesim Pthreads execution times for both test and native inputs ... 59
## Listings

2.1 MareNostrum’s `lshosts` command output extract ............................... 34
2.2 MareNostrum’s `mount` command output extract ................................. 36
2.3 MareNostrum’s `/etc/fstab` file .................................................. 37
2.4 MareNostrum’s `dtcp` command script ............................................ 37
2.5 Job Script, submitter ........................................................................ 39
2.6 MareNostrum’s persistent directories ................................................ 43
2.7 MareNostrum’s boot command line .................................................... 43
2.8 OmpSs Two tasks example ................................................................. 46
5.1 Facesim’s Top-level Makefile ............................................................. 67
5.2 Facesim’s PhysBAM Makefile (Makefile.common) ................................. 68
5.3 Facesim’s PhysBAM Makefile (Makefile.PhysBAM) .............................. 69
5.4 Array Parallel Operations code extract .............................................. 72
5.5 OmpSs Array Parallel Operations code extract .................................... 73
5.6 Pthreads Face Example code extract .................................................. 74
5.7 OmpSs Face Example code extract .................................................... 74
5.8 Deformable Object 3D code extract ................................................... 75
5.9 Conjugate Gradient code extract ....................................................... 76
5.10 Pthreads Conjugate Gradient First loop ............................................ 77
5.11 OmpSs Conjugate Gradient first and second loops .............................. 78
5.12 Add Force Differential function of conjugate gradient iterations ............ 79
5.13 OmpSs Array Parallel Operations without size specification ................ 81
5.14 OmpSs Array Parallel Operations with size specification ..................... 81
5.15 Facesim’s 3D model boundary mesh initialization ............................... 85
5.16 Code which writes the `.obj` file ...................................................... 86
Bibliography


[16] Evan Drumwright, John Hsu, Nathan Koenig, and Dylan Shell. Extending open dynamics engine


[26] Instituto Nacional de Estadística. Sueldos brutos por ocupación. media anual nacional. grupos a, c, d y e (directores y gerentes, otros técnicos y profesionales científicos e intelectuales, técnicos y profesionales de apoyo, empleados de oficina que no atienden al público). http://www.ine.es.


