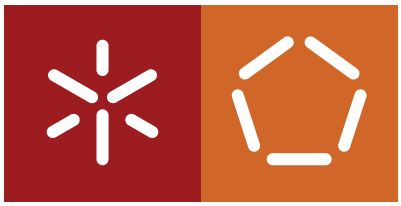
**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

José Ricardo Cunha da Silva Ribeiro

# Improving the performance of liquid surfaces modelling in multicore devices

October 2015



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

José Ricardo Cunha da Silva Ribeiro

# Improving the performance of liquid surfaces modelling in multicore devices

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
**Alberto José Proença**
**José Luís Alves**

October 2015

# AGRADECIMENTOS

Sendo a escrita de uma dissertação, não só a representação do trabalho do último ano letivo, mas sim o culminar de 5 anos de muito trabalho e dedicação no percurso académico de Engenharia Informática em uma das mais prestigiadas academias do país, não posso deixar de incluir os meus sinceros agradecimentos a várias pessoas que contribuíram para o meu sucesso.

Ao meu orientador, Professor Alberto Proença, por todo o acompanhamento não só neste ultimo ano na escrita desta dissertação mas também por todos os anos de aulas exemplares, que contribuíram não só para o meu crescimento académico e profissional mas também sem duvida como pessoa.

Do mesmo modo, agradeço imenso ao Professor Luís Alves, mais uma vez não só pelo acompanhamento nesta dissertação, mas também pela anterior colaboração, naquele que foi o meu 1º projeto na área de Computação Paralela e Distribuída (CPD).

Aos Professores João Sobral, António Pina e Rui Ralha que juntamente com o Professor Proença, foram as pessoas que mais contribuíram para a minha especialização em CPD e sem as quais não teria os conhecimentos que tenho hoje. Agradeço ainda ao Professor José Nuno Oliveira, pelo acompanhamento durante o meu percurso de mestrado e licenciatura, não diretamente relacionado com esta dissertação mas por todo o apoio durante o mestrado.

Um agradecimento muito especial à Bosch e à Universidade do Minho, nomeadamente ao pessoal do projeto HMIExcel (nº 36265/2013 (Projeto HMIExcel - 2013-2015)), pela oportunidade que me deram de aprender e trabalhar durante 1 ano neste projeto com o apoio sob a forma de uma bolsa de investigação.

Não posso ainda deixar de mencionar alguns dos meus colegas que sempre me apoiaram e me ajudaram durante todo o percurso académico. Começando pelo Bruno Araújo, com o qual partilhei a maioria dos meus trabalhos de mestrado, incluindo em parte esta dissertação e ainda todo os meus colegas de CPD pelos momentos de partilha de conhecimento nesta área. Aos meus colegas e amigos Amadeu Andrade, Armando Freitas, Luís Vieira, Marcos Magalhães e Nuno Araújo, entre outros, por toda a ajuda, cumplicidade e amizade ao longo de 5 anos que certamente não ficam por aqui.

*Last but not least*, à minha família e amigos, pelos sacrifícios e paciência e que tiveram em momentos em que estive indisponível devido a este e outros trabalhos durante o meu percurso académico. Sem duvida que o apoio destas pessoas foi fundamental para o meu sucesso e deixo aqui um muito obrigado e a promessa de um dia retribuir tudo aquilo que fizeram por mim.

## ABSTRACT

When assembling bottom terminated components in printed circuit boards, connectivity is extended through metallized terminals. To minimize thermal fatigue failure of the welds, software tools have been developed to model liquid surfaces shaped by various forces and constraints. Surface Evolver (SE) is the software tool used by Bosch in their media entertainment products to model liquid surfaces through the analysis of discrete parts of that surface. However, depending on the level of detail, this process may have long execution times, which is not consistent with the demand of industry and mainly in an interactive software where users expect the results to be obtained quickly.

This dissertation aims to improve the efficiency of SE, through the optimization of the total energy computation, taking advantage of vectorization, parallel computing and other high performance techniques.

The analysis and profile of the current SE version were crucial to support the decisions taken to improve the computational performance of the software. Scalability tests, taking into account the Amdahl's law, call graphs and other profiling analysis helped to identify bottlenecks, where an effort should be invested to improve the software. One of the heaviest computations identified in SE is the computation of the total energy of the configuration.

SE was identified to be a memory-bounded software, mainly due to its current mesh data structure, implemented with linked lists, which limits the use of the vectorization features on current CPU cores and also does not support data parallelization techniques and data locality. A new data structure was proposed to overcome these performance constraints, which led to a faster execution of SE.

The results showed an improvement on the total energy computation, an increase of vectorizable operations, software prefetching techniques and scheduling optimizations which, alongside the alternative data structure, increased the performance of the SE.

# RESUMO

Na montagem de *bottom terminated components* em *printed circuit boards*, a conetividade é expandida através de terminais metalizados. De modo a minimizar as falhas por fadiga térmica das soldaduras, ferramentas de *software* foram desenvolvidas para modelar superfícies liquidas condicionadas por várias forças e restrições. O Surface Evolver (SE) é a ferramenta de *software* utilizada pela Bosch nos seus produtos de *media entertainment* para modelar superfícies liquidas através da análise de partes discretas destas superfícies. No entanto, dependendo do nível de detalhe, este processo poderá ter longos tempos de execução, o que não é condizente com a exigência da industria e principalmente num *software* interativo onde o utilizador espera que os seus resultados sejam obtidos rapidamente.

Esta dissertação tem como objetivo melhorar a eficiência do SE, através da otimização do cálculo da energia total, tirando partido de vetorização, computação paralela e outras técnicas de computação de alta performance.

A análise e o profiling da implementação atual do SE foram cruciais para suportar as decisões tomadas para melhorar a eficiência computacional do *software*. Testes de escalabilidade, tendo em consideração a lei de Amdahl, *call graphs* e outras análises de profiling ajudaram a identificar *bottlenecks*, onde um esforço deverá ser investido de modo a melhorar a performance do *software*. Um dos cálculos mais pesados identificado no SE é o cálculo da energia total da configuração.

O SE foi identificado como sendo *memory-bounded*, principalmente devido à sua estrutura de dados atual, implementada com listas ligadas, o que limita o uso de funcionalidades de vetorização nos cores de CPU atuais e também não suporta técnicas de paralelização e localidade de dados. Uma nova estrutura de dados foi proposta de forma a ultrapassar estas limitações, o que resultou numa execução mais rápida do SE.

Os resultados mostraram uma melhoria no cálculo da energia total, um aumento de operações vetorizáveis, técnicas de *software prefetching* e otimizações de *scheduling* que, juntamente com a estrutura de dados alternativa, aumentaram a performance do SE.

# CONTENTS

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

**AVX** Advanced Vector Extensions. 29, 55
**BTC** Bottom Terminated Component. 12–14, 16, 17, 78
**CAD** Computer Aided Design. 19
**CAE** Computer Aided Engineering. 12, 17
**CPU** Central Processing Unit. 13, 14, 24, 27, 29–31, 33, 35, 39, 40, 42–44, 55, 76, 78, 79
**CUDA** Compute Unified Device Architecture. 27, 31–33, 35
**DPMO** Defects per Million Opportunities. 17
**DRAM** Dynamic Random Access Memory. 32, 33
**FEM** Finite Element Method. 12, 14, 16, 18–20, 25, 26, 52, 78
**FPGA** Field-Programmable Gate Array. 13
**GCC** GNU Compiler Collection. 33
**GDDR** Graphic Double Data Rate. 32, 33
**GPGPU** General Purpose Graphics Processing Unit. 31
**GPU** Graphics Processing Unit. 13, 14, 27, 30–33, 35, 51, 80
**HPC** High Performance Computing. 27, 31
**ICC** Intel C++ Compiler. 30, 33
**MIC** Many Integrated Core. 13, 33
**NUMA** Non-Uniform Memory Access. 27, 28, 35, 46, 47, 78, 85, 86
**PCB** Printed Circuit Board. 12–14, 16, 17, 25, 78
**QFP** Quad Flat Package. 17
**SE** Surface Evolver. iv, v, c, 12–14, 16, 20–22, 24–26, b, 27, b, 30, 33, 35–37, 39, 41, 42, 47–58, 61, 63–70, 73, 75, 78–80, 86
**SIMD** Single Instruction, Multiple Data. 29, 51, 55, 78, 79
**SM** Streaming Multiprocessor. 32
**SMP** Symmetric Multiprocessor. 59
**SMT** Simultaneous Multithreading. 29
**SMX** Extended Streaming Multiprocessor. 32, 33
**SOIC** Small Outline Integrated Circuit. 17
**SP** Streaming Processor. 32
**SPMD** Single Program, Multiple Data. 33
**SSE** Streaming SIMD Extensions. 29, 55

# 1

# INTRODUCTION

The modelling of liquid surfaces shaped by various forces and constraints, is a process used in the study of these surfaces, including simulation and optimization procedures. Several universities and companies in the field of mechanical engineering, electronic engineering, mathematics, among others use this process to study the behaviour of these surfaces when constrained by different forces and energies. This dissertation aims to study the development of a modelling process to simulate the evolution of a surface in order to identify problems of thermal fatigue failure in the design of new Printed Circuit Boards (PCBs), as part of a collaboration with Bosch - Car Multimedia, world leader in media entertainment.

To optimize the procedure and parameters to braze components of the Bottom Terminated Component (BTC) type on PCBs, a case study was chosen to simulate a thermal fatigue failure of the welds, which leads to a premature end of an electronic component [1]. With this simulation, the evolution of the surface can be monitored without the need of field experiences or even discovering problems after large productions, which represent high manufacturing costs for companies.

To generate information about the procedure, this simulation process needs a solution to achieve these optimizations, using Computer Aided Engineering (CAE) computational tools to support the design of new PCBs. The CAE computation tool used by Bosch for this process is the Surface Evolver (SE), an interactive software application to model liquid surfaces using the Finite Element Method (FEM). The FEM is used to model a problem involving continuous surfaces through the analysis of discrete parts of that surface, for which it is possible to know or obtain a mathematical description of their behaviour. However, some surfaces might be discretized with more detail, thus making a larger mesh, or when the mesh evolves to a higher refinement state, it produces more elements and increases the computing requirements. These longer execution times are not consistent with the demand of the industry, particularly in interactive software, where the user expects the results to be obtained quickly and see the evolution of the surface in real-time.

This dissertation addresses this response time problem, aiming to implement a new and more efficient solution using parallel computing techniques: the SE execution times will be reduced,

leading to a more interactive interface with the user and enabling the study of surfaces in greater detail. First of all, the problem was studied and analysed as well as the computing process of the software. The execution of the SE code was profiled to identify potential bottlenecks, the performance of the critical regions in the sequential version were optimized and the adequacy of the data structures required a detailed analysis, before proceeding to parallel implementations. This analysis showed that a significant part of the software is spent to compute the total energy of the configuration. So, the main focus of this work is to implement a new version of this computation, proposing a more efficient data structure, exploring vectorizing features of the available processing units and implementing a parallel version in a shared memory paradigm with OpenMP.

Heterogeneous computing environments, which typically consists of one or more multicore computing devices and an accelerator, such as a Graphics Processing Unit (GPU), will also be taking into account. To achieve a higher performance, the implementation must consider that each processor and accelerator can differ with different architectures, programming paradigms and memory hierarchies. Algorithms may require changes to execute in these environments and deal with load balancing among the available processing units.

An heterogeneous environment might have different architectures, from Central Processing Units (CPUs) to accelerators such as the GPU, the Intel Many Integrated Core (MIC) devices or even Digital Signal Processors (DSPs) and Field-Programmable Gate Arrays (FPGAs). In this dissertation, the parallel implementation executes in an homogeneous shared memory environment: however, all decisions regarding performance improvements, such as an alternative data structure, are made taking into account a future implementation running in heterogeneous environments.

## 1.1 MOTIVATION & GOALS

The key objective of this dissertation is to improve the efficiency of the modelling of liquid surfaces with the SE software. As mentioned, this dissertation uses as a case study, the optimization of the procedure and parameters for brazing components of the BTC type on PCBs, simulating a thermal fatigue failure of the welds, which leads to a premature end of an electronic component.

Both components of profiling and the implementation of a new and more efficient solution are the main outcomes of this dissertation, which will be the result of the following tasks:

- To study the modelling of liquid surfaces with the SE software, including a profile of the code with different sizes of input data sets and resolutions;

- To design and implement a more efficient data structure to deal with vectorization, parallel computing and memory locality;
- To improve the sequential version of the SE software (vectorization, memory accesses among other improvements);
- To implement an efficient parallel version of the application for shared memory environments, with multicore CPUs;
- To study how these improvements can be used for a future heterogeneous implementation, with a GPU or an Intel Xeon Phi as accelerators.

## 1.2 CONTRIBUTION

This dissertation contributes by implementing a new, more efficient version of the SE software used to model liquid surfaces, currently used in related research projects and companies around the world. It also contributes with relevant information about the implementation process, all the studies and decisions made while building a new version of SE that might become useful in the future for other attempts to implement high performance computing software.

## 1.3 DISSERTATION OUTLINE

This dissertation has 6 chapters. Chapter 2 introduces the modelling of liquid surfaces as well as the particular problem of thermal fatigue failure of BTC components welds in PCBs. This chapter also presents the FEM and the software SE used to model these surfaces, as well as the case studies used with SE.

Chapter 3 the required computing background (homogeneous and heterogeneous environments), as well as profilers, compilers, framework and libraries used to improve the performance of the SE. In chapter 4 the SE is presented with greater detail, profiling and analysing the critical regions, performance bottlenecks and the current data structure.

In chapter 5 presents an evolution on the improvements of the SE performance. It starts to describe the total energy computation of the SE (the part of the software identified as critical bottlenecks), an alternative data structure proposal and other techniques to improve the performance, such as vectorization and a shared memory implementation to compute the total energy in parallel.

Finally, chapter 6 presents the conclusions and gives suggestions for future work. This dissertation includes at the end an appendix A, which contains relevant information about the method-

ology and computer nodes characteristics, as well as the Amdahl's law to better understand the limitations to these improvements.

Introductory text explains the structure at the beginning of each chapter, showing key information. Similarly, at the end of each chapter, a summary resumes the chapter and stresses the relevant details.

# 2

# MODELLING OF LIQUID SURFACES

*This chapter introduces the liquid surfaces modelling shaped by various forces and constraints and the main problem studied, the thermal fatigue failure of BTC components welded to PCBs, that leads to a more premature end of an electronic component. The simulation of these procedures is done by a numerical analysis with the FEM that evolves the surface from its initial state to the state where it meets the required stop criteria. This is done by the SE, an interactive software to study these surfaces, and is presented with an example and two main case studies used throughout this analysis.*

The modelling of liquid surfaces shaped by various forces and constraints, is a process used to study these surfaces, including simulation and optimization procedures. Several universities and companies in the field of mechanical engineering, electronic engineering, mathematics, among others use this process to study the behaviour of these surfaces when constrained by different forces and energies.

In this dissertation, it is mostly studied the development of a modelling process to simulate the evolution of a surface in order to identify problems of thermal fatigue failure in the design of new PCBs.

## 2.1 DYSFUNCTIONS/ELECTRICAL ANOMALIES OF A PCB

The thermal fatigue failure of BTC components welds in PCBs yields a more premature end of an electronic component, therefore it is a concern and a process that requires some optimization and maturation. Optimizing the welding process, it is possible to minimize the electrical failure problems due to thermal fatigue of the welded components and thus increasing the life cycle of the BTC components [1].

During its use, these BTC components and the respective PCBs go through successive thermal cycles, which originate anomalies by gradients of thermal expansion coefficients of the various materials present: polymers, composite materials, copper PCBs, encapsulating material of the BTC components, brazing materials, etc. The thermal fatigue, when accumulated, leads

to an interruption of the electrical conductivity, so the brazing is one of the major causes of problems in this system.

It is therefore important to work on solutions to these problems. The optimization of the BTC components placement on the PCB and the brazing procedures and parameters can lead to a substantial increase in the robustness of electronic systems. The analysis of the welding material volume and the design of the PCB copper area to be welded to the component, require careful studies to lower the error rate in production line and extend the life of the weld joint, to assist in the overall quality of the brazing process, since the residual stresses resulting from thermal cycling depend on both the configuration of the PCB, and the amount of solder.

For common components, other than BTCs, the lifetime of the PCB has a fall-off-rate of 2 Defects per Million Opportunities (DPMO) in the production line and an estimated lifetime in reliability tests of more than 15 years, more than 30 years when using nominal values of solder paste volume. The use of components other than the BTC, Quad Flat Package (QFP) or Small Outline Integrated Circuit (SOIC) types are much easier to produce and it can easily reach 30 to 40 years of useful lifetime, since due to the geometry of its terminals, they are easier to produce with these quality levels. These traditional components are more flexible and thus they adapt better to the resulting deformations from the differences in the thermal expansion coefficients of the materials. However, the new BTC components, because of their rigidity, are more susceptible to problems associated with thermal fatigue failure, thereby making it more difficult and demanding to maintain a fall-off-rate of 2 DPMO and 15 years of estimated lifetime. This means that with the introduction of the new BTC components it is necessary to introduce new strategies to oppose the new implications for durability, namely:

- Optimizing the design of the PCB layouts in order to manage the local rigidity, thus minimizing the effects of the thermal fatigue of the solders in the BTC components;
- Optimizing the procedures and parameters of brazing BTC components on PCBs, to scale the size of the welding blisters and minimizing the stress fields associated with the deformation;

To achieve these optimizations, it is essential to generate knowledge and CAE computational tools are required to support the new PCB designing, integrating the BTC components, and generating additional value by incorporating new technologies in smaller (miniaturization) and robuster integrated circuits manufacturing. Moreover, experimental tests to validate a conceptual solution are long, about 4 months, which a CAE computational tool can help to reduce.

## 2.2 FINITE ELEMENT METHOD

The FEM is a numerical method used to model a problem involving continuous surfaces through the analysis of discrete parts of that surface, for which it is possible to know or obtain a mathematical description of their behaviour - discretization. Each discreet element - the finite element - and the mathematical descriptions of its behaviour contributes to the analysis of the global problem. This change of scale from the individual finite element analysis to the analysis of a global problem is called assembling. As a result, a problem with a high complexity or without an analytic solution can be solved by smaller and simpler problems with a mathematical solution (exact or approximated), that when assembled leads to a solution of the global problem [2].

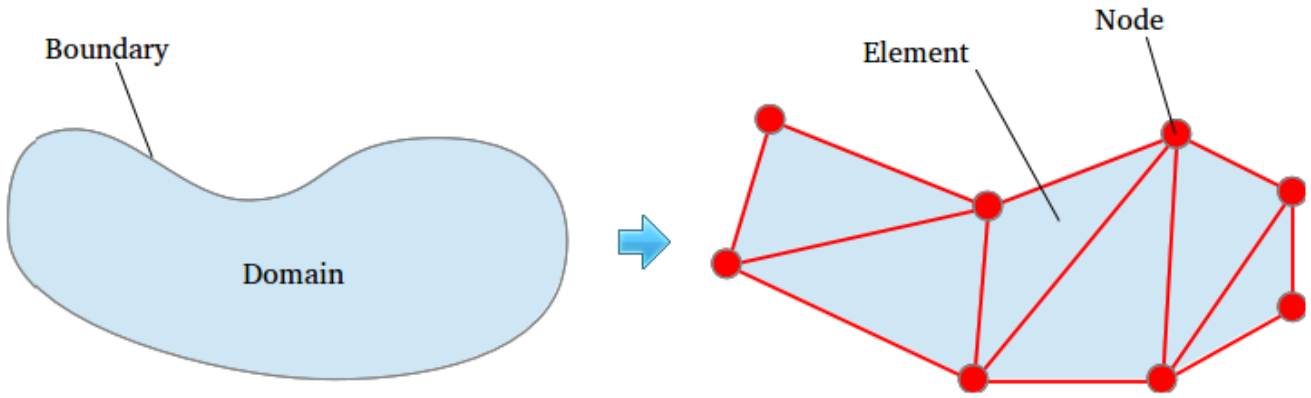



Figure 1.: Spacial discretization of a domain by finite elements [1]

In a numerical simulation process, it must be taken into account that numerical methods such as the FEM, are approximated methods. It is extremely important to identify all the potential sources of error as well as estimate the magnitude of those errors and provide the best model representation in the discretization process. The FEM allows a consideration of a great behaviours diversity and constitutional models such as linear elasticity (Hooke's law), plasticity, viscoplasticity, hyperelasticity, termoelasticity, etc.

The finite elements can assume various geometric forms such as one-dimensional, two-dimensional or three-dimensional. One-dimensional problems are solved by using finite elements in the shape of line segments, two-dimensional problems usually recurs to quadrilaterals

1 http://imagine.inrialpes.fr/people/Francois.Faure/htmlCourses/FiniteElements.html

or triangles and three-dimensional problems usually uses hexahedrons, tetrahedron or pentahedron, among others.

So, for instance in a linear elasticity analysis applied to engineering problems, one of the first analysis to consider is the displacement fields of a finite number of points in the system. These points - the nodes of a mesh - are usually placed in the element vertices, however, depending on the type of formulation they can be placed in the middle of the edges, facets or even inside the elements. So, the numerical analysis with the FEM calculates, in a first stage, the displacements on the nodes for a particular filling applied to the domain being analysed. Thus, it is possible to replace the problem of determining the displacement of an infinite number of points in a continuous domain with the calculations of the displacements in a finite number of points - the mesh nodes.

The FEM, in a generic way, goes through three different stages: the pre-processing stage, the analysis stage and finally the post-processing stage.

### 2.2.1 *Pre-processing*

The pre-processing stage concerns to the geometric modelling of the system being studied and the definition of all the constraints, attributes and physical/mechanical properties to be considered. This stage is usually done with the help of modelling Computer Aided Design (CAD) software that assists the user to create an accurate representation of the continuous problem. The global quality of the FEM analysis depends largely on the quality of these representations and the way the user decides on some simplifications or the element and mesh types.

### 2.2.2 *Analysis*

The analysis stage is the FEM phase where all the calculations are performed. Initially, all the information created by the pre-processing stage is read and analysed, followed by the iterative process of elements calculations and linear systems solving.

### 2.2.3 *Post-processing*

The post-processing stage analyse the output of the previous stage and presents the evolution and/or the final result of the mesh. These results can be presented to the user in friendlier formats to facilitate the analysis of the result, in a color shaped distribution of isovalues or isolines for instance.

## 2.3 THE SURFACE EVOLVER

The Surface Evolver, developed by Ken Brakke from the University of Susquehanna (USA) [3], is an interactive software for the study of surfaces shaped by surface tension and other energies, and subject to various constraints[1]. This software implements a surface as a simplicial complex with a union of triangles made of the basic elements: vertices, edges, and facets. The initial surface can be defined in a datafile that can be loaded to the program. The surface is evolved toward minimal energy by a gradient descent method to find a minimal energy surface, or to model the process of evolution by mean curvature and then it outputs the new surface. SE is an interactive software and the evolving process can be viewed as it evolves towards the final state.

The evolving of the surface is done in three main steps:

- First, the discretization process, when the user writes a model that involves a continuous surface through the analysis of discrete parts of that surface. The initial surface, is defined as a simplicial complex with a union of triangles made of the basic elements: vertices, edges, and faces in a datafile that can be loaded to the SE. The surface is represented by a FEM, where each element is a simplex. In this datafile, it is also defined the element attributes and the surface tension, energies as well as other constraints and boundaries.

- After the model and all the attributes are defined, then the surface can be evolved. This can be done by either defining the iterative process in the datafile, therefore making a script to this process, or interactively typing commands directly to the software. In both cases, the evolution of the surface can be seen graphically.

- Following the iterative process, SE outputs the attributes of the evolved surface and also exports the new surface to several formats.

### 2.3.1 *Surface Evolver example*

The SE has a command line interface and allows the user to start the software with an input datafile defining the surface and other data such as constraints, boundaries and forces [3].

As a basic example of how the SE works, a simple cube can be used to demonstrate all the process. The initial surface is a unit cube, with one body constrained to have a volume of 1. No gravity or other forces are assumed besides the surface tension leading the cube to evolve to a sphere, the minimal energy surface.

1 http://www.susqu.edu/brakke/evolver/evolver.html

```
// cube.fe
// Evolver data for cube of prescribed volume.

vertices /* given by coordinates */
1 0.0 0.0 0.0
2 1.0 0.0 0.0
3 1.0 1.0 0.0
4 0.0 1.0 0.0
5 0.0 0.0 1.0
6 1.0 0.0 1.0
7 1.0 1.0 1.0
8 0.0 1.0 1.0

edges /* given by endpoints */
1 1 2
2 2 3
3 3 4
4 4 1
5 5 6
6 6 7
7 7 8
8 8 5
9 1 5
10 2 6
11 3 7
12 4 8

faces /* given by oriented edge loop */
1 1 10 -5 -9
2 2 11 -6 -10
3 3 12 -7 -11
4 4 9 -8 -12
5 5 6 7 8
6 -4 -3 -2 -1

bodies /* one body, defined by its oriented faces */
1 1 2 3 4 5 6 volume 1
```

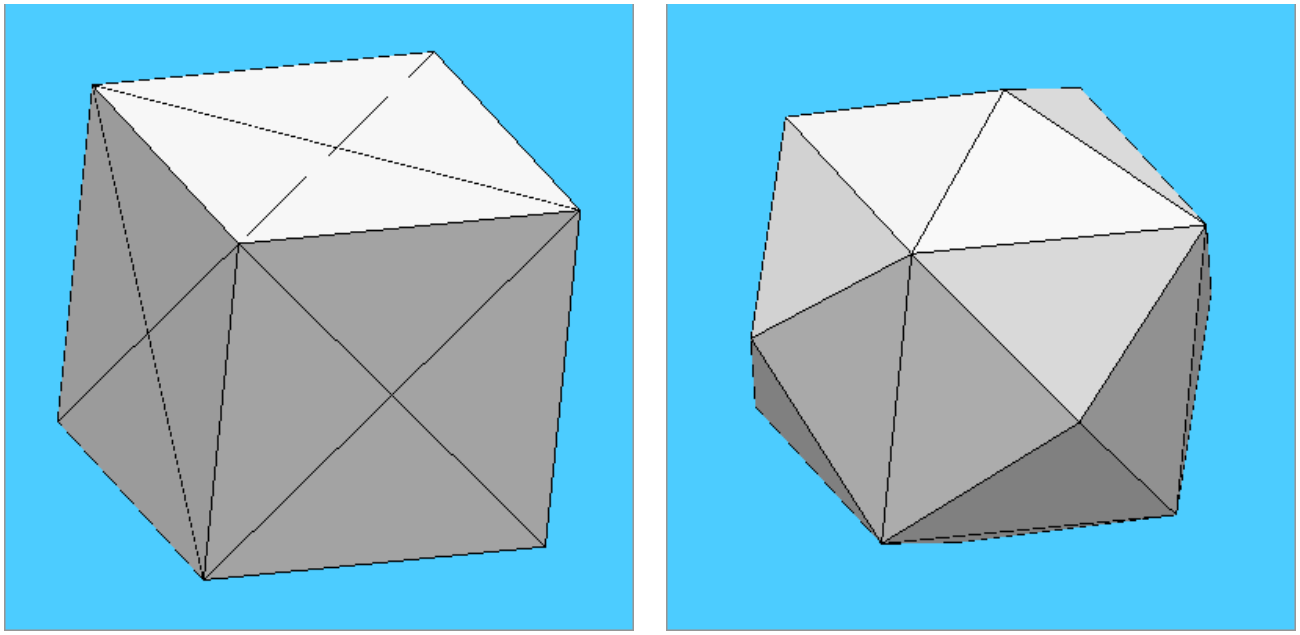
Listing 1: Input datafile to define a surface shaped as a cube in the SE [3]

In the input datafile shown in the listing 1 it is defined the surface. It starts by defining the vertices, one per line, starting by a positive integer, defining the vertex number and then its coordinates. Next, are defined the edges also one per line. It starts also with a positive integer, the identifier and then 2 identifiers, the endpoints, which are the numbers of the tail and head vertices of the edge. The faces are defined next, also starting by the face number and then followed by a list of oriented edge numbers in counter-clockwise order around the face. A negative edge means the opposite direction from the one that was defined on the list. Finally, the bodies, in

this case just 1, are represented by an identifier followed by a list of faces which composite the body. In every definitions (vertices, edges, faces and bodies) are more parameters that can be defined such as the volume, density, constraints, etc. In this case a volume is being defined as a constraint, with the value of 1.

Opening the evolver executable, it is possible to pass the input datafile as a parameter:

```
evolver cube.fe
```



(a) Initial surface (b) Surface after 5 iterations

Figure 2.: Initial state of the cube and the cube after 5 iterations

At this point, the user has access to the command line and can interactively define and evolve the surface, as well as using the GUI to see the surface represented graphically using the command *s*. The appearance of the cube at the start is shown in figure 2a. The faces are converted to facets, since the software only uses triangles. To do a number of iterations it is used the command *g n*, where *n* is the number of consecutive iterations. After the command *g 5*, that is, 5 iterations, the surfaces evolves to the one shown in figure 2b and the output is shown in the listings 2.

The output shows one iteration per line with informations about the area, energy and current scale factor respectively. By default, the SE seeks the optimal scale factor to minimize energy. The current volume can be obtained using the command *v* as shown in the listings 3. It also shows the pressure as well as the volume defined and the actual volume.

```
5. area: 5.11442065156005 energy: 5.11442065156005 scale: 0.186828
4. area: 5.11237323810972 energy: 5.11237323810972 scale: 0.21885
3. area: 5.11249312304592 energy: 5.11249312304592 scale: 0.204012
2. area: 5.11249312772740 energy: 5.11249312772740 scale: 0.20398
1. area: 5.11249312772740 energy: 5.11249312772740 scale: 0.554771
```

Listing 2: Output of running 5 iterations over the surface with the command *g 5*

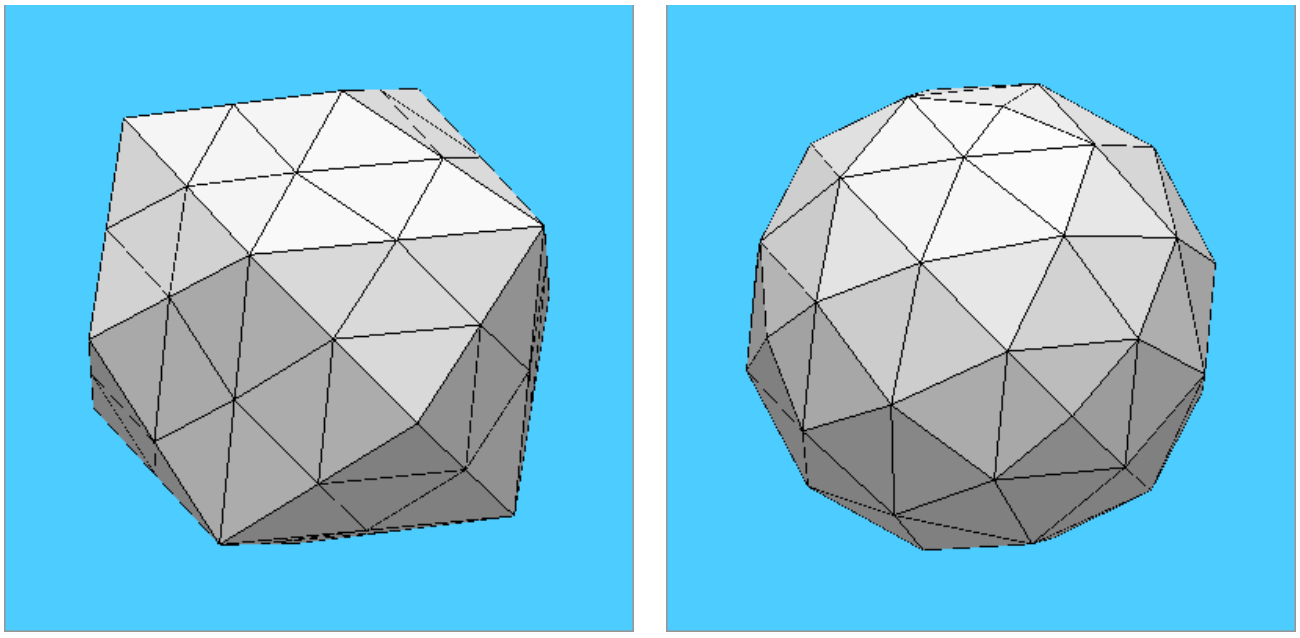
```
Body  target                volume                actual volume pressure
1     1.000000000000000     0.999999779366360     3.408026016427987
```

Listing 3: Volume of the surface obtained by the command *v*

One might want to increase the level of detail by refining the triangulation. This is done with the *r* command. This subdivides each facet into four smaller similar facets as shown in the figure 3a and prints the following message:

```
Vertices: 50 Edges: 144 Facets: 96 Facetedges: 288 Memory: 27554
```

This shows the number of geometric elements and the memory they are currently occupying.



(a) Surface after refinement

(b) Final surface

Figure 3.: Cube after being refined and the final cube after 20 iterations

Finally, the figure 3b shows the surface after running 20 iterations which outputs the information shown in the listings 4.

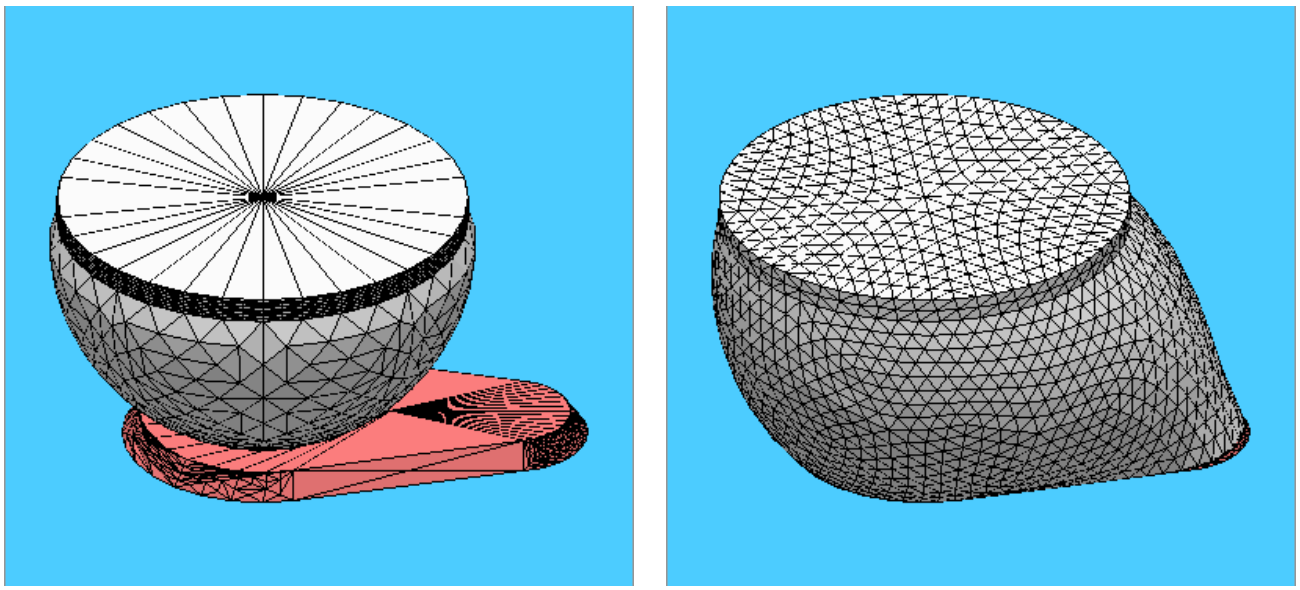
```
20. area:  4.92221107324660 energy:  4.92221107324660  scale: 0.249909
19. area:  4.91072281356264 energy:  4.91072281356264  scale: 0.209411
18. area:  4.90935398167529 energy:  4.90935398167529  scale: 0.264455
(...)
 3. area:  4.90277445806910 energy:  4.90277445806910  scale: 0.457579
 2. area:  4.90266224878650 energy:  4.90266224878650  scale: 0.261398
 1. area:  4.90256564950024 energy:  4.90256564950024  scale: 0.462497
```

Listing 4: Output of running 20 iterations over the surface with the command *g 20*

The surface can continue to be refined and proceed to more iterations. After that, the command *q* quits the software, asking the user to save the final surface.

## 2.4 CASE STUDIES

In order to execute real scenarios with SE, two case studies of simulation used by Bosch were used to measure the performance and to use in development. First, a case study with a smaller number of elements, from now on referred to as *smaller*, to analyse the performance of SE in cases where it fits in the CPU caches and to be used in active development where it is not practical to wait long periods of time between changes in the source code. Second, a case study with a larger number of elements, from now on referred to as *larger*, to analyse the performance of cases that do not fit in the CPU caches and need to access the main memory.

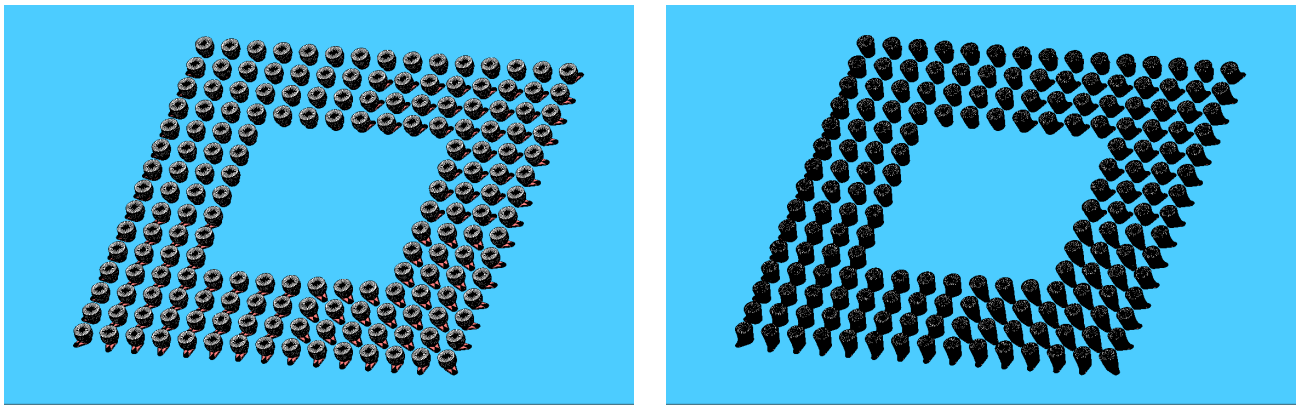


(a) Initial surface (b) Final surface

Figure 4.: *Smaller* case study surface evolving at the initial and final state

The *smaller* case study, shown in figure 4, is initially represented by 10K elements and a memory usage of 1654KB and evolves to a surface with 15K elements and a memory usage of 2366KB. The iterative process is composed by 80 main iterations with computations and mesh refinements throughout the iterations.

The *larger* case study, shown in figure 5, is basically a surface with a representation of 176 elements like the one represented in the *smaller* case with some changes on their attributes and a more refined iterative process. It is initially represented by 500K elements and a memory usage of 314MB and evolves to a surface with 1M elements and a memory usage of 978MB. It also has 80 main iterations but with an even more refined mesh resulting in a considerable increase in the amount of computations and heavier refinements.



(a) Initial surface (b) Final surface

Figure 5.: *Larger* case study surface evolving at the initial and final state

In the measures performed of the original version of the SE, during the profiling phase as shown ahead, the *smaller* case study has an execution time of approximately 2 seconds and the *larger* case study has an execution time of approximately 35 minutes.

SUMMARY

*The modelling of liquid surfaces shaped by various forces and constraints is used in different fields of research and industry to study and simulate the behaviour of these surfaces in different scenarios, such as the thermal fatigue failure of the welds in PCBs which leads to a more premature end of an electronic component. To find an approximate solutions for these problems, numerical methods like the FEM are used to solve continuous and high complex problems by analysing discrete parts of these surfaces, for which it is possible to know or obtain a mathematical description of their behaviour.*

*The FEM goes through three main stages: the pre-processing stage, when the domain is discretized and all the constraints are defined, the analysis stage, when all the calculations are performed and finally the post-processing stage when the results are presented.*
*SE is an interactive program that uses the FEM to study these surfaces and the complexity of the analysed problems shows that an efficient computational solution is required to reduce the execution times, allowing the study of more complex problems.*

# 3

# PARALLEL COMPUTING BACKGROUND

*This chapter introduces the parallel computing background, specially related to parallel computing and its use in High Performance Computing (HPC) to increase the performance of software. Starting with homogeneous environments, it is presented profiling tools, hardware and software multithreading, Non-Uniform Memory Access (NUMA) environments and vectorization and their impact on the performance of software. It is also explored heterogeneous environments, specially with a GPU as an accelerator, its architecture details and also the Compute Unified Device Architecture (CUDA) programming model. In order to develop more efficient code, frameworks and APIs like pthreads, OpenMP, StarPU, Thrust and OpenACC, etc. will also be be presented as potential alternatives to increase the SE performance.*

## 3.1 HOMOGENEOUS ENVIRONMENT

Environments composed by one or more CPUs with a main memory RAM are, until today, the most usual types of systems available. The hardware of each CPU garantees the same storage representation and the same memory space for all the computing units, unlike heterogeneous environments, where different computing devices might have its own memory organization and computing units [4].

### 3.1.1 *NUMA*

Although in homogeneous environments, there is a memory space for each device, if a computing node has more than one device, each one has its own memory space causing a NUMA pattern. In such scenario, each device when it needs to fetch data from a memory space of a different device, it is slower than to fetch data from its own memory space.

In a shared memory paradigm, this is a problem which normally lowers the performance when executing threads in a different device than the one where the data is allocated. All the threads, on a cache miss, will need to fetch from a different memory space which belongs to a

another device. Therefore, a key element in improving the performance on these environments is dealing with memory affinity [5].

The appendix A.1, presents the experimental setup, including the node of the SeARCH cluster at the University of Minho where all the performance tests were realized, with the multicore structure shown in figure 6.

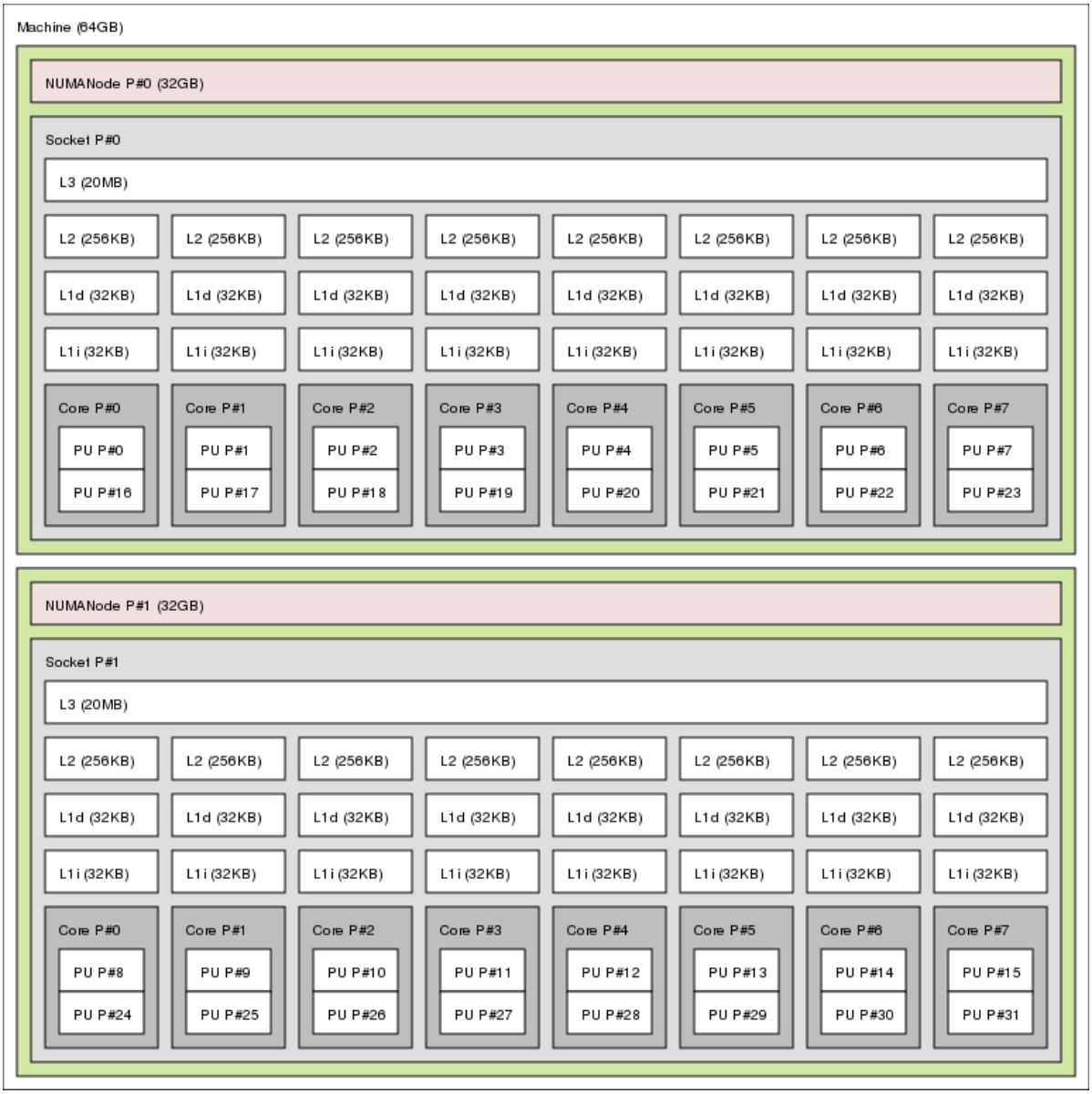



Figure 6.: Shows the node topology used in the experimental setup

Figure 6 shows that the node has two NUMA nodes, one for each of the two computing devices, each one connected to external 32GB of RAM. As mentioned, if the main thread allocates memory in the NUMA node P#0, all threads running in one of the cores of the second device will have a penalized memory access time.

### 3.1.2 *Hardware Multithreading*

Hardware support for multithreading is the ability to execute multiple threads in the same core in a device [6]. This is done by sharing all the resources of the device, such as registers or the ALU and even the caches of that core. It can lead to a more efficient usage of a CPU, if a thread is stalled for some reason, another thread can continue its execution.

### 3.1.3 *Multithreading*

Intel implemented a Simultaneous Multithreading (SMT) approach in its x86 CPU devices, trademarked as *Hyperthreading*, that allows a single processor core to interleave two threads of execution more efficiently, since each core will be able to concurrently balance two threads of execution on a given core. As shown in the node used by the experimental setup, in figure 6, there are 2 Intel devices with 8 physical cores, each supporting 2 threads. This means that after the 16th thread, a new thread will run in the same core as one of the previous threads, sharing all the resources of that core. Software with a particular usage of resources can take advantage of such technology, leading to an increase of performance. However, in cases where the workload is more or less equal for each thread, this will have both threads sharing the same resources, which may not improve the performance and, in some cases, may even drop the performance.

### 3.1.4 *Vectorization*

A particular subset of Single Instruction, Multiple Data (SIMD) is vectorization. This is the ability to transform a scalar implementation, which executes an atomic operation with two scalar operands, in a vector (series of adjacent values) operation. These SIMD operate on multiple data in just one instruction, depending on the presented technology of the CPU. For example, Intel Streaming SIMD Extensions (SSE) has 8 registers of 16 bytes known as XMM0-7, which allows to store 4 floats in each register. Vectorizing in such a scenario, allows to implement an operation with 4 floats at the same time [7].

In Advanced Vector Extensions (AVX), the width of the SIMD registers is increased from 16 bytes to 32 bytes, and renamed from XMM0–XMM7 to YMM0–YMM. This doubles the number of operands to 8 (in the example above, using floats) or 4 doubles. More recently, the AVX-512 was announced, supporting a 64 bytes extension to the AVX-256, meaning that the number will double again soon (by the time this dissertation was written, it was announced to late 2015 or early 2016 in the new Knights Landing Xeon Phi).

Many compilers add support to automatic vectorization in loops and other regions, if the code is written following a particular set of instructions. They can also generate a report with the vectorized operations, and in the case of Intel C++ Compiler (ICC), also tips to improve the efficiency by advising how to vectorize potential loops or regions. In the chapter 5.4.3, this will be explained in greater detail as well as the implementation of SE to increase the performance through vectorization.

## 3.2 HETEROGENEOUS ENVIRONMENT

For decades, microprocessors based on a single CPU were used to execute software. As new CPU models came to the market, the increase of transistors and clock rates led to a more efficient software with low or no effort by the developers. Moore's law [8] states that the number of transistors on a chip will double every 18 to 24 months. However, at the start of the 2000s we hit the power wall, but it is still possible to increase the number of transistors as Moore's law indicates. So, instead of increasing clock speeds, transistors were used to build larger caches, pipelines, branch prediction, prefetching and other architecture enhancements. Not being able to make a single CPU faster, another use for these transistors was to built more than one processing units (CPU + cache) in the same chip/device, also known as cores. However, in the multicore era, developers cannot expect their software to automatically increase the performance of their existing codes on a newly designed chip and software needs to be optimized to run on multicore devices.

CPUs are designed for latency. They have sophisticated control logic to allow instructions from a single thread to be executed in parallel or even out of their order. Moreover, larger caches are build to reduce the latency of memory accesses on more complex applications. A different design approach can be seen on GPUs. GPUs are designed for throughput. During many years, the video and game industry required a massive number of floating-point operations and needed hardware to do so. Reducing the control logic and power of the GPU allows the designers to have more computing units on a chip and thus increase the total execution throughput.

These conceptual differences on CPUs and GPUs should be clear. GPU should not be used when each thread needs more control logic or has intensive memory accesses. Since GPUs have a fast thread switching, thread idle time is covered by quick replace its execution by another. CPU has a more sophisticated logic control but is limited to a more limited number of execution units.

These environments, where both CPUs and GPUs coexist (and other device types such as the Xeon Phi), are referenced to heterogeneous environments. The new challenges involves the

design of efficient solutions to run on different architectures and take advantage of the best of each one.

### 3.2.1 *GPU and CUDA programming model*

The use of GPUs to handle computation other than graphics related is named General Purpose Graphics Processing Unit (GPGPU). A GPGPUs is used in HPC to increase the performance of massive parallel applications. The main manufactures, NVIDIA and AMD, produces GPUs and they can be programmed using either CUDA or OpenCL, however AMD GPUs can only be programmed with OpenCL.

GPU devices have a very different architecture from CPUs so they require a different programming approach. CUDA is an extension to the C programming language and has its own programming model adequate to execute in GPUs [9]. At the time of this dissertation, there are four generations of NVIDIA GPUs to be solely used as computing accelerators: Tesla, Fermi, Kepler and most recently, Maxwell. They differ mostly in terms of capacities, numbers and computing capability. This dissertation focus more on the Fermi and Kepler architectures so they will be the most explored.

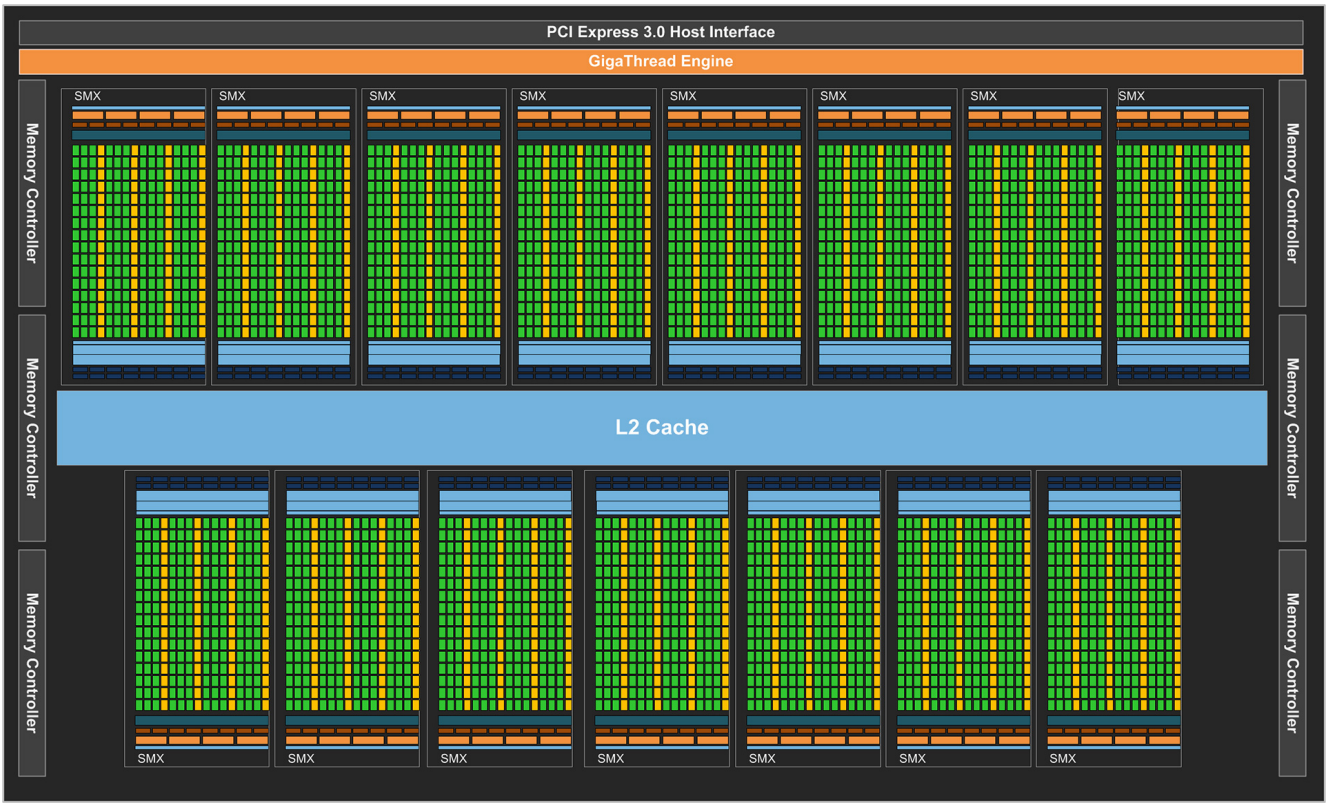



Figure 7.: Kepler architecture - full chip block diagram [1]

A modern GPU architecture is organized into an array of Streaming Multiprocessors (SMs) or Extended Streaming Multiprocessors (SMXs) as shown in figure 7, the former was used by Tesla and Fermi GPUs, while the latter in Kepler and Maxwell. The Kepler architecture has up to 15 SMXs, however the number of SMXs can vary from one generation of CUDA GPUs to another. Each SMX, has shown in figure 8, has a number of Streaming Processors (SPs) that share control logic and an instruction cache [10].


Figure 8.: A SMX from the Kepler architecture [1]

Kepler architecture also has a L2 cache shared by all SMXs and it is connected to memory by 6 memory controllers. GPUs comes with a Graphic Double Data Rate (GDDR) Dynamic

1 http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Random Access Memory (DRAM), referred to as global memory. This GDDR DRAM differs from the DRAM on the CPU board in that they hold graphics information or in the case of massive parallel applications for off-chip memory, thought with higher latency than typical systems DRAM. This impact can be reduced by the study of locality in local memory and caches.

Each Kepler SMX contains 192 CUDA cores (single-precision floating-point compute elements) and 64 double-precision units as well as 32 Special Function Units (SFU), and 32 load/store units. With 16,384 threads, the GTX680 exceeds 1.5 teraflops in double precision [11].

A CUDA device is typically a GPU. They execute CUDA kernels on one or more devices usually called by the host, typically a CPU. The functions or data declarations for both host and device are clearly marked with special CUDA keywords. When a kernel function launched, it is executed by a large number of threads on a device. All the threads that are generated by a kernel launch are collectively called a grid. All threads in a grid will run the same kernel code as according to a Single Program, Multiple Data (SPMD) approach. When all threads of a kernel complete their execution, the corresponding grid terminates.

The execution of a kernel will generate several thread blocks of 32 threads called warps that are scheduled and executed via the warp schedulers and dispatch to a SMX. Each SMX can execute 4 warps, each warp executing two independent instruction per clock cycle.

### 3.2.2 *Intel MIC*

This dissertation work was performed in parallel with another dissertation work by Bruno Araújo, which explores different points, one of them is the efficient use of the Intel MIC accelerator. For further informations about the use of Intel MIC to improve the performance of the SE code, we suggest to consult that dissertation [12].

## 3.3 SOFTWARE

The main software environment which needs to be discussed in this context are compilers, as well as tooling to perform profiling analysis and debugging. The main compilers explored, as shown in the appendix A.1 regarding to the experimental setup, were the GNU Compiler Collection (GCC) and the ICC. As shown throughout this dissertation, both compilers were used without any difference regarding the performance of the SE. However, using the ICC with the Intel Composer pack of utilities, might be useful to detect memory efficiency problems as well as other tools like the Intel VTune Amplifier to inspect performance metrics in a program.

GProf and Valgrind are performance tools to profiling and debug the program. GProf collects data from a program with instrumentation code added by compiling the program with -pg. This

data can then be inspected by the developer or, to ease this analysis, convert the output to a visual call graph by using tools like gprof2dot[2]. Valgrind lets the developer measure other performance metric such as cache misses, CPI, loads/stores, etc. Usually this type of analysis, like it was done in this dissertation, uses PAPI, to specify a standard API for accessing hardware performance counters available on most modern microprocessors [13].

PAPI can be used to support relevant performance decisions and whether to take an effort to optimize or even reimplement a part of the code. Valgrind is a non-obtrusive way to do this, without inserting code, however the margin of error is higher. Valgrind also allows the developer to check its memory performance and identify memory leaks or other memory related constraints like dangling pointers, uninitialized variables, illegal frees, etc.

### 3.3.1 *Shared Memory Environment*

In a shared memory environment, two or more threads share a given region of memory [14]. In these environments of execution, it is normal to have parts of that region that need a synchronized access, e.g. the produces/consumer problem, where the producer writes data to that region and the consumer can only read if a producer has finished writing. Often, semaphores are used to synchronize shared memory access [15].

In such environments, race conditions can happen, causing a program to be non-deterministic, that is, multiple executions producing different results, depending on the order of execution of each thread.

Using threads to create workers is usually done with pthreads (in Unix systems) or OpenMP. Pthreads are the Unix standard to spawn threads and deal with everything during their life cycle [16]. OpenMP is an API for multi-platform shared memory parallel programming in C/C++ and the user only defines parallel regions and the API deals with all the threads life cycle [17]. Pthreads is a low-level API, which means that the user must implement everything, including critical regions, work sharing, scheduling, etc. OpenMP, on the other hand, has all these implemented with relative ease for the developer to include in the code.

OpenMP facilitates the implementation and it is often better to avoid synchronization problems due to implementations of such regions by hand. It has a built-in scheduler, allowing the developer to balance work between threads and implement loop parallelism and parallel regions and other features available through the API [18]. Another advantage is the use of *pragmas* to indicate parallel work, which in compile-time the developer can activate or deactivate, making the program parallel or sequential respectively. It can also use environment variables to define OpenMP primitives like the number of threads or the scheduling mechanism to use as shown ahead. This eases the implementation, but with a cost, since usually this comes with an higher

2 https://github.com/jrfonseca/gprof2dot

overhead. However, it is debatable, since the API has a set of optimized mechanisms to deal with several usual problems in parallel computing and the implementation of a developer might not be as optimized.

### 3.3.2 *Other Frameworks and Libraries*

Although CUDA is the NVIDIA programming model for GPUs, other frameworks and libraries can be used to improve the performance of applications running in heterogeneous environments. StarPU is a scheduling system of graphs of tasks which are assigned to heterogeneous platforms devices, providing a high-level, unified execution model tightly coupled with an expressive data management library. This way, the developer creates tasks that can be schedule to run in parallel, and StarPU assigns a device to run that task, using different scheduling algorithms [19].

Thrust is a parallel algorithms library that provides a high-level interface enabling performance portability between GPUs and multicore CPUs. It enables an abstraction between the heterogeneous environments in terms of unified memory for instance, facilitating transfers between the different devices [20].

The Magma project is a dense linear algebra library, similar to LAPACK but for heterogeneous/hybrid architectures. The latest version also includes some operations on sparse matrices [21]. Other libraries such as cuSP, cuBLAS and cuSPARSE also provides an interface to linear algebra operations but only executing on GPUs from NVIDIA.

OpenACC is a programming standard for parallel computing developed by Cray, CAPS, NVIDIA and PGI. Similar to OpenMP, developers can annotate C/C++ source codes using directives (pragmas) to identify the parallel regions. This can improve the performance of the software without much effort and parallel code can be offloaded to target devices such as GPUs [22].

#### SUMMARY

In this chapter it was explored the computing background, all the hardware and software that are used, specially in homogeneous computing environments as well as others that can be used in the future development of the SE for heterogeneous environments.

Several profiling tools are used to better understand the actual behaviour and performance of the SE, as well as hardware and software multithreading, NUMA environments and vectorization, which need to be taking into account when studying how to improve the performance of software. These will be analysed in greater detail in the next chapters, in the context of the SE.

# 4

# PROFILING THE SURFACE EVOLVER

*Before any new and improved implementation of any software, specifically in this case of the SE for liquid surfaces modelling, a profiling analysis should be performed to verify the current implementation. This analysis must identify the critical regions and which operations are taking longer, to decide whether to optimize or even reimplement those operations.*
*In this chapter, this analysis is made using call graphs of the main components of the software, exploring both normal and Named Quantities modes. Scalability tests to measure the current execution times, speedups and efficiency of this implementation are made using the two case studies explored in the chapter 2. The analysis of the current data structure and locality is also important to explore the impact that the maintenance functions have in the SE.*

The SE, currently in the version 2.7, was implemented to study the modelling of liquid surfaces shaped by various forces and constraints. Depending on the complexity of the problem being analysed, the computational load can be very high, thus having execution times not consistent with the needs of researchers and industry professionals. Before the research and implementation of new solutions to increase the performance of SE, both in homogeneous and heterogeneous environments, it is necessary to evaluate the current computing performance and analyse the current implementation of the software.

## 4.1 EXPERIMENTAL SETUP

All tests were run on the SeARCH cluster at the University of Minho. The methodology followed a k-best approach, where 6 runs were executed and measured and the best result was taken, provided that the difference between that and the 3rd best did not exceed 5%.

The computer node used has the following characteristics:

- 2 x Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
- #Cores: 8 (with *Hyperthreading*, where #Threads are 2)

- L1 Cache (per core):
  - 32 KB instruction cache
  - 32 KB data cache
- L2 Cache (per core):
  - 256KB cache
- L3 Cache (per device):
  - 20MB cache (shared by all cores)
- Main Memory: 64GB

For other informations about the methodology, node characteristics and software used as well as speedups and other calculations used in this chapter, see Appendix A.1.

## 4.2 CALL GRAPHS

A call graph is a directed graph used to represent the relationships between functions of a program as well as the impact in the overall performance. It is an important analysis where it can be identified the functions that take longer to execute.

In the SE, the call graphs were created using both case studies allowing to identify potential critical areas as well as the impact variation between smaller and larger problems.

In the *smaller* case study, a significant portion of the time is spent parsing the input from the user and in graphical operations however, as shown ahead in the *larger* case study analysis, this relative impact on the overall performance is reduced as problems get more complex, since the computation performed per command takes longer.

The most significant and time consuming functions in the *smaller* case study, as shown in figure 9, are:

- *recalc* - 33%: As mentioned, interpreting commands and graphical operations have a high relative impact in less complex problems. The main function responsible for this impact is intended to recalculate and display the new results. This functions may have parallelized computations since the calculation of energies and pressures, can be performed by element without dependencies.
- *calc_energy* - 24%: One of the heaviest functions is responsible for obtaining the total energy of the configuration and it is the heaviest computation included in the *recalc* function. This also, can be parallelized.

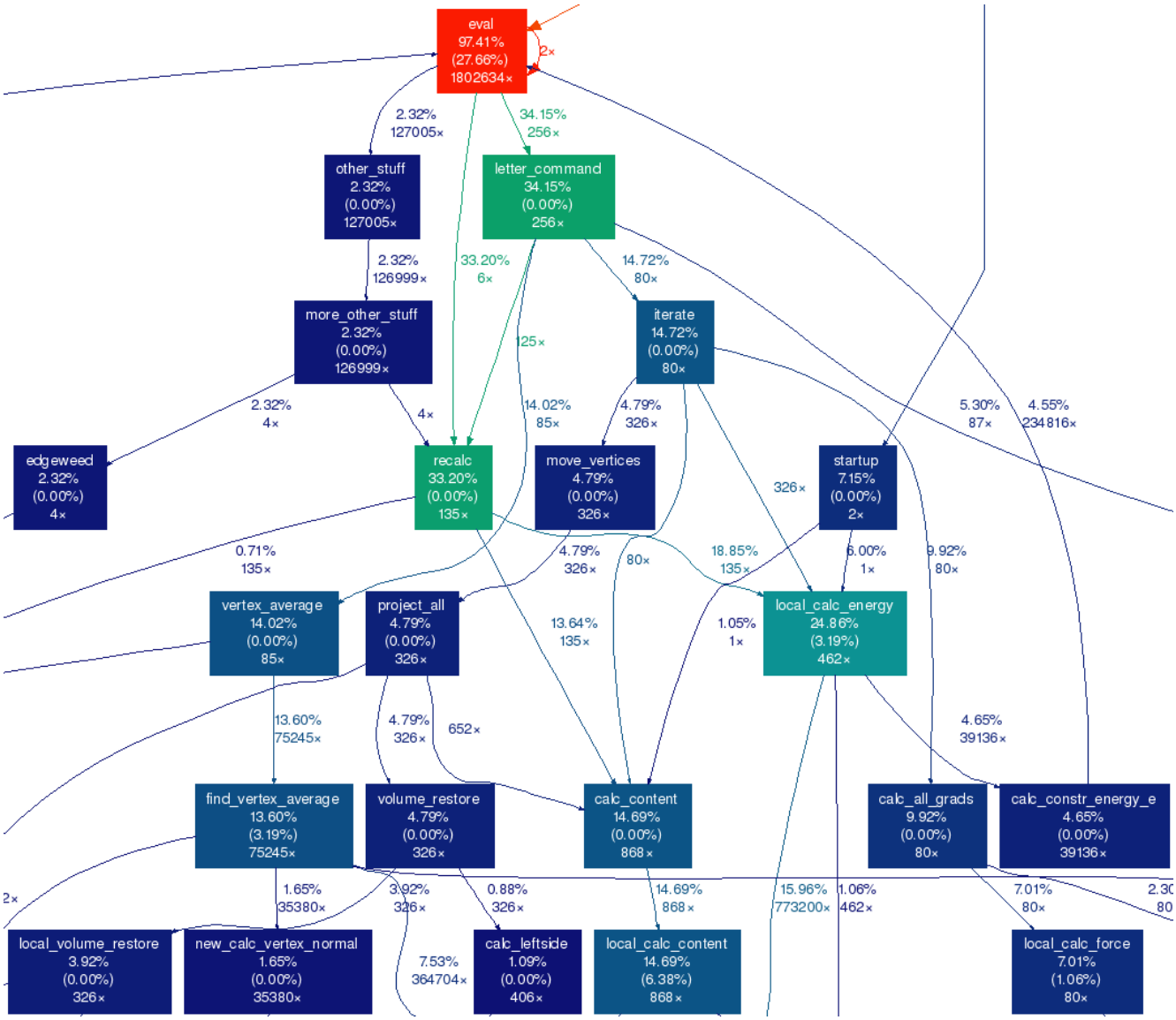



Figure 9.: Main component of the call graph - *smaller* case study

- *vertex_average* - 14%: Another heavy function is used to compute the average of a set of vertices. This also, can be parallelized.
- *calc_all_grads* - 10%: Used to compute the forces and/or gradients from the restriction configuration. This also, can include parallel computations.
- *calc_force* - 7%: This function computes the resulting force on the triangulation vertices due to surface tensions and constraints. This also, can be parallelized.

All the functions involving computations per element are qualified for parallel computing since they can be done without any dependencies between elements. Other computing operations involving all elements, such as obtaining the total energy of the configuration can be done with parallel computing using a reduce pattern.

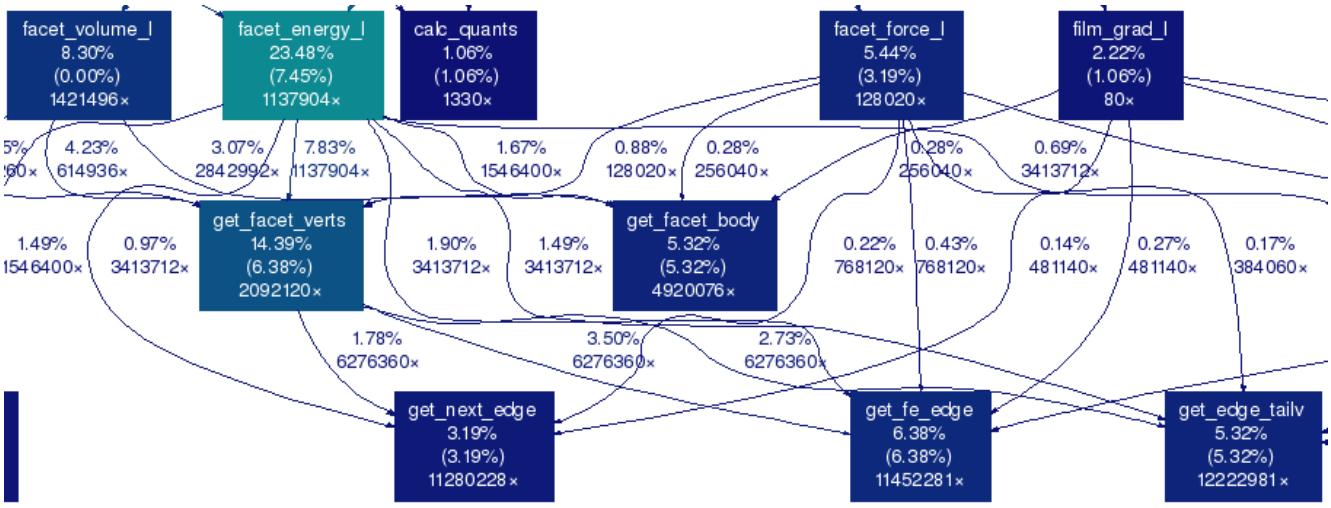



Figure 10.: Impact of the data structure used to represent the finite elements - *smaller* case study

Some of the heavier functions are also those involved in data structure operations namely to search edges, vertices, facets, etc. as shown in figure 10. These functions (*get_facet_verts*, *get_facet_body* containing *get_next_edge*, *get_fe_edge*, among others) have an impact of 14% in the overall performance with a tendency to grow with the complexity of the problems.

The data structure currently implemented, based on linked lists, must be reconsidered and it gets worst with the implementation of parallel computing as analysed ahead.

As shown in figure 11, the *larger* case study has a more complex problem which makes it longer to execute, specially for functions with more computational workload, increasing their respective impact on the overall performance of SE.

- *recalc* - 62%: the *recalc* function, which has an impact of 33% on the overall performance of the *smaller* case study, now has an impact of almost 62% on the *larger* case study due to the complexity increase of the problem, specially in the total energy computation. In the *smaller* case study, all data fits into the CPU caches, with a miss rate below 2% in the L1 cache and less than 1% in the L2 cache, mainly due to compulsory misses. The *larger* case study has a memory usage of almost 1GB and the non-contiguous memory accesses increase the miss rate, which is extremely costly to the overall performance of SE.

- *calc_energy* - 60%: This function also has an increase over the impact on the overall performance.

- *calc_force* - 1%: This function has less calls in this case study so it has its impact reduced.

- *vertex_average* - 7%: This function computes the average of a set of vertices and also has a significant impact on the overall performance.

The functions involved in data structure operations namely to search edges, vertices, facets, etc. have its impact significantly penalized as shown in figure 12. As described, this is actually

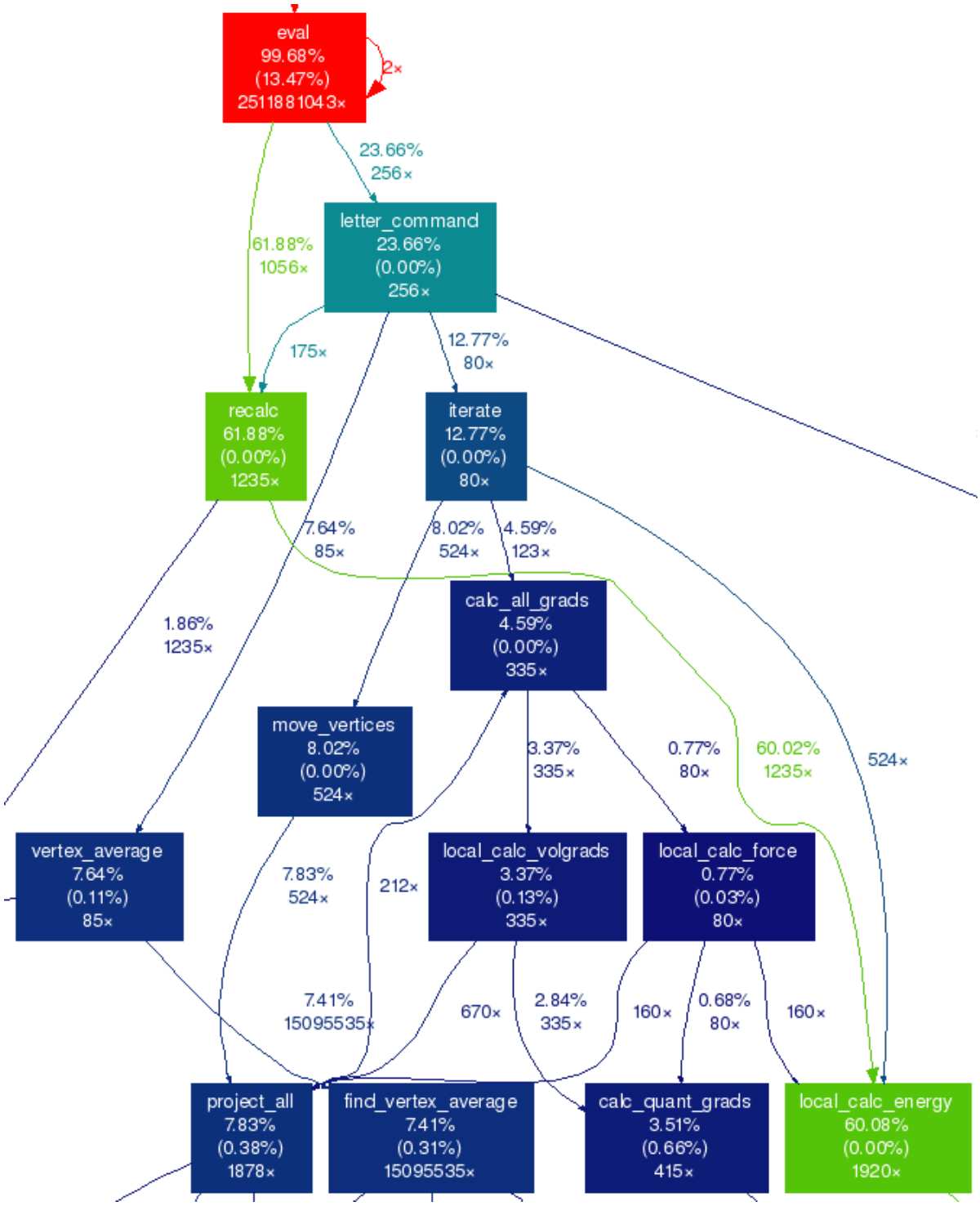



Figure 11.: Main component of the call graph - *larger* case study

where most misses occurs, penalizing the computation functions and the CPU gets stalled due to the miss penalty to receive data from main memory.

This data structure as mentioned is based on linked lists and should be reconsidered, as it will be used in a software with a strong parallel computing component, should be given a particular attention to the partition of the data.

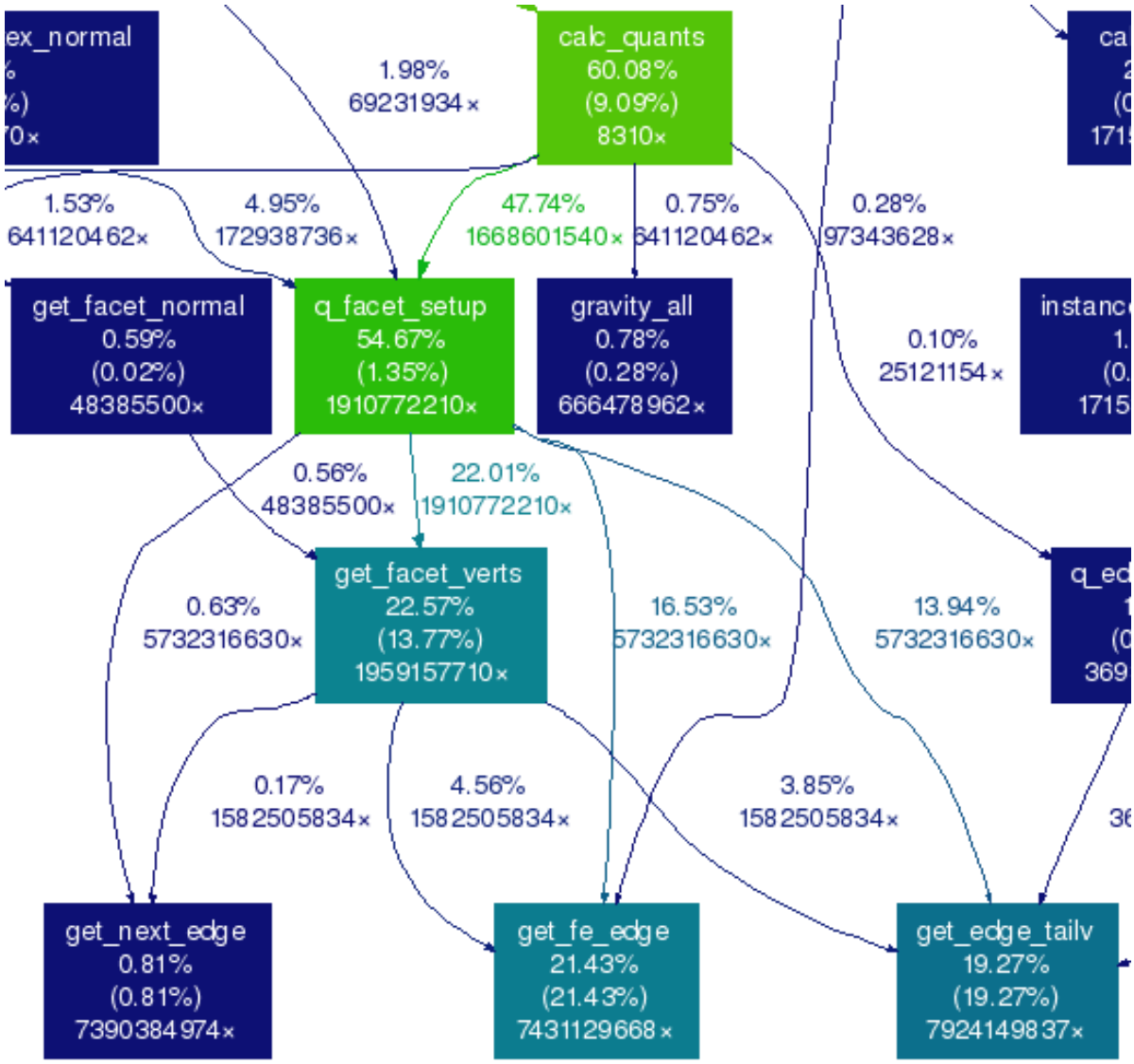



Figure 12.: Impact of the data structure used to represent the finite elements - *larger* case study

## 4.3 CURRENT IMPLEMENTATION ANALYSIS

The SE has implemented some parallel functions, in the normal and *Named Quantities* mode. *Named Quantities* are the systematic scheme of calculating global quantities such as area, volume, and surface integrals that replaces the original ad hoc scheme in the SE. Next, it is studied the scalability of the current implementation for the two case studies - *smaller* and *larger*, with and without the use of *Named Quantities* - which is done with low-level parallelism implementations using POSIX threads.

All the tests were made using up to 14 threads. The SE has a problem which needs to be solved, when running with more than 14 threads, SE returns a memory corruption error. So it was not possible to include the results with more than 14 threads.

### 4.3.1 *Smaller case study*

The *smaller* case study has too low-complexity to draw any significant conclusions however, as some of the problems have this kind of complexity, the analysis of such case studies is important.

*Without Named Quantities*

In terms of execution times, the *smaller* case study is relatively fast and the sequential version has an approximate execution time of 2.25 seconds. In fact, for problems of low-complexity, that is, problems for which the surface dimension fits in the CPU cache, and for some number of threads, the parallel implementation has higher execution times than the strictly sequential version.

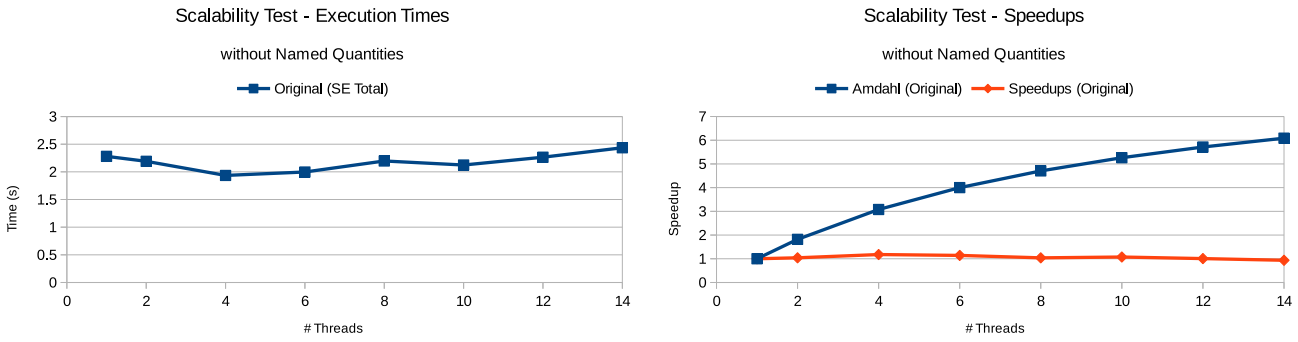



Figure 13.: Execution times and speedups using up to 14 threads without *Named Quantities* - *smaller* case study

The thread creation overhead is not enough to compensate the use of parallelism in such small case studies, thus not scaling. The speedups reflect the behaviour of the execution times and Amdahl's law states that the parallel version is limited by the sequential portion of the software (see A.2).

Analysing the software behaviour, the currently implementation needs approximately 10% of sequential code for parsing, initial settings, computations between each iteration, etc. So, the maximum speedup attainable with a certain number of threads can be represented as shown in figure 13 and the difference between the current speedups and the maximum attainable speedups are significant. Also, these 10% of the sequential proportion are taken from the current implementation of the SE. Some portions may, with deeper analysis, be reduced thus increasing the maximum attainable speedup.

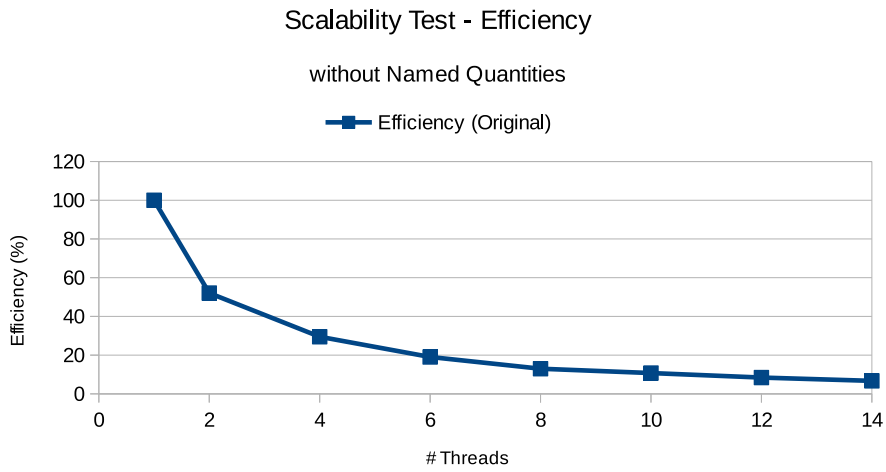



Figure 14.: Efficiency using up to 14 threads without *Named Quantities* - *smaller* case study

Another important metric to analyse is the efficiency. This metric estimates how well the resources, the CPUs in this case, are being used. Since the efficiency directly depends on the speedups, the results show in figure 14, as well as the speedups analysis, that the current multithreaded implementation is inefficient. Using 2 threads, each thread only works 52% of the execution time and with 14 threads the efficiency drops to 7%.

*With Named Quantities*

Using *Named Quantities* to compute quantities such as the total energy of the configuration adds a more schematic way of computing these quantities. However, to facilitate the way the user can provide the computational method, the software has more workload, thus decreasing the overall performance.

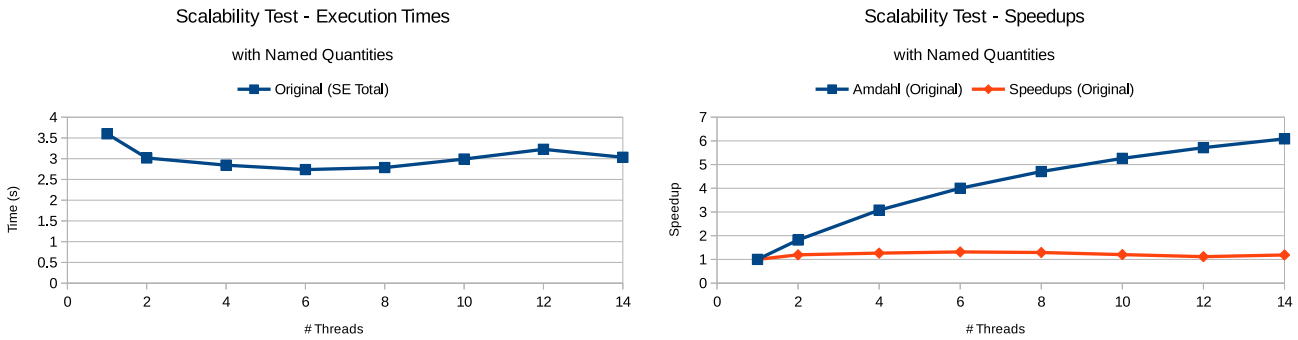



Figure 15.: Execution times and speedups using up to 14 threads with *Named Quantities* - *smaller* case study

Both *smaller* and *larger* case studies have worst execution times than in the normal mode as show in figure 15. This trade-off of functionality and code usability against performance can be observed in many scenarios of software development.

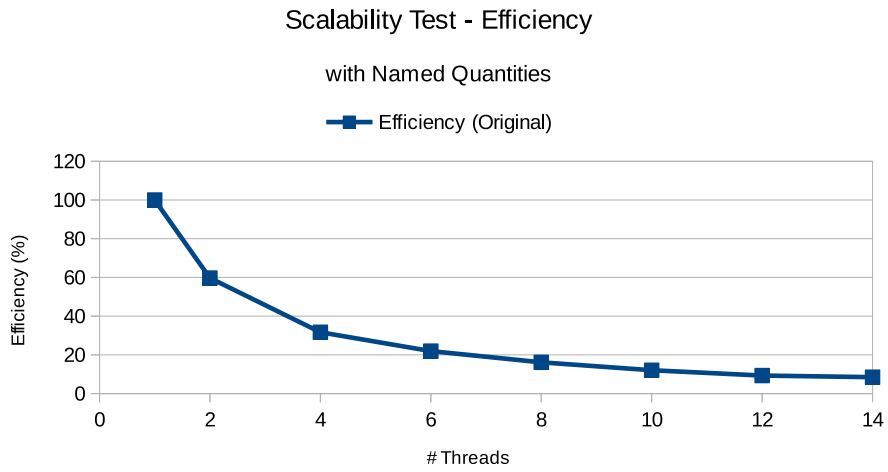



Figure 16.: Efficiency using up to 14 threads with *Named Quantities* - *smaller* case study

The *smaller* case study is also fast and the sequential version has an approximate execution time of 3.65 seconds. Here, the surface dimension also fits in the CPU cache. Since it has more workload per element than in the normal mode it scales a bit better, however without any significant speedup. The number of elements is still too low to compensate the overhead creation of the threads.

Observing the efficiency, which directly relates to the speedups, the results show in figure 16, that the multithreaded implementation is also inefficient. Using 2 threads, each thread only works 60% of the execution time and with 14 threads the efficiency drops to 8%.

### 4.3.2 *Larger case study*

In the *larger* case study, since it has significantly more elements, the thread creation overhead pays off in the normal mode. However in the *Named Quantities* mode, the amount of additional workload to maintain the computational processes over each quantity, reduces the parallelism capability and makes it almost sequential to compute over all elements. As mentioned, this trade-off is noticeable here also, where to increase the functionality of the software and to facilitate the interaction with the user, the overall performance is reduced.

*Without Named Quantities*

In the *larger* case study, the use of parallel computation has better speedups than in the *smaller* case study as shown in figure 17.

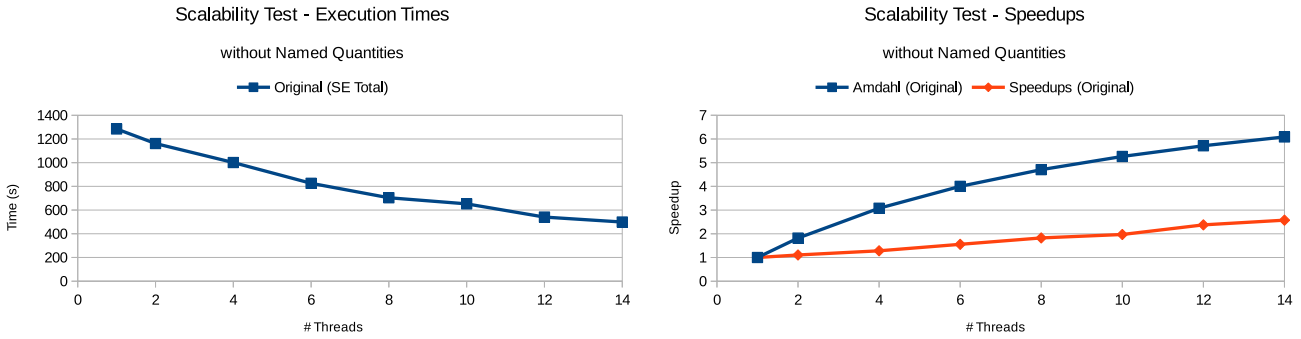



Figure 17.: Execution times and speedups using up to 14 threads without *Named Quantities* - *larger* case study

The computation done by each thread regarding to the number of cache hits is higher, thus reusing the data in the caches from the cores where each thread is running. The speedups shown in figure 17 reflect the execution times and as mentioned, are far from the maximum attainable speedups calculated according to Amdahl's law with the same 10% of sequential portion of the code. In fact, the maximum speedup is just below 3 with 14 threads.

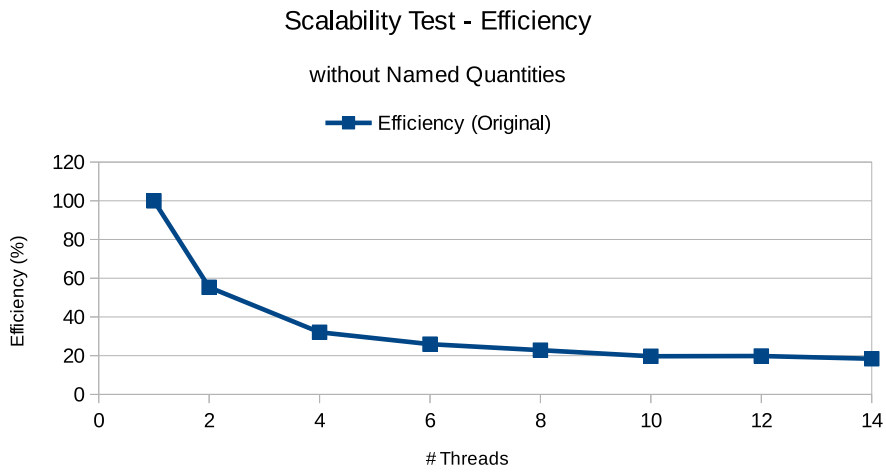



Figure 18.: Efficiency using up to 14 threads without *Named Quantities* - *larger* case study

The *larger* case study has a more efficient use of the processors using parallel computations as shown in figure 18. Each thread is reusing data in the cache of the core where it is running which makes it possible to increase the efficiency. However the efficiency is still very low with 58% using 2 threads and 20% using 14 threads.

*With Named Quantities*

The *Named Quantities* as shown in figure 19, have a worst performance than the normal mode. In fact, these differences are noticeable not just in the sequential version but also in the multithreaded version. The thread creation overhead is also prejudicing the performance because it has more data associated per thread which causes this overhead to be higher than in the normal mode.

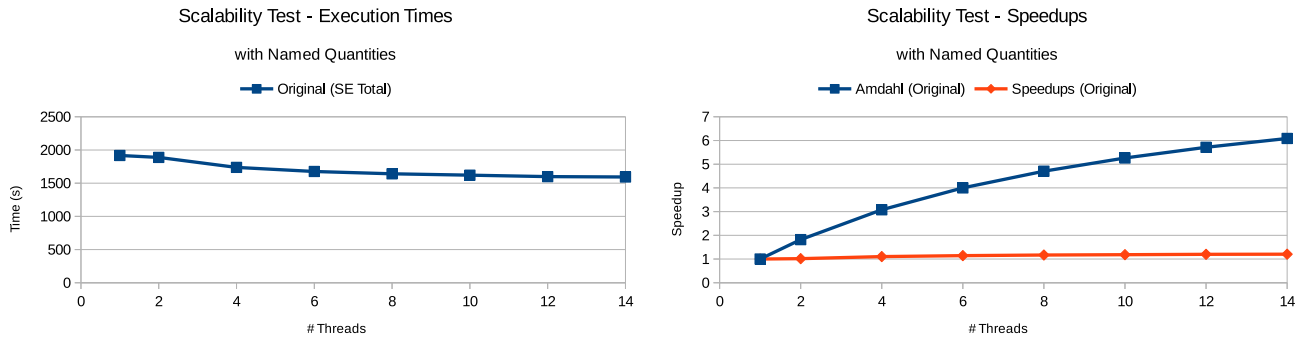



Figure 19.: Execution times and speedups using up to 14 threads with *Named Quantities* - *larger* case study

The speedups shown in figure 19 reflect the execution times and are far from the maximum attainable speedups, calculated according to Amdahl's law, with the same 10% of sequential portion of the code. In fact, the additional workload prevents this case study from scale.

The efficiency as show in figure 20 also demonstrates that the use of *Named Quantities* is worst than using the normal mode. It is also very low with 50% using 2 threads and 8% using 14 threads.

## 4.4 DATA STRUCTURES AND LOCALITY

As seen in the call graphs, functions involved in data structure operations namely to search edges, vertices, facets, etc. have a great impact on the overall performance. This impact is even greater when running with parallel computations in a NUMA environment, as the one where

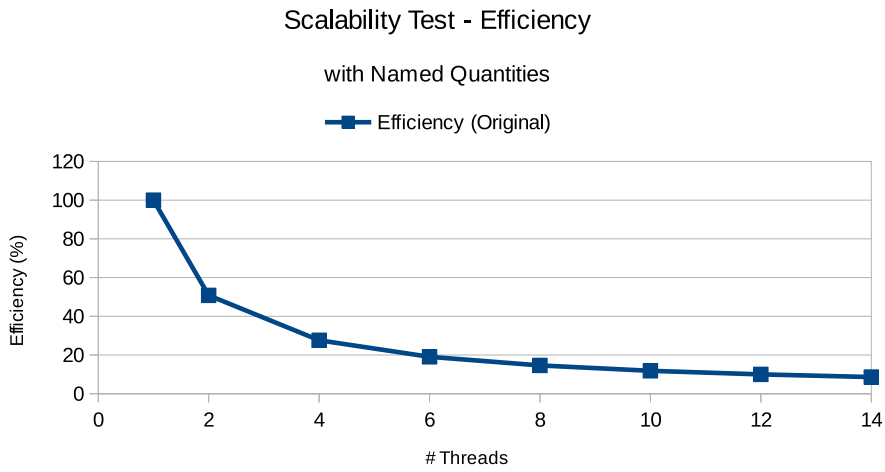



Figure 20.: Efficiency using up to 14 threads with *Named Quantities - larger* case study

these tests were performed. In NUMA environments with more than one device, each device has its own local memory which is faster to access than other non-local memory spaces belonging to other devices. For instance, when running SE using 2 threads, if the second thread is allocated to run in a different core from another device, memory accesses to the local memory of the first thread are slower. The second thread will have to access a non-local memory that is in a different device, thus the access will be penalized.

### 4.4.1 *Current Data Structure*

As mentioned, some of the heavier functions in SE are those that deal with the data structure. Search for a particular element, insert/remove and other operations have an impact of almost 15% in the overall performance. It also reduces the performance of other computations since it reduces memory locality and prevents vectorization.

Analysing the current implementation of SE as shown in figure 21, it is noted that the main data structure that contributes to all computation processes (excluding auxiliary structures used in parsing, GUI, etc.) is the *web*.

The *web* is the structure that supports the entire content of a surface. In the *web*, it is contained the entire information of the surface, including the elements that build the surface as well as the various boundaries and constraints, size, total area and total energy of the surface, among other information. Inside the *web*, are contained the elements of the surface. These elements are separated into five types: *Vertex*, *Edge*, *Facet*, *FacetEdge* and *Body*.

There is also a special type *Element*, used to represent any type of element, useful for computations that iterate over all the elements (independent of their type). Also to learn information from a particular element without even knowing its type, just by having a pointer to it.

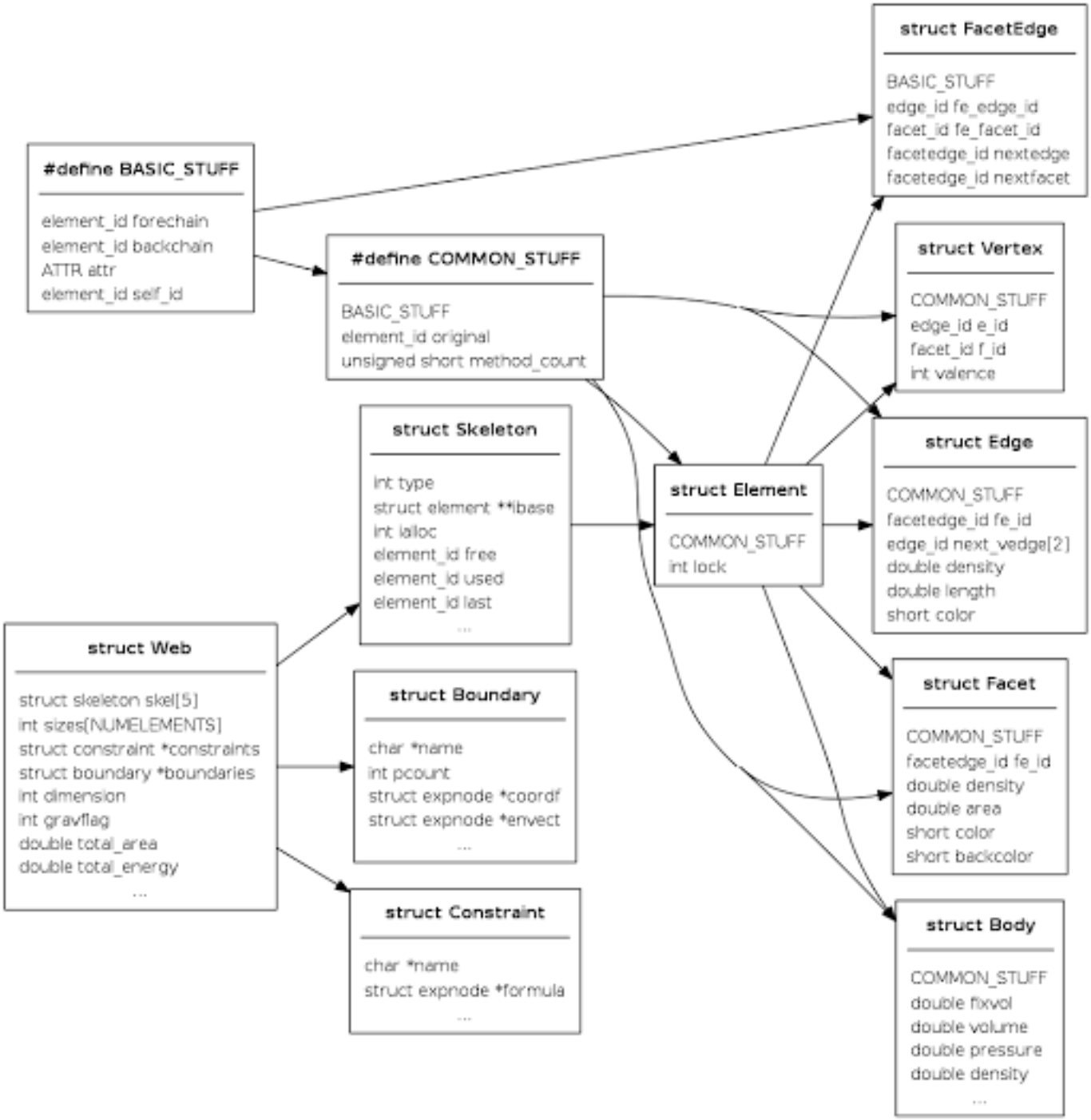



Figure 21.: Main data structure scheme of SE, representing the *web* and the different types of elements and boundaries/constraints

These elements are inserted into the *web* through an array (with 5 positions) of *Skeletons*. A *Skeleton* is a structure used to represent each group of elements, accessible from the *web* through their respective position:

- Position 0 - *Vertex*
- Position 1 - *Edge*

- Position 2 - *Facet*
- Position 3 - *Body*
- Position 4 - *FacetEdge*

Inside the *skeleton* of each type of elements, a flag is saved to identify its type and also a linked list. This linked list has its start marked by a pointer named *used*, has a pointer named *free* which points to a position where it can be inserted an element and also a pointer named *last* which points to the last element. The linked list is used to save all the elements corresponding to a specific type.

The elements that compose each of the respective lists may be one of five types as described. Each element has its own set of attributes defined internally by SE as well as an extra set of attributes that can be defined by the user.

- *Vertex*: A *vertex* is a point in space represented by its coordinates (these coordinates can change as the surface evolves). In addition to a set of properties, common to other types of elements, a *vertex* is connected to the overall surface through one of the edges and facet that compose the *vertex*.
- *Edge*: An *edge* is a one-dimensional geometric element. It is connected to the global structure through two vertices and a *facetedge*. It also has a set of attributes, common to all types of elements as well as type specific attributes such as the density, length and the color with which it is drawn in the GUI.
- *Facet*: A *facet* is a structure that represents a face in the shape of a triangle. In the input datafile there is also an auxiliary element, a *face*, which allows the user to represent a *face* with other geometric shapes, a rectangle for example. However, SE only recognizes triangles so in the pre-processing phase, while creating the surface, all the faces are converted to a set of *facets*. It is connected to the global structure through a *facetedge* and in addition to attributes in common with other types of elements, a *facet* has its own attributes such as density, area and also the front and back color with which it is drawn in the GUI.
- *FacetEdge*: A *facetedge* is an internal structure, defined by SE as a way to represent an oriented pair between a facet and one of its edges, so that the orientation of the edge remains consistent with the orientation of the facet.
- *Body*: A *body* is a set of elements that composes all the dimensions of a region in space. It also has associated a set of attributes, common to other types of elements, as well as other specific attributes such as the volume, density and pressure of a region.

As noted in the data structure scheme shown in figure 21, some attributes are specific to each element type and others are common to all the elements. In the structure of the SE, this separation is made with two macros: *BASIC_STUFF* and *COMMON_STUFF*. The *BASIC_STUFF* macro (which is part of all types of elements) has a set of variables used to manage the linked list. It contains the *forechain* and *backchain* variables, that are pointers to the next and previous elements of the list. It also contains the inner *id* of the element, assigned to the element by SE as well as a bitmap *ATTR* used for internal attributions.

The *COMMON_STUFF* macro contains the *BASIC_STUFF* set of variables and also the *original id* of the element, that is defined by the user in the input datafile. It is necessary to save the *original id* because the id assigned internally by the SE may not be the same as specified by the user, however, the element can still be referenced by the user, in a computation process, by its *original id*. All types of elements may have an *original id* so the *COMMON_STUFF* macro is included in their structure definition, except for the *facetedge* type, which is only defined internally and it just has the internal id assigned by the SE.

In addition to the elements, the *web* also has a list of *boundaries* and *constraints* that can be defined by the user in the input datafile. These *boundaries* and *constraints* can be associated to each element (or a set of elements) and they are identified by a name and an expression. These expressions (that can be a formula or other SE specific expression) act on the elements and are represented as a structure of the *expnode* type after the parsing phase.

### 4.4.2 *Linked lists*

SE main data structure is based on linked lists. Linked lists can be resized dynamically which is very useful in programs like SE, where mesh refinements produces more elements. Also, linked lists allow different element types which is not possible in arrays. Other interesting operations are the insertions at the head or tail of a list with a complexity of $O(1)$. Typically, programmers are attracted to linked lists to minimize the memory usage. Also, it is the easiest dynamic data structure to write from scratch [1].

However, when performance is critical, linked lists have several disadvantages that lowers the efficiency, not only in sequential but also in parallel computing and specially when executing in heterogeneous environments:

- *Out-of-order*: the performance is increased with out-of-order execution of instructions. However, accesses in linked lists are dependent as the address of the next element is only computed after the current element has been loaded from memory.

1 http://www.futurechips.org/thoughts-for-researchers/quick-post-linked-lists.html

- *Hardware prefetching*: modern microprocessors computations are faster than memory accesses so prefetching data from memory before is actually needed increases the overall performance. However, it is not possible with linked data structures.
- *Locality of reference*: linked lists have the data elements (even the adjacent ones) spread all over the memory. This means that every time the list is iterated, the next element will often be loaded from memory.
- *SIMD*: data level parallelism can be exploited with SIMD operations. A single instruction can operate on multiple elements thus increasing the performance. However elements in a linked list have to be fetched one at a time.
- *GPUs*: in heterogeneous environments, specially using GPUs as computing accelerators, memory address spaces are different. Thus, when sending a list over to the GPU, all pointers have to be converted from one memory space to the other. Having a lot of pointers also reduces considerably the performance of the GPU execution, since the global memory accesses are highly penalized.

### 4.4.3 *Arrays*

Arrays, unlike linked lists, are not resizable. To increase the size it is needed to reallocate the array which may end up either in the same position or in a new position, moving the data somewhere else. However, this impact can be amortized. If possible, starting with a predictable size and double the current amount when the array is full could increase the performance of SE. One of the advantages of linked lists, allowing different element types, are not needed on SE. Parallel computing with arrays is also faster than with linked lists. The data partitioning without contiguous elements in memory could create a load balancing problem since the elements are spread in memory, each thread can take longer to iterate over its partitioned set of elements.

In heterogeneous environments, arrays are also more efficient. All pointers in a linked list need to be converted from one memory space to the other. The amount of pointers in a linked list makes it considerable expensive to send it to the GPU. Global memory accesses are also more efficient with arrays, since the elements are contiguous in memory.

### 4.4.4 *Concurrent Data Structures*

Standard data structures, with concurrent operations applied to their elements need synchronization primitives to make it thread safe. In parallel computing, this is one of the major bottlenecks when global operations are applied. Until a few years ago, the concern was not so great as

it is today, since sequential computing was the main paradigm, data structures have never been designed to be inherently parallel[1]. Today, in the multicore era, data structures with a minimum of synchronization possible are needed. Other data structures like octrees or graphs can induce a partitioning in which can be identified potential critical regions in mesh refinement and other operations of the FEM.

### SUMMARY

In this chapter it was analysed the performance of the current SE implementation. Observing the call graphs, one of the main critical areas are the data structure operations. These not only reduce the performance of the sequential implementation, as also reduce the performance of the current parallel implementation and any future implementation, specially in heterogeneous environments. The main data structure of SE, current implemented with a linked list of finite elements, it is not the best when computing performance is taken into account, as shown by the speedups and efficiency analysis of the current implementation. Another important computation is the total energy computation of the configuration. The function - *calc_energy* - computes the energy of the surface and has an impact close to 60% on the overall performance of the *larger* case study. Thus, reducing the execution time of this function will increase the overall performance of SE.

1 http://www.drdobbs.com/architecture-and-design/parallel-data-structures/231601211

# 5

# IMPROVING THE PERFORMANCE OF THE SURFACE EVOLVER

*Previous chapters introduced the problem of the liquid surfaces modelling and described the software used by Bosch as well as two case studies. Being the main goal of this dissertation the performance improvement of the SE, the next step was to research the necessary parallel computing background and the profiling of the software to better understand the bottleneck performancing critical regions. The most computational heavy function, the calc_energy is used to compute the total energy of the configuration and it has, in the* larger *case study, a workload of 60%. This chapter presents an alternative to the web data structure in SE and other techniques to improve the performance of this function, which will improve the overall efficiency of the SE.*

## 5.1 TOTAL ENERGY COMPUTATION OF THE SURFACE EVOLVER

The function that computes the total energy of the configuration, the *calc_energy* function, has a workload of 60% of the total execution time in the *larger* case study, as observed in figure 12. This function obtains the total energy configuration and has several computations over all *facets*, *edges* and *bodies*.

The *calc_energy* function starts with a series of initializations and verifications, followed by a computing process over three types of elements:

- *For all Facets*: a computing process that computes the energy of each *facet* in the configuration. It is a *for* loop that iterates over all the elements of the *facet* type.
- *For all Edges*: Similar to the previous, this computing process also iterates over all elements of the type *edge*, computing the energy for each one. However, this part of the computing process has a constraint evaluation, which is done by adding a critical region, processing each edge at a time.
- *For all Bodies*: After the energy for each element that contributes to the total energy configuration is computed, this final computing process computes the energy of the *body* and reduces with a sum of all *bodies*, thus obtaining the final value.

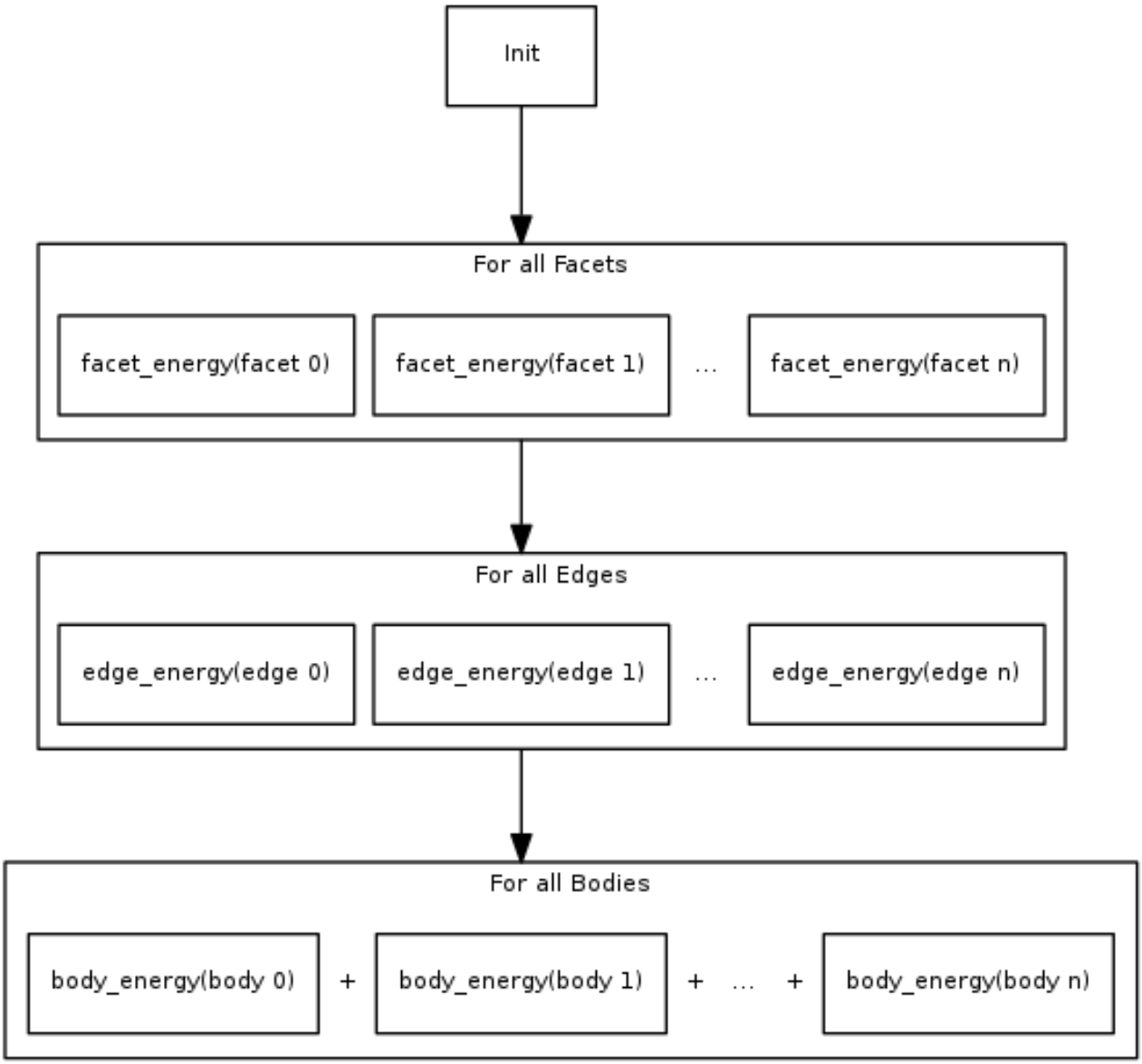



Figure 22.: Main flow of the calc_energy function with computations over all the elements of a specific type

These *For all ...* computing processes, iterate over all elements, performing a computation on each of the elements: they are parallelizable. However, in some cases, synchronization and communication between different elements are required which serializes that part of the process and reduces the efficiency of a parallel implementation.

The *calc_energy* function, has 3 main iterations over 3 element types with an impact of 60% for the *larger* case study, mainly due to an intensive memory access, mostly in the 3 iterations over all elements in a linked list as well as a lack of vectorized code and a parallel implementation.

To improve the performance of SE through the optimization of one of the heaviest computations, the *calc_energy* function, we propose an alternative data structure, that can take advantage

of vector operations on commodity CPU devices (such as x86 architecture), with improved memory locality, as well as a parallel implementation of this function.

## 5.2 AN ALTERNATIVE DATA STRUCTURE

The proposal of an alternative data structure to SE, focusing on performance, aims to replace the existing data structure to store the different types of elements, which is currently implemented with linked lists, with the following goals:

- *Contiguous memory allocation*: as observed, one of the major bottlenecks in the SE performance involves the maintenance of linked lists (functions like *get_facet_verts* or *get_fe_edge* are responsible for more than 20% of the overall performance). These functions have intensive searches of elements, which are not stored in contiguous memory addresses, thus increasing the number of memory accesses resulting in a high number of cache misses. The current data structure implementation is not taking advantage of spatial locality. This suggests that this new data structure must store the elements (*facets*, *vertices*, *edges*, *facetedges* and *bodies*) contiguously in the memory.

- *Vectorization*: one of the potential techniques for increasing the performance is to take advantage of vectorization capability of current commodity processors (SSE, AVX) through SIMD computations. To enable the vectorization capability, the elements of the array must be contiguously accessed (or with a fixed offset). That is, a loop must maintain the same offset when iterating over the elements necessary for the computation. With linked lists, this is not possible due to the dynamic computation of the address for each element.

- *Complexity*: accessing a particular element of a linked list has a complexity of $O(N)$ in the worst case. That is, searching for an element, in the worst case, forces to iterate over all the elements of the list and, in the best case, a complexity of $O(1)$, having an average complexity of $O(N/2)$. A data structure with direct mapping between the *id* of the element and its address always has a search complexity of $O(1)$.

- *Parallelism*: A linked list has several problems, as explained in the last chapter. Thus, a new data structure must also be optimized to execute in a parallel implementation, not only in a shared memory paradigm but also in heterogeneous environments such as those presented in chapter 3 with computing accelerators like GPUs and the Intel Xeon Phi, where memory accesses and data communication are even more penalized.

### 5.2.1 *Implementation*

The proposal for the new data structure is based on arrays with an array of auxiliary indices. This way the mentioned points are implemented:

- *Contiguous space*: arrays store their elements in contiguous memory spaces.
- *Vectorization*: since the spaces are contiguous, the offset will also be constant in the operands, making it possible to vectorize some of the computing processes over all the elements.
- *Complexity*: an array of elements have their *id* mapped directly to the position where they are stored in the array. However, SE might not actually remove an element from the linked list and instead mark it as *deleted*. That way, if an element needs to be reused after its removal, it is not created again, just unmarked as *deleted*. As one of the objectives is to enable vectorization, the array cannot have elements that are with the *deleted* tag because a verification will break the vectorized computation. Since it is not possible before the loop execution to verify if an element is marked as *deleted*, in this data structure the element must be removed from the array, making the vectorization to be possible. However, simply remove the element will leave that position as empty, which also prevents vectorization. To correct this, it was added an index array that maps each *id* to the element index in the array, maintaining a complexity of $O(1)$ using the index array to find an element, and a second array, the actual elements array which is used in the loop computations throughout the SE.
- Parallelism: arrays are excellent structures for adding parallelism, however, problems of inefficiency may arise due to false sharing which can be solved by padding to a line of cache (typically 64 bytes).

This new data structure adds these new maintenance operations:

- *init*: initializes a new structure (allocates space for an initial number of elements)
- *reset (l)*: the structure *l* is reset: (releases the used space and initializes a new structure)
- *set (key, value)*: adds or updates an element *value*, accessible through a *key*, which in this case is always the *id* of the element.
- *unset (key)*: removes an element via its *key* (its *id* in the SE).

This data structure is built not just to store elements in the SE, but also to store other types of elements.

### 5.2.2 *Maintenance*

In the SE project, the structures are defined in the respective *ds.h* and *ds.c* header file. They are then used to store all the *facets*, *vertices*, *edges* and *bodies* in the *storage.c* file. The headers *ds.h* are included in the SE header file *include.h* which in turn is included in all files of the project, making available to use all the functions of the new data structure (*init*, *reset*, *set*, *unset*) throughout the project.

### 5.2.3 *Usage*

To exemplify this process, the structure can be demonstrated storing integer values for better understanding:

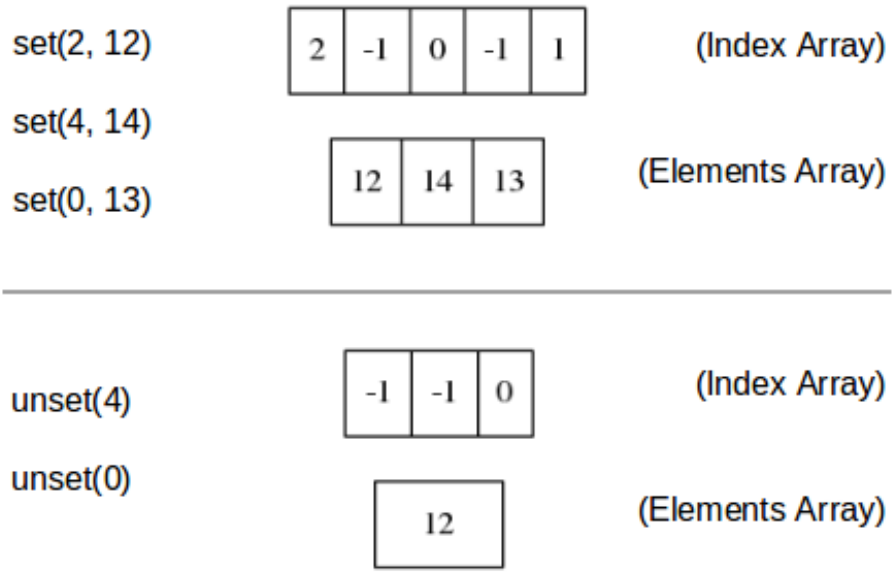



Figure 23.: Data structure usage example, using integer values as elements

The array, using the function *init* to initialize, has 3 *set* operations that define the elements of the data structure. In this example, the elements 12, 14 and 13 were defined with the keys 2, 4 and 0 respectively as show in the figure 23. As the elements with keys 1 and 3 have not been defined, these are marked with -1 in the index array. The elements that have been defined have their indices in the index array pointing to the respective position where it is stored in the elements array. For example, the element with the key 2 is obviously in the index array at the position 2, which has the value 0 stored, corresponding to the position in the elements array where the value is, in this case it has stored the value 12 in that position.

The reverse operation, *unset*, only receives the *key* to remove. Note that to save space, the index array is cut to the right until it finds an element different from the value -1.

To enable the usage in the *calc_energy* function and in other functions, all the functions and macros that creates or removes an element (defined in *storage.h* and *storage.c*) as well as all the functions and macros that updates any attribute of an existent element (defined in *skeleton.h*, *skeleton.c* and *inline.h*) needs to be consistent with the old data structure. These changes add an overhead to the overall execution time to support both data structures. As this data structure is a proposal to replace the current data structure to store the elements, it is needed to maintain both, since this proposal is only being used in the *calc_energy* function and in the *calc_quants* function, used to compute in the *Named Quantities* mode.

## 5.3 PARALLEL IMPLEMENTATION IN SHARED MEMORY

As mentioned, one of the heaviest computational functions is the *calc_energy*. This function was chosen to measure the improvement of the previously suggested data structure as well as other high performance computing techniques such as vectorization, improving the memory locality and parallel computing. Thus, if these suggestions improve this function and therefore improve the overall performance of SE, they can also be adapted to the entire software. Not just other computational heavy functions (pressure, forces and other computations) but also in the commands/datafile parsing and even the graphical user interface. The implementation of the *calc_energy* parallel version with OpenMP allows to maintain the data organization of the *web* as well as easily specifying if the software should be compiled sequential (without any OpenMP overhead) or parallel in compile-time, just by including the -openmp directive to the compiler.

### 5.3.1 *Data partitioning*

As observed in the chapter 5.1, the *calc_energy* function has 3 main parallelizable computations:

- *For all Facets*: This is the most computational heavy part of the function. It is completely parallelizable without any dependencies. It is a reduction loop, which computes the total energy of a set of facets, by computing the energy of each individual facet, finishing by adding each one.
- *For all Edges*: Similar to the previous, it is also a parallelizable loop, however, for each edge it is evaluated all the constraints of the edge. This evaluation is done with a critical

region, which means that only one edge is evaluated at a time, thus decreasing the performance of the parallel computation.

- *For all Bodies*: After the energy computation of each body, the total energy it is computed by adding the energy of each body. However, it is difficult to parallelize this loop due to the low number of bodies that normally compose a surface. Even the *larger* case study, only has 176 bodies, which is low to pay-off the thread creation overhead.

Thus, these *For all ...* loops can be implemented in parallel by assigning each thread a chunk of the respective data set involved in the computation. In the *For all Facets* loop, for instance, each thread is assigned with a subset of facets, each thread computing the energy of each facet and then adding in a reduce pattern the total energy of all the facets. The same methodology is also used for both *For all Edges* and *For all Bodies* loops, assigning to each thread a subset of edges and bodies respectively.

### 5.3.2 *False Sharing*

After the parallel implementation with OpenMP, the results observed showed almost no speedups. After an analysis of the miss rate inside the parallel region, it was noticed that the miss rate increased almost 30% with a tendency to grow as the number of threads increase. The resulting high cache miss rate is a cause of concern, since it can significantly limit the performance of multiprocessors causing a false sharing scenario [23].

Since in Symmetric Multiprocessor (SMP) systems, each processor has a local cache, the memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which drops the overall performance [24].

As suggested by the Intel Guide for Developing Multithreaded Applications [25], a padding was added to the elements data structures, to complement the size of a typical cache line which is 64 bytes. This avoids the need to update a value in the cache that might be used by another thread as shown in listing 5. Besides the manual padding of the structure, this can also be ensured by specifying the alignment with compiler directives as also shown in listing 5. It not just prevents a false sharing scenario but it is also helpful when dealing with vectorization as shown ahead. Different compilers has different ways to specify this directives. In the GNU compiler, it can be specified before the structure definition with an aligned attribute: `__attribute__ ((aligned(x)))`. In the Intel compiler it is specified at the end of the structure definition with a declspec align - `__declspec(align(x))`, where in both cases, x is the alignment size, in this case a cache line which is typically 64 bytes.

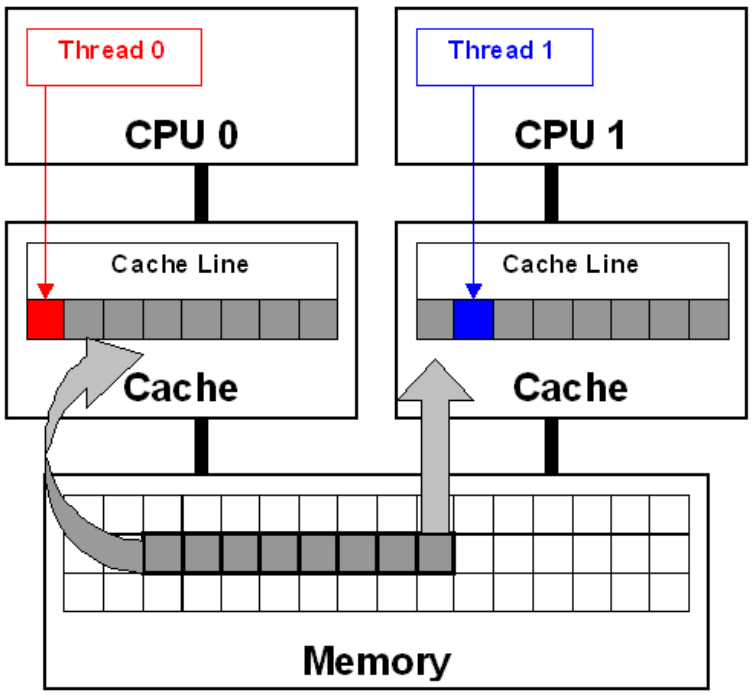



Figure 24.: Falsh sharing example where threads 0 and 1 require variables that are adjacent in memory and reside on the same cache line. Even though the threads modify different variables, the cache line is invalidated, forcing a memory update to maintain coherency.

```
#define CACHE_LINE 64

#ifdef __INTEL_COMPILER
#define _IF_INTEL_ALIGN(x) __declspec(align(x))
#define _IF_GNU_ALIGN(x)
#else
#ifdef __GNUC__
#define _IF_INTEL_ALIGN(x)
#define _IF_GNU_ALIGN(x) __attribute__ ((aligned(x)))
#endif
#endif

struct _IF_INTEL_ALIGN(CACHE_LINE) facet
{
    COMMON_STUFF          /* 32 bytes */
    REAL  density;        /* 8 bytes */
    REAL  area;           /* 8 bytes */
    facetedge_id fe_id;   /* 4 bytes */
    short color;          /* 2 bytes */
    short backcolor;      /* 2 bytes */

    int padding[2];       /* 8 bytes */

} _IF_GNU_ALIGN(CACHE_LINE);
```

Listing 5: Facet structure padded to a cache line size of 64 bytes. The structure is also aligned to 64 bytes with a compiler directive to both GNU and Intel compilers

### 5.3.3 *Parallelism overhead*

Performing a computation in parallel, always inserts some overhead to the software. In a shared memory paradigm, using OpenMP in this case, both fixed startup and per-thread overhead are added to the software. In some cases, these overheads are high enough to make it worthless to parallelize some determined region [26]. Some of the factors that increase the OpenMP overhead are:

- OpenMP startup overhead: when the software starts, there is an additional overhead to initialize the library. It is just noticeable at the software startup and it is not significant for most software.
- Thread startup overhead: the overhead to create the worker threads. Since OpenMP reuses a pool of threads, it is also a one-time cost;
- Per-thread overhead: in every parallel region, for instance in a parallel for, there is an overhead to assign to each worker thread a chunk of data to compute. This overhead is noticeable in every start of a parallel region of the software;
- Lock management overhead: the overhead spent to manage blocks on critical regions. In the *For all Edges* loop of the SE, each edge must validate a set of constraints which must be done an edge at a time, not just preventing parallelism but also adding a synchronization overhead.

In the SE, the *smaller* case study starts with just 10K elements which makes the overhead to be more significant than in the *larger* case study which starts with 500K elements.

## 5.4 PERFORMANCE OPTIMIZATION DISCUSSION

Subsection below presents the results of the new shared memory implementation with OpenMP, both in the normal and *Named Quantities* modes as well as using both the *smaller* and *larger* case studies. These results also include the alternative data structure described above and other optimizations such as scheduling, vectorization and software prefetching, as described ahead.

### 5.4.1 *Results*

*Smaller case study*

In the smaller case study, as mentioned above, the *calc_energy* function has a workload of 24% and it has 3 main parallel regions:

- *For all Facets*: a computing process that computes the energy of each *facet* in the configuration. This region has a workload of 60% of the total energy computation.

- *For all Edges*: Similar to the previous, this computing process iterates over all the *edges*. However, this part of the computing process has a constraint evaluation, which is done by adding a critical region, processing each edge almost one at a time and it has a workload of 45%.

- *For all Bodies*: This final computing process computes the energy of each *body* and reduces with a sum of all energies, thus obtaining the final value. However, since the *smaller* case study only has a single body, parallelization only causes overhead of the new thread creation. In fact, it was decided not to parallelize regions with less then approximately 1000 bodies since the overhead does not pay-off. This computation and the final updating region have a workload of 5%.

Besides the *calc_energy* function, it was also parallelized the *vertex_average* function (with a 14% workload in the *smaller* case study) and the *calc_force* (with a 7% workload also in the *smaller* case study) [12].

#### WITHOUT NAMED QUANTITIES

The first analysis uses the normal mode to compute the total energy of the configuration. As observed, the *For all Facets* is the most significant region of the computation as well as the best candidate to parallelize due to not having a need for synchronization, unlike the *For all Edges* region.

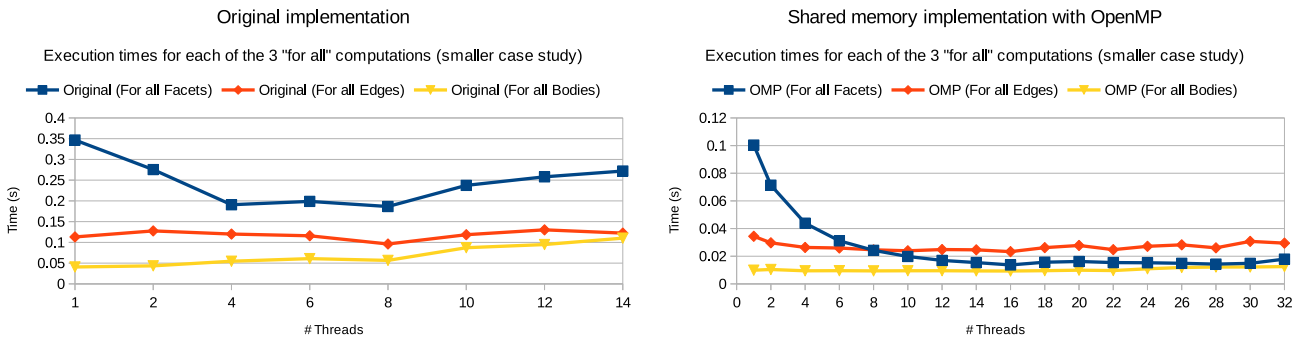



Figure 25.: Comparing the execution times of the 3 ”For All ...” loops of the total energy computation for the original and the new shared memory implementation with OpenMP

In the figure 25 it is shown the results of the new parallel implementation with OpenMP, as well as other performance improvements:

- An alternative data structure used to compute the total energy of the configuration as described above;
- A vectorization, false sharing, scheduling and software prefetching improvement by modifying some computations and with the help of the new memory organization in the alternative data structure as described ahead;

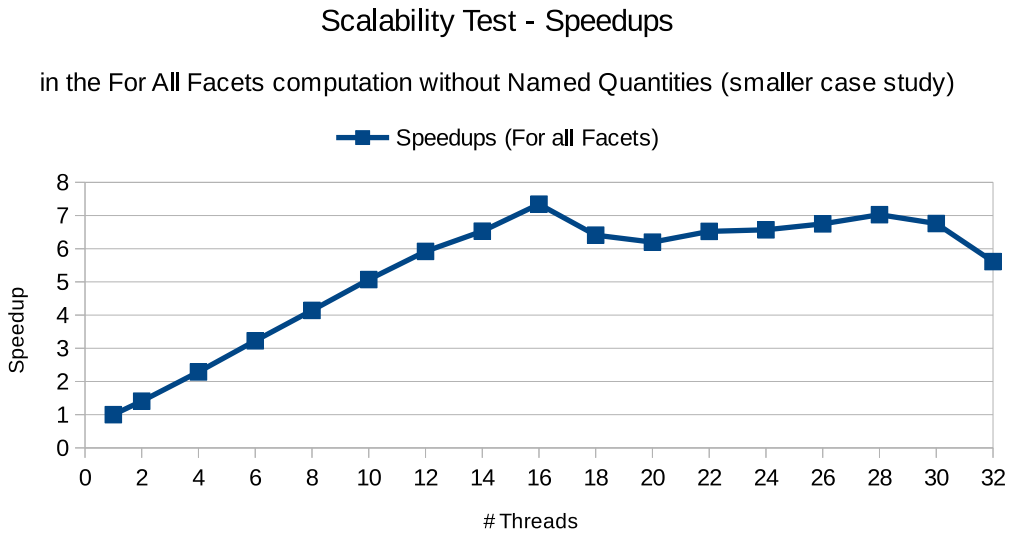



Figure 26.: Speedups of the facets energy computation, the heavier part of the total energy computation

These results show that the *For all Facets* is in fact more suitable of parallelizing due to not having any type of synchronization, not like the *For all Edges* that is almost sequential. The *For all Bodies* does not have any type of parallelism. The speedups of the *For all Facets* region, in the figure 26, shows an improvement over the original version.

It also shows that the hardware support for multithreading does not help in these types of problems. Since each thread uses the same units, which are shared, they do not benefit of this technology as it can be observed in the figures 25, 26 and 27. As demonstrated in the experimental setup, the node has 2 devices, each with 8 cores plus *Hyperthreading*. So after the 16th thread, the results show no improvement and in some cases are even worst.

Figure 27 shows the new execution times of the SE with the new performance optimizations as mentioned. As shown, the new data structure, as well as the more efficient parallel implemen-

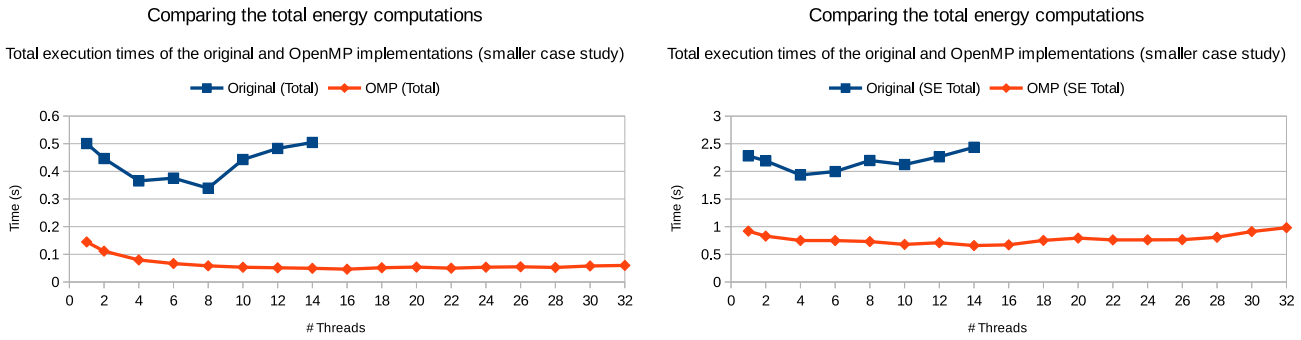



Figure 27.: Comparing the total execution times of the energy computation (at the left) and the total execution time of the SE (at the right) for the original and the new shared memory implementation with OpenMP

tation with OpenMP and other improvements, reduce the execution times more than 2x in the smaller case study of the SE.

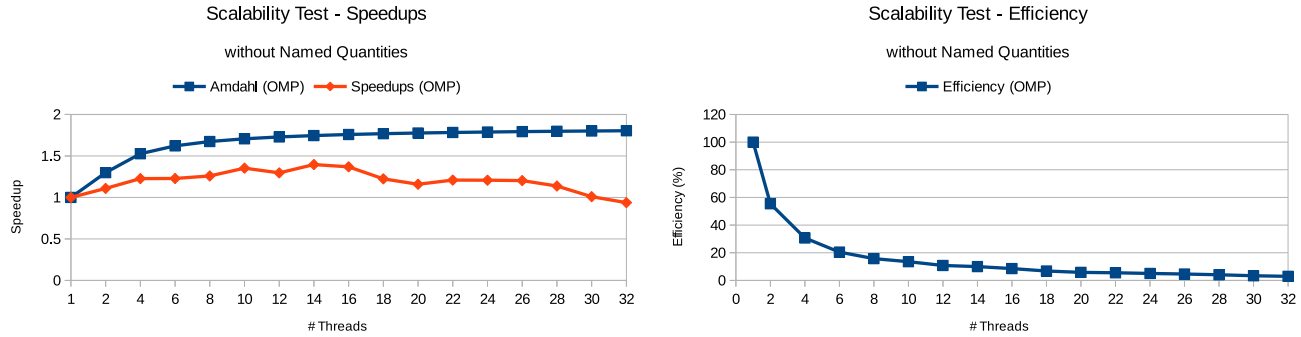



Figure 28.: Speedups and efficiency of the total energy computation (at the left) and of the SE with the energy computation parallelized (at the right)

The speedups shown in figure 28 have the Amdahl's law stating the maximum attainable speedups by just parallelizing the regions that were mentioned (*calc_energy*, *calc_force* and *vertex_average*). This way it is demonstrated the improvement with just these 3 functions and not all the parallelizable regions like in the chapter 4.3. The Amdahl's law is calculated as shown in the appendix A.2 with a B=0.55 (sequential weight of the algorithm). This value was obtained by subtracting the weight of the 3 parallelized regions (*calc_energy*=24%, *calc_force*=7% and *vertex_average*=14% which is 0.45) to 1. This shows that although the performance was improved with these optimizations, it can still be better with an even deeper analysis of the current algorithm, specially in the energy computation for all the edges, in order to remove the critical region of the code that is lowering the performance of the total energy computation in the SE.

WITH NAMED QUANTITIES

The *Named Quantities* mode uses a set of methods as a way of computing a scalar value from some particular type of element (*vertex*, *edge*, *facet*, *body*). However, this more schematic mode of computing these values adds an extra work that lowers the performance but increases the ease of adding new methods to the SE.


Figure 29.: Comparing the total execution times of the energy computation (at the left) and the total execution time of the SE (at the right) for the original and the new shared memory implementation with OpenMP with the *Named Quantities* method

The new implementation, with all the optimizations mentioned in the previous section, as shown in figure 29, improves the total energy computation. In fact, the new SE has a performance more than 2x efficient than the original implementation but is just by optimizing the data structure since the additional overhead to parallelize does not pay-off using *Named Quantities* with the *smaller* case study.

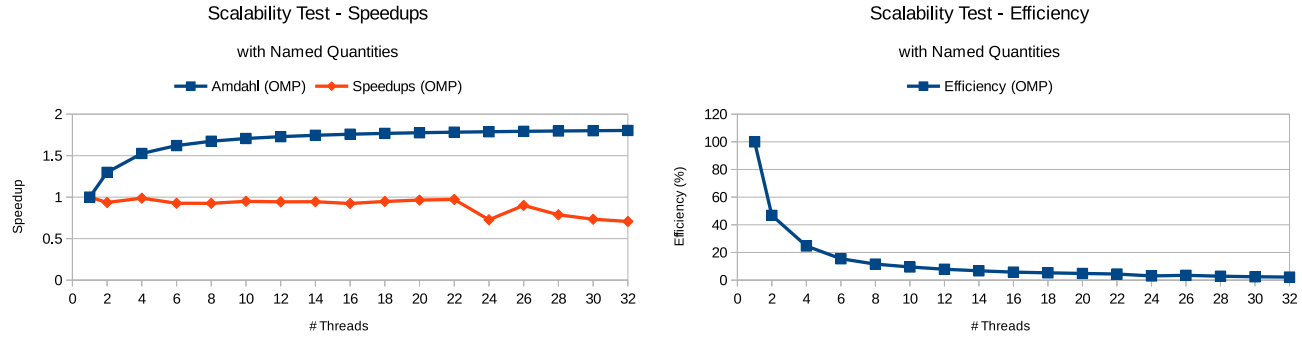



Figure 30.: Speedups and efficiency of the total energy computation (at the left) and of the SE with the energy computation parallelized (at the right) with *Named Quantities*

In the figure 30, the speedups and efficiency are calculated in the same way as in the normal mode and show that it does not scale.

*Larger case study*

In the larger case study, as mentioned above, the workload of the *calc_energy* function is 60% of the total SE execution time and each parallel region has a workload of:

- *For all Facets*: This region has a workload of 80% of the total energy computation;
- *For all Edges*: This region has a workload of 18% of the total energy computation, however this is almost sequential due to the critical region;
- *For all Bodies*: The *larger* case study only has 176 bodies which still does not feasible the parallelization overhead for this region. It has a workload of 2%.

WITHOUT NAMED QUANTITIES

The first analysis, like in the *smaller* case study, uses the normal mode to compute the total energy of the configuration. In this case study, the *For all Facets* is also the most significant part of the computation as well as the best candidate to parallelize.

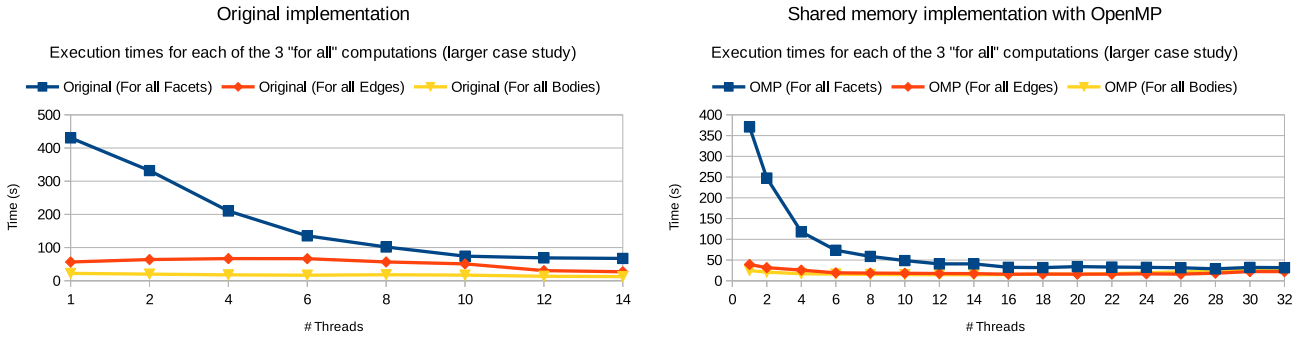



Figure 31.: Comparing the execution times of the 3 ”For All ...” loops of the total energy computation for the original and the new shared memory implementation with OpenMP

The results in figure 31 also shows, as in the *smaller* case study, that the *For all Edges* region suffers from the critical region which prevents it to scale. Also, for the *For all Bodies* the parallelization is not being used since it only has 176 bodies.

However, the *For all Facets* is a region without any synchronization and with the false sharing problem solved by the alternative data structure as described in the chapter 5.3.2, this region shows significant speedups in figure 32.

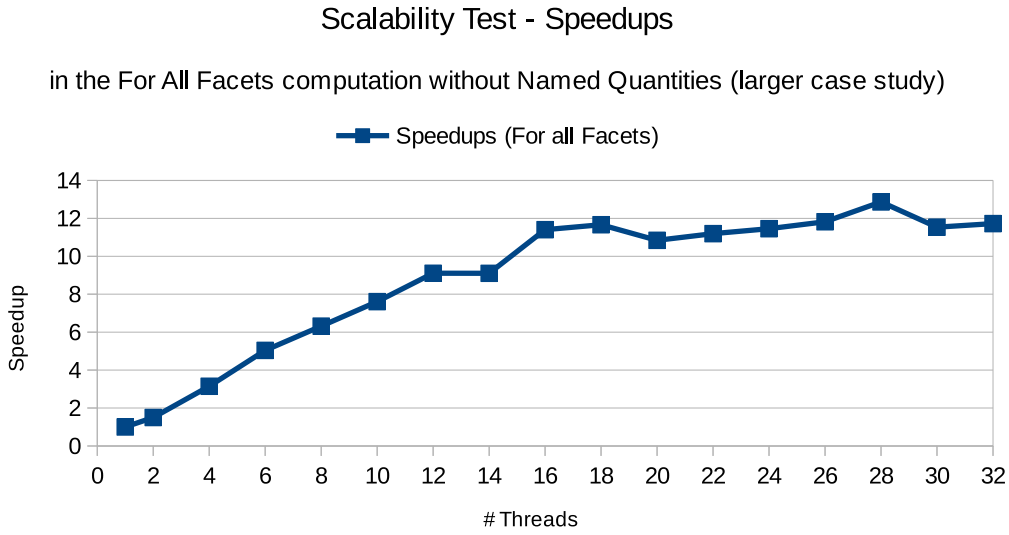



Figure 32.: Speedups of the facets energy computation, the heavier part of the total energy computation

It also shows, as in the *smaller* case study, that the hardware support for multithreading does not help in these types of problems. Also, after the 16th thread, the results show almost no improvement and in some cases are even worst.


Figure 33.: Comparing the total execution times of the energy computation (at the left) and the total execution time of the SE (at the right) for the original and the new shared memory implementation with OpenMP

Figure 33 shows the new execution times of the SE with the new performance optimizations as mentioned. The improvement is not as significant as the *smaller* case study because one of the main problems with the original parallel implementation is the thread creation overhead for smaller problems. This overhead for larger problems is reduced, however it still shows a significant improvement over the original implementation.

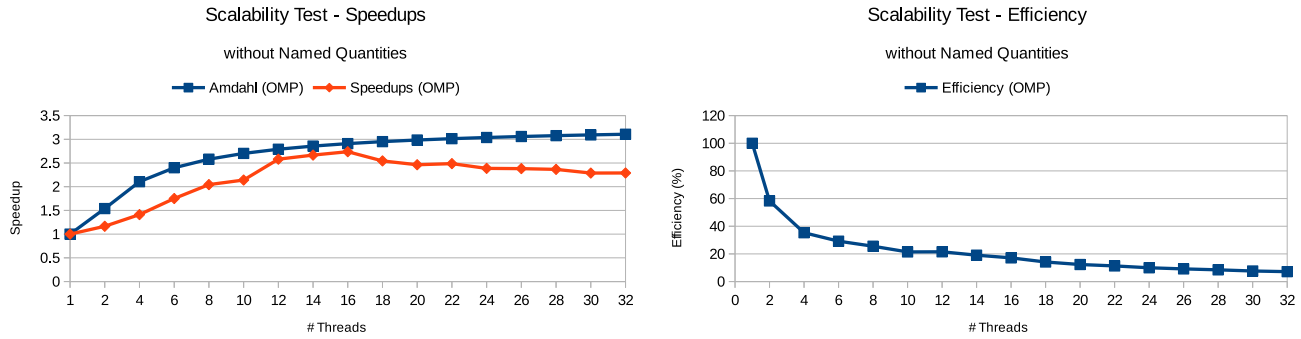



Figure 34.: Speedups and efficiency of the total energy computation (at the left) and of the SE with the energy computation parallelized (at the right)

The speedups shown in figure 34 also have the Amdahl's law stating the maximum attainable speedups by just parallelizing the regions that were mentioned (*calc_energy*, *calc_force* and *vertex_average*). The Amdahl's law is calculated as shown in the appendix A.2 with a B=0.32 (sequential weight of the algorithm). This value was obtained by subtracting the weight of the 3 parallelized regions (*calc_energy*=60%, *calc_force*=1% and *vertex_average*=7% which is 0.68) to 1. Due to having more workload per thread, the impact overhead is significantly reduced when comparing to the *smaller* case study, approaching more the Amdahl's curve, specially for 14 and 16 threads. This also shows that the *For all Edges* is preventing a higher performance of the energy computation, but not as much as in the *smaller* case study where the relative workload is higher.

### WITH NAMED QUANTITIES

The *Named Quantities* mode uses a set of methods as observed ahead. This mode does not have the overhead as the normal mode so the improvement observed is only due to the alternative data structure and not the new shared memory implementation.

This shows, in figure 35, an improvement on the total energy computation but is not significant on the overall performance of the SE.


Figure 35.: Comparing the total execution times of the energy computation (at the left) and the total execution time of the SE (at the right) for the original and the new shared memory implementation with OpenMP with the Named Quantities method

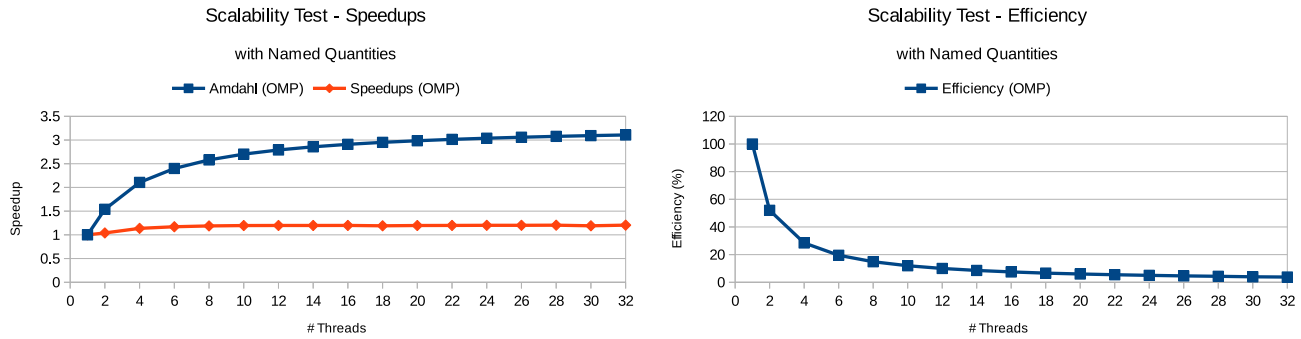



Figure 36.: Speedups and efficiency of the total energy computation (at the left) and of the SE with the energy computation parallelized (at the right) with Named Quantities

In the figure 36, the speedups and efficiency are calculated in the same way as in the normal mode and shows that it does not scale.

It is important to note that the *Named Quantities* mode was not much explored since it was not part of the initial objectives. One of the advantages of using the *Named Quantities* is the ability to compute more than one quantity in parallel (for instance, computing parallel energies and forces), which was not necessary in these case studies. The *Named Quantities* mode can be useful and even scale more than the normal mode in such cases and also adds an easier way to define new methods for calculating quantities.

### 5.4.2 *Scheduling*

A scheduler is a mechanism that assigns a subset of the workload to the parallel workers. In OpenMP, specially in parallel loops, a loop scheduler assigns a specific subset of the iterations to different threads. The default scheduling mechanism assumes an equal load balance which is frequently noticed in some problems [27]. That is, the same static amount of iterations is assigned to each thread.

However, in some cases this causes a load imbalance. Some of the threads are assigned with more work than other threads and the algorithm does not process until the last thread finishes while the other threads are doing nothing as they finished earlier.

Since the proposal of OpenMP [28], it was suggested to include in the default implementation of OpenMP different types of schedulers to deal with these balancing problems. Currently, there are 5 different loop scheduling types in OpenMP [29]:

- *Static*: This type of scheduler divides the loop into chunks of equal size or as equal as possible in the case where the number of iterations is not divisible by the number of threads. The chunk size can be defined and by default is the loop count divided by the number of threads. If set to 1, it interleaves the iterations by the amount of threads;
- *Dynamic*: The dynamic scheduler uses an internal work queue to give a chunk of the loop iterations to each thread. However, when a thread is finished it receives more work from the top of the work queue. By default the chunk size is 1;
- *Guided*: The guided scheduler is similar to the dynamic scheduler but the chunk size starts off large and decreases to better handle imbalance between the iterations;
- *Auto*: This special type of scheduler delegates the scheduling decisions to the compiler;
- *Runtime*: Runtime is not a kind of scheduler. It is a way of define what type of scheduler to use just by defining an environment variable named *OMP_SCHEDULE*. It is useful for problems where a particular dataset or number of threads change the performance with a specific scheduler so the user can decide without recompiling the code, which one to use just by defining the variable;

Using a non-static scheduler adds more overhead to the OpenMP implementation of a software. Dealing with schedulers increases the workload to manage the work queues and assigning data to each thread so, in some cases, even with a small load imbalance problem, assigning a non-static scheduler could not increase the performance of the software.

To measure the OpenMP different types of scheduling influence in the SE, it was used the runtime variable *OMP_SCHEDULE* as mentioned above, to assign a different scheduler and measure its performance in the function *calc_energy* to compute the total energy of the configuration.

The figure 37 shows that up to 16th threads there are no significant differences for the *smaller* case study. In fact, the work to compute each subset the *For all ...* loop is not much different so there are not a real load balancing problem. After the 16th thread, the behaviour is different due to *Hyperthreading*, where a thread might be preempted, and with a smaller case study this affects significantly the performance of each scheduler.

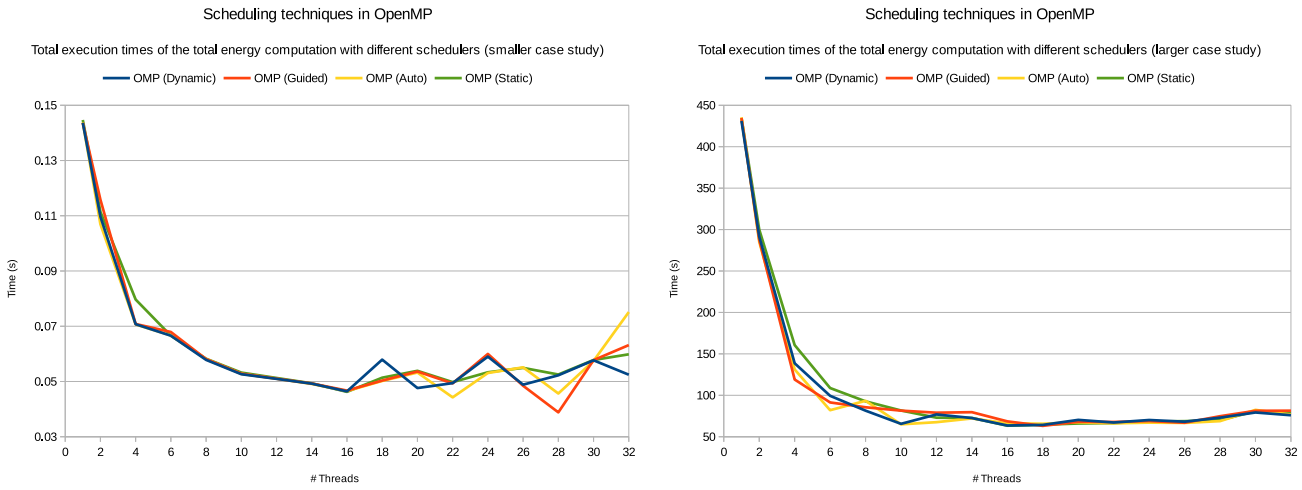



Figure 37.: Comparing the performance of the different types of schedulers in the *calc_energy* function with the *smaller* case study (at the left) and the *larger* case study (at the right)

In the larger case study, due to the increasing number of computations, each thread have more work than another, depending on the number of elements that each facet, edge or body, in the subset, is connected. In the *smaller* case study, since this number is not high, the load imbalance is not noted, however in the *larger* case study it can be observed that the static has a worst performance, not with 1 thread where the overhead is smaller or even with 2 threads where the subsets are almost equal, but thereafter where each subset is smaller and the imbalance is more noticeable.

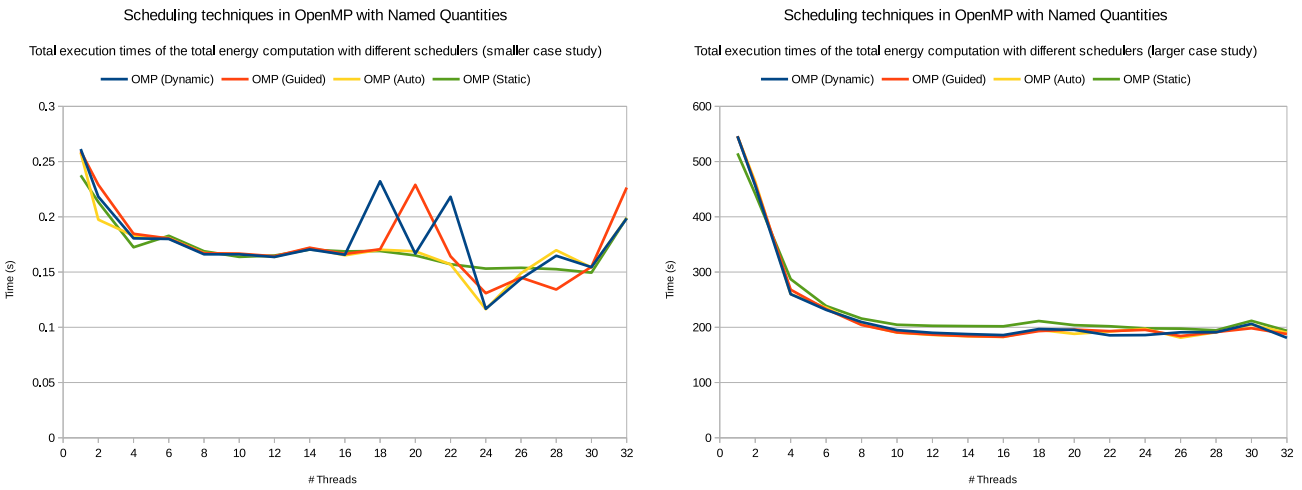



Figure 38.: Comparing the performance of the different types of schedulers in the *calc_energy* function with the *smaller* case study (at the left) and the *larger* case study (at the right) with *Named Quantities*

In the *Named Quantities* mode it is noted almost the same, with a smaller difference in the non-static scheduling types as shown in figure 38. Unlike the normal mode, where for some number of threads, there is a better performance for a particular scheduler.

### 5.4.3 *Vectorization*

As detailed in the chapter 5.2 referring to the alternative data structure as well as in the false sharing chapter 5.3.2, the total energy computation was optimized in order to increase the number of SIMD computations. The alternative data structure, now based on arrays has its elements contiguously stored in memory, and with an alignment as defined in the chapter 5.3.2 makes it possible to automatically vectorize some of the loops in the total energy computation.

```
LOOP BEGIN at filml.c(214,9)
<Multiversioned v2>
   remark #15388: vectorization support: reference x has aligned access
   remark #15388: vectorization support: reference x2 has aligned access
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 10
   remark #15477: vector loop cost: 1.500
   remark #15478: estimated potential speedup: 6.230
   remark #15479: lightweight vector operations: 2
   remark #15480: medium-overhead vector operations: 1
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Figure 39.: Vectorization report generated by the Intel C Compiler with a level 5 vectorization report

The Intel C Compiler allows to generate a report based on a level of detail with a set of informations of which loops were vectorized and a justification as well as tips on loops that were not vectorized [7].

As shown in figure 39, one of the loops that was not vectorized, is now vectorized as well as other loops that benefit from the alternative data structure and data alignment. In the vectorization report is described the variables involved in the operation, *x* and *x2* in this case, showing that both have aligned access. Another relevant information is the scalar and vector costs and the resulting estimated potential speedup that in this case is 6.230.

To measure the vectorization impact, it was disabled the automatic vectorization just for the *calc_energy* function and measured the performance of that function that computes the total

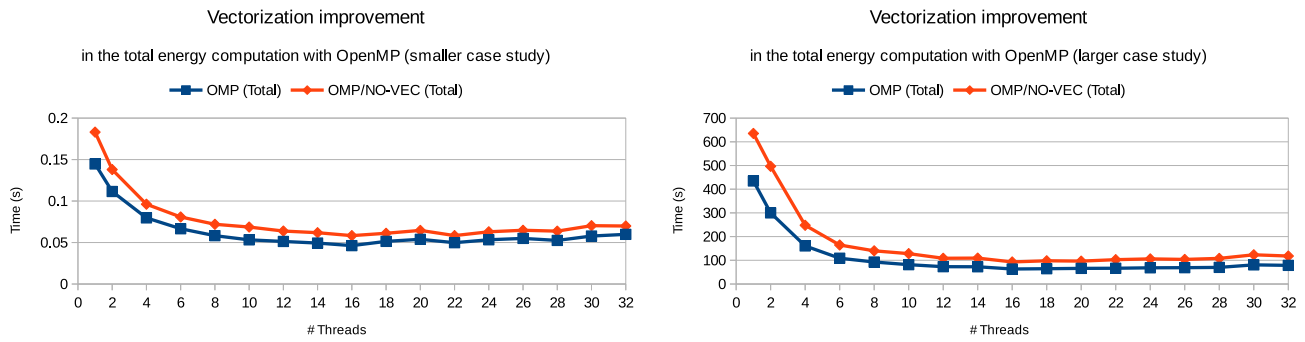



Figure 40.: Comparing the vectorization performance in the *calc_energy* function with the *smaller* case study (at the left) and the *larger* case study (at the right)

energy of the configuration as shown in figure 40. This shows that in the normal mode, removing the vectorization from the function, drops the performance in more than 30% in some cases.

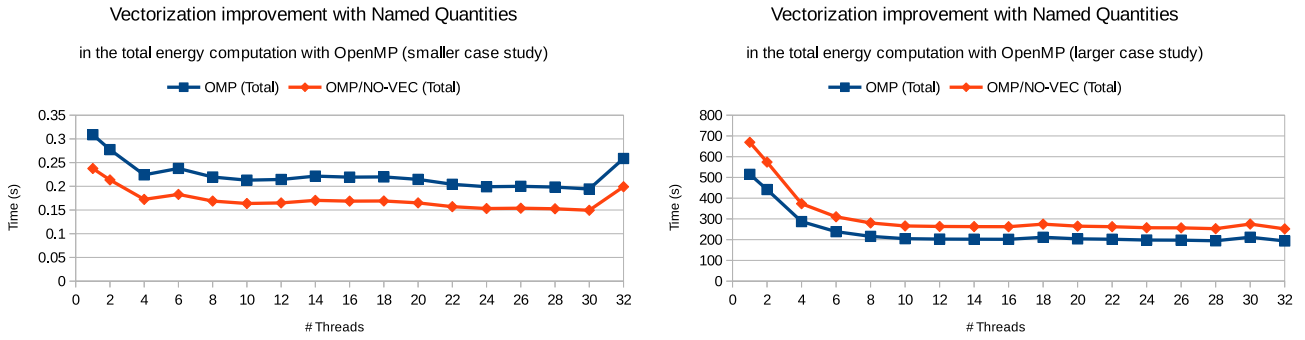



Figure 41.: Comparing the vectorization performance in the *calc_energy* function with the *smaller* case study (at the left) and the *larger* case study (at the right) with *Named Quantities*

Also, for the *Named Quantities* mode shown in figure 41, the difference tends to be even higher as there are more loops in this mode.

### 5.4.4 *Software Prefetching*

The alternative data structure, as the results show, increases the performance of computing the total energy of the configuration. By having contiguous data allocation, with a fixed stride, enables the ability to vectorize loops but it also has influence on the data locality. However, this alternative data structure just stores the elements of the surface and all the other information still uses the linked list and other data structures of the SE. Some of this information is still needed inside the *calc_energy* function and it is lowering the performance of this function. One of the

explored techniques to reduce the impact of these data accesses to the linked list among others, is to explicitly load the elements to the cache through software prefetching.

An array, unlike a linked list and depending on the memory access pattern might have the element used in the following iteration using an hardware prefetch capability. An hardware prefetcher operates transparently, without developer intervention and it is triggered when successive cache misses occur in the last-level cache and a stride in the access pattern is detected, such as in the case of loop iterations that access array elements [30]. This does not mean that the developer does not have to do anything, it is still needed to have knowledge of how this can be optimize, through cache friendly code.

Software prefetching and locality optimizations are techniques for overcoming the speed gap between processor and memory. However, this relies on the developer or the compiler to insert explicit prefetch instructions into the code for memory references that are likely to result in a cache miss. At runtime, the inserted prefetch instruction will bring the data into the cache of the processor, thus overlapping the memory access cost when the element is needed, hiding the memory access latency [31].

Explicitly prefetching has several benefits [32]:

- *Irregular memory accesses*: Data elements can be typically prefetch even in various kinds of irregular data structures by inserting appropriate prefetch intrinsics;
- *Cache locality hint*: Hardware prefetchers place the data in the lowest level cache of the requesting core, whereas by software prefetching data, this can be placed directly into the L1 cache;
- *Hide latency*: Asynchronously prefetching data before it is actually needed by inserting a prefetch declaration hides the memory accesses latency to load the data element;

There is also some negative impacts in software prefetching [32]:

- *Increased instruction count*: Unlike hardware prefetching, software prefetching inserts a set of instructions to load the elements from the memory;
- *Code structure change*: Specially in a loop, the position in the code where a load might occur has an impact on the performance, so there might be additional computations to measure the best place to prefetch the data element;

Detecting the right place to prefetch data is crucial to increase the performance, otherwise it can even decrease the performance by having an unnecessary memory load that is replacing data in the cache. If the prefetch occurs too early in the code it might be replacing other useful

data (cache pollution) or be replaced before being used. If it occurs too late, it might not hide the processor stall [31].

In the GCC, software prefetch can be achived by adding a prefetch built-in function as shown in listing 6:

```
/* Prefetching in GCC */

for (i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
    /* ... */
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
}
```

Listing 6: Built-in prefetch function from GCC loop example

The function receives 3 arguments[1]: the address of the memory to prefetch and two optional arguments, rw and locality. The rw argument is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The locality must be also a compile-time constant integer between 0 and 3. A value of 0 means that the data element has no temporal locality, that is, it needs not to be in the cache after the access. A value of 3 means that it has a high degree of temporal locality and should be kept in all the levels of cache as long as possible.

In the SE it was included software prefetch to load the elements from the linked list that are still needed in the total energy computation done at the calc_energy function.

Figure 42 shows the results using software prefetch to load the elements in the normal mode. It can be observed that for the *smaller* case study there are no significant improvement and for 10 threads there is even a lost of performance. This case study, due to its size, fits in the L3 cache, so prefetching only causes overhead. However, for the *larger* case study, it is noted a small but significant improvement using software prefetching. One of the main bottlenecks are in the elements access inside the linked list. Hiding the memory latency by prefetching the elements as soon as the pointer to the next element is computed reduces the cache misses.

In the Named Quantities, shown in the figure 43, the difference between the time that the pointer is computed and the time it is needed is very close, so in some cases, the element is not

1 https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

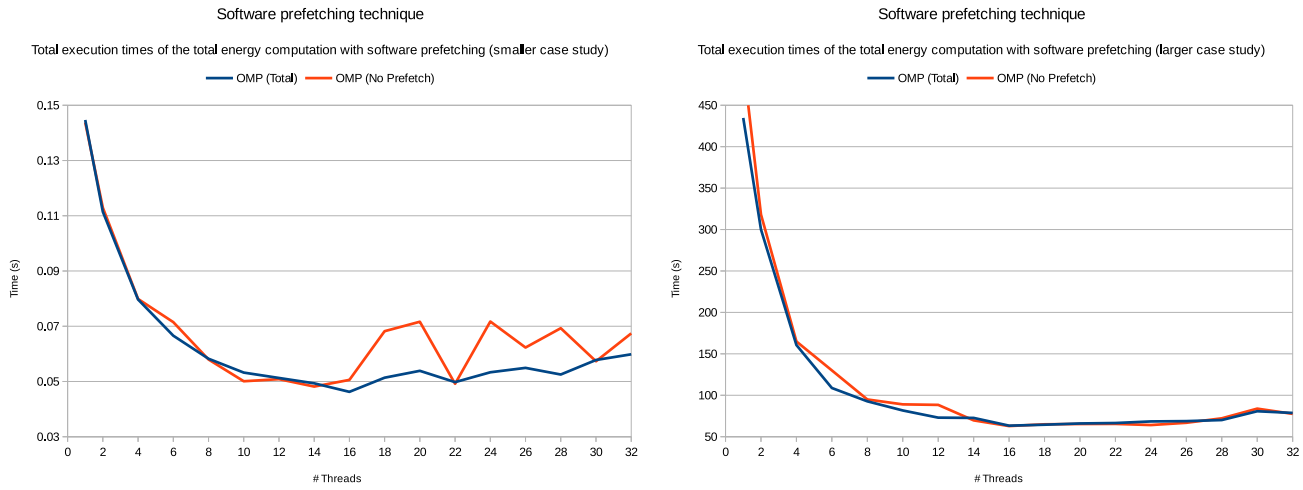



Figure 42.: Comparing the software prefetching performance in the *calc_energy* function with the *smaller* case study (at the left) and the *larger* case study (at the right)

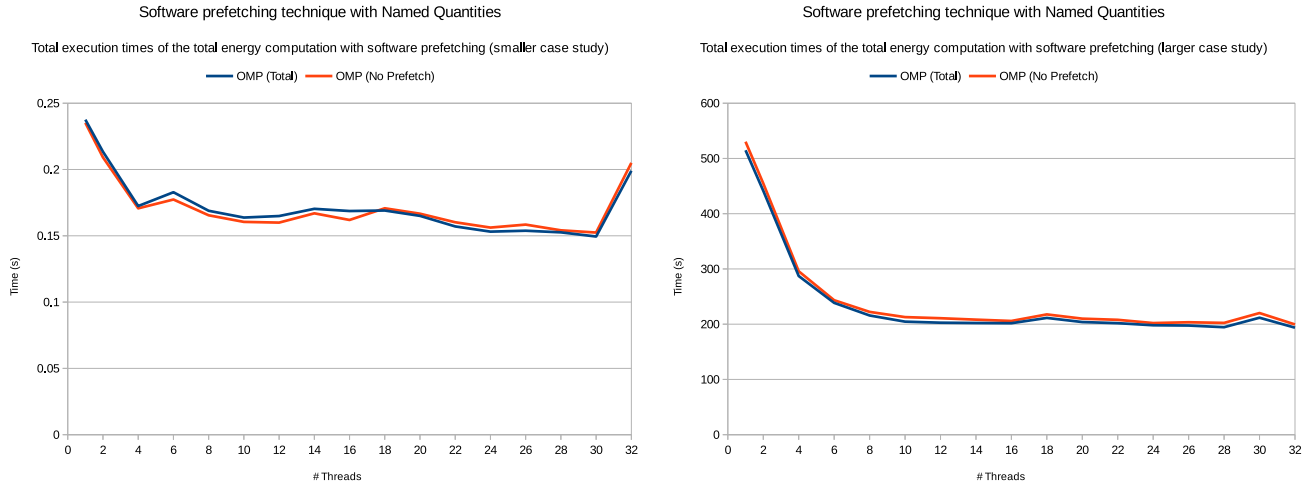



Figure 43.: Comparing the software prefetching performance in the *calc_energy* function with the *smaller* case study (at the left) and the *larger* case study (at the right) with *Named Quantities*

yet in the cache, stalling the CPU. However, with a deeper analysis, both the normal and *Named Quantities* can be improved.

In fact, this is one of the general alternatives for software using linked lists or other irregular data structure. Prefetching the elements as soon as the pointers for the next iterations are computed, a prefetch might reduces significantly the number of cache misses, thus increasing the software performance.

## SUMMARY

This chapter presented several ways to improve the computational performance of the *calc_energy* function, used to compute the total energy of the configuration. To increase memory locality and

vectorization, an alternative data structure was proposed, mainly to replace the elements storage from linked lists to arrays.

Next, it presented how to increase the performance through a parallel implementation in a shared memory environment, using OpenMP. The results show a significant increase of the performance, specially in the *smaller* case study where the parallelism overhead is more noticed. It was also explored how to increase the performance exploring the workload scheduler of OpenMP, the vectorization improvement after using the alternative data structure and finally a suggestion for software prefetching, that even thought it was not fully explored, shows a performance improvement that can be higher with a deeper analysis.

# 6

# CONCLUSIONS

This dissertation studies the liquid surfaces modelling shaped by various forces and constraints, specially the thermal fatigue failure of BTC components welded to PCBs, that leads to a premature end of an electronic component. The simulation of these procedures is done by a numerical analysis with the FEM that evolves the surface from its initial state to the state where it meets the required stop criteria.

This is a process used by universities and companies in the field of mechanical engineering, electronic engineering, mathematics, among others to study the behaviour of these surfaces when constrained by different forces and energies. The software used by Bosch to study these surfaces is the SE and it was presented with two main case studies used throughout this dissertation.

This dissertation studied the development of a modelling process to simulate the evolution of a surface, to identify problems of thermal fatigue failure in the design of new PCBs. This has a very high computational load, which is not consistent with the requirements of the industry and researchers, which required to improve the performance of the current software, the SE, to increase the complexity of the problems to be addressed and to obtain a solution in valid time.

The approach followed to increase the performance of the SE, started by studying the parallel computing background, namely the target homogeneous environment with multicore CPUs. It was noticed that NUMA systems, as well as hardware multithreading capabilities and Intel *Hyperthreading* can be useful to increase the performance with a small effort by the developer.

Vectorization proved to be very critical in achieving high performance computing software. Using SIMD allows one single instruction to execute with multiple operands, depending on the technology. Although the main focus of this dissertation was to implement an improved version to execute in a homogeneous environment, heterogeneous environments were also introduced and all the decisions regarding efficiency, were also made by taking into account these environments.

From the profiling analysis of the SE, it was noticed that the current implementation has several performance issues. By the analysis of the SE call graphs, it was shown that one of the heaviest computational functions is the *calc_energy*. This function is used to compute the total

energy of the configuration and has a workload of 24% in the *smaller* case study and 60% in *larger* case study. So, to improve the computational efficiency of the SE it was decided to use this function as prove of concept of all the decisions to increase the efficiency of the SE.

Another major bottleneck identified is the current data structure, which stores the elements using linked lists. This type of data structure, although very efficient in memory usage, it is not consistent with high performance computing so it was proposed an alternative data structure. This data structure has the main purpose to replace the linked lists implementation with an array based implementation.

The shared memory implementation uses OpenMP to implement parallelism and has several advantages over the actual version implemented with *pthreads*. As shown in the results discussion, after fixing the false sharing and other memory locality related problems, the new implementation increased the performance more than 2x in the *smaller* case study.

OpenMP also allowed to implement a scheduling study, where different mechanisms were explored. For the *smaller* case study, since all the data fits in the CPU caches, a load balancing problem is not as much reflected as in the *larger* case study. For this case, a non-static type of scheduling has advantages over the static mechanism, due to the fact that some elements take longer to compute its energies than others.

One of the major impacts on the performance comes from the alternative data structure and that is the vectorization. Having the data contiguous in memory, as well as some memory alignments, enabled the vectorization and allowed to perform several computations using SIMD. In the most notorious loop, this reflected in a speedup of 6.230 over the scalar loop.

However, the *calc_energy* function also needs to use the current data structure and not only the alternative one. This means that some of the computations are still using the elements from the linked list. To reduce the impact of these memory accesses, a software prefetching technique was also explored. The main idea to this technique was to add a prefetch directive to load asynchronously an element from the list before it was actually needed. This can be hard to do, if a load occurs to early, it might be replacing important data from the cache and it might even be replaced before it is used. If the load occurs too late, it might not yet been loaded from memory, stalling the CPU waiting for the data. Although this technique has only been implemented by the end of this dissertation and not fully explored, the results show a small but significant increase of performance, specially in the *larger* case study.

## 6.1 FUTURE WORK

At the end of this dissertation, several results specially using the alternative data structure, capable of vectorization and with improved memory locality, motivate further work to improve the performance of the SE, namely:

- Adapt the alternative data structure to all the functions of the software. This can be done with the *calc_energy* function as a guideline to implement an improved and vectorizable version of other functions.
- Reimplement the parser and GUI to also use the alternative data structure. Although more challenging than the previous point, the performance of the SE fully depends on the efficiency of these functionalities.
- One of the major bottlenecks, that is preventing the *calc_energy* function to achieve higher speedups, is the critical region in the *For all Edges* computation. Rethinking the algorithm or at least reimplement this computation to reduce the penalty of the critical region, also increase the SE performance.
- Further research on the software prefetching technique presented can also help to reduce the impact on the linked lists and other memory accesses that are penalizing the performance of the SE.
- All the decisions regarding performance, specially in the alternative data structure, were made taking into account the possibility of a future heterogeneous implementation, specially using the GPU as a computing accelerator. This dissertation also includes information about this computing device as well as libraries and frameworks that can be used to achieve a higher performance in such an environment.

# BIBLIOGRAPHY


[1] BOSCH. Optimização dos procedimentos e parâmetros de brasagem de componentes do tipo btc em pcbs. 2014.

[2] F. Teixeira-Dias, J. Pinho da Cruz, R.A. Fontes Valente, and R. J. Alves de Sousa. *Método dos Elementos Finitos - Técnicas de Simulação Numérica em Engenharia*. ETEP – Edições Técnicas e Profissionais, 2010.

[3] Kenneth A. Brakke. *The Surface Evolver Manual*. Mathematics Department, Susquehanna University, August 2013.

[4] Jack Dongarra. Homogeneous and heterogeneous computing. 1996.

[5] Laércio L. Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Abhinav Bhatele, Philippe O. A. Navaux, Jean-François Méhaut, and Laxmikant V. Kale. Improving parallel system performance with a numa-aware load balancer. 2011.

[6] Alexandra Fedorova, Margo Seltzer, Christopher Small, and Daniel Nussbaum. Performance of multithreading chip multiprocessors and implications for operating systems design. 2005.

[7] Mark Sabahi. A guide to vectorization with intel c++ compilers. 2012.

[8] Gordon E. Moore. Cramming more components onto integrated circuits. 1965.

[9] NVIDIA. Cuda c programming guide. 2013.

[10] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110 (whitepaper). 2012.

[11] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler. 2013.

[12] Bruno Araújo. *Efficient modelling of liquid surfaces on multi-core CPU and Xeon Phi devices MSc. Dissertation*. Department of Informatics, University of Minho, October 2015.

[13] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[14] David Cortesi, Arthur Evans, Wendy Ferguson, and Jed Hartman. Sharing memory between processes. In Christina Cary, editor, *Topics in IRIX Programming*. Silicon Graphics, Wake Forest University, 2000.

[15] Iqbal Mohomed. Systems programming v (shared memory, semaphores, concurrency issues). Technical report, 2004.

[16] IEEE. Ieee p1003.1c-1995: Information technology-portable operating system interface (posix). 1995.

[17] OpenMP Architecture Review Board. Openmp c and c++ application program interface. 1998.

[18] Bob Kuhn, Paul Petersen, and Eamonn OToole. Openmp versus threading in c/c++. 2005.

[19] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. 2011.

[20] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. 2011.

[21] Azzam Haidara, Stanimire Tomov, Piotr Luszcze, and Jack Dongarra. Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing. 2012.

[22] OpenACC Directives for Accelerators. Theopenacc application programming interface. 2013.

[23] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. 1994.

[24] Intel. Avoiding and identifying false sharing among threads. In *Intel Guide for Developing Multithreaded Applications*. 2011.

[25] Intel. *Intel Guide for Developing Multithreaded Applications*. 2012-2015.

[26] Paul Lindberg. Performance obstacles for threading: How do they affect openmp code? 2009.

[27] M.D. Jones. Practical issues in openmp. 2014.

[28] OpenMP. Openmp: A proposed industry standard api for shared memory programming. 1997.

[29] Intel. Openmp loop scheduling. In *Intel Guide for Developing Multithreaded Applications*. 2014.

[30] Ravi Hegde. Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers. 2008.

[31] Steven P. VanderWiel and David J. Lilja. Data prefetch mechanisms. 2000.

[32] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. 2012.

[33] Bull J.M. Measuring synchronization and scheduling overheads in openmp. 1999.

[34] Abdel-Hameed Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. The efficacy of software prefetching and locality optimizations on future memory systems. 2004.

[35] Chunhua Liao, Zhenying Liu, Lei Huang, and Barbara Chapman. Evaluating openmp on chip multithreading platforms. 2006.

[36] Z. Sun, L. Benabou, and P.R. Dahoo. Prediction of thermo-mechanical fatigue for solder joints in power electronics modules under passive temperature cycling. *Engineering Fracture Mechanics, Volume 107*, 2013.

[37] L. Benabou, Z. Sun, and P.R. Dahoo. A thermo-mechanical cohesive zone model for solder joint lifetime prediction. *International Journal of Fatigue, Volume 49*, 2013.

[38] Q.K. Zhang and Z.F. Zhang. Thermal fatigue behaviors of sn–4ag/cu solder joints at low strain amplitude. *Materials Science and Engineering: A, Volume 580*, 2013.

[39] Shoho Ishikawa, Hironori Tohmyoh, Satoshi Watanabe, Tomonori Nishimura, and Yoshikatsu Nakano. Extending the fatigue life of pb-free sac solder joints under thermal cycling. *Microelectronics Reliability, Volume 53*, 2013.

[40] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. 1967.

# A

# APPENDICES

## A.1 EXPERIMENTAL SETUP

All tests were run in the SeARCH cluster at the University of Minho with shared nodes among other MSc. students. However, the nodes were always entirely reserved so that no external influence were noticed on the results. The methodology followed a k-best approach, where 6 runs were executed and measured and the best result was taken, provided that the difference between that and the 3rd best did not exceed 5%.

### A.1.1 *Node Characteristics*

The computer node used has the following characteristics:

- 2 x Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
- #Cores: 8 (with *Hyperthreading*, where #Threads are 2)
- L1 Cache (per core):
  - 32 KB instruction cache
  - 32 KB data cache
- L2 Cache (per core):
  - 256KB cache
- L3 Cache (per device):
  - 20MB cache (shared by all cores)
- Main Memory: 64GB

This node has 2 Xeon Processors, each one with 8 Cores plus Intel's technology *Hyperthreading* making it able to support 2 virtual hardware threads, thus 32 threads can run simultaneously. Memory accesses are non-uniform for these 2 NUMA processors as shown in the node topology.

*Node Topology*

In the Figure 44 it is presented the topology of this node from the operating system point of view, to better understand its functioning.

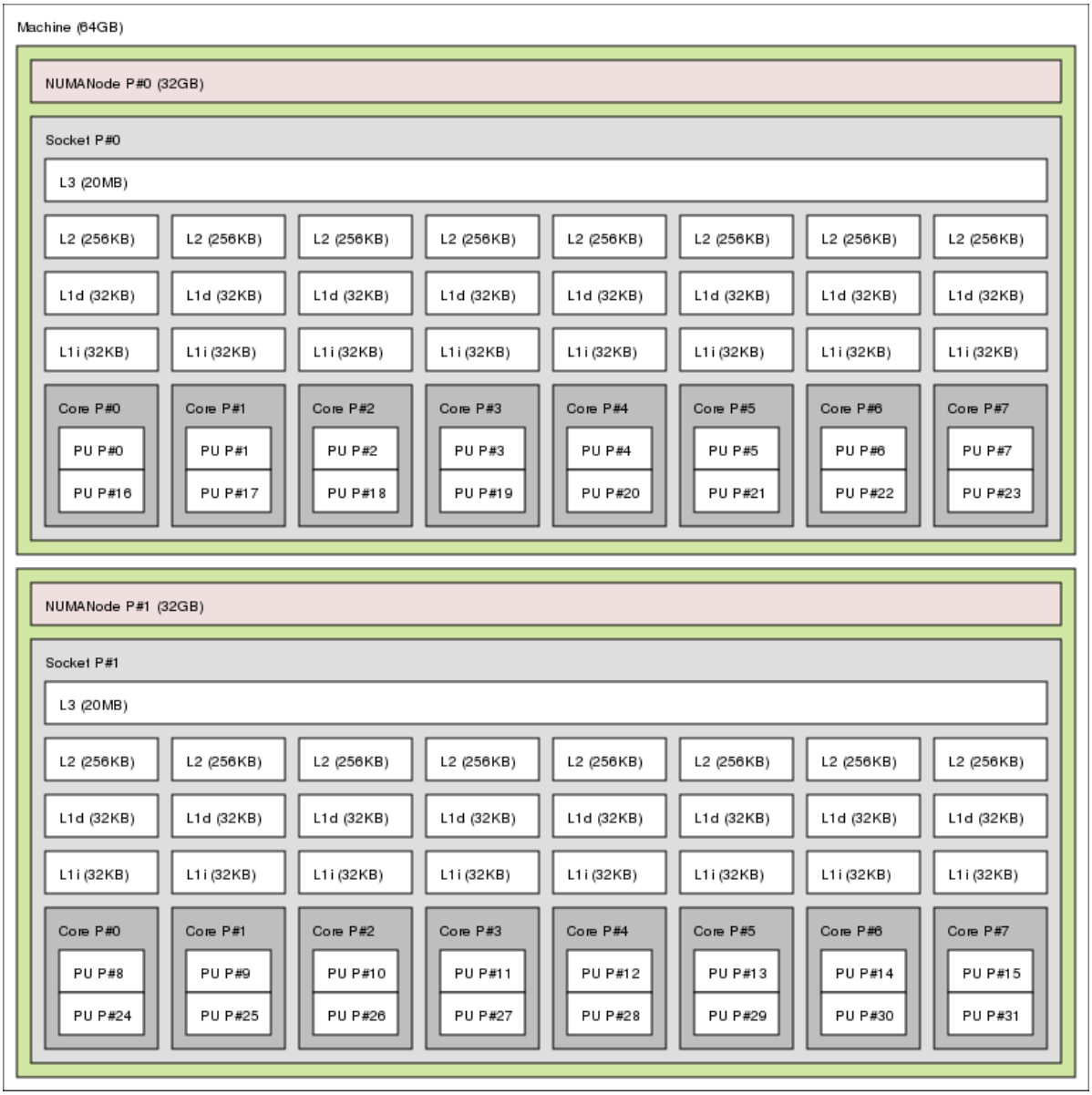



Figure 44.: Shows the node topology used in the experimental setup

As mentioned, there are 2 NUMA nodes, each one with half of the main memory available (32 GB for each) and each of the 8 cores has its own L1 and L2, with a shared L3 for the 8 cores of each processor. This has a great impact on memory and processor affinity since processor units from one NUMA node often need to access data from the other node. Another impact on the performance is the cache conflicts that *Hyperthreading* causes in L1 and L2 since these caches are shared by both threads.

### A.1.2 *Software Versions*

Several profilers and software were used in the SE analysis but mostly the profilers gProf and the Valgrind Tools (Valgrind, Callgrind, PAPI counters with cachegrind). These profilers have a measuring error inferior to 1% on the miss rate measures and a residual error on the call graphs creation. As for the software used, the versions are:

- GCC - Version 4.9
- Intel Composer 2015 SP1 (package 2015.1.117)
- ICC - Version 15.0.1
- PGI - Compilers & Tools 2014 (release 14.7)
- gProf - Version 2.20.51.0.2-5.34.el6
- Valgrind - Version 3.8.1
- GCC level 3 of optimizations

## A.2 SPEEDUP AND EFFICIENCY

Speedups were calculated for each scalability test performed according to:

$$S_p = \frac{T_1}{T_p}$$

where:

- $S$ is the speedup
- $p$ is the number of processors
- $T_1$ is the execution time of the sequential algorithm

- $T_p$ is the execution time of the parallel algorithm with $p$ processors

Also, it was calculated, according to Amdahl's Law [40], the maximum speedup that can be obtained:

$$S_p = \frac{1}{B+\frac{1}{n}(1-B)}$$

where:

- $S$ is the speedup
- $p$ is the number of processors
- $n$ is the number of threads of execution
- $B \in [0,1]$ is the weight of the sequential part of the algorithm

Another performance metric used - Efficiency - was computed to estimate how well-utilized the processors are in the execution of the program.

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

where:

- $E$ is the efficiency
- $p$ is the number of processors
- $S$ is the speedup
- $T_1$ is the execution time of the sequential algorithm
- $T_p$ is the execution time of the parallel algorithm with $p$ processors

This work was funded by Bosch, and it is included in the HMIExcel project (nº 36265/2013 (Project HMIExcel - 2013-2015)) as part of the case study 3 of the research line 12.