Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Tiago Abreu de Araújo

Efficient modelling of liquid surfaces on multi-core CPU and Xeon Phi devices

October 2015

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Tiago Abreu de Araújo

Efficient modelling of liquid surfaces on multi-core CPU and Xeon Phi devices

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
Alberto Proença
Luís Alves

October 2015

AGRADECIMENTOS

Começo por agradecer aos meus pais, por abdicarem do que fosse preciso para que eu pudesse concluir o meu objetivo. No final deste meu percurso de cincos anos, espero que eles sintam que tudo valeu a pena, e que tenham em mim tanto orgulho como eu tenho neles.

Ao Professor Proença por tudo aquilo que me ensinou académica, profissional e pessoalmente. É sempre um prazer conversar com alguém que sabe tanto sobre tudo e que uma simples observação é sempre tão cheia de conteúdo. Ao Professor Luís Alves pela ajuda nesta ponte entre a Engenharia Mecânica e a Informática, a sua colaboração foi essencial para que este trabalho fosse realizado. Ao Ricardo pela amizade e companheirismo ao longo dos dois anos de mestrado, que culminou neste trabalho desenvolvido com o Surface Evolver.

A todos os professores da Universidade do Minho que fizeram parte da minha vida académica.

Aos amigos que fiz na universidade. Todos eles me ensinaram que a amizade é a melhor lição que levamos da universidade. Aos meus amigos de sempre, pela paciência e pela força que sempre me deram, por sempre me mostrarem o caminho certo mesmo quando eu insistia no contrário.

Agradecimento à Bosch pelo apoio financeiro sob a forma de uma bolsa de investigação para desenvolver este trabalho e dissertação.

Trabalho desenvolvido no âmbito do projeto HMIExcel (nº 36265/2013 (Projeto HMIExcel - 2013-2015)), fica também o agradecimento pela confiança que depositaram no meu trabalho.

ABSTRACT

The assembly of miniature electronic components requires an adequate scale of the size of the welding terminators in printed circuit boards to minimize the stresses due to deformation. An optimum terminator layout minimizes the surface tension of the liquid solder, but requires efficient simulation algorithms to compute the results in an acceptable time slot. Current Surface Evolver is a software tool to study surfaces, shaped by surface tension and other energies, and its execution efficiency can be improved to take advantage of shared memory systems based on multi-core and many-core computing devices.

This dissertation aims to analyze the Surface Evolver, identifying the computational bottlenecks and working on solutions to improve the overall performance of the application. Parallel algorithms were developed to explore the architectural features of current multi-core and many-core computing devices namely the Xeon Phi, and including the growing vectorization features of newer processing devices.

After an analysis of the application and its profiling, the original data structure was identified as the critical bottleneck for software performance: it is implemented with linked lists, which prevents the use of the vectorization features of current devices and leads to inefficient parallel algorithms, both key elements to improve the performance of the Surface Evolver. The modification of the data structure was a key task in this dissertation.

The calculation force was identified as one of the most time consuming tasks of Surface Evolver and it was the target function of this work. This algorithm iterates over all vertices, edges and faces so is a good example to conclude how vectorization and parallelism affects the performance of simulation software used in the variety fields of science and engineering. In the end of this work it is possible to see that vectorization can greatly improve the performance of an application, bringing significant speedups to Surface Evolver.

The measured execution times are presented and discussed, throughout the various development stages of the application, aiming to analyze the impact of the application of high performance techniques on the Surface Evolver, suggesting yet further future improvements that were well identified in the end of this work.

RESUMO

A montagem de componentes eletrónicos mais pequenos requer um tamanho adequado da solda nas placas de circuito impresso para minimizar as tensões devido às deformações. Uma disposição ótima do terminal minimiza a tensão superficial da solda líquida, mas requer algoritmos eficientes para calcular os resultados num intervalo de tempo aceitável. O Surface Evolver é uma ferramenta de software para estudar superfícies, moldadas pela tensão de superfície e outras energias, e a sua eficiência pode ser melhorada para tirar proveito dos atuais sistemas paralelos.

Esta dissertação tem como objetivo analisar o Surface Evolver, identificando os estrangulamentos computacionais e trabalhando em soluções para melhorar o desempenho global da aplicação. Algoritmos paralelos foram desenvolvidos para explorar as características das arquiteturas multi-core e dispositivos de computação many-core, nomeadamente Xeon Phi, e também as novas características de vetorização presentes nos dispositivos mais recentes.

Depois da análise da aplicação, a estrutura de dados original foi identificada como o principal problema da aplicação: é implementada com listas ligadas, o que não possibilita o uso de vetorização e leva a algoritmos paralelos ineficientes, dois elementos cruciais para o aumento de performance no Surface Evolver. A alteração da estrutura de dados foi o trabalho mais importante ao longo desta dissertação.

O cálculo das forças foi identificado como uma das tarefas mais pesadas do Surface Evolver e foi por isso o alvo principal deste trabalho. O algoritmo itera sobre todos os vértices, arestas e faces, sendo por isso um bom exemplo para se tirar conclusões sobre como a vetorização e o paralelismo pode melhorar a performance de aplicações de simulação usadas nos vários campos da ciência e engenharia. No final deste trabalho será possível constatar que a vetorização consegue trazer melhorias de performance a uma aplicação, trazendo essas mesmas melhorias ao Surface Evolver.

Os tempos de execução foram medidos e discutidos, durante os vários períodos de desenvolvimento da aplicação, tendo como objetivo analisar o impacto da aplicação das técnicas de alto desempenho no Surface Evolver, sugerindo ainda futuras melhorias que foram identificadas e explicadas no final deste trabalho.

CONTENTS

Contents iii
1 INTRODUCTION 3
1.1 Context 3
1.2 Motivation & Goals 4
1.3 Contribution 4
1.4 Dissertation outline 5
2 HIGH PERFORMANCE COMPUTING 7
2.1 Heterogeneous Computing Platforms 7
2.2 Vectorization 8
2.3 Accelerators 9
2.3.1 MIC - Intel Xeon Phi 9
3 MODELLING THE LIQUID SURFACE 13
3.1 Surface Evolver 13
3.2 Testbed environment 14
3.3 Profiling the sequential code 15
3.4 Force calculation 17
3.5 Data structures 19
3.6 A shared memory implementation 22
3.7 A distributed memory implementation 24
3.8 Identified problems and decisions 24
4 AN EFFICIENT IMPLEMENTATION 27
4.1 Testbed environment 27
4.2 Improvements to the algorithm and data structures 27
4.3 Vectorization 30
4.4 Shared memory on multi-core 32
4.5 Distributed memory to many-core 33
5 INTEGRATED APPROACH 37
5.1 Testbed environment 37
5.2 Unified data structure 37
5.3 Results discussion 39
6 CONCLUSION 43

i APENDICES 47
A CALL GRAPH 49
B PROFILERS USED AND SOFTWARE VERSIONS 51
C EXPERIMENTAL SETUP 53
C.1 Node topology 54
D SPEEDUP AND EFFICIENCY 55

LIST OF FIGURES

Figure 1 Xeon Phi Coprocessor Core[1] 10
Figure 2 The Vector Processing Unit of Xeon Phi[2] 11
Figure 3 Ring system in the Interconnect of Xeon Phi[3] 12
Figure 4 Tilted view of the simulated **slow** case 14
Figure 5 **Slow** case after refinement 15
Figure 6 General view of call graph for the **fast** case 16
Figure 7 Zoom on some of the heaviest routines 17
Figure 8 Calc_force function flow 18
Figure 9 The functions that search on data structures 21
Figure 10 Execution time in parallel with Named Quantities - **Slow case** 22
Figure 11 Speedups in parallel with Named Quantities - **Slow case** 23
Figure 12 Efficiency in parallel with Named Quantities - **Slow case** 23
Figure 13 Calculation force algorithm with new data structure but without vectorization - **Slow case** 29
Figure 14 Comparison between with vectorization and without vectorization using dynamic scheduling - **Slow case** 31
Figure 15 Comparison between dynamic and static scheduling with OpenMP - **Slow case** 32
Figure 16 Calc_force function flow 34
Figure 17 Calc_force function flow with directives to Xeon Phi 35
Figure 18 Comparison tests with Xeon Phi version and dynamic schedule in CPU device 36
Figure 19 Execution time with OpenMP in the unified version - **Fast case** 39
Figure 20 Execution time with OpenMP in the unified version - **Slow case** 40
Figure 21 Speedups with OpenMP in the unified version - **Fast case** 41
Figure 22 Speedups with OpenMP in the unified version - **Slow case** 42
Figure 23 General view of Profiling the **slow** case 50
Figure 24 Topology of used node 54

1

INTRODUCTION

1.1 CONTEXT

The work in this dissertation aimed to solve an actual problem that engineers at Bosch Car Multimedia face in the production of printed circuit boards: how small can be the welding bubbles of the BTC (Bottom Terminated Components) devices, without failing during the life cycle of a product inside a car.

To reduce the problems of electrical failure due to the thermo-mechanical fatigue of welded components and to create conditions to increase their life cycle, the welding process can be optimized, using adequate simulation software.

Under normal use, BTC components and PCBs (Printed Circuit Board) undergo successive thermal cycles, which originate anomalies by gradients of thermal expansion coefficients of the various constituent materials, namely polymers composites and copper in PCB, BTC encapsulating material and brazing materials (Sun et al., 2013). The thermo-mechanical fatigue, when accumulated, leads to an interruption of the electrical conductivity of the connection.

The optimization of the placement of the BTC components on PCB and brazing procedures and parameters can lead to a substantial increase in the strength of electronic systems (Benabou et al., 2013). 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 line production and extend the life of the weld joint (Ishikawa et al., 2013), which are the two paths that assist in the overall quality of the brazing process, since the residual stresses resulting from thermal cycling depend on the configuration of the PCB and on the amount of solder.

To improve the life cycle of the electronic components two lines of work can be explored: (i) to optimize the design of the layout of the PCB to *manage* the local rigidity and thus minimize the effects of thermal fatigue of the solders of the BTC components, thereby minimizing the residual stresses of thermal origin, (ii) to optimize the procedures and parameters of brazing components of type BTC in PCBs, to scale the size of the *blisters* welding so as to minimize the stress fields associated with the deformation and buckling loads for the associated thermal cycling (Ishikawa et al., 2013).

For common components, not the type BTC, the lifetime of the PCBs are 2 DPMO (Defects Per Million Opportunities) Fall-Off-Rate in the production line and estimated life - in reliability tests -

is longer than 15 years, or even longer than 30 years when using nominal values of solder paste volume (Benabou et al., 2013). Results top class in the world, which is intended to maintain when the intention is to use components with BTC type miniaturization - such as BGA (Ball Grid Array), QFN (Quad Flat No-leads), LGA (Land Grid Array). The use of components other than the BTC type - QFP (Quad Flat Package), SOIC (Small Outline Integrated Circuit) - are much easier to produce and easily reach 30 to 40 in life cycle, due to the geometry of its terminals, and are easier to produce with such quality levels. These traditional ingredients are more flexible and thus adapt better to the deformations resulting from differences in the thermal expansion coefficients of the materials. Rather, the new BTC components, the rigidity is more susceptible to problems associated with thermal fatigue failure, thereby making it more difficult and demanding to maintain the defect level on 2 DPMO and 15 years lifetime. This means that with the introduction of the new BTC components it is necessary to introduce new strategies to the new implications for durability (Benabou et al., 2013).

In order to accomplish the best size for the welding bubbles in this context of fluid mechanics, the Bosch CM uses the software Surface Evolver, the application that will be deeply reviewed in this dissertation.

1.2 MOTIVATION & GOALS

The motivation to do this work is to improve the software used for the study of surfaces shaped by surface tension in a way that its execution times are more in line with the industry demands, improving the efficiency of the application. The objective is to develop two parallel versions: in multi-core CPU with shared memory and using the many-core Intel Xeon Phi, an accelerator. This work will take in consideration the specificities of each of these paradigms to achieve better results. Some comparison tests, like scalability and usability, between the paradigms will be presented to justify the choices made through the process. In resume, the main goals are:

- To implement a new data structure to take advantage of new features present on devices
- To implement a parallel version for shared memory systems with multi-core CPU devices
- To implement a version to take advantage of a many-core computing accelerator, the Intel Xeon Phi

1.3 CONTRIBUTION

This dissertation aims to contribute to the two areas in question, Mechanical Engineering and Computer Science. The contribution to Mechanical Engineering will be the implementation of a more reliable and faster version of the Surface Evolver, this way the testing of electronic components will be made more quickly, allowing these same components come faster to the market and with better

solutions in terms of welding of components. To the Computer Science, will be essentially the restructuring of data structures, by localization and resizing, to increase the computational efficiency in order to contribute to new forms of data organization, which is a major area of work currently in High Performance Computing and often - almost always - the difference between having a parallel version with or without improvements.

1.4 DISSERTATION OUTLINE

This document contains six chapters. The first is an introduction with focus on explaining the problem to solve with this work in the software. The chapter 2 briefly presents the state of the art of High Performance Computing in terms of their architecture. The chapter 3 introduces the Surface Evolver and analyses the original version of the software application. The chapter 4 presents a new data structure to overcome the performance bottlenecks identified in a previous profiling of the application code. In chapter 5 the fusion with the data structure presented in (Ribeiro, 2015) is explained and the profiling of this unified version is discussed. The last chapter presents the conclusions and suggestions for future work are presented.

2

HIGH PERFORMANCE COMPUTING

2.1 HETEROGENEOUS COMPUTING PLATFORMS

Manufacturers are moving from high-frequency designs to multi-core chips, instead of improving single-threaded performance. Besides programming CPUs, general-purpose computation on graphics processing units (GPGPUs) has become increasingly popular as computing accelerators. Studies show that state of art multi-core CPUs are able to compete with accelerators, by exploring single instruction multiple data processing, multi-threading and cache efficiency techniques. Upgrading a convolution algorithm on a CPU, it was shown that the processing is only a little slower than on a GPU. Nowadays, hardware designers have to design and implement a hardware architecture specific to the algorithm, costly in terms of production time as compared to software engineering. This hardware design processes has consider the problem of power consumption, situation where FPGAs has an advantage because are powerful devices with low power consumption for certain applications.

Heterogeneous platforms consists of at least one multi-core CPU and one accelerator device that typically is a GPU or an Intel Xeon Phi. These platforms raise several concerns like choose efficient memory structures to run on GPU, share work between the CPU and GPU to have both working at the same time - no device wait for the other - and assign the type work that each device can perform better. The accelerators, for example, can improve the application efficiency performing the heaviest computing tasks.

The challenge with the heterogeneous platforms is handling with the hardware constraints, the memory limitation of accelerators and the fact that CPUs and accelerators do not share the same memory - which does not allow a traditional multi-threaded approach - raises problems that it must deal to get maximum performance of these heterogeneous platforms - that it is the sum of the maximum performance every device can provide. CPUs and accelerators not sharing the same memory is the key for performance degradation in heterogeneous computing platforms because implies the communication of data elements between devices. Some techniques as pipelined DMAs (Direct Access Memory) for data transfer - it is extremely important to fully exploit the DMA engine capabilities to maximize network bandwidth, specially in large transfers -, dynamic chunk sizing, and better asynchronous progress, try to decrease communication latency as shown on the literature review (Vaidyanathan

et al., 2014), but these techniques are always specific to the algorithm in study and are not used as a general purpose solution.

2.2 VECTORIZATION

Vectorization is implemented as the execution of a single instruction on multiple data objects, i.e., to apply operations to whole arrays instead of individual elements. It is the way to take advantage of AVX or SSE in the Intel x86 line of processors. For example, the AVX instructions can perform eight 32-bit or four 64-bit floating point operations per clock cycle. It is a huge improvement on performance, beyond the capacity of task parallelism, this way we also have data parallelism.

To take advantage of vectorization and improve the efficiency of applications, it is important to identify which type of loops can be vectorized (Corporation, 2012):

- **Countable.** The number of loop iterations must be known at the beginning of the loop at runtime, though it need not be known at compile time. The loop count can be a variable, but the variable cannot vary during the loop execution.

- **Straight-line code.** SIMD instructions perform the same operation on data elements from multiple iterations of the loop, so it is not possible have 'if' conditions on those iterations because different iterations cannot have different paths, they must not branch. As expected, 'switch' conditions are not allowed. The 'if' statements is allowed if they are masked assignments, i.e., the calculations is made for all elements, but the assignment is only performed for elements whose mask returns true.

- **The innermost loop.** In a situation of nested loops, only the innermost is vectorized. The exception is if the compiler - in compiling phase - transform an outer loop in a inner loop due to optimization techniques such as loop unrolling or loop collapsing.

- **No function calls.** Call functions inside a loop can result in a unvectorizable loop. Some exceptions are the known math functions - e.g. cos, sin, floor, sqrt, exp - and for inline functions. Inline functions are only keywords to substitute the body of the function performing inline expansion and can be intended as function replacing and not a function call.

On the other hand, to vectorize loops is crucial to avoid the most known inhibitors of this feature (Corporation, 2012):

- **Non-contiguous Memory Accesses.** A single SSE instruction loads/stores from/into memory sets of integer and real numbers; if these data elements are not adjacent in memory, multiple instructions are required to access memory, causing a negative impact on performance. The most common examples of non-contiguous memory access are loops that iterate through linked lists, a data structure that scatter the data all over the memory without any sense of contiguous

data. The compiler seldom vectorizes such loops, unless the amount of work is large compared to the cost of loading data from not contiguous locations.

- **Data Dependencies.** If the data elements that are written in one iteration do not appear in any other iteration, this mean the loop are data independent and possible to be vectorized. When a variable is written in one iteration and read in a subsequent iteration, there is a "read-after-write" dependency, and when a variable is read in one iteration and written in a subsequent iteration, this is a write-after-read dependency, resulting both in vectorization inhibitors.

The Intel C compiler (ICC) has some options in the compiling phase that allows to automatically generate vector code, but this feature can not work in perfection by the variations of each code. To help on this task, ICC has a helpful tool called vec-report that aids the developer find out which loops were not vectorized, making a vectorization report and providing guidelines with the reason why such loops were not vectorized. When this happens, it is necessary the intervention of the programmer to modify the code in such a way to enable those loops to be vectorized.

2.3 ACCELERATORS

In recent years this area has evolved to computing with accelerators, GPU and MIC. The GPUs are no longer only graphics processors, but are devices with a key role in high performance computing, with high ability to perform calculations and are therefore a valuable ally to increase the efficiency of applications. The GPU architecture may contain thousands of smaller and more efficient cores designed to handle multiple tasks simultaneously, making it a massively parallel architecture.

As GPUs will not be explored in this dissertation, only the Intel Xeon Phi architecture will be detailed in the next section. The architecture of GPU devices are further detailed in (Ribeiro, 2015).

2.3.1 *MIC - Intel Xeon Phi*

The Xeon Phi is Intel response to the dominance of Nvidia in the segment of accelerators for HPC. The coprocessor is connected to an Intel Xeon processor, also known as the host, through a PCI Express bus, and is under this PCIe bus that could be implemented a virtualized TCP/IP stack to access the coprocessor as a network node and allowing the user to connect to the Intel Xeon Phi through a secure shell (ssh) and run jobs interactively inside the device [1]. It is also possible build applications wherein a part of the application executes on the host while a part executes on the coprocessor.

In a single host system can be installed multiple Intel Xeon Phi coprocessors and they can communicate through the PCIe peer-to-peer interconnect, InfiniBand or Ethernet, without any influence from the host, preventing further loss of performance in the communication process between coprocessors and host, making a direct connection in all devices present in a single host.

1 https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

The Intel Xeon Phi comes with a private L2 cache in each core that is kept fully coherent by a global-distributed tag directory a memory controllers that provide a direct interface to the GDDR5 memory on the coprocessor and a PCIe client logic that provide the connection with the PCIe bus (Jeffers and Reinders, 2013). All these components are connected together by the ring interconnect.

This device contain 61 cores - most common - that run in low clock frequency (1.238 GHz), with hardware support for 4 simultaneous threads in each core, equipped with 32KB L1 instruction cache and 32KB L1 data cache, and shares 512KB of L2 cache, giving a total of approximately 30 MB for the entire device(Jeffers and Reinders, 2013). These caches are fully coherent and implement the x86 memory order model and provide an aggregate bandwidth faster compared to the aggregate memory bandwidth. Is designed to be power efficient while providing a high throughput for highly parallel workloads(Jeffers and Reinders, 2013).

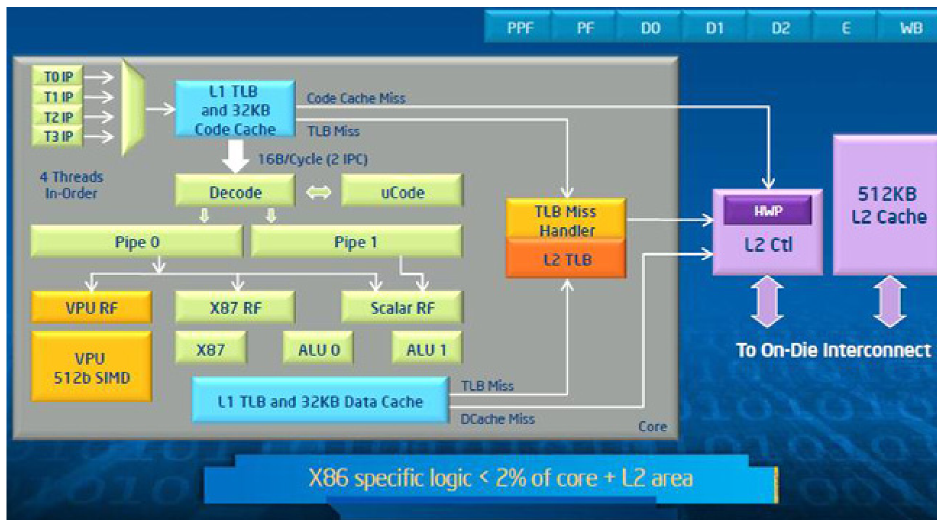

Figure 1.: Xeon Phi Coprocessor Core[2]

Xeon Phi has a Vector Processing Unit (VPU) that can execute 16 single-precision or 8 double-precision operations per cycle. Since this unit supports Fused Multiply-Add (FMA) instructions, each core can execute 32 single-precision or 16 double-precision floating point operations per cycle. The VPU also supports gather and scatter instructions directly in hardware. This features helps in keeping the code vectorized with sporadic or irregular access patterns(Jeffers and Reinders, 2013).

To help in complex math operations, such as square root, reciprocal and log, the VPU execute this operation in a vector way calculating polynomial approximations of these functions(Jeffers and Reinders, 2013). This feature is known as Extended Math Unit (EMU) and show how Intel designed a device aiming to give efficiency to heavy computational software, bringing news ways to improve the most common calculations of simulation applications in the variety of fields of Science and Engineering.

2 https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

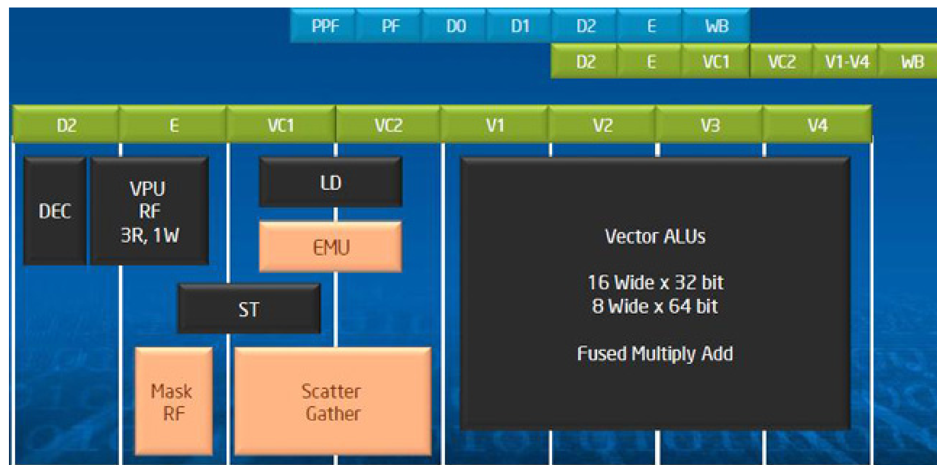

Figure 2.: The Vector Processing Unit of Xeon Phi[3]

The communication between those many cores is implemented as a bidirectional ring. Each direction is comprised of three independent rings: first, largest, and the biggest, the data block ring. The first ring is the acknowledgement ring, the smaller, and is responsible to send flow control and coherence messages(Jeffers and Reinders, 2013). The largest is address ring, used to send read/write commands and memory addresses. The data block ring - the largest - is 64 bytes wide to support the high bandwidth requirement due to the large number of cores(Jeffers and Reinders, 2013).

When a miss L2 cache occurs, an address request is sent to the tag directories. If the requested block is found in L2 cache of another core, the request is sent to this L2 cache over the address ring and request block is sent through the data block ring. If the requested block is not found in any L2 caches, a request is sent to the memory controller. This is the scheme of how a miss cache is treated by the Interconnect of Intel Xeon Phi coprocessor(Jeffers and Reinders, 2013).

3 https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

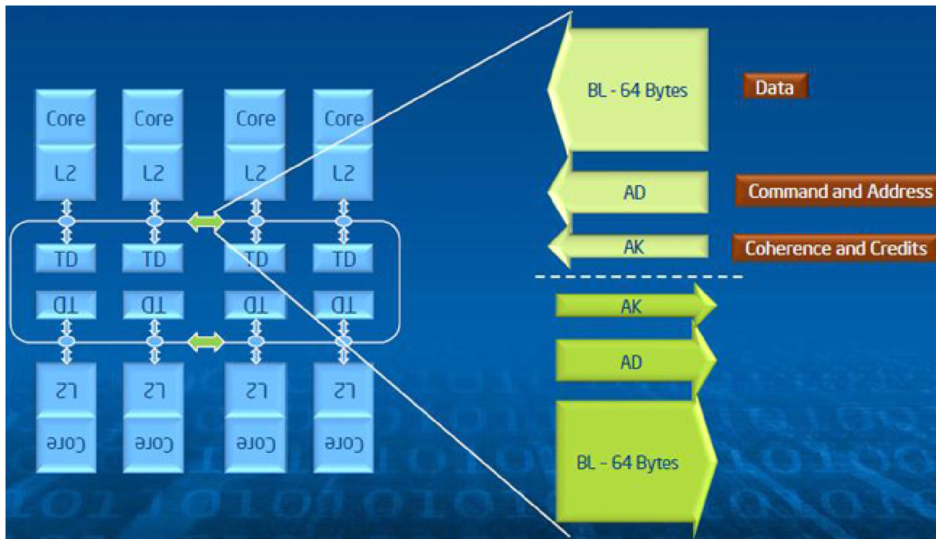

Figure 3.: Ring system in the Interconnect of Xeon Phi[4]

The floating peak performance in Xeon Phi is given by:

- 16 (SP SIMD) x 2 (FMA) x 1.1 (GHz) x 61 (# cores) = 2 147,2 GFLOP/sec for single precision
- 8 (DP SIMD) x 2 (FMA) x 1.1 (GHz) x 61 (# cores) = 1 073,6 GFLOP/sec for double precision

Another great advantage of Xeon Phi is that it is not necessary to learn a new language, it is possible to use OpenMP code, MPI or PThreads - for example - to run. It also offers different ways to run applications: native code, automatic offload or manual offload. Using native code, the code is compiled to run entirely on Xeon Phi, while when using offload only some parts of the code are sent to the device to run. In the case of offload, it is possible to delegate this task to the compiler or to do it manually, task done by the programmer using directives defining which code regions to send to the device. The two possibilities - native or offload code - have, expectedly, different results. For the hardware memory architecture, a code that requires plenty of space to allocate data turns out to have a worse performance when compiled as native code, being preferable in this case to use offload to define the pieces of code to be sent to the device, selecting the heaviest computational routines, where the Xeon Phi can really make a difference with the 61 cores that are at your disposal.

4 https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

3

MODELLING THE LIQUID SURFACE

3.1 SURFACE EVOLVER

The way to improve the life cycle of the electronic components is optimizing the layout of the PCB and the procedures and parameteres of brazing components, through a better size of the welding bubble to minimize the stress fields associated(Ishikawa et al., 2013).

To achieve these optimizations, it is essential to take advantage of available computational tools (CAE) to support the new PCB design integrating components of type BTC, and generate additional value by incorporating new technologies on manufacturing smaller integrated circuits - the new trend of the market, miniaturization. It is also noted that according to the existing literature, the optimization of the weld volume is a key factor and is still poorly explored. Another point that may help CAE is the speed of the validation process, this time testing the experimental validation of a conceptual solution (PCB) stand at about 4 months (Ishikawa et al., 2013), quickly reach the market if the time of these tests is reduced.

In summary, the differentiation factor is attempting to virtualize the optimization process of brazing procedures and parameters of BTC type components on PCB, through the study of the influence of the quantity of liquid solder in the final geometry of the bond between the copper PCB and the *legs* of the BTC component. This objective should be determined through optimization algorithms and minimization of surface tension of the liquid solder to later be able to study the impact of the thermal cycling stress fields generated and thus optimize the lifetime of the component.

Developed by Ken Brakke in University of Susquehanna, Surface Evolver (SE) is an interactive program to model liquid surfaces shaped by various forces and constraints. SE evolves the surface for a minimum of energy by a gradient descent method. The evolution is intended to be a computer model of the process of evolution by mean curvature for energy surface tension. The energy in the SE can be a combination of surface tension, gravitational energy, mean square curvature, surface integrals defined by the user or nodal energy. SE can, also, handle arbitrary types, volume restrictions, border restrictions, limit contact angles, gravity and restrictions as surface integrals. The surface may be an arbitrary scale environment, which can have a Riemannian metric and the environment space can be a space under a quotient group. The user can modify the interactive surface to change its properties or to maintain a well-behaved evolution.

3.2 TESTBED ENVIRONMENT

All tests in this chapter were run on the Cluster SeARCH at the University of Minho. The computing node used has the following characteristics:

- 2 x Intel Xeon CPU E5-2650 @ 2.00GHz
- 8 cores per device, each with 2-way Hyper-Threading
- L1 cache per core: 32KB for instructions and 256 KB for data
- L2 cache per core: 256KB
- L3 cache per device: 20MB (shared for all cores)
- Main memory: 64GB

Three case studies were selected to work on this project: a fast simulation with a small input data set, a medium and a more complex and time consuming. Due to a version mismatch of the Surface Evolver, the medium case introduces some errors during its execution and therefore could not be used for these tests. The execution times for cases **fast** and **slow** will be shown later when proceeding to the presentation of SE profiling. The methodology to measure the code execution time used the k-best approach, where 6 executions for each test were made and chosen the best measure that had a difference of not more than 5% in relation to the second and third best measurement. An overview of simulations that are being addressed follows, where images of the **slow** case are shown, as well as the object state at the end of the simulation and another image of the object after a few more iterations and refinement.

Figure 4.: Tilted view of the simulated **slow** case

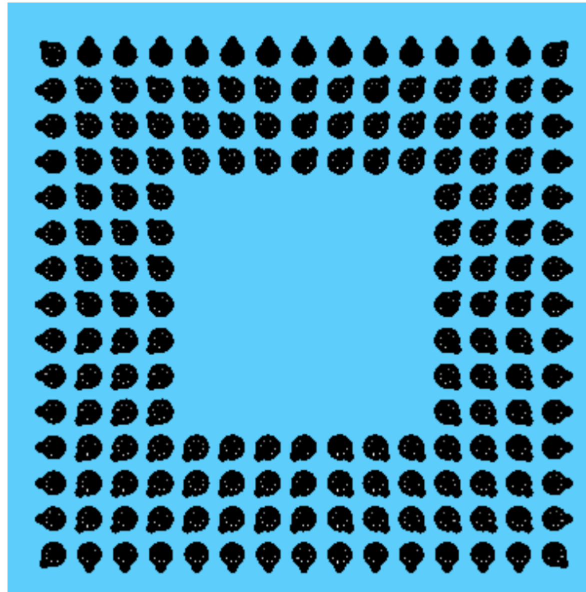

Figure 5.: **Slow** case after refinement

3.3 PROFILING THE SEQUENTIAL CODE

Currently the Surface Evolver has a fully functional sequential version. This version is not created as a starting point for a parallel version and has some limitations when it wants to migrate the code to a parallel version. This version in cases of relatively complexity, takes some time to complete its execution, taking about 40 minutes in one case study that will be presented throughout this dissertation. The sequential version does not have much to refine, and so in order to improve computing efficiency a parallel version will be developed.

A call graph is a directed graph used to represent the relationship between the functions of a program. With this graph we can identify the functions that take a larger share of the overall execution time and the number of calls those functions.

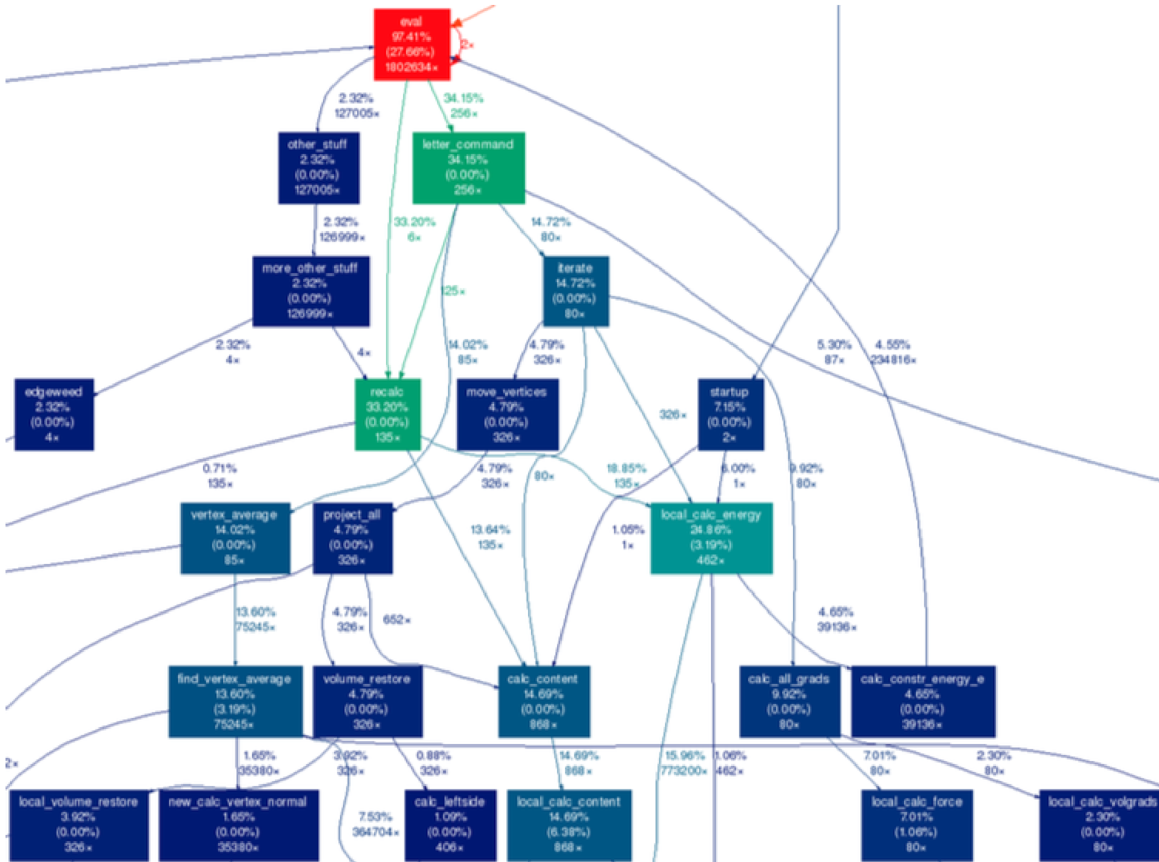

Figure 6.: General view of call graph for the **fast** case

Heaviest routines:

- **recalc** 33%: the interpretation of commands has a relative weight increase in small case studies; the main function responsible for this weight aims to recalculate and display the new results to the user;
- **calc_energy** 24%: is responsible for obtaining the total energy configuration;
- **calc_all_grads** 10%: used to recalculate the forces and/or restrictions gradients.

The other functions mainly compute energies, volumes and their gradients at the vertices.

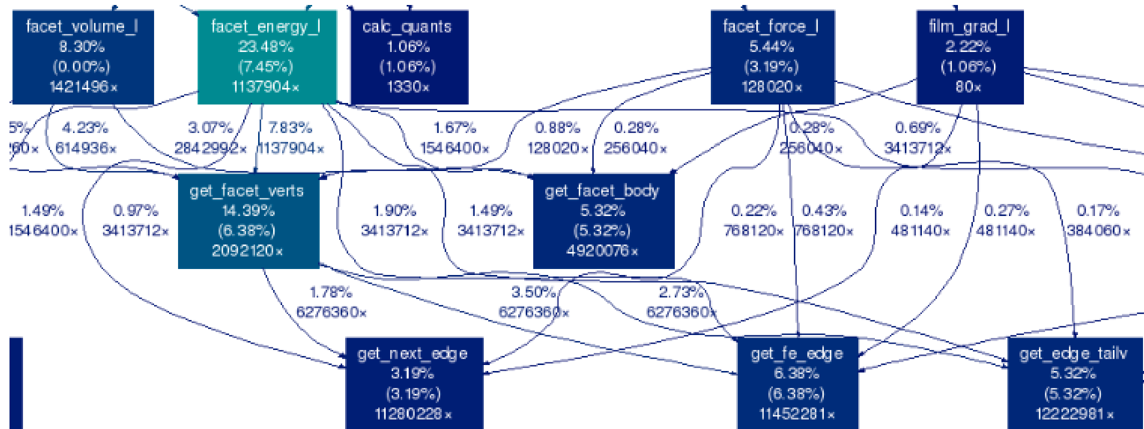

Figure 7.: Zoom on some of the heaviest routines

The functions that search for edges, corners, facets and vertices belonging to the data structure (including get_facet_verts, get_facet_body, get_next_edge containing, among other get_fe_edge) has a weight of approximately 14%. This structure should be reconsidered to reduce the impact of these functions on the overall performance.

An analysis of the resulting call graph of the **slow** case (details in Appendix Call graph) shows that the recalc function, which accounted for 33% of the runtime now has a weight of 62%. This significant increase in weight is due to the increase of the problem size. While in the case **fast**, all data entirely fit into the cache, and miss rate in some of these routines less than 1% in L1 cache is nearly 0% in L2 cache due to compulsory misses, these are always available in cache. In the **slow** case, whose data occupy dozens of MB, the miss rates increase and the access to the main memory are extremely costly.

3.4 FORCE CALCULATION

This function calculates the force caused by superficial tension and other constraints on vertices of triangulation. Surface Evolver is a iterative application that calculate the evolution of the mesh over time, but this function is only called in the end of each iteration - calc_energy is called thousand times each iteration - but still called thousand of times in the execution of the whole application. In this particular case study is not heavy, but is a function that can be more time consuming on other simulations.

This diagram shows the flow of calc_force function:

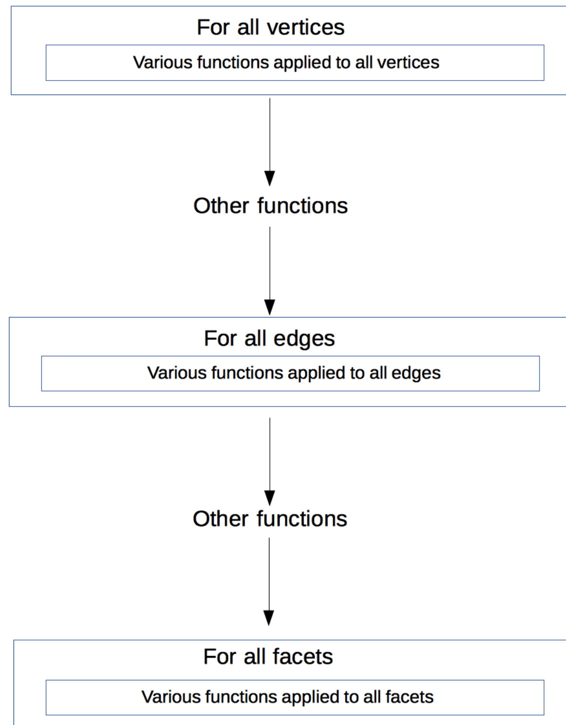

Figure 8.: Calc_force function flow

The function has three main loops - that runs on all vertices, edges and facets - and apply various functions to every elements. In the original data structure, to find the elements is mandatory to search on the linked list and it is necessary start the search at the beginning of the list, making this routine computationally expensive due to the large number of elements for each type. This is a common problem when using linked lists with a large number of elements.

Inside each main loop, the working element needs to be applied to various functions - some functions are quite heavy - that are expecting pointers to do their work. On a task to change the data structure of Surface Evolver, this functions need to be fully rewritten to maintain the functionality of the application.

Besides this functions applied inside major loops, between those loops other functions are applied to some elements of the web that need the elements updated with the functions applied inside the loops. This restriction invalidate any attempt of parallelism between those calculations - functions inside loops and functions outside loops - or change the order of execution because the need of updated elements implies that wait by the finish of the previous major loop. These strategies could be very effective for accelerators - explained in detail on next chapter of many-core devices.

Calc_force is one of the main functions related to computational work. It is strictly related with calculations and do not have IO (input-output) work, so it is a good candidate to deliver an effort to

improve the efficiency of the function. As a calculation function, it is always possible to implement strategies from High Performance Computing to gain performance, and without IO operations it is possible to implement a version that targets on heterogeneous platforms.

3.5 DATA STRUCTURES

The original data structure of the web - the mesh of the elements - is based on linked lists. This type of data structure is very useful for dynamic allocation of data - probably the reason of the choice - because it can easily append and remove elements from the list. The linked lists have $\theta(1)$ for insertion - when last member is known -, $\theta(n)$ for searching and $\theta(n)$ for removing assuming that search is needed. The main disadvantages are: a linked list must be read in order from the beginning as linked lists are inherently - sequential access; are stored incontiguously, greatly increasing the time required to access individual elements within the list.

In Surface Evolver, an element called web is defined by a data structure responsible for all the mesh - the webstruct. This struct has various properties about the mesh, like lagrange_order, dimension, number of elements and so on. Inside this struct, has an array of struct of type skeleton that will be explained next. This array of structs has five positions, the indices are 0 for vertices, 1 for edges, 2 for facets, 3 for bodies and 4 for facetedges. These indices have a pointer to the respective list of elements. Besides this, the webstruct has some fields to manage the size of elements in the list to reallocation proposes.

Part of webstruct data structure:

```
struct webstruct {
    struct skeleton skel[NUMELEMENTS];
    int sizes[NUMELEMENTS];  /* allocated space for element structure */
    int usedsizes[NUMELEMENTS];  /* used space for element structure */
    struct element **elhashtable; /* id hash list of element pointers */
    int elhashcount;  /* actual number of live entries */
    int elhashmask;  /* for picking off index bits of id hash */
    int elhashsize;  /* size of hash table; power of 2 */
    int sdim;  /* dimension of ambient space */
    int dimension;    /* where tension resides */
    int representation; /* STRING, SOAPFILM, or SIMPLEX */
    int modeltype;    /* QUADRATIC, LINEAR, or LAGRANGE;  */
    int lagrange_order; /* polynomial order of elements */
    int headvnum;
    ...
  }
```

The struct skeleton is the responsible to manage the linked list of each element. It has a field to identify the type of element - vertex, edge, facet, body or facetedge - the dimension in bytes of the

element, the first element and pointers to free positions. This struct was well designed in terms of memory allocation, but for strategies used in High Performance Computing this structure does not fit and with the evolution of devices, this type of data structure will be responsible for performance degradation because do not help the new features present on newer CPU and accelerators.

Part of skeleton data structure:

```
struct skeleton {
    int type; /* type of element*/
    int dimension;  /* dimension of element */
    int ctrlpts;    /* number of control points */
    INDIRECT_TYPE  *ibase; /* to indirect list */
    int ialloc; /* number elements allocated to ibase */
    long maxcount;   /* elements allocated */
    int alloc;      /* number actually in use */
    element_id  free;       /* start of free list */
    element_id  freelast;   /* end of free list  */
    int  sparse_spot; /* used by sparse_ibase_flag */
    struct element *freehead;  /* start of freelist
    ...
  }
```

To validate these assumptions about the original data structure, the functions that works directly with this data structure will be analyzed above:

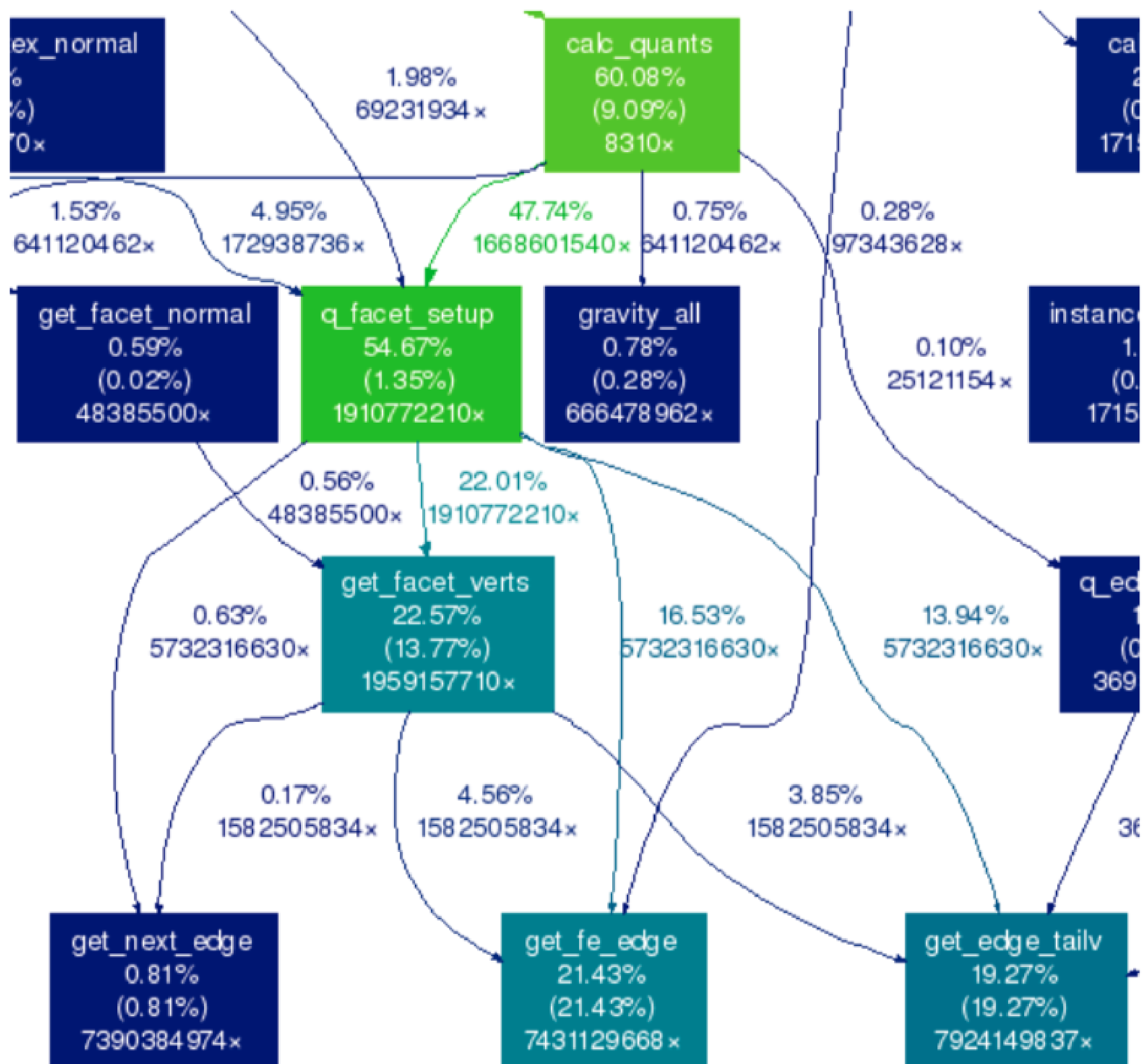

Figure 9.: The functions that search on data structures

The functions responsible for search and maintenance of data structures has a significant weight. As mentioned earlier, this data structure should be changed.

The search by edges, vertices and facets are quite heavy. This difference is even more clear when we are running in parallel and even more pronounced in environments with Non-Uniform Memory Accesses (NUMA), as the environment where these case studies have been performed. One of the largest bottlenecks in the performance of the Surface Evolver involves memory accesses. Such accesses should be minimized studying a data partition algorithm and allocate threads to run on the data that are allocated to your device or a data structure that minimizes the time to find an element, that can be a structure based in arrays by the fact that have $\theta(1)$ as search complexity, very different of $\theta(n)$ in linked lists, which means that with this change memory accesses will be largely decreased.

The SE data structures are also a source of problems because it is based on linked lists. This structure dispersed the data throughout memory preventing the vectorization, which could greatly improve the application efficiency. So it is important to think in a structure which favors the allocation of contiguous data and in that way take advantage of vectorization features of newer devices.

3.6 A SHARED MEMORY IMPLEMENTATION

The only parallelized functions in the original SE version belong to the class **Named Quantities**, using PThreads. It is a class that aims to give a more systematic way to add new energy and constraints. When using the flag **-q**, everything is converted to **Named Quantities** and this is the only class whose functions use parallel computing. The goal of parallel SE includes working with other functions and not just with this **Named Quantities**, will therefore be a more robust and more specific version of Surface Evolver.

As mentioned, the implementation of parallel calculation is made through the Named Quantities using PThreads. Then, the scalability analysis of this implementation will be studied for the **slow** case. Those tests only was performed at 14 threads. The application have problems with this shared memory version and never finish with more than 14 threads. This is another concern with Surface Evolver, build a robust and clean version in shared memory maintaining the functionality of the application with consistent outputs and also improving the parallel version bringing speedups.

The use of parallel computation by the Named Quantities bring improvements, but only 10 threads.

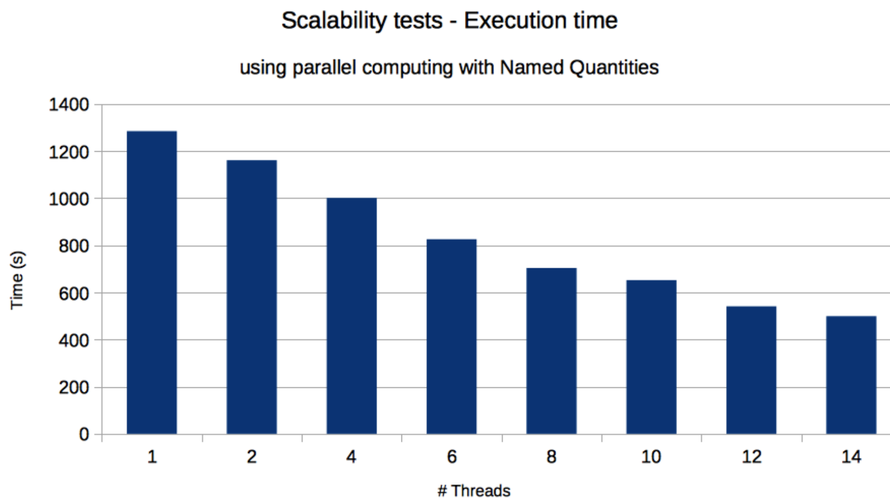

Figure 10.: Execution time in parallel with Named Quantities - **Slow case**

Average CPU time per thread remains low but the difference between the total execution time and by thread does not compensate the use of 8 threads or more.

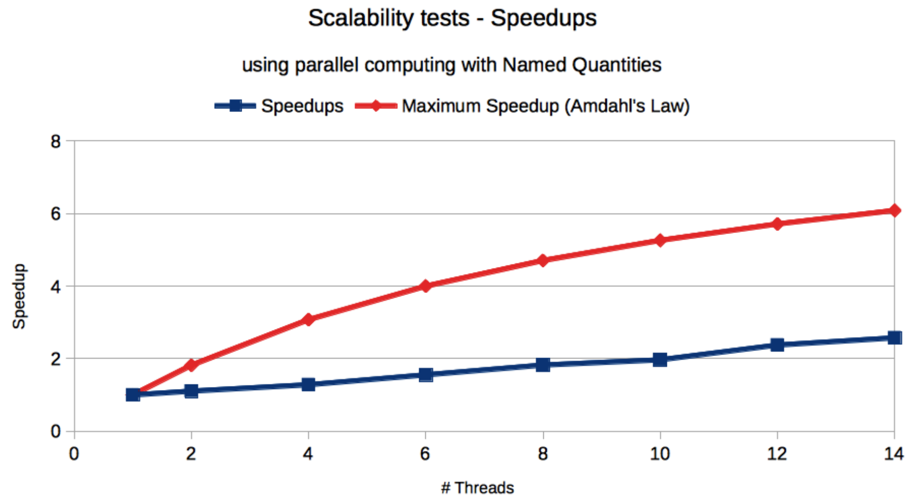

Figure 11.: Speedups in parallel with Named Quantities - **Slow case**

The speedups show the execution times and as mentioned are the maximum possible distance given by Amdahl's Law.

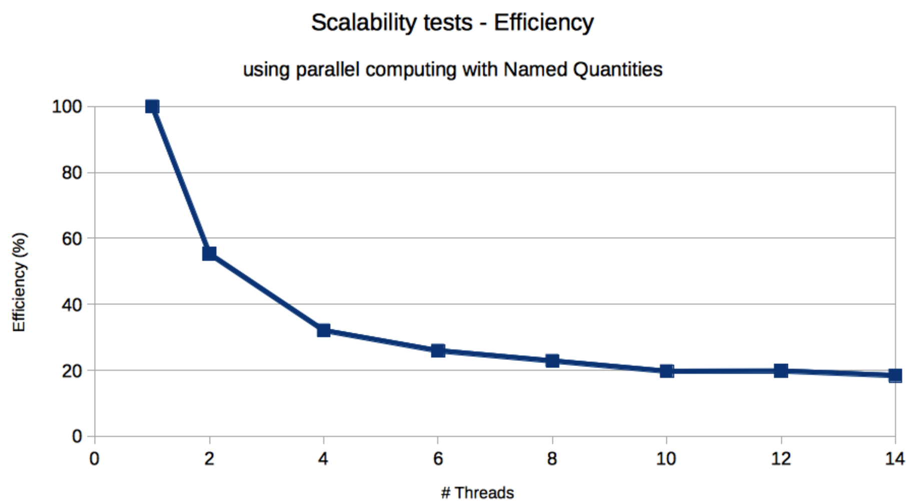

Figure 12.: Efficiency in parallel with Named Quantities - **Slow case**

This case study makes use of the threads most effective using parallel computing in Named Quantities. Most reuse of cached data in cores where the threads are running, makes possible this efficiency increase.

3.7 A DISTRIBUTED MEMORY IMPLEMENTATION

On the SE website there is an MPI version available. This version was tested and we came to the conclusion that it is not even consistent on the outputs, giving different results with different number of processes, making it not reasonable to use because this version not even preserve the application functionality. On the webpage there has a warning to use this version with care as it is an experimental version. Since it was decided that the aim of this work was to build a version of shared memory, the MPI version will not be improved, and the work will be invested in developing a OpenMP version.

3.8 IDENTIFIED PROBLEMS AND DECISIONS

The Surface Evolver, as explained above, is one of the solutions to the problem, but as a solution it brings with it, as expected, new problems and challenges. One of the situations that we need to deal with is the computational weight of such simulations. In more complex cases, the Surface Evolver can take hours to calculate this same simulation, something completely unacceptable in a factory environment.

Another problem arises from the parallel calculation used in Named Quantities, where sometimes the result when running with multiple threads is inconsistent with the sequential version, which proves that is not well implemented. One more time, this type of problem can not be acceptable in a business environment, an application without consistent and trustful results, is an application that not does its purpose.

The solution proposed is to parallelize the application. The strategy will depend almost exclusively the result of Profiling SE, with the results and the analysis of what are really bottlenecks during run-time, the way forward will be completely different. First it is necessary to understand if the application is memory or CPU bound. Next is to realize if communication affects the overall performance. The solutions are essentially in parallel using CPU or through the use of accelerators. In order to use accelerators is necessary a further study of the application, because the impact of communication affects the performance when using accelerators. What is gained in computing performance is often lost only in communication. The ideal solution is, in many cases, to use a hybrid solution: pieces of code run in parallel in the CPU while others run in the accelerator. When the goal is to run parts of the application in Xeon Phi, and by its specificity, it is necessary a deeply study of the data structures. As current Xeon Phi has 61 cores with 4 threads per core - which makes a total of 244 threads - this architecture gives applications a breakthrough in computing power. But as L1 cache has only 32 KB per core and 512 KB of L2 cache per core, in turn, requires some adaptation of applications to really take advantage of this architecture, because the volume of data can not be very high due to the less space of caches of each core.

After the profiling made and analyzed in this chapter, it was identified that the data structure based in linked lists is a huge problem of Surface Evolver. This type of structure is a inhibitor of vectorization

and do not benefit a parallel version for multi-core CPU and many-core devices. To benefit of those features and strategies is crucial to implement a data structure that works well in this context. The choice was to implement a version of this data structure with static arrays. With this type of structure the data will be contiguous in memory - very useful for vectorization - and the search for elements is in constant time when using the index of each element. In addition to these improvements, arrays is more used in the context of High Performance Computing because is easily implemented in parallel version for multi-core CPU and many-core devices, unlike the linked lists that does not fit well in CPU and accelerators approaches, because it brings many problems and in most cases it can not be implemented for these platforms.

This software has years of development and probably several developers, increasing the complexity of the application and unreadability code. The code is very large - above hundred of thousand lines of code - and not very well commented and documented. This is a problem to take in account before start implementing a new data structure, because the original version of data implement is present in all application, so changes are required in all source code. As starting point, the application was divided in three large stages: parsing, computation and graphic user interface (GUI). The heaviest and more important stage is computation, so the focus of this work will be on the functions strictly computational and calc_force raises as good candidate to this work. It is a heavy function with three large loops that iterates through all elements and fits in the intention of test vectorized loops with new data structure based in static arrays and also a good case to run the loops in parallel on CPU and many-core devices.

As mentioned earlier, the MPI version of the SE has some problems. After analyzing the application code it has been decided that it was not worth the effort to rewrite this version, but work in a completely new solution. It would be much more costly trying to work on a experimental version and nothing stable, with only a few commented code, than starting a new version in which all decisions to parallelize the software is already taken with knowledge of what is the application, therefore a specific parallel version for SE: taking into account all types of data used, the size of the structures and target architectures.

4

AN EFFICIENT IMPLEMENTATION

4.1 TESTBED ENVIRONMENT

Due the necessity of a Intel Xeon Phi device to perform some comparisons, the tests in this chapter were run on a node with this type of device and have the following characteristics:

- 2 x Intel Xeon CPU E5-2670 v2 @ 2.50GHz
- 10 cores each with 2-way Hyper-Threading
- L1 cache per core: 32KB for instructions and 256 KB for data
- L2 cache per core: 256KB
- L3 cache per device: 25MB (shared for all cores)
- Advanced Vector Extensions (AVX)
- Intel Xeon Phi with 61 cores @ 1.238 GHz (total of 244 simultaneous threads)
- Main memory: 64GB

4.2 IMPROVEMENTS TO THE ALGORITHM AND DATA STRUCTURES

To improve the efficiency of Surface Evolver - as already explained before - it was decided that the best solution would be working on an array version of the data structure. In this way, the application can take advantage of the vectorization and also the constant memory access to an element of the array when needed. The first version of this implementation was built through running loops for all the vertices, edges and faces and copy the elements data for the arrays. The loops would be done prior to every iteration of calc_force function. This process is extremely heavy and costly, which would not bring great results in terms of efficiency. The choice for this approach was made because it was the quickest way to implement and it would be important to take some initial results of using arrays and make comparative tests with the linked lists version. All application depend on the linked lists -

parsing input file, GUI and computation functions - thus also important to work with this first version to determine which were the side effects of changing a complex data structure present throughout the application execution. With the results obtained in this first version, it was easier to leave it for a more robust version that does not imply copying the linked list content to arrays at every calc_force iteration.

The second and closer to the end version, to extend the functions of insert, remove and update of application elements, thereby allowing each time a new element was added, removed or modified, to happen at the same time in the new data structure, avoiding the weight of always copying the data of linked lists for arrays on all calc_force iterations. This modification is more efficient, readable and logical to use in this specific situation.

The implementation of this version caused some problems in software functionality, making the simulations not to end in both case studies. After some research work and code tests, some problems have been discovered in the original implementation of the Surface Evolver. Contrary to what would be expected, the Surface Evolver does not update the properties of the elements through functions specific to it: the application has a get_force function that returns the force of a element, but does not have a function to update the force, i.e., a set_force function. Along the code, whenever the force of an element is updated, this is done by going directly to the element and not with a function, which makes this complex work of changing the data structure more difficult, because all the code has to be revised to find which functions are updating the force to make this also happen in the new data structure. This caused problems in Surface Evolver functionality because the force was not conveniently maintained in the two structures, requiring the rewriting of various functions that were not expected to be changed in the first place. The next step was to find all the functions that were directly updating the value of force to create the set_function to update the force on both structures and, therefore, the application is more consistent with good practices of programming. After all this work to find and change the functions, the Surface Evolver was able to execute all the simulations without errors or problems with outputs, giving consistent results.

After these changes, and according with vec-report that the ICC displays, some loops continued without being vectorized. The reason has to do with the fact that some functions present within the major loops that iterate the arrays, are still using elements present in the linked list. Despite the major loops that run the vertices, edges and faces are in contiguous memory spaces, some functions used inside the loop still use the original structure. This forced the rewriting of more functions were needed to change the code in order to use only arrays inside the loops that run all the elements.

The original version with linked lists took, in all iterations of the application, 68 seconds to calculate the force. Then, it will be shown a chart presenting the calc_force execution times using the new version of the data structure with arrays but still without vectorization.

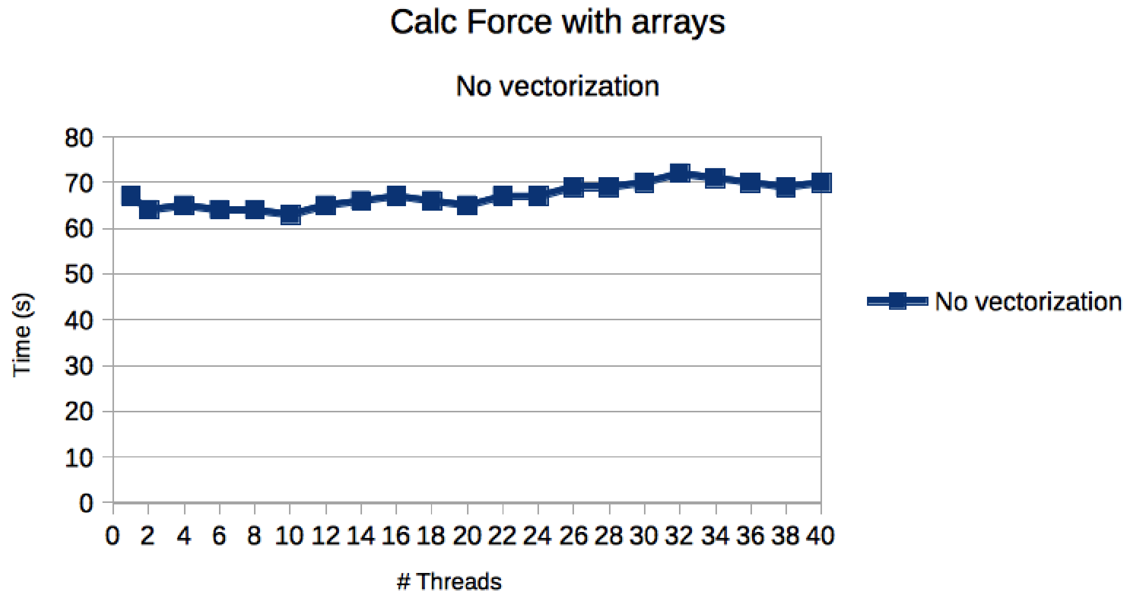

Figure 13.: Calculation force algorithm with new data structure but without vectorization - **Slow case**

In the sequential version with arrays, to calculate the force, Surface Evolver takes 67 seconds, only one second less than the version with linked lists. The difference between them – in code terms – refers to the fact that one is using arrays and the other one is using linked lists. In this chart it is not possible to compare the two versions because the original version with linked lists is not prepared to run in parallel, only sequentially, so this results are only for the new version with OpenMP version implemented, but still without vectorization. It was expected to gain some performance with the change of linked lists for arrays, because with arrays the access to an element is constant and not linear O(n). But, in fact, this did not happen, instead only a minimum gain was identified, practically insignificant. The expectation to increase performance goes to the vectorization and parallelization of main loops, which is expected to bring more encouraging results.

After rewriting all these functions, the loops were finally vectorized and some simulations could be executed to test if this vectorized version of the loops brings performance improvement. Meanwhile, tests were done to identify possible false sharing between threads(Liu and Berger, 2010). False sharing is a well-known performance issue on SMP systems, where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line and this invalidates the cache line and forces a memory update to maintain cache coherency(Liu and Berger, 2010). This circumstance is called false sharing because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access. To ensure data consistency across multiple caches, multiprocessor-capable Intel processors follow the MESI (Modified/Exclusive/Shared/ Invalid) proto-

col. On first load of a cache line, the processor will mark the cache line as ‘Exclusive’ access. As long as the cache line is marked exclusive, subsequent loads are free to use the existing data in cache. If the processor sees the same cache line loaded by another processor on the bus, it marks the cache line with ‘Shared’ access. If the processor stores a cache line marked as ‘S’, the cache line is marked as ‘Modified’ and all other processors are sent an ‘Invalid’ cache line message. If the processor sees the same cache line which is now marked ‘M’ being accessed by another processor, the processor stores the cache line back to memory and marks its cache line as ‘Shared’. The other processor that is accessing the same cache line incurs a cache miss. There are some techniques to avoid this problem of false sharing, one of them consists in adding a padding to the data structures of the elements of 64 bytes (a typical size of nowadays processors cache line), so that the different threads do not overlap an element in the cache (Liu and Berger, 2010).

4.3 VECTORIZATION

A vector or SIMD enabled-processor can simultaneously execute an operation on multiple data operands in a single instruction. An operation performed on a single number by another single number to produce a single result is considered a scalar process[1]. An operation performed simultaneously on N numbers to produce N results is a vector process (N ¿ 1). This technology is available on Intel processors or compatible, non-Intel processors that support SIMD or AVX instructions. The process of converting an algorithm from a scalar to vector implementation is called vectorization[2].

1 https://www.nersc.gov/users/computational-systems/edison/programming/vectorization/
2 https://computing.llnl.gov/?set=code&page=intel_vector

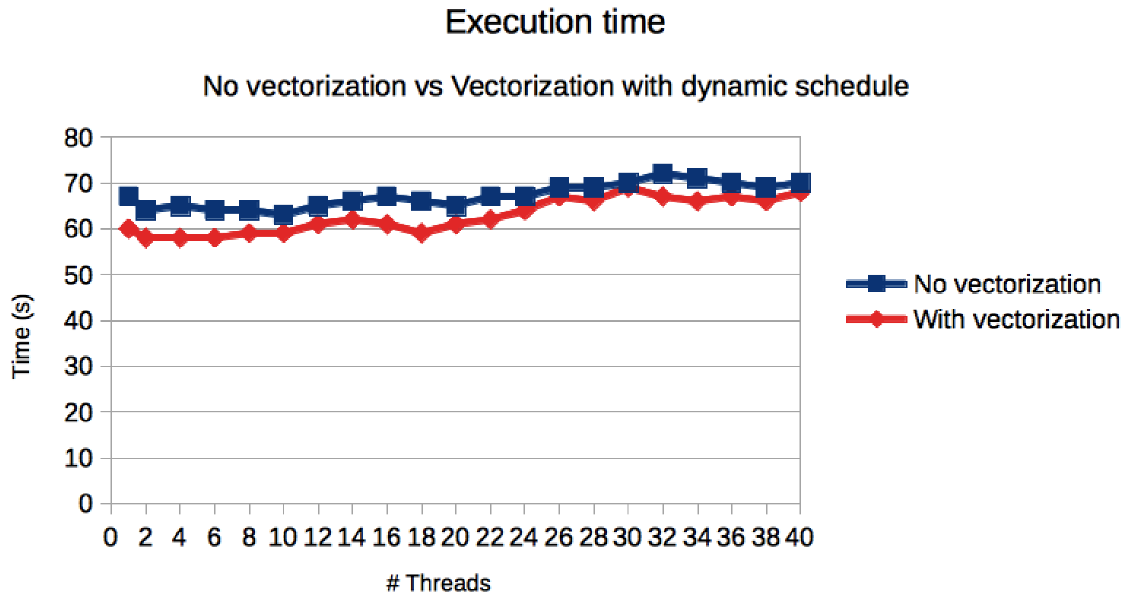

Figure 14.: Comparison between with vectorization and without vectorization using dynamic scheduling - **Slow case**

After finishing the vectorization of the three major loops, it is possible to have performance gains in Surface Evolver. As seen in the chart, there was a gain of around 13%, and this is how a function, that in this case study only takes 68 seconds running in original version with linked lists, can be considered as a significant gain. With a case study needing more intense calculations of force it is expected that vectorization still has a greater impact on application performance, bringing further improvements.

Although this case study only occupies 7% of the time to run the calc_force, this function is computationally heavy as it is a function that gets called much less than the other exclusively computational functions. And this is because the calculation of the force is only done at the end of each iterative process, while the energy - for example - is calculated thousands of times in each of those iterations.

Still, is possible to see the improvements that vectorization brought to the Surface Evolver, without looking for the parallel implementation. Vectorization improves performance because more operations are made to the same loaded data, thereby taking advantage of the AVX - these tests were run on an Intel Xeon Processor E5-2695 containing AVX - using all loaded line of 256 bits. This is a major reason for the change of linked lists to arrays, taking advantage of this line loaded by the Intel Xeon processor. Is necessary that all the data is in contiguous memory locations, which does not happen in the linked lists, because the data are scattered throughout the memory and, because of that, the vectorization is not possible using linked lists.

In resume, Single Instructions Multiple Data (SIMD).

4.4 SHARED MEMORY ON MULTI-CORE

Another goal of this work with the Surface Evolver was to take advantage of parallelization and also have improve performance through this way. With the original data structure, it was not possible to take advantage of parallelization, for, among other things, the linked lists are unfriendly to cache. Adjacent items in a list tend to be scattered in memory. With cache misses costing on the order of 100x a cache hit, this can be a significant performance issue. The specific problem with parallel programming is that traversing a linked list is inherently serial.

With this in mind, it was also important to change the data structure for arrays to be able to parallelize the loops. Thus, each iteration of major loops that run all the vertices, edges and faces, could divide the array by multiple threads and thus run simultaneous work on the same array, which is expected to bring higher performance of those loops. With the paralellized loops iterating over the array and the functions inside these same loops completely changed to the new structure, building the combination of vectorization together with parallelization, brought interesting results when was measured the performance of calc_force function.

Again, it was not possible to get more out of parallelization because, in this case study, the function that calculates the forces does not have an extremely high computational weight - that will be different with other case studies - and so the overhead of creating threads and parallelization of loops, mitigates the improvements with the parallelization. Still, the parallelization can bring positive results, although not very efficient in a cold analysis of the parallel version.

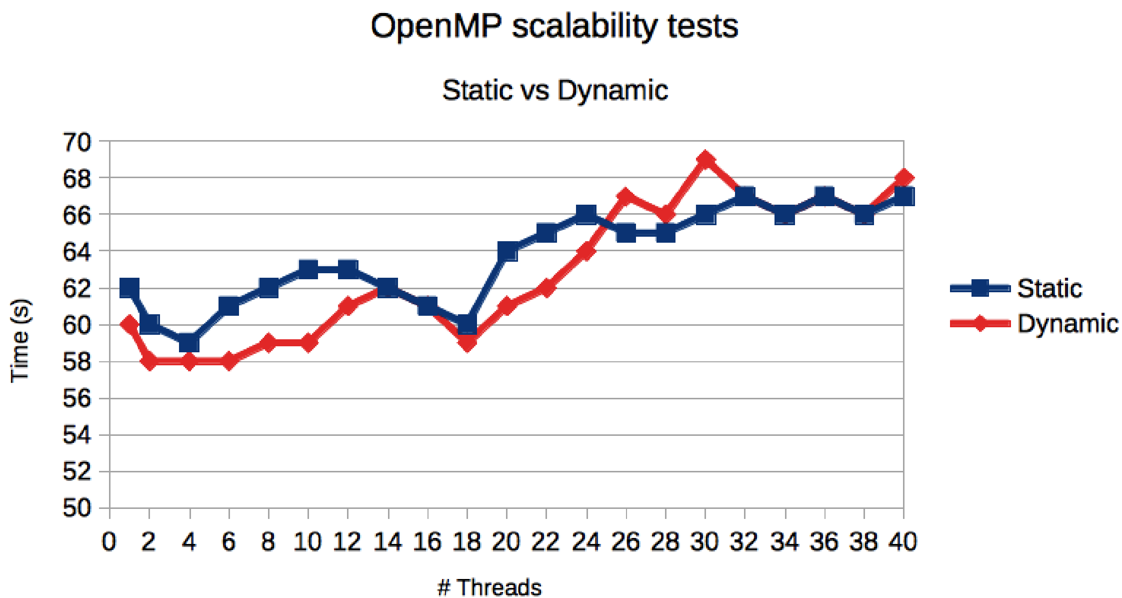

Figure 15.: Comparison between dynamic and static scheduling with OpenMP - **Slow case**

Once implemented and tested the version of OpenMP of calc_force, tests were made with static and dynamic schedules. Static schedule means that iterations blocks are mapped statically to the execution threads. With static scheduling is that OpenMP runtime guarantees that if you have two separate loops with the same number of iterations and execute them with the same number of threads using static scheduling, then each thread will receive exactly the same iteration range in both parallel regions. This is quite important on NUMA systems: if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop the same thread could access the same memory location faster since it will reside on the same NUMA node. On the other hand, dynamic schedule uses the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue.

In this case, the dynamic scheduling has better performance results and that happens by the fact that some threads receive less work to do, because several indices of the arrays are empty due to the elimination of edges, vertices and faces by refinements or functions added by the end user of Surface Evolver. As some threads have parts of the array that have less elements to work, they finish their work faster and thus are available to work with other parts of the array. This does not happen in the static scheduling, because each thread gets a percentage of specific work and in the end is still waiting for all the other threads to finish without getting back to work again.

The dynamic schedule seems at the outset a very good solution for all cases, but this is not always true. That is because the overhead of job switches between threads also has to be taken into account, which also causes delays in computing. In the case of Surface Evolver, there is a slight gain because there are some contiguous areas of the array that have enough unallocated elements making the trade-off, between work reallocation overhead and threads always working, positive.

4.5 DISTRIBUTED MEMORY TO MANY-CORE

The development of this work with Surface Evolver also aims to implement a version in distributed memory to many-core, namely the Intel Xeon Phi. With a software based in pointers, a version of Surface Evolver in native mode was not in question because Intel Xeon Phi does not allow to copy structs with pointers (Jeffers and Reinders, 2013) and, besides that, Surface Evolver has a lot of IO operations that cannot be used in this type of platform(Corporation). With these informations, the solution used for this version of Surface Evolver was a hybrid solution with large part of the application executed in CPU device and calc_force function in Intel Xeon Phi. As strictly computational function and with the new structure with elements in arrays, calc_force is the best fit for this tests with the accelerator of Intel.

Before explain the offload version of calc_force to Xeon Phi it is important to remind the algorithm of the function.

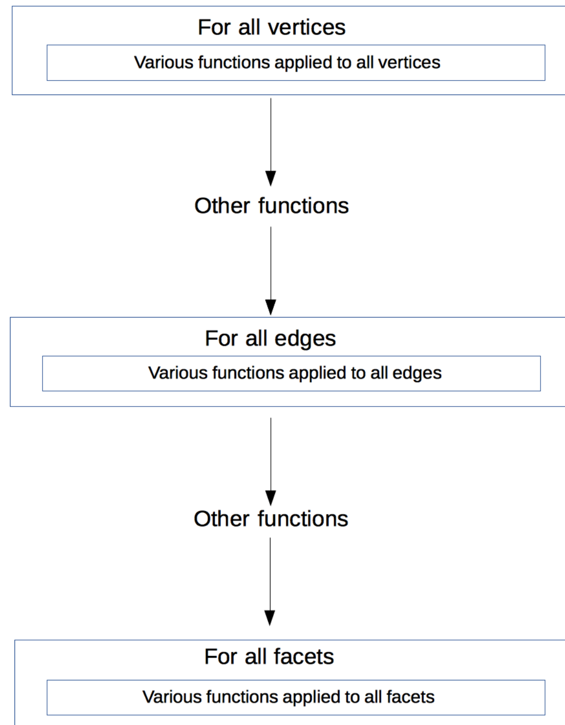

Figure 16.: Calc_force function flow

To compute the function in Xeon Phi is mandatory to copy the necessary data to the device, in this case the arrays of elements. Normally, this task of communication is the source of problems with the applications executed in Xeon Phi due to the PCIe bus - explained in Chapter 2. To get improvement of performance using this device is crucial to minimize data communications between host and coprocessor to decrease the impact of the overhead of communications between devices. Looking at the calc_force, the way to avoid those large communication overheads is to copy all the data in the beginning of the function to the coprocessor and copy data back to the host in the end of all computation, but this is not possible due to the functions between the three major loops in algorithm. To copy structs to Xeon Phi that must be bitwise copyable, and a struct is bitwise copyable if it meets some conditions, such as: members are either scalars, arrays or aggregate members that are themselves bitwise copyable, the size of the struct is the same between host and coprocessor or the offset of each member is the same between host and coprocessor; and not bitwise copyable if: contains a pointer, contains a bit-field, defines a virtual function, any of its base classes defines a virtual function, contains a user-written constructor or copy constructor or contains a "static" data member; the problem with those in-between functions is that they keep working with structs with pointers and bit-fields to identify the type of element, not allowing to run those functions in Xeon Phi (Jeffers and Reinders, 2013). The only computation present in calc_force that can be executed in the

accelerator is the three major loops because all the functions inside those loops are using the new data structure and therefore are bitwise copyable.

In this case, the version of calc_force for the Intel Xeon Phi was implemented in that way:

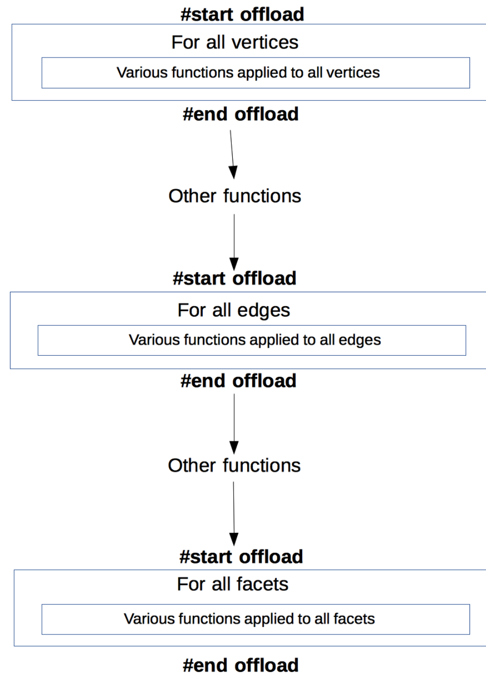

Figure 17.: Calc_force function flow with directives to Xeon Phi

This implementation implies various communications between host and coprocessor, but due to the original implementation of calc_force function is the only way to maintain the functionality of Surface Evolver.

In Intel Xeon Phi it is possible to use persistent data between offloads to maintain data in the coprocessor and avoid the weight of communication between devices, useful to use when the data does not change between offloads. This feature of persistent data cannot be used in this function because some elements can be updated in between the major loops with the additional functions presents in the code, implying the various send and receive of all arrays in every use of offload.

With this version implemented the execution times are:

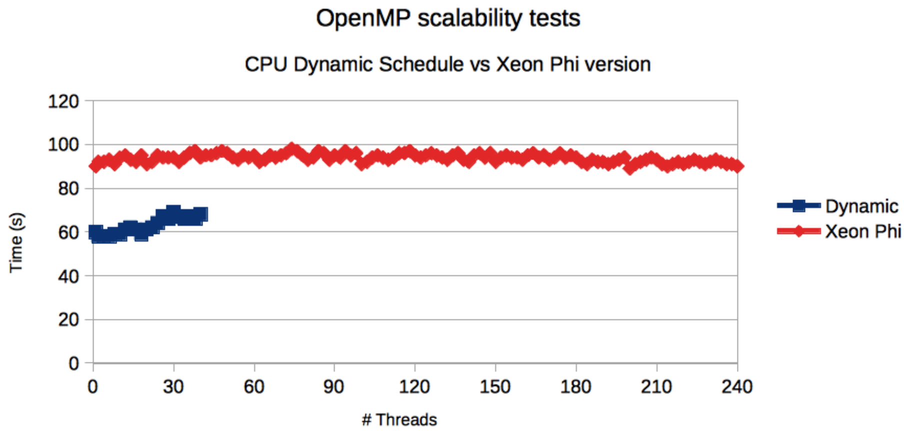

Figure 18.: Comparison tests with Xeon Phi version and dynamic schedule in CPU device

When implementation calc_force in the MIC device, this degradation of performance was expected. The overhead of communication is very high because the three arrays sent to the device have a very large size and delays all the computation, resulting in a worst execution time when compared with the version in arrays executed in CPU device. In computation terms, by the fact of this function has low weight in this simulation, the overhead of creation of the threads do not compensate the use of this type of device.

After some dig in the source of Surface Evolver to change the data structure, this results were - in some way - expected by the implementation of this function, but this work was important to develop a first version to run on MIC and take a real notion of the limitations in the architecture analyzing the results. In conclusion, it was an important work for future improvements on the application.

5

INTEGRATED APPROACH

5.1 TESTBED ENVIRONMENT

To make a fair comparison, the tests in this chapter were run on the same computing node as the tests of the original version of Surface Evolver presents on chapter 3. The node used has the following characteristics:

- 2 x Intel Xeon CPU E5-2650 @ 2.00GHz
- 8 cores each with 2-way Hyper-Threading
- L1 cache per core: 32KB for instructions and 256 KB for data
- L2 cache per core: 256KB
- L3 cache per device: 20MB (shared for all cores)
- Main memory: 64GB

5.2 UNIFIED DATA STRUCTURE

At same time of this work with calc_force function, a work with another computational function was developed by José Ricardo Ribeiro, namely with calc _energy (Ribeiro, 2015). In that work, another data structure was implemented in Surface Evolver to take advantage of vectorization, data localization and heterogeneous platforms - as the work present in this dissertation - also based in static arrays. The data structure implemented in that work has some similarities with the data structure presented in this dissertation. The data was allocated in static arrays separated by type of element and with the properties specific of each element, also with paddings to benefit the vectorization and avoid problems with false sharing, previously explained. The major difference was in the variables of control the last index inserted and the size of memory allocation, one approach with global variables and other with all information inside the struct of element.

The two solutions were analyzed and discussed to came to the best solution to fit on Surface Evolver, in terms of code maintainability, readability and performance. In performance both solutions are very

similar and therefore the choice was for the solution with better readability. The global variables to control the size of memory allocation and number of elements stored were replaced to variables inside the struct of each element and the structs were separated by type of element in files.

Each element has a file with the respective structs and functions to help on debugging and to avoid mix the code related to data with functions that do not directly interact with data structures. This was a problem with the original code, data structures of all elements and functions are mixed, which does not help on understanding the way the data was structured, situation that delayed this work with Surface Evolver.

Content of ds_vertices.h file

```
#ifndef INIT_CAPACITY
#define INIT_CAPACITY 10000
#endif

typedef struct {
    struct vertex val;
    int index;
} Vertex_Node;

typedef struct {
    int alloc_lista;
    int alloc_indices;
    int size_lista;
    int size_indices;
    Vertex_Node *lista;
    int *indices;
} Vertices_List;

extern Vertices_List vertices_list;

void initVertices_List(Vertices_List *l);
void resetVertices_List(Vertices_List *l);
void setVertices_List(Vertices_List *l, int index, struct vertex val);
void unsetVertices_List(Vertices_List *l, int index);
```

To better maintain the code in future, this approach was followed to all elements.

This unified version of data structures resulted also in a version of Surface Evolver with calc_force and calc_energy functions using the list of elements based in arrays. The two functions were also adapted to the slight changes of the structures but only for the functions inside the three major loops, that where the vectorization and parallelism can bring more efficiency to the code. This means that in the functions between the loops the original data organization of Surface Evolver was maintained.

The relation between effort and possible gains in performance do not compensate the change of those functions, taking in account the time frame to finish this changes in the software.

5.3 RESULTS DISCUSSION

With this unified version with calc_force and calc_energy using the arrays, improvements in performance are expected because in the case studies used in this work the function calc_energy occupies 60% of the execution time and, when working with heavier functions, the weight of overhead in parallelization will be less noticed.

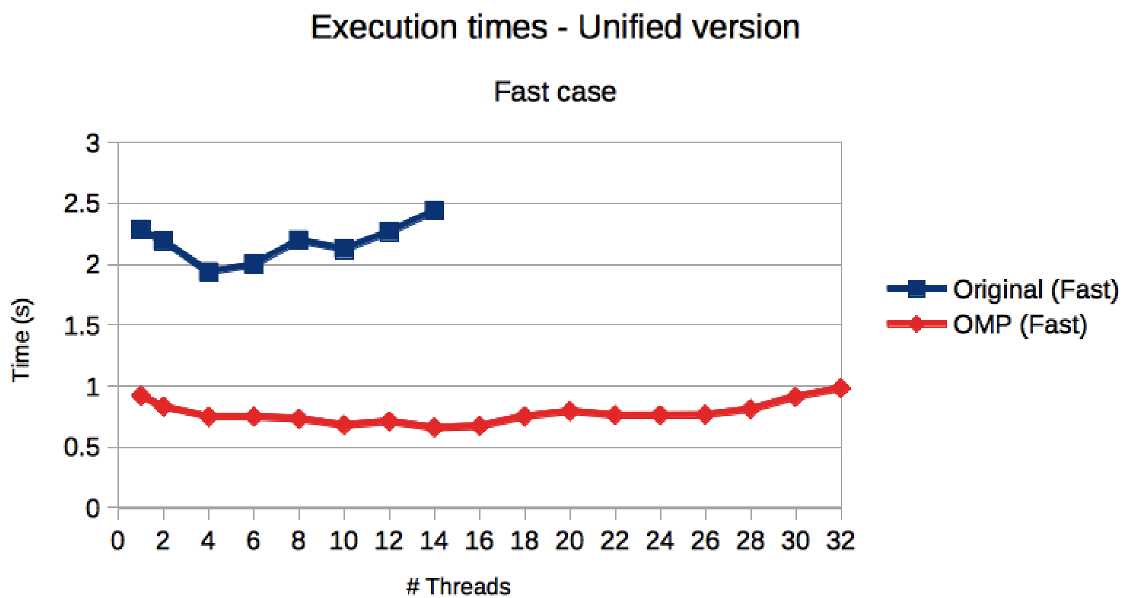

Figure 19.: Execution time with OpenMP in the unified version - **Fast case**

With the two functions using arrays and not linked lists, the execution time of the application have a considerable drop, even in the fast case. With sequential version - exemplified in graph with one thread - the drop of execution is caused by the vectorization, so is possible to see the importance of this feature. The results with OpenMP are not very positive, but this is explained with the overheads caused by parallelization in a case that last a second to execute, so it is more vulnerable to those overheads.

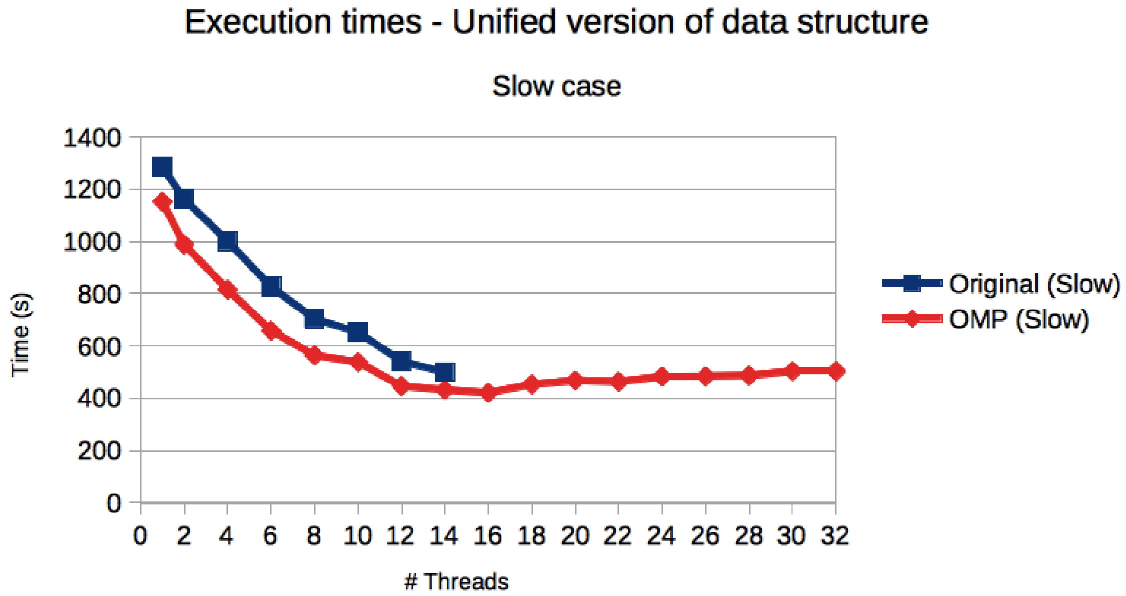

Figure 20.: Execution time with OpenMP in the unified version - **Slow case**

In slow case the vectorization result, also, in performance improvement. Grabbing the example of four threads, the original version of Surface Evolver last almost seventeen minutes to execute, with unified version this execution time drop to less than fourteen minutes. A significant improvement achieved by the vectorization and parallelization of the major loops on both functions, calc_force and calc_energy.

This is the average difference of times between original and unified version.

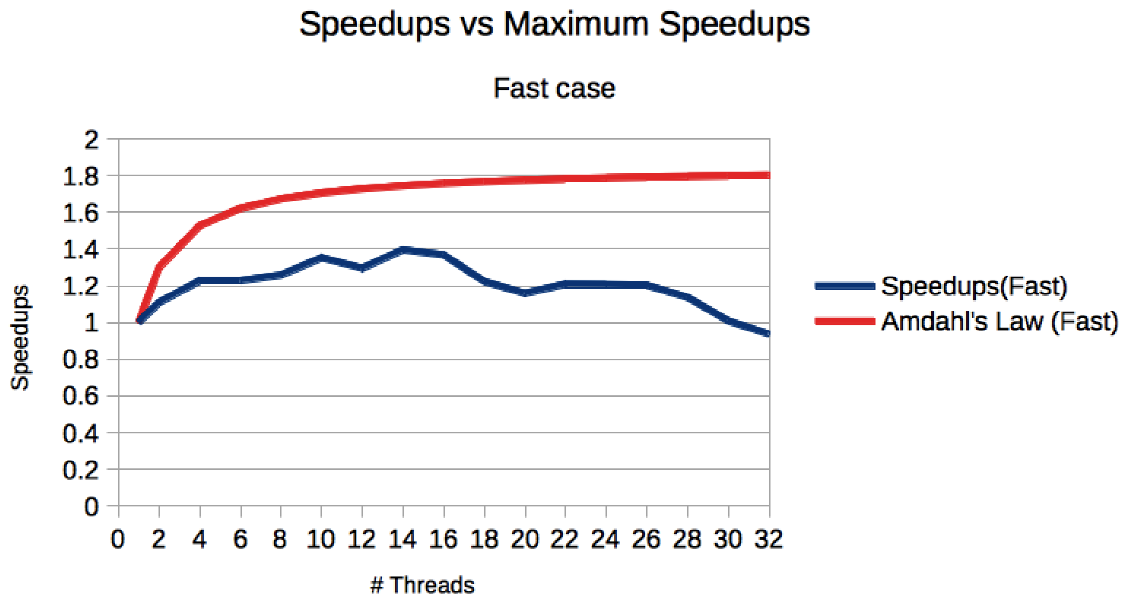

Figure 21.: Speedups with OpenMP in the unified version - **Fast case**

To compute the Amdahl's Law curve of maximum speedup that can be achieved, the factor of sequential code was 54%. This factor comes from the analysis of the call graph in fast case and the knowledge of each parallelized function, resulting in 46% of parallel code.

As can be seen, from 16 threads the performance substantial decreases and this is due to the SMT (Simultaneous Multi-Threading). These tests were executed in a node with two Intel Xeon with eight cores each, so from 16 threads they are using the same core and therefore the same caches, resulting in a degradation of performance.

The scalability of application is a little far of the maximum speedup given by Amdahl's Law, but this can be explained by the overheads of threads creation that are more noticed in an application that executes in less that one second.

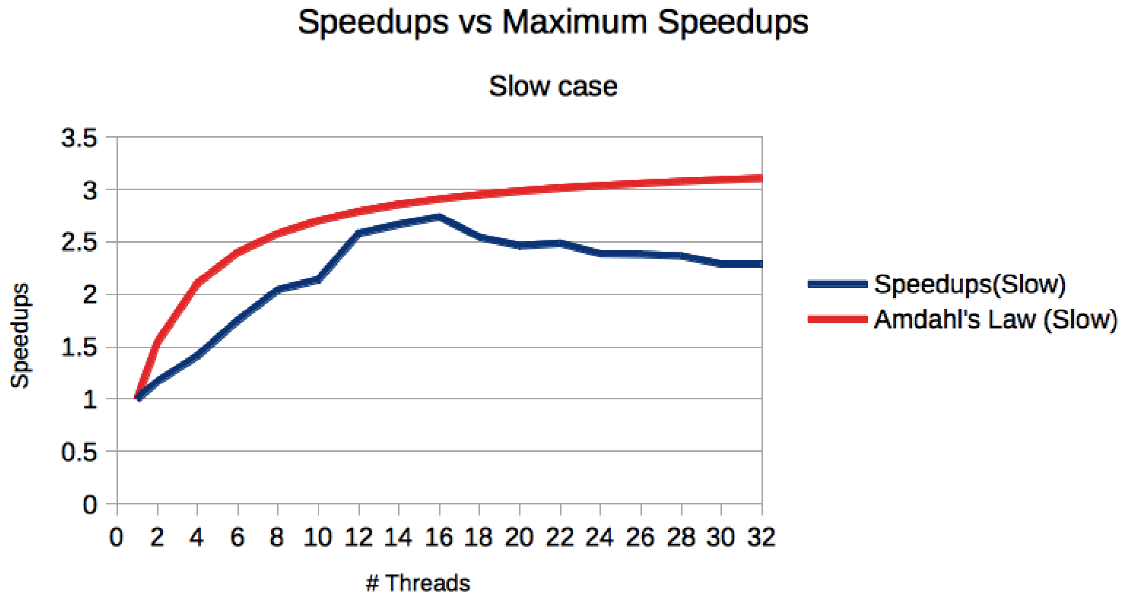

Figure 22.: Speedups with OpenMP in the unified version - **Slow case**

To compute the Amdahl's Law curve of maximum speedup that can be achieved, the factor of sequential code was 30%. In this test case, the percentage of parallel code is higher because other functions are used that were updated with parallel loops, e.g. vertex_average function.

The application scales very well until sixteen threads, always very close to the expected speedup given by Amdahl's Law. After the sixteen threads the same problem of SMT is noticed as seen in the fast case, being more prevalent in this slow case because the data stored is a lot higher, making the sharing of same caches between threads more critical and resulting in a high degradation of performance.

The efficiency of parallelization can be considered high because the results obtained are close to the expected in theorical maximum speedup, resulting in that way in a efficient parallel implementation.

In resume, the peak performance of Surface Evolver is with sixteen threads and automatically put aside any execution that use SMT.

6

CONCLUSION

Surface Evolver is an application with several years of development and with various developers, where some obsolete development methodologies that do not help the code maintenance. Not using functions to update some properties of elements is one of those cases, forcing the change dozens of functions over the code to be more readable and easier to maintain in the future. It was not an application developed with High Performance Computing in mind, and the linked lists are a good example, fact that limits the application when is intended to take advantage of new features that appear on processors. It was a long job to get into the code and understand some of the functions and algorithms, but without this task it was impossible to change the data structures and rewrite functions.

This work with Surface Evolver can be analyzed from two different points of view: scientific contribution and the results for the industry.

In the scientific point of view, the work ends with an extremely positive balance because it was possible to get results and analyses from vectorization, parallelization and many-integrated core devices. In vectorization, it was possible to have a real perception that it is something that should be further explored and looked carefully at the High Performance Computing field because it is a feature that when it is possible to take advantage of it, greatly increases the efficiency of applications that require a large computational weight. With the parallelization good results were also achieved, which were noted speedups until the use of SMT. With this feature present in the Intel processor, performance considerably drops, mainly because the two threads present in the same core share the same caches, and this is most noticeable in an application like Surface Evolver where the need for loads and stores is extremely high, working with a lot of data. With the accelerator from Intel, the Xeon Phi, this work do not take performance increases, but it was possible to take good notes for future work with the Surface Evolver, namely with the change of all data structure, which will drop the communication between the host and the coprocessor, currently the largest limitation of the Intel Xeon Phi. This was explained earlier in detail.

Still in the scientific field, it was possible to see which points to future explore in the Surface Evolver. A change to the whole data structure is one of the strongest possibilities, to take advantage of vectorization throughout the application and not only in calc_force and calc_energy. By the results obtained, it was concluded that vectorization can greatly accelerate the application and so is the point that should be explored. To be able to take advantage of vectorization, it is necessary to change all

linked lists into arrays, which will also make it possible to parallelize many more loops over the application and not only the loops that were in the updated functions. As seen in the results obtained, OpenMP was efficient and so it is also a point to explore as future work. With data structures in arrays instead linked lists, it will be possible to use the Xeon Phi to offload the most computational parts of the application - the 244 threads present in this device also can help increase application performance - and also to use the Intel Math Kernel Library (MKL) to calculate some functions and reorder the mesh of Surface Evolver. This purely mathematical library is highly optimized for this type of simulation software and is therefore also a candidate to invest some work in the future.

The starting point for future work is to change all data structures to arrays, so that the application can take advantage of three major ways explored in this work: vectorization, parallelization and accelerators. Change all data structures to arrays implies almost changing the entire application, updating the parser that processes the input file, all the computational functions and even the graphical interface of Surface Evolver, used to show the elements in the final of computation.

As this work was carried out in a collaborative project with the Bosch Car Multimedia, it is also important to analyze the results from another point of view. For the industry, these results are positive because there was a decrease of execution time maintaining the full functionality of the software. These are the most important points to show, also the fact that several paths were left open for future work and therefore the application can still be improved in the future.

BIBLIOGRAPHY

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.

Hu Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. *international Conference on Supercomputing (ICS)*, (c):353–360, 2006.

Intel Corporation. Is Intel Xeon Phi Coprocessor right for you?

Intel Corporation. A Guide to Vectorization with Intel ® C ++ Compilers. *Intel Corporation*, 2012.

Jianbin Fang, Ana Lucia Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An Empirical Study of Intel Xeon Phi. *arXiv preprint*, (Section III):137–148, 2013.

Leslie Greengard, Kenneth L. Ho, and June-Yub Lee. A fast direct solver for scattering from periodic structures with multiple material interfaces in two dimensions. *Journal of Computational Physics*, 258:738–751, 2014.

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.

Jeffers and Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, 2013.

Seid Koric, Qiyue Lu, and Erman Guleryuz. Evaluation of massively parallel linear sparse solvers on unstructured finite element meshes. *Computers and Structures*, 141:19–25, 2014.

Bo Li, Hung-ching Chang, Shuaiwen Leon Song, Chun-yi Su, Timmy Meyer, John Mooring, Kirk Cameron, and Virginia Tech. The Power-Performance Tradeo ff s of the Intel Xeon Phi on HPC Applications. *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1448–1456, 2014.

Tongping Liu and Emery D Berger. Sheriff : Detecting and Eliminating False Sharing. *Science*, pages 1–11, 2010.

Murat Manguoglu. A domain-decomposing parallel sparse linear system solver. *Journal of Computational and Applied Mathematics*, 236:319–325, 2011.

Rolf Rabenseifner. Hybrid Parallel Programming on HPC Platforms. *5th European Workshop on OpenMP*, pages 185–194, 2003.

José Ricardo Ribeiro. Improving the performance of liquid surfaces modelling in multicore devices. Master's thesis, University of Minho, 2015.

Marcin Sieniek. Fast graph transformation based direct solver algorithm for regular three dimensional grids. *14th International Conference on Computational Science*, 2014. doi:10.1016/j.procs.2014.05.092.

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.

Hari Sundar, Rahul S. Sampath, Santi S. Adavani, and George Biros Christos Davatzikos. Low-constant parallel algorithms for finite element simulations using linear octrees. *University of Pennsylvania, Philadelphia, PA*, 2007. Tech Report.

Tiankai Tu and David R. OHallaron. Balance refinement of massive linear octrees. *Carnegie Mellon University, Computer Science Department*, 2004. Tech Report.

Karthikeyan Vaidyanathan, Kiran Pamnany, Dhiraj D Kalamkar, Alexander Heinecke, and Mikhail Smelyanskiy. Improving Communication Performance and Scalability of Native Applications on. pages 1083–1092, 2014.

Chenhan D. Yu and Weichung Wang. Performance models and workload distribution algorithms for optimizing a hybrid cpu–gpu multifrontal solver. *Computers and Mathematics with Applications*, 67:1421–1437, 2014.

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.

Part I

APENDICES

CALL GRAPH

Figure 23.: General view of Profiling the **slow** case

B

PROFILERS USED AND SOFTWARE VERSIONS

The profilers used were gProf and Valgrind Tools - Valgrind, Callgrind and PAPI counters.

The versions of software used were:

- GCC - Version 4.9
- Intel Composer 2013 SP1 (package 2013.1.117)
- ICC - Version 13.0.1
- gProf - Version 2.20.51.0.2-5.34.el6
- Valgrind - Version 3.8.1
- Level 3 of optimization

C

EXPERIMENTAL SETUP

In chapter 3 and 5 were used one node with:

- 2 x Intel Xeon CPU E5-2650 @ 2.00GHz
- 8 cores each with 2-way Hyper-Threading
- L1 cache per core: 32KB for instructions and 256 KB for data
- L2 cache per core: 256KB
- L3 cache per device: 20MB (shared for all cores)
- Main memory: 12GB

In chapter 4 were used one node with the following characteristics and with one Intel Xeon Phi device:

- 2 x Intel Xeon CPU E5-2670 v2 @ 2.50GHz
- 10 cores each with 2-way Hyper-Threading
- L1 cache per core: 32KB for instructions and 256 KB for data
- L2 cache per core: 256KB
- L3 cache per device: 25MB (shared for all cores)
- Main memory: 12GB

These node has two Xeon processors with Intel Hyper-Threading functionality capable of supporting two virtual threads in hardware. Memory accesses are not uniform in these 2 processors (NUMA).

C.1 NODE TOPOLOGY

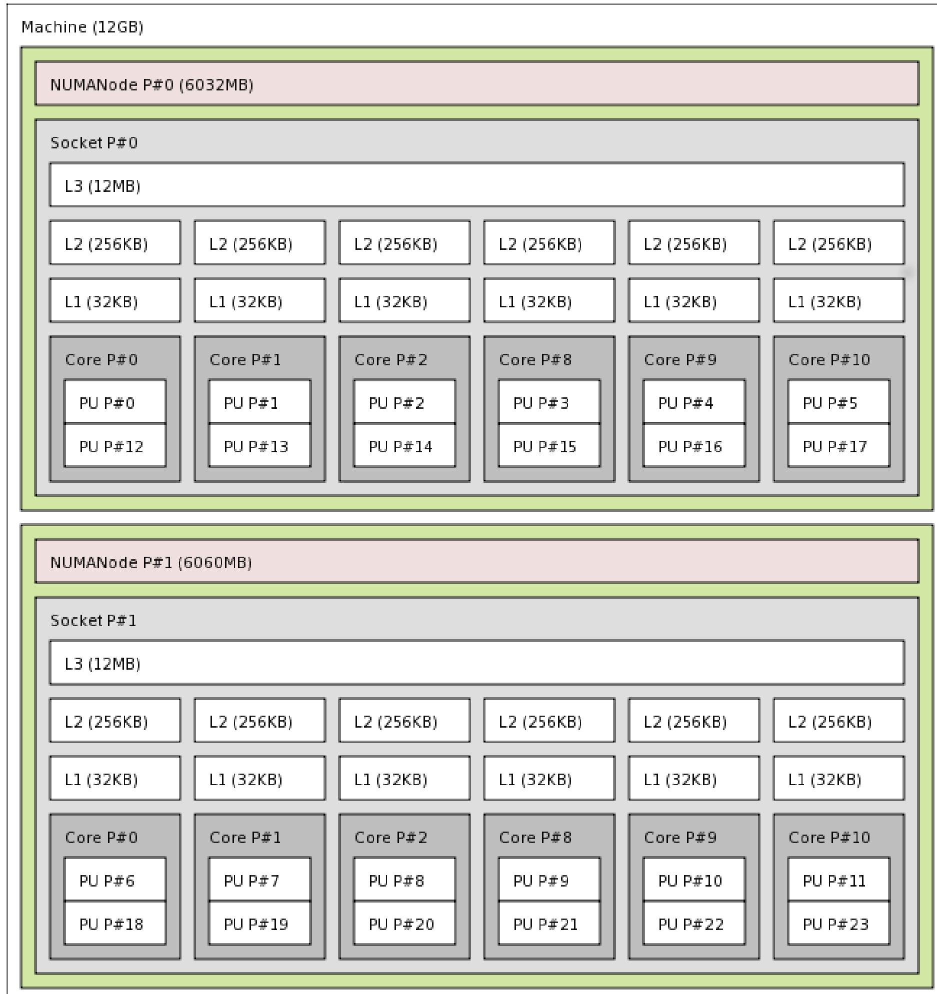

Figure 24.: Topology of used node

As mentioned, there are 2 NUMA Nodes, each one with half of the Main Memory available and each core has its own L1 and L2, with a shared L3 for all cores of each processor. This have a great impact on Memory and Processor Affinity since Processor Units from one NUMA Node often needs to access data from the other Node. Another impact is the cache conflicts that Hyper-Threading causes in L1 and L2 since these caches are shared by the 2 hardware threads.

D

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, to evaluate the maximum speedup we can obtain, we calculate according to Amdahl's Law:

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

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

Projeto HMIExcel (nº 36265/2013 (Projeto HMIExcel - 2013-2015))