

Parallelization of Plasma Physics Simulations on Massively Parallel Architectures

Luis Diego Chavarría Ledezma

A dissertation submitted for the degree of

Magister Scientiae in Computing
Computer Science Concentration

in the

School of Computing
Costa Rica Institute of Technology

Supervisor: Esteban Meneses, PhD

March 2017

APROBACIÓN DE LA TESIS

"Parallelization of Plasma Physics Simulations on Massively Parallel Architectures"

TRIBUNAL EXAMINADOR



Doctor Esteban Meneses Rojas
Profesor Asesor



Máster Luis Alexander Calvo Valverde
Profesor Lector



Máster Elvin Rojas Ramírez
Profesor Externo



Dr. Roberto Cortés Morales
Coordinador del Programa
de Maestría en Computación

Marzo, 2017

*To my Parents
and In Loving Memory of my Grandfather Memo*

Acknowledgements

First of all, I would like to thank to GOD for having me blessed with all the opportunities that He gave me in my life.

Next, I would like to thank my adviser, Dr. Esteban Meneses, for all his support and dedication to this thesis.

Also, to Iván Vargas y Esteban Zamora, who developed the original SOLCTRA, which sets the bases for this research. Thanks for allowed me to work with this application.

To Lars Koesterke for his support on the Stampede supercomputer. Thanks to him I could have access to the Stampede supercomputer, so I could run my experiments on it.

Then, I need to thank to my family, friends and colleagues who supported me on this endeavor.

And finally, I thank the reviewers of this dissertation.

Contents

Contents	5
List of Figures	8
1 Introduction	12
1.1 Motivation	13
1.2 The Problem	14
1.3 Hypothesis	14
1.4 Objectives	14
1.4.1 Overall Objective	14
1.4.2 Specific Objectives	15
1.5 The Approach	15
1.6 Contributions	16
1.7 Thesis Overview	16
2 Background	18
2.1 Massively Parallel Architectures	18
2.1.1 Multicore	18
2.1.2 Simultaneous Multi-Threading and Hyper-threading	19
2.1.3 Manycore	19
2.1.4 Clusters	21
2.2 Performance Metrics	21
2.2.1 Latency and Throughput	22

2.2.2	Speedup and Efficiency	22
2.2.3	Amdahl's Law and Gustafson's Law	23
2.2.4	Scalability	24
2.3	Parallel Programming	25
2.3.1	Vectorization	25
2.3.2	Shared Memory with OpenMP	26
2.3.3	Distributed Memory with MPI	28
2.4	Plasma Physics Applications	29
2.4.1	Epoch	30
2.4.2	BOUT++	30
2.4.3	NESCOIL	30
2.4.4	VMEC	30
2.4.5	SOLCTRA	31
3	Parallelizing a Plasma Physics Application: SOLCTRA	33
3.1	Code Cleaning	33
3.2	Implementing Distributed Memory, Vectorization and Multithreading	35
3.2.1	Implementing Distributed Memory	35
3.2.2	Implementing Vectorization	36
3.2.3	Implementing Multithreading	38
3.3	Experiments on Stampede	44
3.3.1	What is STAMPEDE?	44
3.3.2	Weak Scaling Experiment Results	45
3.3.3	Strong Scaling Experiment Results	46
3.3.4	Incrementing the Size of the Problem While Keeping the Workers Fixed Experiment Results	47
4	Executing SOLCTRA on a Many Integrated Core Architecture: Knights Landing	49
4.1	Intel's Xeon Phi Architectural Overview	50
4.2	Environmental Changes on Supercomputer	52
4.2.1	Thread affinity	52
4.2.2	Loop scheduling	53
4.3	Experiment Results	54
4.3.1	Vectorization Results	54

4.3.2	Selecting the Best Configuration for SOLCTRA on KNL	54
4.3.3	Weak Scaling Experiments Results	61
4.3.4	Strong Scaling Experiment Results	63
4.3.5	Incrementing the Size of the Problem While Keeping the Workers Fixed Experiment Results	64
5	Conclusions and Future Work	66
5.1	Conclusions	66
5.2	Future Work	67
	Bibliography	69

List of Figures

2.1	C++ example with OpenMP	27
2.2	C++ example with MPI	29
2.3	Formula to calculate the roof of the n point	31
2.4	Pseudo-algorithm for the fourth order Runge-Kutta	32
3.1	Division transformation into the multiplication of the inverted divisor.	34
3.2	Analysis results on the original code version of SOLCTRA	35
3.3	MPI implementation with particles independently distributed across MPI ranks.	36
3.4	Example of the compiler report on vectorization	37
3.5	Vectorization analysis results on the original code version of SOLCTRA	37
3.6	From AoS to SoA.	38
3.7	Implementation of the <code>_mm_malloc</code> function for memory allocation.	38
3.8	Vectorization of one loop in SOLCTRA.	39
3.9	Adding threading through OpenMP to SOLCTRA.	39
3.10	Saving the value per thread in an array.	40
3.11	Strip-mining implementation example.	42
3.12	Strip-mining implementation in the SOLCTRA application.	43
3.13	Implementing the <code>omp</code> for collapse to increase the thread count.	43
3.14	Total of threads calculation after strip-mining and <code>omp</code> for collapse	44
3.15	Experiment results of SOLCTRA on Stampede supercomputer for the weak scaling.	46
3.16	Experiment results of SOLCTRA on Stampede supercomputer for the strong scaling.	47
3.17	Results of SOLCTRA on Stampede supercomputer with fixed worker count.	48

4.1	Composition of Knights Landing’s tile.	50
4.2	Composition of Knights Landing’s mesh.	51
4.3	Distribution of four OpenMP threads according to the affinity type.	52
4.4	Schedule example for two threads and a chunk size of two.	53
4.5	Experiment results of SOLCTRA on on the Knights Landing architecture.	55
4.6	Experiment results of SOLCTRA on on the Knights Landing architecture contrasting MPI with the Threads per Core instead of OpenMP.	56
4.7	Average speedup contrasting the schedule policy.	58
4.8	Average efficiency contrasting the schedule policy.	58
4.9	Average speedup contrasting the schedule chunk size.	59
4.10	Average efficiency contrasting the schedule chunk size.	59
4.11	Average efficiency contrasting the schedule chunk size with the threads per core for only 8 MPI ranks.	60
4.12	Average speedup contrasting the affinity and the granularity.	61
4.13	Average efficiency contrasting the affinity and the granularity.	61
4.14	Experiment results of SOLCTRA on the KNL for the weak scaling.	62
4.15	Experiment results of SOLCTRA on the KNL for the strong scaling.	63
4.16	Average speedup of SOLCTRA on a KNL with worker count fixed.	65
4.17	Average efficiency of SOLCTRA on a KNL with worker count fixed.	65

Abstract

Clean energy sources have increased its importance in the last few years. Because of that, the seek for more sustainable sources has been increased too. This effect made to turn the eyes of the scientific community into plasma physics, specially to the controlled fusion. This plasma physics developments have to rely on computer simulation processes before start the implementation of the respective fusion devices. The simulation process has to be done in order to detect any kind of issues on the theoretical model of the device, saving time and money. To achieve this, those computer simulation processes have to finish in a timely manner. If not, the simulation defeats its purpose. However, in recent years, computer systems have passed from an increment speed approach to a increment parallelism approach. That change represents a short stop for these applications. Because of these reasons, on this dissertation we took one plasma physics application for simulation and sped it up by implementing vectorization, shared, and distributed memory programming in a hybrid model. We ran several experiments regarding the performance improvement and the scaling of the new implementation of the application on sumpercomputers using a recent architecture, Intel Xeon Phi - *Knights Landing* - manycore processor. The claim of this thesis is that a plasma physics application can be parallelized achieving around 0.8 of performance under the right configuration and the right architecture.

Resumen

Las fuentes de energías limpias han venido incrementando su importancia durante los últimos años. Debido a esto, la búsqueda de fuentes sostenibles se ha incrementado también. Este efecto hace que la comunidad científica vuelva sus ojos hacia la física de plasmas, en especial hacia la fusión controlada. Estos desarrollos de física de plasmas tienen que confiar en procesos de simulación por computadora antes de empezar a implementar los respectivos dispositivos de fusión. Estos procesos de simulación tienen que hacerse con el fin de detectar cualquier tipo de problema que puedan tener los modelos teóricos del dispositivo, ahorrando tiempo y dinero. Para lograr esto, estos procesos de simulación computarizada tienen que ser ejecutados en un tiempo determinado. Si no, la simulación pierde su propósito. Sin embargo, en los últimos años, los sistemas de computadores han pasado de un enfoque en el que se incrementaba su velocidad a un enfoque en el que se incrementa el paralelismo. Este cambio representa un obstáculo para estas aplicaciones. Por estas razones, en este trabajo tomamos una aplicación para simular física de plasmas y la aceleramos al implementar vectorización junto con memoria distribuida y memoria compartida en un modelo híbrido. Ejecutamos una serie de experimentos respecto al desempeño y la escalabilidad de la nueva implementación de la aplicación en supercomputadoras, usando la reciente arquitectura de Intel, el procesador *manycore* Intel Xeon Phi - *Knights Landing*. La pretensión de esta tesis es que una aplicación de plasma puede ser paralelizada, alcanzando alrededor de 0.8 de desempeño bajo la configuración correcta y bajo la arquitectura correcta.

Chapter 1

Introduction

Plasma, the fourth fundamental state of matter. Different to the other states. When matter is into this state the particles are unbounded. This state can be achieved by increasing the heat or by applying magnetic or electric fields to a gas. On this state the temperature is so high that urge the plasma to be confined by magnetic fields instead regular physically containers that could work to a solid, liquid or gas.

We can find applications for plasma physics from medicine to biology (like disinfection and cancer treatment), from propulsion in spaces and to nuclear fusion for clean energy production. Those last two applications are very relevant to our society. On the propulsion in space, we have the Variable Specific Impulse Magnetoplasma Rocket (VASIMR) developed by Ad Astra Rocket Company, which creates plasma thermal motion to convert the plasma into a directed jet. This project has great importance to power spacecraft to travel in the outer space [6].

On the other hand, on the 2016, Costa Rica was under the eye of the global scientific community with the first plasma discharge in Latinoamerica. This event was possible with the Stellarator Costa Rica 1 (SCR-1) developed by the Plasma Group of the ITCR. A stellarator is like a small nuclear reactor that confines toroidal plasma with magnetic fields. This achievement is really important on the road to achieve the "*Costa Rica carbon neutral*", since "*with one gram of this fuel, in this kind of device, it will be possible to produce until 26MWh, which could supply 80 homes for a month*" (Dr. Ivan Vargas, [15]).

As any scientific process, before the real life implementation of the plasma, the scientific or engineer has first to go for the steps of modeling and simulation of the system. These simulations of the model to implement are important because: [1]

- Real experiments could be dangerous.
- Simulation is usually cheaper than real life experiments, specially if the experiments would have to be at high scale.
- Simulation provides more controlability and visibility of the inputs and outputs of the model.
- The research could be constraint by time. This makes experimentation not feasible since simulation is faster than real life experiments.

1.1 Motivation

The development of computer systems had helped and improved the simulation process on different sciences. We can find how computers are being used on a wide spectre of areas, such as astronomical, earth and ocean sciences, space research, fluid mechanics, and of course, plasma physics. This list could go on and go on[27].

The passed years the size and the data of the simulations has been increased. On the contrary, the top speed of computers have shown a decrement in the growth. This follows a physical reason: when the processor speed is increased, so is the power consumption, reaching a point where the power is transform into heat instead of increase the frequency. This *power-performance dilemma* has pushed the computer architects to avoid increasing the speed of the processors.

However, what happens when you need to have a fence painted by the end of the day but the painter cannot paint faster? You hire more painters. Following this metaphor, the alternative to the power-performance dilemma has follow to the increment of the core count, leading to the multicore and manycore architectures that we have today.[12, 16]

But, achieving high performance based on parallel hardware like multicore or manycore architectures is not as easier as having more painters to paint a fence. This is because these architectures benefit parallel applications at the expense of serial applications, like most of the applications developed today. Because parallel development opens the door for additional difficulties like synchronization, resources management and exploding hardware features, the parallel development pushes for a wide spectrum of paradigms for parallel programming such as vectorization and shared and distributed memory.

1.2 The Problem

Plasma physics applications are math intensive applications. In table 1.1 we have the time consumption of the SOLCTRA, a software to simulate plasma physics, for a small number of particles and an estimation of the time for a high number of steps and particles.

# Particles	# Steps	Tera Flops	Serial Run-time (1 GFLOP/s)
1	0.5M	0.63	10min
10	0.5M	6.3	2h
100	2M	25	7h
1K	2M	250	3days
10K	2M	2500	30days

Table 1.1: Approximation of the execution for the serial SOLCTRA application.

In the table, we can see that for the cases when a scientist wants to simulate a high number of particles and/or a high number of steps (this will help him to achieve more accurate results) he might have to wait up to a month. And we all know that a model like that is not sustainable. For a case like that, the results might be late and for the scientist it would be faster to develop a real life experiment, even when it is more expensive or dangerous.

The challenge we face is: can we improve a plasma physics simulation application by exploring these parallel architectures? And if so, what level of parallelism can it be achieved and how much would it be the maximum improvement?

1.3 Hypothesis

On this dissertation we propose that by having the right configuration and setup, an application for plasma physics simulation can be parallelized to achieve an efficiency of 0.8 on massively parallel architectures.

1.4 Objectives

1.4.1 Overall Objective

Evaluate and understand the architectural features of massively parallel processors to accelerate plasma physics applications.

1.4.2 Specific Objectives

- Identify parallelization opportunities in plasma physics simulations.
- Evaluate architectural features in massively parallel processors Intel Xeon Phi to optimize plasma physics simulations.
- Explore mechanisms to match algorithmic and architectural features to accelerate plasma physics simulations.
- Evaluate and quantify the scalability of a parallel application on plasma physics.

1.5 The Approach

The overall objective of this thesis is to evaluate and understand the architectural features of massively parallel processors to accelerate plasma physics applications. This, by implementing vectorization and distributed and shared memory, while maximizing the speedup and the efficiency of the application.

To prove that a plasma physics application can be accelerated by implementing parallelism, we selected one application, the SOLCTRA, and within the application added vectorization and shared memory and distributed memory in a hybrid memory model.

- **Vectorization:** Vectorization is one implementation of the SIMD technique (Single Instruction, Multiple Data). This is a feature that most of new processors include at some level. We can find that some processors have more power on vectorization than others. Following in this text, we will go deep into what vectorization is and how we implemented into the selected plasma physics application.
- **Shared Memory:** Shared memory is a paradigm based where a single process spawns independent threads which share the same address space. This is intended for small tasks where the threads could be created and destroyed several times across the application execution. This is also useful when the different cores are sharing the same memory RAM.
- **Distributed Memory:** On the contrary to shared memory, on the distributed memory paradigm, several process are spawned each with its own address space. This is intended for big tasks where the process have to be permanent instead volatile like the threads.

- **Hybrid Model:** The hybrid models is when we mix the shared and the distributed memory paradigms. This is intended for cluster architectures where the application is being executed across several nodes, each of theses with more than one core.

1.6 Contributions

The major contributions of this thesis are:

- Understanding of a plasma physics code and opportunities for speeding it up with parallel computing. Part of the work of this research was to understand and study a plasma physic application to seek performances improvements through parallelism techniques, such as vectorization, shared, and distributed memory models.
- An experimental evaluation of a plasma physics code on top-of-the-line massively parallel architectures, exploring the features offered by the architecture and exposing the strengths and weaknesses of this architecture on a CPU-bound application such as the plasma physics application. We performed and evaluation and quantification of the scalability of a plasma physics application on multiple HPC architectures.
- A performance improved plasma physics simulation application for the local plasma community. The plasma physics application that we used for our research is an application of the Plasma Physics Group at the Costa Rica Institute of Technology. During the work of this research we improved the application, moving from taking hours and even days for one execution, to take just a few minutes.

1.7 Thesis Overview

The rest of this thesis is organized in chapters covering the most important knowledge areas relevant to the discussion. Beginning from the general concepts to particular results for every proposed experiment.

First of all, chapter 2 offers an introduction to the area of High Performance Computer, like massively parallel architectures, the theory of performance metrics and parallel programing paradigms, and plasma physics application.

Then, chapter 3 is intended to show the work on implement parallelism on a plasma physics application and the results of this implementation on a multicore architecture as is the Stampede supercomputer.

Once shown the results of the application on a multicore arhitecture, in chapter 4 we show the experiments results of the same application, but on a manycore architecture as it is the latest Intel Xeon Phi, *Knights Landing* (KNL), processor.

Finally, on chapter 5 we have the conclusions of this dissertation.

Background

2.1 Massively Parallel Architectures

2.1.1 Multicore

Around the year 2005 [23], the processors started to hit the "heat barrier". That means that there is a point where instead of increasing the frequency of the processor, the power is consumed by heat and cooling the processor becomes one of the main efforts and commercial concerns. On the other hand, the need to increase the processing power is driven mainly by architectural advances. This *power-performance* dilemma pushed vendors to focus on *multicore* designs. [12]

A multicore processor is a processor with multiple *subprocessors* or *cores*, where each one of these is capable of fully support at least one hardware thread [23]. The technical motivation behind this is the observation that the power dissipation of the CPUs is around the third power of clock frequency. So, by reducing the clock frequency, the vendors are able to place more than one core into the same die/package while keeping the same power envelope. [12]

When there is more than one core inside a die or package, there are some considerations to keep in mind:

- **Cache:** when there are more than one core per die, the programmer has to consider how the cache is set. Normally there are more than one level of cache, where at higher levels the caches are separated but at lower levels are shared. The cache arrangement could impact the performance of a parallel application on cases like false sharing or bandwidth impact. Also, this could have more logic regarding having to keep the cache coherence.

- **Integrated memory controller:** previously, the memory controller was out of the die. Now, with multicore designs, it was moved inside the die. This helped on reducing the latency on accessing main memory and increased the velocity of intersocket networks.

We can find this kind of processors in most of today computers, from a smartphone like the Google Nexus 6, which has 4 cores, to the latest Intel Xeon processor, which has 24 cores.

2.1.2 Simultaneous Multi-Threading and Hyper-threading

Simultaneous Multi-Threading (SMT) is a technique to improve performance on the processor. Instead on focusing on higher clock speeds which increase the size of the die and the power consumption, SMT seeks improve the performance by having multiple threads executing on one processor without context switching. Unlike other techniques, SMT maximize performance without adding significant power consumption [29] and only requires some extra hardware instead the multicore [2].

Hyper-threading is the implementation of SMT on Intel's architecture. It makes one single core appears as two logical cores to the operative system by replicating the architectural state of each thread or logical core, while sharing the physical execution resources among them [2, 26]. This increase the processor utilization and reduce the performance impact of I/O latency by overlapping the latency of one thread with the execution of the another. For the sharing resources, we can find three policies for sharing resources: partition (equal resource per thread), threshold (limited flexible sharing) and full sharing (flexible sharing without limits). [29]

The side effects of this implementation are not well-know and they will depend on each application. This is because simultaneous sharing of resources might create a potential performance degradation of the hardware.[2, 21]

2.1.3 Manycore

The next step for multicore platform is the manycore platforms. We can describe a manycore processor as a processor with so many cores that in practice we do not enumerate them; there are just "lots" of cores. This term is generally used for processors with 32 or more cores, but there is not a precise definition or an exact bound for a processor to be called from multicore to manycore processor. [23]

Another difference between the multicore and the manycore architectures, is the fact that, while the multicore architectures abound, we can find only a few manycore architectures. Basically we can

say that currently there are two main approaches: GPUs and the Intel Xeon Phi processor/coprocessor. Now we are going to describe each of these.

2.1.3.1 Graphic Processor Unit (GPU)

While a regular processor consists of a few cores optimized for sequential serial processing, a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously [24]. As its name stands for, the GPUs were originally designed for graphics processing, but they have become general-purpose enough to be used for other purposes, taking advantage of its parallelism. [23]

For programming purposes, the GPUs can be programmed using a API called OpenCL or, in case of NVIDIA, a proprietary language called CUDA [23]. The focus on programming GPUs is to have most of the application code executed in the main CPU, while having the compute intensive parallelizable application code executed on the GPU. [24].

2.1.3.2 Coprocessor

Similar to a GPU, a *coprocessor* is an external card connected, mostly, through the PCIe port that can be used for additional computational power. The application that uses a *coprocessor* can work as a native application on the *coprocessor*, where basically the application is executed completely on the *coprocessor*. Alternatively, using an *offload* approach, similar to the GPUs, most of the application is executed on the host and the compute intensive is offloaded and executed on the *coprocessor*.

Nowadays, the main coprocessor in the market is the Intel Xeon Phi. The first generation of the Intel Xeon Phi was a *coprocessor* with from 57 to 61 cores. This was in essence a manycore processor trapped in a coprocessor body.

But, the *coprocessor* has its limitations. Since a *coprocessor* is a PCIe device, it is limited to the memory inside its package and does not have direct access to the main memory of the machine. Also, since the *coprocessor* is fed by the host processor, the data transfer will be back and forth between them, making the data bandwidth a potential issue on performance. These limitations pushed the programmers to look forward for an Intel Xeon Phi *processor*. [16]

On the year 2016, Intel released the second generation of the Intel Xeon Phi code named *Knights Landing*. Because of the limitation presented, this second generation can work both as a processor and as the legacy *coprocessor*. The new processor presentation is almost entirely about

programming in the same way as we could for an Intel Xeon processor system, but with extra attention on exploiting the high degree of scaling possible in the Knights Landing.

The major difference between the GPUs and the Intel Xeon Phi is that while a GPU accelerate some applications through scaling combined with vectorization, they do not offer the programmability of a Knights Landing. Applications that show positive results with GPUs should always benefit from Knights Landing because the same fundamentals of vectorization or bandwidth are present in that architecture. The opposite is not true. The flexibility offered by the Knights Landing includes support for applications that cannot be executed on a GPU. For example, while Knights Landing supports all the features of C, C++, Fortran, OpenMP, etc, GPUs are restricted to special GPU specific models, like NVIDIA CUDA, or subsets of standards like OpenMP. Therefore, the Knights Landing offers broader applicability and greater portability and performance portability. Also, tuning and debugging an application on a GPU is rather difficult in comparison with an architecture like the Knights Landing.

2.1.4 Clusters

When we talk about *multicore* and *manycore* we are talking about cores inside the same die/package. But, what about having cores not in the same die or even in the same computer. That is a cluster. A cluster is a distributed memory system composed by a collection of systems connected by an high-speed interconnection network. On a cluster we have n nodes interconnected where each of these nodes can be any kind of processing machine, from single-core nodes to manycore machines. Nowadays, it is usual to find clusters whose nodes are multicores or manycores with shared-memory.[25]

2.2 Performance Metrics

When a programmer is working on parallel computing, it is because that programmer wants to increment the performance of his application. That is the main objective of parallel programming. But, what is performance? As we can find in [23], performance can be seen as:

- The reduction of the total time for a computation.
- Increase the rate at which a set of computations can be get done.
- Reduce the power consumption of a computation.

Performance can be measured empirically on real hardware or estimated by using analytic models based on ideal theoretical machines. Empirical measures account for real world effects, but often give little insight into root causes and therefore offer little guidance as to how the performance can be improved or what could it be limited by. On the other hand, analytic measures ignore some real world effects but give insight into the fundamental scaling limitations of parallel algorithm. These approaches also allow the programmers to compare parallelization strategies at a lower cost than going directly into the implementation.

2.2.1 Latency and Throughput

Now, we are going to present some concepts related to performance theory, starting with the basic ones, such as latency and throughput. **Latency** is the time it takes to complete a task and for this document we denote it as T . **Throughput** is the rate at which a series of tasks can be completed. While latency is measured in unit of time like seconds (s), the throughput is measured in works per unit time, and while a lower latency is better, a higher throughput is better. [23]

2.2.2 Speedup and Efficiency

Having defined the latency and throughput, we can define the speedup and efficiency. **Speedup** is the ratio of the comparison between the latency of an identical computational problem on one hardware unit or worker (also called the serial latency, depending on the consulted literature like [25]) versus that work on P hardware units (or the parallel latency). The speedup equation is defined in 2.1 [23].

$$\text{Speedup} = S_P = \frac{T_1}{T_P} \quad (2.1)$$

where T_1 the latency taken by the computational work with a single worker, while T_P is the latency taken by the same work on P workers. [23]

Efficiency measures the return of investment (ROI), and is defined by the speedup between the number of workers:

$$\text{Efficiency} = E_P = \frac{S_P}{P} = \frac{T_1}{PT_P} \quad (2.2)$$

If T_1 is the latency of the parallel program running with a single worker, the equation in 2.2 is called **relative speedup**, because it shows the relative improvement by using P workers. However, sometimes the serial algorithm is not the best to solve the problem in parallel, which makes the programmer change the algorithm for a better performer in parallel. If the T_1 in equation 2.2 is the

original serial algorithm, 2.2 is called **absolute speedup**, because both algorithms are solving the same computational problem.

Something that we as scientists have to be aware of and careful about is that using an unnecessarily poor T_1 as baseline could inflate the speedup and efficiency. Also, we always have to be clear and show both, speedup and efficiency, because while speedup can be a large number, the efficiency is a number between 0 and 1. A speedup of 100 sounds better than an efficiency of 0.1 even when we are talking about the same program/hardware [23].

Ideally, the efficiency would be 1, this is called as **linear speedup**, but it is unusual to achieve. Also, it is expected that as P increases the E will become smaller [25]. However, we could find some exceptions of applications with efficiency higher than 1. Cases like this are called **superlinear speedup**. We can find causes of this kind of behavior in cases like:

- If we use absolute speedup on cases when we are using different algorithms on the serial and the parallel applications. We have to be careful with this.
- Sometimes, the restructuring made for the parallel executions could improve the cache usage and handling.
- The performance of an application is being limited by the cache accessible for a single worker, so, adding more workers would increase the cache size accessible for the program, allowing the program to improve its performance.
- Parallelizing an applications could allow it to avoid extra work that its serialization could force.

2.2.3 Amdahl's Law and Gustafson's Law

What does make an application to not achieve a linear speedup? In the 60s, Gene Amdahl made the observation that unless virtually all of a serial program is parallelized, the possible speedup is going to be limited, no matter the number of workers available [25]. This is known as the **Amdahl's Law**. This argued that the latency of a program falls into time spent doing non-parallelizable work (W_{ser}) and parallelizable work (W_{par}). So, given P workers we can say that:

$$T_1 = W_{ser} + W_{par} \quad ; \quad T_P \geq W_{ser} + \frac{W_{par}}{P} \quad (2.3)$$

From equation 2.3 we get the speedup for Amdahl's Law:

$$S_P \leq \frac{W_{ser} + W_{par}}{W_{ser} + W_{par}/P} \quad (2.4)$$

Now, let say that f is the non-parallelizable fraction of the T_1 . By substitution we can get the speedup in terms of f :

$$S_P \leq \frac{1}{f + (1 - f)/P} \quad (2.5)$$

So, the maximum speedup achievable when P tends to infinite will be:

$$S_\infty \leq \lim_{P \rightarrow \infty} \frac{1}{f + (1 - f)/P} = \frac{1}{f} \quad (2.6)$$

And with this, the conclusion of Amdahl's Law is that the speedup will be always limited by the fraction of work that cannot be parallelized.[23]

A consideration with Amdahl's Law is that it does not take into consideration the problem size [25]. Jonh Gustafson noted that the problem size grows as computers become more powerful and as the problem size grows the parallel part grows much faster than the serial part. This is known as Gustafson's Law. So, from the equation 2.6, if f decreases, the speedup will be improved. [23]

In the end, both approaches are right. It will depend on the given problem if it is about running the same problem faster or running at the same time a bigger problem.

2.2.4 Scalability

Scalability can be defined as the ratio on where a technology can handle ever-increasing problem sizes. Applied to parallel computing, if we can increase the problem size at the same rate of the number of workers that the efficiency remains the same, then we can say that the program is scalable. [25]

Matching the Amdahl's Law and Gustafson's Law from section 2.2.3 to scalability we can find two special cases: [23, 25]

- **Strong scalability:** This is the case for Amdahl's Law. If the number of workers is increased and the efficiency is unchanged without increase the problem size, the application is said to be *strongly scalable*.
- **Weak scalability:** This is the case for Gustafson's Law. If the the efficiency is unchanged while the problem size and the number of workers increase at the same rate, the application is said to be *weakly scalable*.

2.3 Parallel Programming

On section 2.1.3 we mentioned two paradigms: shared memory and distributed memory. Based on how the hardware setup, the programmer can take one of those two approaches on parallel programming or an hybrid of both if the hardware allows it. On shared memory we can find several APIs or languages such as OpenCL, OpenMP and CUDA (for GPUs). While there can be found several choices on shared memory, on distributed memory there is one mostly used API: MPI (Message Passing Interface). Something to keep in mind is that the paradigm of shared or distributed memory is something at higher level of abstraction of the hardware. Therefore, besides one paradigm can be limited or improved by one of the types of hardware architectures seen in 2.1, they are not strictly attached to a hardware architecture in specific.

Additionally, if we have shared or distributed memory, the programmer has to be also worried about vectorization, which is a SIMD (Single Instruction Multiple Data) process where the compiler takes into account the processor capabilities and compiles certain groups of operations into vector instructions. [25]

2.3.1 Vectorization

We are starting with vectorization because it's the first step on parallelizing an application, since taking this into the application could decrease the size of the problem by using the very same hardware and the workload for each future worker.

SIMD systems are parallel systems that operate on multiple data streams by applying the same instruction to multiple data items. This could be a system with a single control unit but multiple ALUs (Arithmetic-Logic Unit) or vector processors. Vectorization applies to vector processors. [25]

A vector processor or VPU (Vector Processing Unit) is a processor capable to operate on vectors of data instead of a conventional CPU that can operate only on scalar data. To perform this, a VPU must have among other features vector registers and vector instructions to apply to those vector registers. So, in vectorization the compiler will generate assembly code to those vector instructions and registers, instead of creating assembly code to use the regular ALU [25]. For example, the Knights Landing processor supports vector register/instructions of 512 bits, allowing it to handle 16 single precision or 8 double precision mathematical operations at the same time, instead having to perform 16 or 8 separated instructions [16].

The vectorization process can be triggered by two methods. The first methods is to tell the compiler to identify vectorization opportunities. Here what the compiler will do is to identify loops

that can be vectorized. The second relies on the programmer, and it is basically to vectorize the application by explicit usage of libraries or SIMD languages directives.

In [16] we can find a methodology to achieve vectorization that we summarize next:

1. Measure baseline release build performance.
2. Determine hotspots.
3. Determine loop candidates by using compiler reports.
4. Use a vectorization analysis tool to identify the components that will benefit most from vectorization.
5. Implement vectorization recommendations.
6. Repeat the process until performance is achieved or there is vectorization left identified.

Something really important to achieve high performance on vectorization is the data layout. This is important since a bad memory layout would push the compiler for a non-optimal vectorization, requiring some extra instructions and data access that will reduce the performance. To improve the memory layout, memory alignment is required. **Memory alignment** is a method to force the compiler to create data objects in memory on specific byte boundaries. This is done to increase the efficiency of data operations to and from the processor.

The memory alignment leaves the programmer with two tasks: align the data and make sure that the compiler knows that the data is aligned through language directives.

2.3.2 Shared Memory with OpenMP

A **shared-memory** application is an application that is parallelized at the thread level. In here, the different workers share the physical address space [12]. On this paradigm the variables can be shared or private. Therefore, when developing a shared memory application, the programmer must be careful on the memory access of the shared variables and use synchronization to access them if it is required. Avoiding synchronization could create race conditions and lead applications to have non-deterministic behavior and wrong results. [25]

There are several libraries to write shared-memory code. In this document we are going to review only OpenMP since it is the one that we are going to use for this thesis. **OpenMP** is the acronym for Open Multi Processing and it is composed by a set of compiler directives. Those directives or pragmas are hints to the compiler that certain pieces of codes can be parallelized. This approach has the advantage that a well-written parallel OpenMP program is a valid program in a serial environment. This is because a non-OpenMP compiler will just ignore those directives and produce a sequential code. [12]

All the different OpenMP directives have their own options or configurations that the programmer must test and play with in order to find the configuration that fits better in the hardware/problem combination. For example, depending on if we are decomposing the problem from a task perspective or a data perspective [22], we could configure the OpenMP directive to set the affinity and distribute the threads scattered or compact through the different processors.

In any OpenMP application, a single thread runs from the start. Then, the user defines parallel regions where the master thread forks and joins a team of threads that executes the instructions inside the region concurrently.

```
double error = 0.0;
double errorFor = 0;
#pragma omp parallel for reduction(max:error) num_threads(n)
for(int i=0; i<X; i++)
{
    for(int j=0; j<Y; j++)
    {
        const int currentLinear=linear(i, j);
        grid_new[currentLinear]=calculateNewGridValue(grid);
        if(fabs(grid_new[currentLinear]-grid[currentLinear])>errorFor)
            errorFor=fabs(grid_new[currentLinear]-grid[currentLinear]);
    }
    if(errorFor > error)
        error = errorFor;
}
```

Figure 2.1: C++ example with OpenMP

In figure 2.1 we can find a simple code to solve a stencil 2D. In there we have two nested `for` loops to iterate on the matrix *grid* and calculate the new grid. Before the first `for` we have a `pragma omp parallel for`. This compiler directive tells to the compiler that the `for` of the next line has to be parallelized. This configuration will create `n` threads to work each on a round robin fashion on the different iterations of the `for`. For example, if `n=4`, the thread 0 will work with $i \in \{0, 4, 8, \dots\}$, the thread 1 with $i \in \{1, 5, 9, \dots\}$ and so on. In that `pragma` we also have an example of the `reduction` directive. In order to maintain the data integrity, as the variable `error` is shared for all the threads, the directive `reduction` treats the variable as if it were not shared by the different threads and will perform a reduction using the operator `max`, saving the *max*

value prompted to be saved in the variable.

2.3.3 Distributed Memory with MPI

Starting from what we saw on the previous section, we can say that differently to shared-memory, a **distributed memory** is an application that is parallelized at the process level. Hence, the address space is exclusive. This non-shared memory model breaks the restriction of having the different processes being executed on the same physical machine, allowing to have node-level parallelism in a cluster (see section 2.1.4).

Since the memory is distributed, the programmers need to explicitly transfer data between the different processes and nodes. This can be achieved by message passing through the OS. Today, this message passing is implemented in most of distributed memory application by using the MPI API.

MPI (Message Passing Interface) is a library that defines a set of more than 500 functions *s* [12] called from C/C++ or Fortran. In an MPI application, multiple instances or *ranks* of the same program run concurrently and use the library routines to communicate between each other by exchanging messages. From the perspective of the programmer, the place where the ranks are being executed and how the communication is between them could vary without affecting the behavior of the application. [16]

Between the set of functions that MPI includes, there are functions for operations such as send, receive, and broadcast, and more advanced operations like gather, scatter, and reduction. MPI also provides operations for process synchronization, such as barriers, and also provides options for message passing synchronous or asynchronous. An additional requirement that MPI has, is that an MPI program cannot be executed on its own, it needs of a set of wrappers for its execution. [25]

In figure 2.2 we have a code that solves the same of the code of figure 2.1, but implemented using MPI. We have to clarify that these two examples are examples and might not be fully functional. In every MPI code, we have an opening and a closure calls, which are the `MPI_Init` and the `MPI_Finalize`. Outer those functions there is no MPI functionality available. In this example, first we request for communicator size and the `rank` of the current process. These values work as the total count of process being executed and the *id* to the calling process.

Differently to OpenMP, that disperse the different iteration through the different threads, on MPI this kind of work has to be manually. In the example of the figure 2.2 we splitted the workload of the for loop into slices, so every process is going to work on a given slice. At the end we have the reduction operation that selects for the `max` error of the executions.

```

int main(int argc, char** argv)
{
    MPI_Init(argc, argv);
    int worldSize;
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    int myRank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    double myError = 0.0;
    double errorFor = 0;
    const int sliceSize = X/worldSize;
    for(int i=myRank*sliceSize; i<(myRank+1)*sliceSize-1; i++)
    {
        for(int j=0; j<Y; j++)
        {
            const int currentLinear=linear(i, j);
            grid_new[currentLinear]=calculateNewGridValue(grid);
            if(fabs(grid_new[currentLinear]-grid[currentLinear])>errorFor)
                errorFor=fabs(grid_new[currentLinear]-grid[currentLinear]);
        }
        if(errorFor > error)
            myError = errorFor;
    }
    double error;
    MPI_Reduce(&myError, &error, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}

```

Figure 2.2: C++ example with MPI

2.4 Plasma Physics Applications

Nowadays, in the plasma physics community, there is not a standard software for simulations, instead there are several simulation applications developed based on the specific requirements of the given group. We researched on the state of the art on the main applications currently used by the plasma physics community for plasma physics simulations.

Now, we are going to make a review on those applications, presenting their main functionalities and if they have been parallelized, and if so, how.

2.4.1 Epoch

EPOCH is the Extendable PIC (Particle in Cell) Open Collaboration project developed for the CCP-Plasma (Collaborative Computational Project in Plasma Physics) UK community [14]. It is a plasma physics simulation code which uses the PIC method where collections of physical particles are represented using a smaller number of pseudoparticles. It calculates the fields generated by the motion of these pseudoparticles, then uses the force generated by those fields to calculate the velocity and then the velocity to update the positions of the pseudoparticles. At the end, this works to reproduce the behavior of a collection of charged particles. [3]

Currently, this application has been parallelized by using MPI and used dynamic load balancing for optimal usage of the processors.

2.4.2 BOUT++

Also under the CCP-Plasma, BOUT++ is modular C++ 3D plasma fluid simulation code developed at York in collaboration with the MFE group at LLNL and the MCS division at ANL [14]. It has been developed from the *Boundary Turbulence 3D 2-fluid tokamak edge* simulation code BOUT. Among the features that BOUT++ has we can cite: arbitrary number of scalar and vector fluid equations solver, time integration schemes and it has implemented high-order differencing schemes, generalised differential operators. [11, 10]

On parallelization, BOUT++ can be parallelized on two dimensions. It has enabled the usage of PETSc, which is a library for partial differential equations. This library supports MPI and GPU parallelization through CUDA and OpenCL.

2.4.3 NESCOIL

NESCOIL (NEumann Solver for fields produced by external COILs) is a code for the calculation of the surface current on the exterior surface of two toroidally closed surfaces such that the normal field on the interior surface is minimized [18]. This code has been an important coil design tool for large aspect ratio stellarators.

In terms of parallelization, currently there is no public information on the state of this application.

2.4.4 VMEC

VMEC is an application, also from PPPL, for solving for MHD (magnetohydrodynamics) equilibrium configurations in magnetically confined fusion devices by finding a minimum in the total

energy of the system [19].

2.4.5 SOLCTRA

SOLCTRA is a CPU-bounded code used on development of the SCR-1 (Stellarator of Costa Rica 1) by the Plasma Group of the ITCR (Costa Rica's Institute of Technology). [31]

The SOLCTRA code was used to calculate the magnetic fields and the magnetic structure that confines the plasma produced by the twelve modular cooper coils on the SCR-1. These magnetic fields are important to define the best electron cyclotron frequency heating (ECH) system. These magnetic fields are also needed to evaluate the confinement of the device. [30]

This application was initially written in Matlab in order to draw the generated magnetic fields by the coils on a complex way. To solve this, the application uses a Biot-Savart solver and a reduced model of the twelve modular coils of the SCR-1. The application estimates the 3D magnetic field, magnetic surfaces, rotational transform profile and magnetic well. [30]

During the SCR-1 development, performance issues were found on the Matlab code related to the estimation of the magnetic fields. That pushed the team on migrate that section of the code to C++, splitting the application in two parts: the computationally intensive one in C++ for the magnetic fields simulation and calculations and the light one in Matlab for the drawing of the magnetic fields resulted on the module in C++.

To calculate the magnetic fields, the application implemented a fourth order Runge-Kutta's method based on [5] and [17].

At high level, the code first load the information of the twelve modular coils, one Cartesian point per grade. This information needed to be loaded only once per iteration and remains constant thought all program execution.

After the load of the coil information, the tool has to calculate \hat{e}_n , which is defined as the unit vector along each segment of each coil [13]. This value is calculated as the difference between the next point and the current point divided by the norm of the current point (see Fig 2.3).

$$\hat{e}_n = \frac{P_{n+1} - P_n}{\text{norm}(P_n)}$$

Figure 2.3: Formula to calculate the roof of the n point

Once we have the coil information loaded and the \hat{e}_n for every point, a loop is executed to get the Runge-Kutta calculation. This loop keeps its flow as follows the pseudo-algorithm in Fig 2.4.

Note that the roof calculation has to be made only once for all the execution. This is because the coils are fixed, hence the \hat{e}_n are same for all the Runge-Kutta calculations.

```
 $P_0 \leftarrow StartPoint$   
loop  
   $K_1 \leftarrow \text{MAGNETIC FIELD}(P_0)$   
   $P_1 \leftarrow \frac{K_1}{2} + P_0$   
   $K_2 \leftarrow \text{MAGNETIC FIELD}(P_1)$   
   $P_2 \leftarrow \frac{K_2}{2} + P_0$   
   $K_3 \leftarrow \text{MAGNETIC FIELD}(P_2)$   
   $P_3 \leftarrow K_3 + P_0$   
   $K_4 \leftarrow \text{MAGNETIC FIELD}(P_3)$   
   $P_0 \leftarrow P_0 + \frac{K_1 + 2 * K_2 + 2 * K_3 + K_4}{6}$   
end loop
```

Figure 2.4: Pseudo-algorithm for the fourth order Runge-Kutta

For every loop in 2.4, the application ends generating a point that will be saved in a file and then loaded by the Matlab module of its visualization.

Parallelizing a Plasma Physics Application: SOLCTRA

For the purpose of this research, we worked on the SOLCTRA application (see section 2.4.5). In this chapter, we will present the work on the parallelization of the SOLCTRA application and how we implemented vectorization, distributed memory and multithreading to improve the performance of the application. Additionally, we will present the Stampede supercomputer of the Texas Advance Computer Center and the results of the execution of the parallelized SOLCTRA, in the Stampede platform.

3.1 Code Cleaning

The original code was written in Matlab and then migrated to C++, but, this migration was not made keeping performance or C++ code practices in mind. It was just a code translation from Matlab to “C++”, which was more a translation to an old C. Because of that, before implementing any parallelization technique, we had to clean up the code so vectorization, distributed memory and multithreading implementation could be easier.

Besides making the code more readable and understandable, another major reason that pushed us to start with cleaning the code is that, as we can find in [32], some simple changes can help the compiler to improve the efficiency of the generated binaries.

By code exploration we found that for every iteration of the algorithm, the code was opening and closing the output file, which is an expensive way of handling I/O operations. We modified the

code from doing that, to do one `fopen` before the start the algorithm execution for every given point and a `fclose` at the end, allowing during the iterations to only have one I/O operation, the writing of the result of the iteration.

Another *Best Known Method* (BKM) from [32] is to change several divisions with the same divisor (see 3.1a) into the multiplication of the inverted of that divisor (see 3.1b). This is based on the fact that a multiplication takes fewer CPU cycles than a division.¹

```
double result, divisor;
double[n] dividend;
for i in n
{
    result += dividend[i] / divisor;
}
```

(a) Regular division.

```
double result, divisor;
double[n] dividend;
double divisorInverted = 1 / divisor;
for i in n
{
    result += dividend[i] * divisorInverted;
}
```

(b) Multiplication of the inverted divisor.

Figure 3.1: Division transformation into the multiplication of the inverted divisor.

Also, in the application we have several structures that were at global level. This is a bad software development practice, so we changed the application to make those structures local and pass them as parameters to whoever needs them.

The previous changes were made by just reviewing the code. But to get a better picture of the performance issues of the applications we had to start using the Intel VTune Amplifier. This tool is a performance profiler that assists the developer on finding issues and the bottlenecks of the code. The first time that we executed the analysis tool we got the results from figure 3.2 where the operations `pow` and `sqrt` were the ones that were taking most of the execution time. From a parallel computing perspective, these functions can be marked as noise because we cannot make

¹All code examples bellow are simplified to avoid overload them with unnecessary information.

them parallel themselves, but it does not mean that they cannot be improved. On the case of the `pow` operation, the exponent was always 2, so we changed the operation `pow(a, 2)` for a multiplication: `a*a`. For the `sqrt` operation, there was no way to improve it and that was something that had to stay in the code.

Function Stack	CPU Time: Total			Sp.. Time	Ov.. Time
	Effective Time by Utilization				
	Idle	Poor	Ok		
main	99.5%			0.2%	0.0%
RK4	99.5%			0.2%	0.0%
magnetic_field	98.8%			0.0%	0.0%
pow	31.2%			0.0%	0.0%
sqrt	19.1%			0.0%	0.0%
R_vectors	4.8%			0.0%	0.0%
save_data	0.6%			0.0%	0.0%
printf	0.1%			0.2%	0.0%
norm_of	0.0%			0.0%	0.0%

Figure 3.2: Analysis results on the original code version of SOLCTRA

After implement the changes above, we decided to start on the parallelization work as is shown on the next section.

3.2 Implementing Distributed Memory, Vectorization and Multithreading

In section 2.3 we introduced the concepts of MPI, Vectorization and OpenMP. In this section we are going to talk more deeply about these frameworks and show how we had applied them into the SOLCTRA application.

3.2.1 Implementing Distributed Memory

For the distributed memory we used MPI. On this implementation we have to split the application into two parts. The first one is going to parse the parameters, load the coil information and calculate the \hat{e}_n (see section 2.4.5). The second one is the one that iterates on the list of start points per particle and executes the Runge-Kutta algorithm for each of those start points.

For the first section of the code, since the calculations had to be made only once and the calculations were just a few, we just limited them to a single rank, which after the calculations are going to spread the data into the other ranks.

For the second section of the code, after reviewing the code, we noted that the algorithm executions for every particle are completely independent on each other. This led us to implement the MPI parallelization at the particle level. With this, when the application receives a list of start points, for each particle, the application is going to distribute them among the MPI ranks that the application has (see figure3.3). One of the reasons of implementing the MPI parallelization is that there is no need of adding any MPI communication between the ranks for this section of the code, which could lead to overhead during the execution of the application.

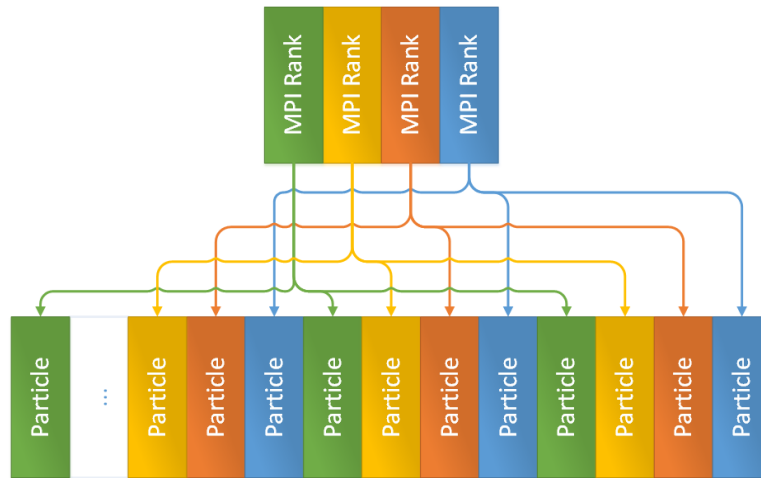


Figure 3.3: MPI implementation with particles independently distributed across MPI ranks.

3.2.2 Implementing Vectorization

To start on the vectorization we took the compiler report² and the Intel Advisor. Those tools helped us on finding issues and vectorization opportunities.

In figure 3.4 we can see that how the compiler is reporting issues on vectorization due to *unaligned access* which can decrease the vectorization performance or block it completely.

Meanwhile, on figure 3.5 we can find that we have a vectorization opportunity on the `magnetic_field` function that has not been vectorized due dependencies assumed by the compiler.

Unaligned data is one of the major constraints for vectorization, so, we changed the code to use aligned data. Above, we described the data structures that were global and we moved them to a local scope. These structures were not aligned since they were AoS (Array of Structs, see figure 3.6a).

²Created by adding option `-qopt-report-phase:vec` on compilation time.

```

LOOP BEGIN at main.c(151,5) inlined into main.c(412,5)
remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at main.c(153,9) inlined into main.c(412,5)
remark #15389: vectorization support: reference num_coil.x has unaligned access [ main.c(155,13) ]
remark #15389: vectorization support: reference num_coil.x has unaligned access [ main.c(155,13) ]
remark #15388: vectorization support: reference leng_segment has aligned access [ main.c(161,13) ]
remark #15389: vectorization support: reference vec_e_roof.x has unaligned access [ main.c(163,13) ]
remark #15381: vectorization support: unaligned access used inside loop body
remark #15305: vectorization support: vector length 2
remark #15309: vectorization support: normalized vectorization overhead 0.203
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 6
remark #15451: unmasked unaligned unit stride stores: 3
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 167
remark #15477: vector loop cost: 91.000
remark #15478: estimated potential speedup: 1.830
remark #15488: --- end vector loop cost summary ---
LOOP END
LOOP END

LOOP BEGIN at main.c(295,5)
remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at main.c(243,17) inlined into main.c(298,27)
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between Rmi.x line 183 and Rmi.x line 231
remark #15346: vector dependence: assumed ANTI dependence between Rmi.x line 231 and Rmi.x line 183

LOOP END
LOOP END

```

Figure 3.4: Example of the compiler report on vectorization

Function Call Sites and Loops	🔥	Vector Issues
⊕ [loop in magnetic_field at main.c:212]	☑	
⊕ [loop in RK4 at main.c:295]	☑	💡 1 Data type conversions present
⊕ [loop in __libc_start_main]	☑	
⊕ [loop in magnetic_field at main.c:212]	☑	💡 1 Assumed dependency present

Figure 3.5: Vectorization analysis results on the original code version of SOLCTRA

Because of that, the first improvement that we did was to attack them in order to transform them into SoA (Struct of Arrays, see figure 3.6b).

With the SoA implementation, we also have to change the way code handles the memory. Originally, the memory was allocated statically in the stack. By using this way, the developer does not have the control to allocate the memory in an aligned way. Additionally, we had not used the `malloc` function for the memory allocation since this does not have the capability to force the memory alignment required for vectorization. Instead, we used the `_mm_malloc`, which receives an additional parameter to force the alignment (see figure 3.7).

Having the data structures vectorized, the compiler report and the results of the Intel Advisor, we started the implementation of the code changes required for vectorization. For this process we

<pre> struct cartesian { float x; float y; float z; }; cartesian coil[361]; </pre>	<pre> struct Coil { double* x; double* y; double* z; }; </pre>
(a) Previous AoS implementation.	(b) SoA implementation vectorization friendly.

Figure 3.6: From AoS to SoA.

```

Coil coil;
coil.x = _mm_malloc(361, 64);
coil.y = _mm_malloc(361, 64);
coil.z = _mm_malloc(361, 64);

```

Figure 3.7: Implementation of the `_mm_malloc` function for memory allocation.

followed the code execution. Therefore, the first loop that we attacked, was on the function to get the length of each segment and the \hat{e} , at the beginning of the program, and the calculation of the vectors from each segment end points to the observation point [13]. Once having the AoS to SoA code implementation and the use of the `_mm_malloc` function, the vectorization of the \hat{e} was as simple as mark the for loop with the `pragma ivdep` macro (see figure 3.8). When we were sure that the memory was aligned, we also added a `pragma vector aligned` macro to explicitly indicate to the compiler that the memory of the array is aligned. We have to clarify that sometimes the compiler could determinate that the loop can be vectorized, but we have to add those macros to explicitly mark the given for loop and help the compiler on this task. We continued doing similar until all the innermost for loops were vectorized.

3.2.3 Implementing Multithreading

When we were done with the changes to improve the vectorization, we move on the thread parallelization through OpenMP.

```

#pragma ivdep
#pragma vector aligned
for (int i = 0; i < TOTAL_OF_GRADES; i++)
{
    Rmi.x[i] = point.x - x[i];
    Rmi.y[i] = point.y - y[i];
    Rmi.z[i] = point.z - z[i];
}

```

Figure 3.8: Vectorization of one loop in SOLCTRA.

3.2.3.1 Adding threading

The places selected to add threading with OpenMP was on the `magnetic_fields` and another function called `inside_magnetic_fields`. These two functions had a common structure with two nested loops, where the outer loop iterates per coil and the inner one on the coil information.

During this process we noted that the size of the problem per particle is defined by the quantity of coils and the information of each coil. So, it is fixed to $12 * 360$ iterations. With that, on this stage we parallelized these sets of for loops at the outer level, allowing a maximum threading of 12 per particle. At this point, it was as easy as just add the `omp parallel for pragma` above each outermost for loop inside those functions as is shown in figure 3.9.

```

#pragma omp parallel for
for (int i = 0; i < TOTAL_OF_COILS; i++)
{
    // Do something
}

```

Figure 3.9: Adding threading through OpenMP to SOLCTRA.

Also, during this process of adding threading through OpenMP, we had to do other changes to the `magnetic_fields` function. This was because, different to the other loops where the iterations are completely independent, the `magnetic_fields` function has a set of calculations that are being accumulated in a set of variables. So, in order to avoid race conditions on this variables we would have to add mutex or another synchronization method. But adding a serial code like that would impact the execution time of the code. Another option was to implement an OpenMP reduction, but this will induce an issue on having to move the variables from the cache of one core

into another. To solve this predicament, we implemented a solution showed in [32]. This solution consisted on creating a temporary array where each thread will save the results of its calculations and, when the parallel region is done, the main thread iterating on these arrays. A collateral change, we had to split the `omp parallel for` for that loop so we could initialized the variables per thread (see figure 3.10).

```
Point total = 0;
#pragma omp parallel for
for (int i = 0; i < TOTAL_OF_COILS; i++)
{
    Point temp;
    // Do something some calculation temp
    total += temp
}
```

(a) Before code splitting.

```
Point[totalOfThreads] totalPerThread;
#pragma omp parallel
{
    const int myThread = omp_get_thread_num();
    totalPerThread[myThread] = 0;
    #pragma omp for
    for (int i = 0; i < TOTAL_OF_COILS; i++)
    {
        Point temp;
        // Do something some calculation temp
        totalPerThread[myThread] += temp
    }
}
Point total = 0;
for (int i = 0; i < TOTAL_OF_COILS; i++)
{
    total += totalPerThread[i];
}
```

(b) After code splitting.

Figure 3.10: Saving the value per thread in an array.

3.2.3.2 Moving from `pragma ivdep` to `pragma simd`

Other change that we did was to change the `pragma ivdep` macro for the OpenMP `pragma omp simd` macro for two reasons: a) the `pragma ivdep` is not an standard and only applies for some compilers, and b) that according to [12] and [16], while the `pragma ivdep` is an explicit hint to the compiler, the `pragma omp simd` macro forces the vectorization. This is not without a risk. Since with this mode the compiler does not perform any check on alignment or possible data issues, the responsibility relies on the developer. With this change, we also have to add a `reduction` clause to the `pragma omp simd` per variable that was accumulating the calculations.

At this point, we started to execute the application with the Intel VTune Amplifier again after the first stage of parallelization. With this we found that every time that the `magnetic_fields` function is called, it was allocating memory for the temporal variables and it was taking significant time. Since the data was always written first on the function called at the beginning of `magnetic_fields` and then read for the rest of the `magnetic_fields` function, there was no chance to read dirty values. Because of those reasons, we move the memory allocation to an upper level of the application call stack and adding those vectors as parameters of the `magnetic_fields`. With this change we moved the application from performing four memory allocations per iteration to perform only one memory allocation per Runge-Kutta execution.

Even with the previous change, the performance related to this temporary calculation was low. Because of that, we focused our efforts on how to get rid of these vectors and the function called by `magnetic_fields`. But those temporal arrays were used only on the loops inside the `magnetic_fields` and there was not a cross index calculation (the *i*-th value was used only by the *i*-th iteration). Therefore, there was no need to calculate those values before they were going to be used. This change allowed the application to get rid of an unnecessary for loop affected by load imbalance without changing the results of the application, improving the performance of the application by around 9%.

3.2.3.3 Strip-Mining Implementation

Previously, we explained that when we added the parallelization at thread level, we did it at the coil level and that with this way, we were limited by the quantity of coils, which for the current usage of the application were only 12. But most of the processors nowadays the worker count are multiple of 2 (1, 2, 4, 8, 16, etc). So if we were going to execute the application in a hardware where the node has 8 workers, the application will have a load imbalance since for the second iteration only 4

workers will have work to do and the other 4 will have to wait those to finish. On the other hand, if we have 16 workers, what we are going to have is that 12 workers are going to be busy but 4 are going to be unused at all, which is a waste of resources.

To solve this predicament, we implemented a technique called *strip-mining*. This technique consist in transform a for loop into two nested for loops by splitting the data into *strips* or *tiles*, so the outer loop iterates on the per strip and the inner iterates inside the strip [32]. In figure 3.11 we have an example of how the original for loop 3.11a is transformed into two 3.11b.

```
for(int i=0; i<n; ++i)
{
    DoSomething(i);
}
```

(a) Before strip mining.

```
const int myTile = 100;
for(int ii=0; ii<n; ii+=myTile)
{
    for(int i=ii; i<ii+myTile; ++i)
    {
        DoSomething(i);
    }
}
```

(b) After strip mining.

Figure 3.11: Strip-mining implementation example.

In[32], the authors recommend to select the size of the tiles to be a multiple of the vector length so the strip-mining implementation will not interfere with the vectorization in the inner loop. To avoid false sharing across the different cores, we decided to not use the vector length. Instead, we decided to use the page size so every thread will be accessing its own independent page on every iteration. In our application we are working with the `double` data type, which has a size of 8 bytes, and since the page size is of 64 bytes so we are able to fit 8 values in the inner loop.

In the example 3.11 we are using a tile size of 100, but to have it working, the value of n must be a multiple of 100. If this restriction is not met, the developer would have to add an additional code to check the inner loop in order to not go beyond the size of n . For the SOLCTRA application, we had to implement this code since the n is of 360, leaving the for loops as is shown in the figure 3.12.

```

for (int jj = 0; jj < TOTAL_OF_GRADES; jj += GRADES_PER_PAGE)
{
    int top;
    if(TOTAL_OF_GRADES < jj + GRADES_PER_PAGE)
        top = TOTAL_OF_GRADES;
    else
        top = jj + GRADES_PER_PAGE;
    for (int j = jj; j < top ; ++j)
    {
        // magnetic fields main code
    }
}

```

Figure 3.12: Strip-mining implementation in the SOLCTRA application.

Where *TOTAL_OF_GRADES* is the total of grades per coil (360) and *GRADES_PER_PAGE* is the quantity of grades that fits into one single page (8). So, since the last iteration will not fully fit into a single page, we have to add a check so the last iteration will stop at the end of the data and not at the end of the page.

Currently, we have shown how this implementation affected the inner loop of the two originals. Now we are going to explain what we have changed in the outer one and how it solved the predicament of maximum 12 threads.

```

#pragma omp for collapse(2)
for (int i = 0; i < TOTAL_OF_COILS; i++)
{
    for (int jj = 0; jj < TOTAL_OF_GRADES; jj += GRADES_PER_PAGE)
    {
        // top calculation
        for (int j = jj; j < top ; ++j)
        {
            // magnetic fields main code
        }
    }
}

```

Figure 3.13: Implementing the omp for collapse to increase the thread count.

In figure 3.13, we are showing how we have used the `collapse` clause to increment the parallelism level of the application. This allows the application to add threading at a granularity level of the total of pages used, instead the total of coils. After this, we would be able to run, theoretically, a total of 540 threads (see equation 3.14).

$$total = TOTAL_OF_COILS * \frac{TOTAL_OF_GRADES}{GRADES_PER_PAGE}$$

Figure 3.14: Total of threads calculation after strip-mining and omp for collapse

3.3 Experiments on Stampede

Once we have modified the code, we started to run the application and collect data in a the Stampede supercomputer.

3.3.1 What is STAMPEDE?

Stampede is the main supercomputer of the Texas Advance Computer Center funded by the National Science Foundation of the USA. Deployed in 2012, the Stampede system is powered by Dell PowerEdge nodes with Intel Xeon E5 processors and Intel Xeon Phi (Knights Corner) coprocessors, the first generation of processors based on Intel's (MIC) architecture. [28]

In this section we describe only the host architecture of the nodes. This is because at this stage of our research we were not focused on the Stampede's coprocessors.

Each of the nodes consisted of two Intel Xeon E5-2680 of the Sandy Bridge family with 32GB of memory. These processors or sockets have a base frequency of 2.7GHz and each has 8 cores. Even though these processors includes the hyper-threading technology, the Stampede's nodes have the feature off. An additional feature of these processors to our concern is that they include the AVX instruction set. This instruction set allows the user to use vector instruction of 256 bits wide, in our case to execute 4 operations at the same time with the proper vectorization.

In the experiments to follow, in order to have more reliable values, we executed least 10 times each configuration. So, the results that we are going to show are the average of those executions. The reason of using 10 is because our queue size on Stampede allowed us to run only 10 executions simultaneously.

For the experiments in Stampede we are setting up only three variables:

- **OMP_NUM_THREADS:** This is an environment variable used to define and control the quantity of OpenMP threads executing every `parallel` region.
- **Number of ranks:** The number of ranks is set by the `-n` parameter of the MPI execution (`mpirun` or `mpiexec`, and `ibrun` in the cases of Stampede).
- **Particles:** The number of particles to simulate is set in the SOLCTRA application by the `-particles` parameter. This is the only variable that we are going to set to modify the size of the problem in all our experiments.

3.3.2 Weak Scaling Experiment Results

The first experiments we present are on the frame of the weak scaling. How we explained in section 2.2.4, the weak scaling is when the size of the workers increase as the same ratio as the size of the problem is increased.

For these experiments, we “played” with the different configurations of MPI and OpenMP, changing the quantity of ranks and threads to analyze the behavior of the application. We changed variables in exponents of 2. For the MPI we used the set of $\{1, 2, 4, 8\}$ and for OpenMP the set of $\{2, 4, 8, 16\}$. We did not analyze the combination of `MPI=1` and `OpenMP=1` since it is the serial time. Instead, we used the average of that configuration as the T_0 . On the problem size, the number of particles, for each experiment, we configured it equal to the MPI rank, so both are increased as the same ratio while the particles per rank remain fixed.

In figure 3.15a we have the graph for the speed up achieved by the application on different MPI/OpenMP configurations. We can see that for all the OpenMP configurations the line is incremental as the MPI is incremented. Also, we can note on those lines that higher the OpenMP, higher the speedup, except for the OpenMP of 16. For that case, we can see that it follows a similar increment as the other lines but with a speedup lower than even the OpenMP of 2.

In figure 3.15b we have the results for the efficiency. In this figure we can see that overall the four lines follow the trend to be flat. The exception for that is for the lines of OpenMP set to 2, 4 and 8. In those cases we can see that there is an increment when the MPI pass from 1 to 2 for all the OpenMP. Also, similar to the speedup, we can see that there is a big difference between the OpenMP of 16 with the other configurations.

So, what happens from the OpenMP thread count of 8 to 16 to see this downfall in the performance that does not apply from 2 to 8? On section 3.3.1 we described the Stampede supercomputer. There, we mentioned that every computing node has 2 processors with 8 cores for 16 cores per

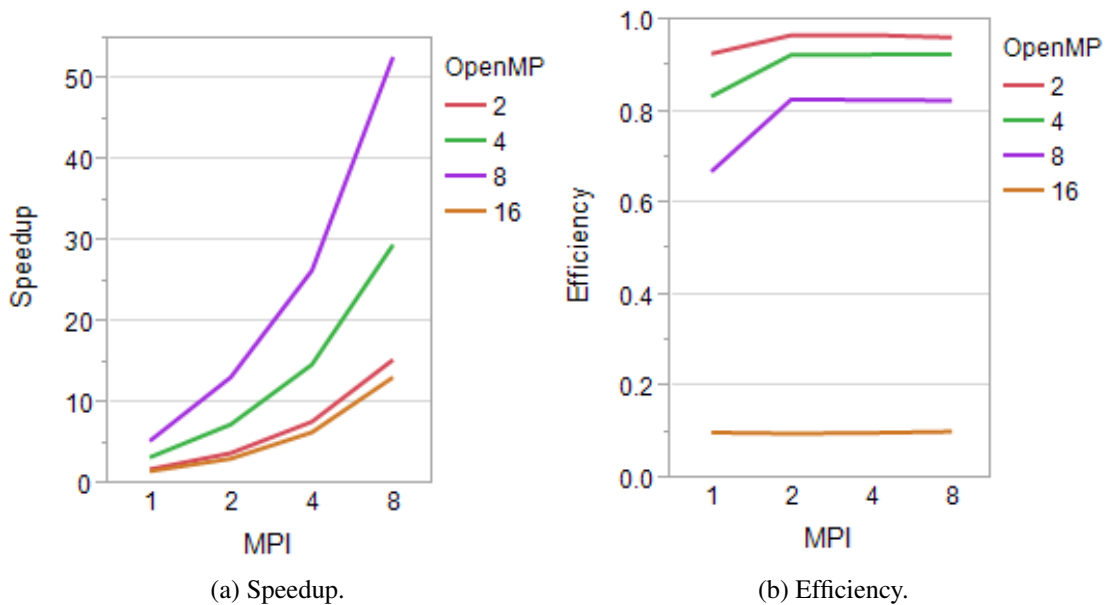


Figure 3.15: Experiment results of SOLCTRA on Stampede supercomputer for the weak scaling.

node. When we have a OpenMP thread count equal or lower than 8, all the threads remains in the same processor, but when the thread count is 16, the application has to use both processors for the same rank. This inter processor communication has a high impact on the performance. Other research [4] found there is a performance degradation when threads are spread across different sockets on the same node. Our experiments show the same phenomenon, which may be due to the cost of synchronization between threads on different processors.

From these experiments, we can say the the application is weakly scalable in the Stampede architecture while the threads remain in the same socket.

3.3.3 Strong Scaling Experiment Results

After the results of the weak scaling, we moved to run the experiments on the strong scaling. We saw in section 2.2.4 that strong scaling is when the size of the problem is fixed to a certain value while the number of workers is increased.

For these experiments on strong scaling we are going to use the OpenMP only on the values from 1 to 8. But again on the analysis we did not include OpenMP to 1 because that is the T_0 . We did not ran experiments for OpenMP set to 16 because we already saw in section 3.3.2 that for that case the application is not weakly scalable. On the problem size, we fixed it to 8 and then execute

all the configurations.

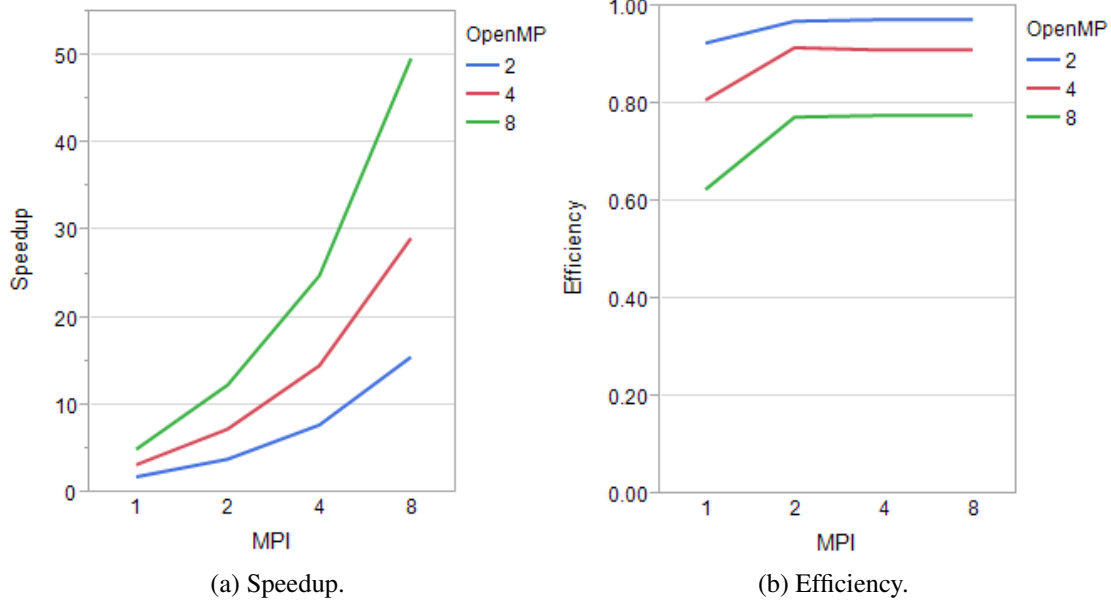


Figure 3.16: Experiment results of SOLCTRA on Stampede supercomputer for the strong scaling.

In figure 3.16a we have the speedup for the strong scaling results. In there we can see that the speedup increase at a ratio close to the number of MPI ranks for each OpenMP thread count. On the efficiency side (figure 3.16b), we can see an increment of the efficiency when we pass from 1 to 2 MPI ranks, and then it remains almost unchanged for the other MPI ranks.

Since the efficiency is increased or flat when we increase the number of MPI ranks while having the size of the problem fixed, we conclude from these experiments that the application on this architecture is strongly scalable.

Additionally, we can note that the speedup and efficiency figures for weak scaling are quite similar to the figures for strong scaling with weak scaling lines a little bit higher than the strong scaling lines.

3.3.4 Incrementing the Size of the Problem While Keeping the Workers Fixed Experiment Results

We tried an additional experiment where, differently to strong and weak scaling, we fixed the workers configuration while increase the size of the problem.

For these experiments, we selected the configuration of 2 MPI ranks and 8 OpenMP threads per rank. We chose this configuration because the OpenMP=8 was the one that showed better balance between high speedup and high efficiency for both weak and strong scaling. In order to maximize the resources for execution and to fill up a whole node, we chose MPI=2. Then we executed the application with the particle's count from 2 to 32.

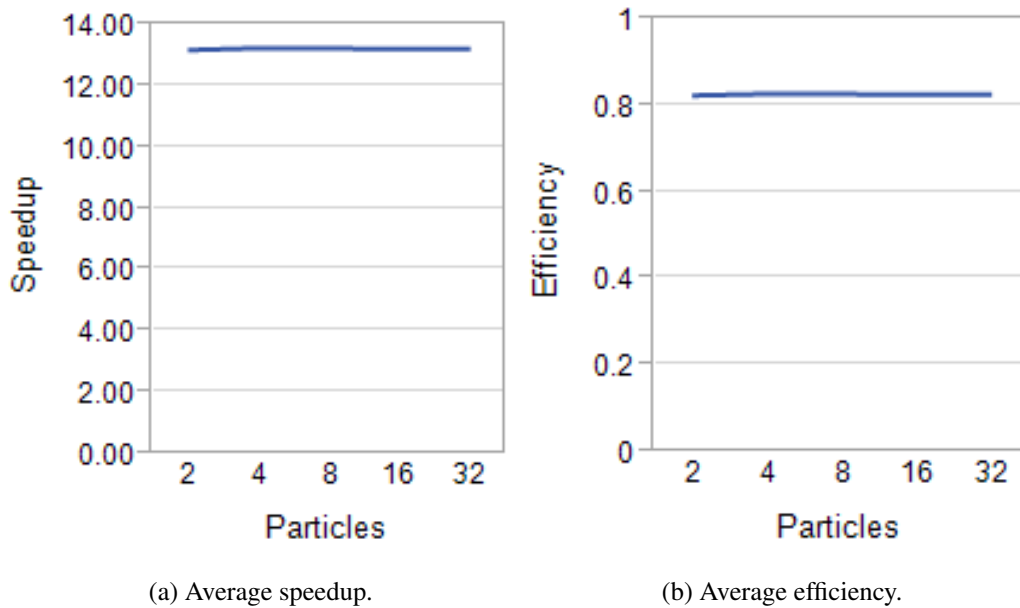


Figure 3.17: Results of SOLCTRA on Stampede supercomputer with fixed worker count.

On figure 3.17 we have the speedup (3.17a) and the efficiency (3.17b) results for this set of experiments. In there we can see that both remains linear from 2 particles to 32, from 1 to 16 per MPI rank. That was what we were expecting: to keep the speedup and the efficiency unaffected as the size of the problem got increased.

After these sets of experiments, we can conclude that this plasma physics simulation application is strongly and weakly scaleable while the threads remains in the same socket, achieving a performance higher than 0.8 and that this performance remains as the problem size is increased.

Executing SOLCTRA on a Many Integrated Core Architecture: Knights Landing

On chapter 3 we showed the parallelization of SOLCTRA and the results of the application on the Stampede supercomputer. After having a successful test on a multicore system, the next step was to test the SOLCTRA application on a manycore system. For this experiments, we selected the second generation of the Intel's Xeon Phi processor codenamed *Knights Landing* (KNL).

The main reasons of this selection was based on:

- Top of the line architecture: The Intel Xeon Phi Knights Landing is the latest processor in the many-core line. It was released in July of 2016. According to the latests top 500 list released in November, the top 5th supercomputer in world, Cori, was made using KNL [20].
- Portability: The Knights Landing is binary compatible with prior x86 processors. That means, that any application executed for a regular processor can be executed on a Knights Landing with no change to the executable at all. So, programming for the new Knights Landing is as easier as programming for any x86 architecture. [16]

4.1 Intel's Xeon Phi Architectural Overview

In section 2.1.3 we introduced the concept of a many-core architecture and in section 2.1.3.2 specifically the Knights Landing. In this section we are going to do a deeper review of the architecture of the Knights Landing manycore processor.[16]

The Intel Xeon Phi Knights Landing is the latest processor of the the Intel's Many Integrated Core (MIC) architecture family, which is the name used by Intel to identify its manycore processors and coprocessors. The previous product of this family was the Xeon Phi codenamed *Knights Corner* (KNC). In section 2.1.3.2 we introduced this coprocessor. In that section we also mentioned that it has several limitations like being constrained to the memory inside the package and the bandwidth of the PCIe connection with the host resources.[16, 8]

Differently to the Knights Corner, the Knights Landing is a standard standalone processor. Hence, it does not have the constraints related to memory and bandwidth of of being inside a PCIe package.[16]

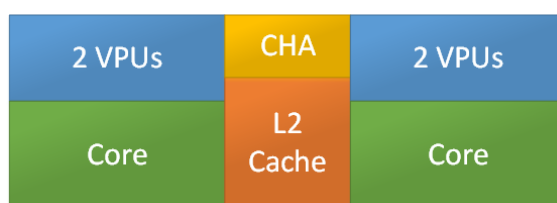


Figure 4.1: Composition of Knights Landing's tile.

Architecturally, the Knights Landing includes the concept of *tile*. Each tile includes two cores, a cache level 2 with 1MB, shared by the two cores, and a structure to manage the distributed tag directory called CHA (Caching/Home Agent) (see figure 4.1). Every Knights Landing has 38 tiles but only between 32 and 36 are enabled due to manufacturing restrictions (between 64 and 72 cores per Knights Landing). Each Knights Landing's core includes two VPU (Vector Processing Unit) and four hyper-threads (see section 2.1.2). To communicate every tile with other tiles or other resources, the Knights Landing implements a cached-coherent two dimensional (2D) mesh interconnect. This provides higher bandwidth and lower latency vs the 1D in Knights Corner (see figure 4.2). [16, 8]

The Knights Landing introduced the AVX-512 (512-bit Advance Vector Extension) ISA which will be also on the next Xeon family codenamed *Skylake*. This AVX-512 supports vector instructions of 512 bits wide, differently to the previous AVX2, which only supports vector instructions of 256 bits. This increment of the vector instructions marks an improvement of 2x only on the vectorized side of the applications. [16]

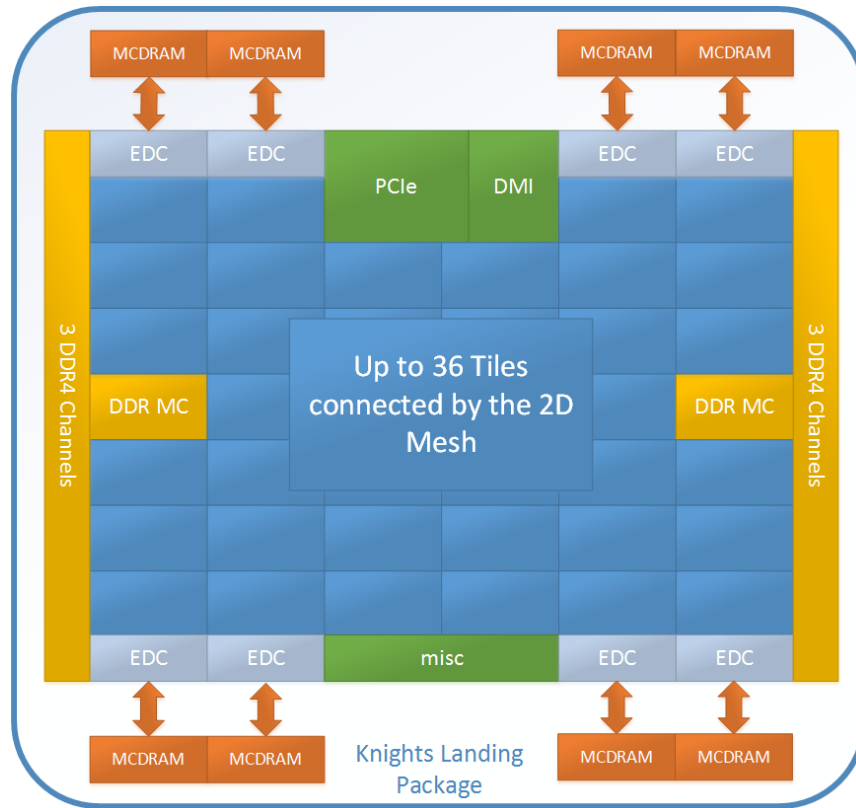


Figure 4.2: Composition of Knights Landing's mesh.

The Knights Landing processor has two types of memory, the MCDRAM for high bandwidth and the DDR for large capacity. The MCDRAM is integrated on the Knights Landing package with 16GB of capacity. These MCDRAM have their own memory controller called EDC. This can be configured at boot time to be used as cache of the DDR, to be an extension of the DDR (sharing the same address space), or a hybrid, where a percentage can be used as cache and the remaining as flat. The Knights Landing has two DDR4 memory controllers, allowing to handle until 384GB of DDR memory. [16, 9]

The Knights Landing also has other important features like the Omni-Path fabric for high speed intercommunication, but we have are not going to go deeper into it since they are not relevant for the purpose of this research.

4.2 Environmental Changes on Supercomputer

Opposed to Stampede which is a multicore architecture, with the Knights Landing we are moving from a configuration with 8 cores and threads per package, to a configuration of 64 cores and 256 threads per package. This change opens a spectrum of possible configurations, beyond the ones described in section 3.3.1 which might have an impact on the performance of our application.

Next, we are going to describe some additional settings that we changed to improve the performance of SOLCTRA on the Knights Landing.

4.2.1 Thread affinity

This is a configuration set through the environment variable `KMP_AFFINITY`. This variable sets how the OpenMP threads are distributed across the cores, and how they are distributed and bound inside each core. This is one of the more complex environment variables. This is because, differently to most of the variables related to OpenMP, the variable has several arguments to be set. For our experiments we only care about the `granularity` and the `type`:

- **Granularity:** the granularity sets if each OpenMP thread (software thread) is bound to a thread of the core (hardware thread) or if it will be attached to the core but not to a specific thread inside that core. To have them attached to a thread inside the core, the variable has to be set to `fine` (or `thread`), and if we want them to move across the threads in the core, it has to be set to `core`.
- **Type:** the type sets how the threads are distributed across the different cores. For our experiment we only used `scatter` and `balanced`.

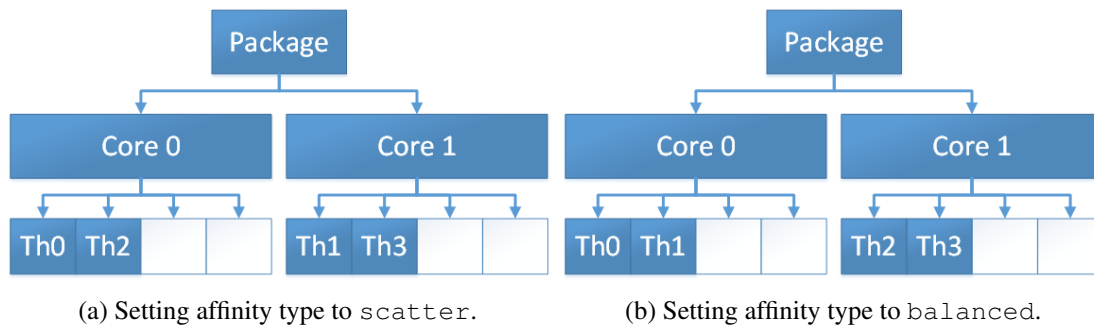


Figure 4.3: Distribution of four OpenMP threads according to the affinity type.

In figure 4.3 we have the distribution of four threads on a hypothetical hardware with two cores and four threads per core. For the `scatter` affinity (4.3a), the threads are going to be distributed evenly across the cores as possible, following a round-robin distribution. Meanwhile, for the `balanced` affinity (4.3b), the threads are going to be balanced on the cores with continuous threads closed to each other, following a block distribution.

4.2.2 Loop scheduling

The loop scheduling defines how the different loops iterations are being scheduled across each OpenMP thread in a `parallel for` region. It can be defined inside the code or through the environment variable `OMP_SCHEDULE`. It has two parameters, the schedule policy and the chunk size.

The chunk size is described as the number of loop iterations that each thread is going to execute on each of its iterations on the `for` loop. For example, if this value is set to 2 and we have 2 threads, when the thread 0 starts the loop, it is going to execute the first 2 iterations, then the thread 1 is going to execute next 2 loop iterations, then the thread 0 will execute the next 2 loop iterations, and so on (see figure 4.4).

# Thread	Loop Iteration
0	0
	1
1	2
	3
0	4
	5
...	...

Figure 4.4: Schedule example for two threads and a chunk size of two.

For our experiments we used the policies `static`, `dynamic`, and `guided` described as follow:

- **Static:** this divides the loop into chunks of similar size as much as it can across the threads. This is defined statically before the loop starts. This is intended for cases when the programmer expects that the iterations will have similar workload.

- **Dynamic:** this distributes the chunks in a FIFO manner. Every time a thread finish its given a chunk, it will continue with the next chunk. This is defined dynamically as the loop is being executed. This is intended for cases when the iterations workload are going to be unbalanced.
- **Guided:** similar to `dynamic` but the chunk size is going to be decreased as iterations are executed in order to improve the load balance across iterations.

4.3 Experiment Results

For our experiments, we used the infrastructure of the CENAT (*Centro Nacional de Alta Tecnología*). There, we have available four servers, each with one Intel Xeon Phi 7210, Knights Landing. Each of these processors has 64 cores, with a base frequency of 1.3GHz. [7]

4.3.1 Vectorization Results

The first experiment we executed on KNL were related to the vectorization. We decided to start with the vectorization because this is one of the features to improve the performance that the KNL provides over other architectures.

To have a fair comparison on these results, we had to isolate the vectorization from other variables and make it independent of the OpenMP and the MPI configurations. To achieve this, the experiments that we ran were with the basic configuration of only 1 MPI rank and only 1 OpenMP thread.

Additionally, we took the same source code and generate two executable, one with vectorization on and another with the vectorization off, and then execute them both in the KNL. To turn the vectorization off, we added the options `-no-vec` and `-qopenmp-stubs` to the compiler. We turned off also the OpenMP because inside the source code we are using the `omp simd` pragma.

The results we got from this two executions were that while the executable without vectorization last an average of 829.28s, the one with vectorization last an average of 97.93s. This results showed us that only on the vectorization side of features that the KNL provides, we can get an improvement of 8.47x.

4.3.2 Selecting the Best Configuration for SOLCTRA on KNL

Because in a manycore architecture such as KNL we have more variables to configure than a multicore architecture as Stampede, we decided to select the configuration that would fit best for use

before run any strong or weak scaling experiment.

In chapter 3 we saw that the 8 OpenMP thread per MPI rank was the configuration that allowed us to achieve the peak performance of the application on Stampede. Also, we concluded that it was because when we cross that number, the threads have to be executed across sockets. In the KNL that should not be the case. We do not mean to say that at 16 or more threads per rank the scaling will remain. This is something that is still subject to experiment. What we say is that for the KNL there is not cross socket penalty as on Stampede when there are more than 8 threads per rank. Because of that, we decided to use from 1 to 8 MPI ranks to explore how the application performance behaves on this architecture when we “play” with those other variables. Also, we wanted to keep each experiment execution contained into one KNL. So, when we have from 1 to 8 MPI ranks, we will have from 8 to 64 cores per each MPI rank.

On the OpenMP thread count we decided to configure it according to the MPI rank count, so that we could have 1, 2 and 4 threads per core. One example will be that if we have 4 MPI ranks, to have 1 thread per core the OpenMP thread count must be configured in 16, to have 2 threads per core, it must be in 32 and go on.

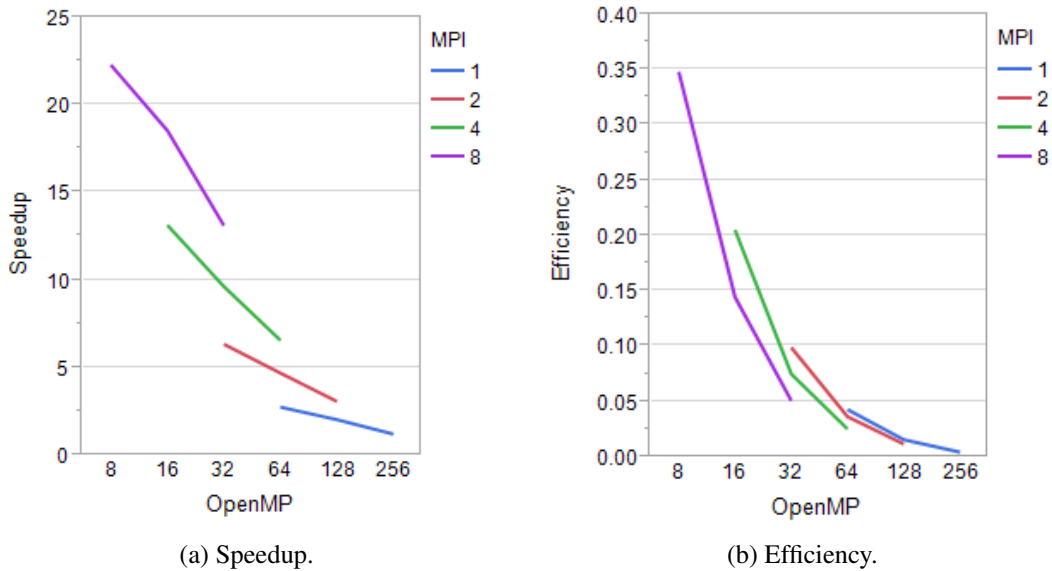


Figure 4.5: Experiment results of SOLCTRA on on the Knights Landing architecture.

The overall results for the different configurations that we executed are shown in figure 4.5a for the the speedup and in figure 4.5b for the efficiency. Similar to the experiments on chapter 3, these results are the average from at least 10 executions of each configuration.

In these figures, we can see that x axis alignment (OpenMP thread count) varies according to the quantity of MPI ranks. This is expected since the ranges of OpenMP thread count varies with the MPI ranks. But, in order to facilitate the analysis of our results and to normalize the x axis, we decided to use the thread per core as a variable, instead the OpenMP thread count. After this change, the overall speedup and efficiency of the experiment results are shown on figures 4.6a and 4.6b.

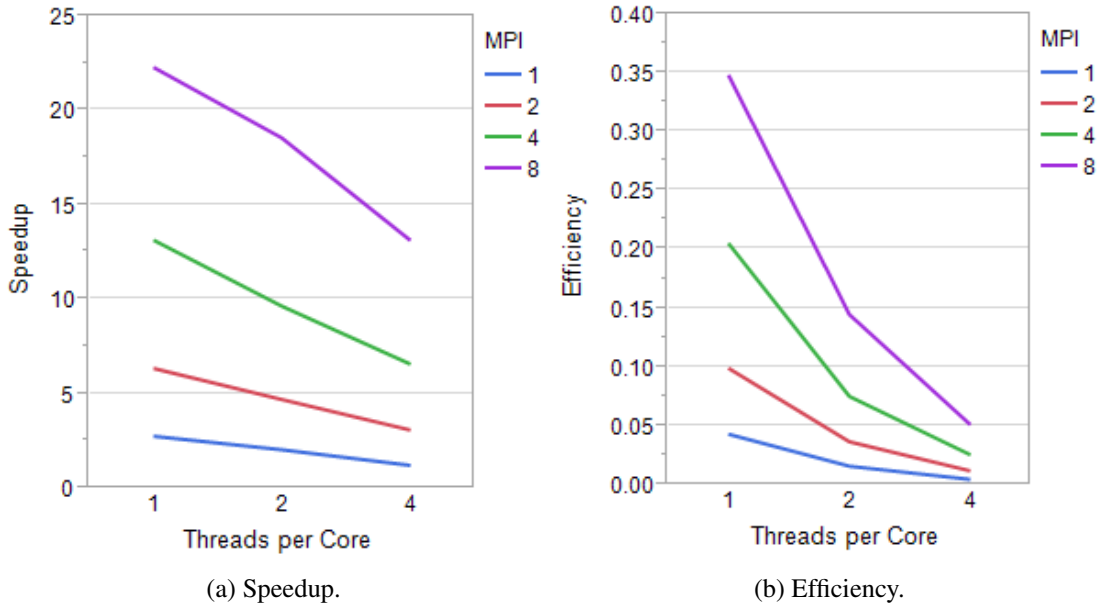


Figure 4.6: Experiment results of SOLCTRA on on the Knights Landing architecture contrasting MPI with the Threads per Core instead of OpenMP.

In these two graphs we saw that higher performance is achieved when the MPI rank count is in 8 and the threads per core is set to 1, that is, 8 OpenMP threads. This is a result similar on the number of workers of the results we shown on chapter 3. Overall, we can see that the performance decreases as the thread per core is increased and/or the cores per MPI rank is also increased. This raised a couple of questions like: why is this phenomenon? Why is this performance degradation happening when the thread per core is increased or when the number of cores per MPI rank is also increased? Have we achieved the maximum level of parallelization for this application, at least for this architecture?

For the case when the thread per core increments, we noted that the decrement trend becomes more drastic as the number of MPI ranks is increased. Because it can be isolated to the threads per core, we conclude that this issue is strictly related to the hyper-threading technology implemented

on the Knights Landing's cores. Related to this, we found in [21] and [2] that how efficient is the performance of the hyper-threading is strictly related to the characteristics of the application. Additionally, we found in [26] that hyper-threading is not good for applications CPU-bound, such as SOLCTRA.

But in [26] the author used a desktop processor, the Intel Core *i5-2410M*, which is not built for HPC. That processor has only replicated the basic requirements for Hyper-Threading. In our case, we are using Knights Landing architecture. Because of that, we started to wonder: why are not we getting at least an acceptable performance for two threads per core if, differently to the *i5*, we have replicated the VPU and other resources ([16]) than just the registers? To respond this question we have a limitation: the architectural documentation for the Knights Landing is restricted and the deeper architectural documentation is what we have also in [16]. While for the first part of the question the answer is related to the architecture, for the second part we found that it is more related to the application than the architecture itself.

In section 3.2.3.1, we showed that the magnetic fields calculation is the main point of parallelization at thread level in the SOLCTRA application. This function also contains most of the mathematical operations. After an analysis of this code and with the help of the Intel VTune Amplifier, we concluded that our constraint is related to the size of each iteration of the loop of the magnetic fields calculation. The smaller the workload of one iteration, the higher overhead of the thread synchronization across the cores, hence, more the communication through the KNL mesh. But, on the Stampede multicore architecture, the application did scaled ok with threads across the processor. Why is it not the same on the KNL? From these results, what we can conclude is that the efficiency of the 2D mesh of the KNL is not as good as the one on the processors on Stampede.

Once having the overall results of the application on the KNL, we started comparing the performance of the application per schedule policy. On the speedup (figure 4.7), we see that for the MPI rank count 1 and 2, there is no notable difference. While for 4 and 8, there is a slightly difference between the policies, with dynamic being higher than static and that higher than guided. For the efficiency (figure 4.8), the difference is barely noted. From these results, we concluded that for the SOLCTRA application, the schedule policy does not make a significant impact on the performance.

The next variable to explore is the schedule chunk size. Similar to the schedule policy analysis, for the schedule chunk size variable the speedup and efficiency remains similar (figure 4.9 and 4.10) for the MPI rank count of 1, 2 and also 4, but with slightly differences when the MPI rank count is set to 8. Because of that, we decided to focus only on this last scenario.

In figure 4.11 we have a graph of the efficiency contrasted with the chunk size but only for the

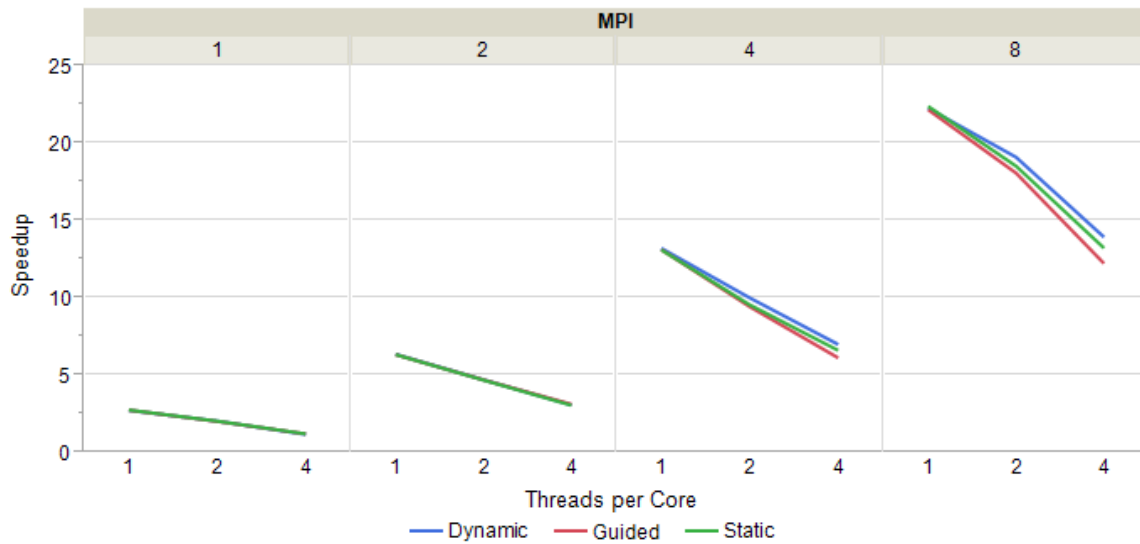


Figure 4.7: Average speedup contrasting the schedule policy.

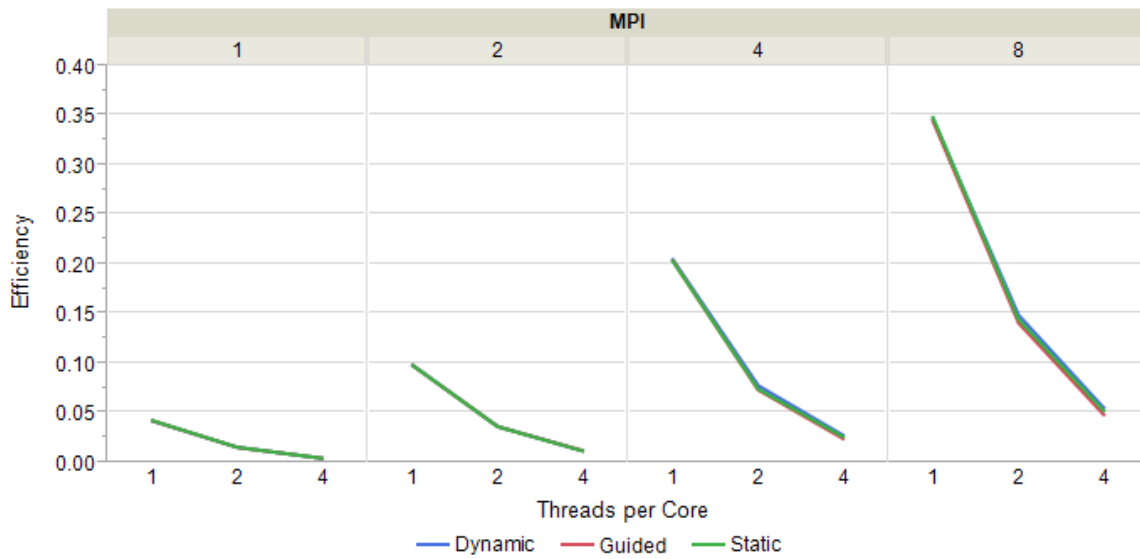


Figure 4.8: Average efficiency contrasting the schedule policy.

case when the MPI rank count is set to 8. For the case of 4 threads per core, we can see that the line remains practically constant until the chunk size of 32, where the efficiency drops and remain at that level for the chunk size of 64. Meanwhile, for 2 threads per core, the line has an increment for the chunk size from 1 to 2, but remains stable for the rest of the line, and then there is a drop for

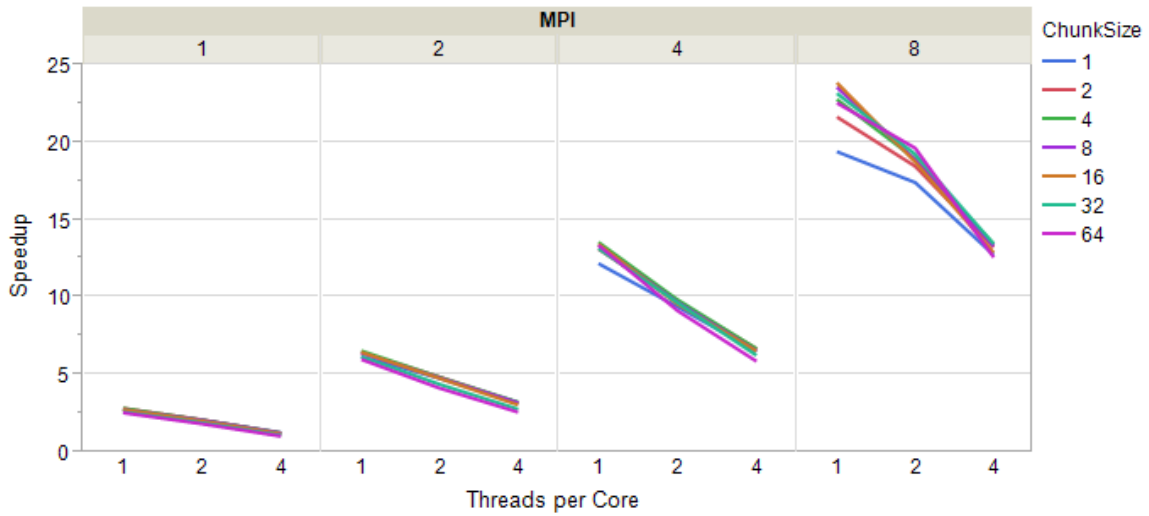


Figure 4.9: Average speedup contrasting the schedule chunk size.

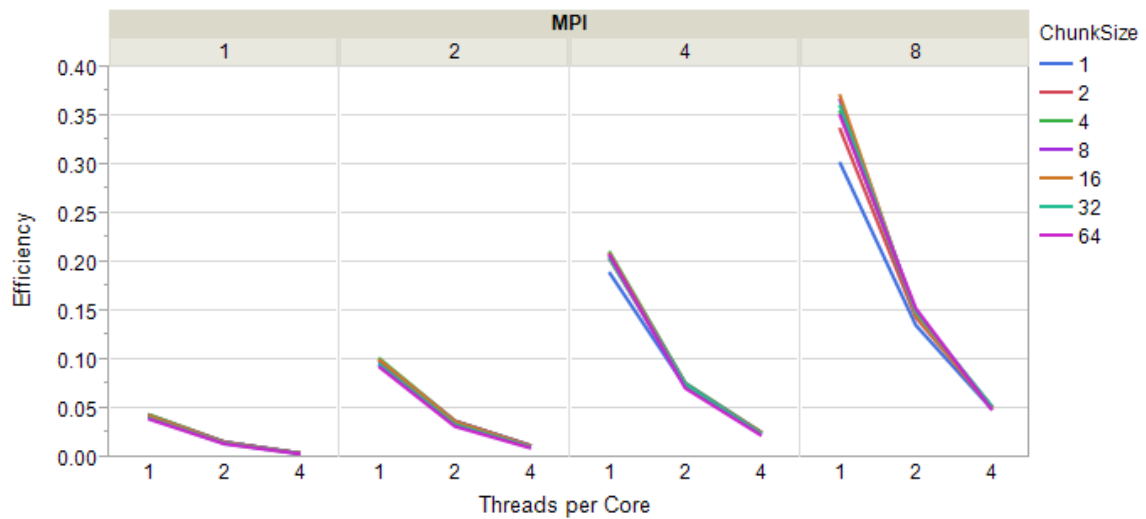


Figure 4.10: Average efficiency contrasting the schedule chunk size.

the chunk size of 32 and 64, similar to 4 threads per core. Finally, for the 1 thread per core case, instead having a straight line, what we have is an incremental curve that start above 0.2, reach its peak below 0.25 when the chunk size is in 16, and then it suffers from the same behavior of the other cases: a drop of the efficiency for the chunk size of 32 and 64.

What we conclude here, is that for the size of the problem (the workload of each iterations and

the amount of iterations), we are able to achieve the peak of performance when the chunk size is set to 8 or 16. If we decrement this number, we will have more granularity than what is needed, forcing the application to oversynchronize the threads. But, if we increment this number, we will have less granularity of the required, generating an artificial load imbalance among the threads of the application. So, 8 and 16 provide us the balance where we can maximize the thread count without compromised the performance. This results are for the current size of the loop in the magnetic fields function, where we have only 12 coils. If we had more coils or more points per coil, new experiments would be needed in order to explorer which would be the appropriate chunk size.

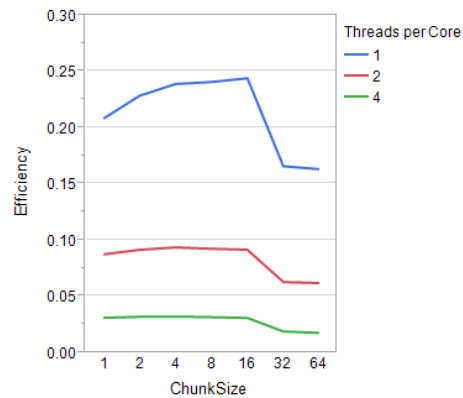


Figure 4.11: Average efficiency contrasting the schedule chunk size with the threads per core for only 8 MPI ranks.

Our next analysis is focused on the affinity and the granularity. The results of our experiments for the combination of those configurations are shown in figure 4.12 and 4.13.

In both figures, we can see that most of the lines remains pretty similar. The exception are the lines for the combinations of balanced-core and balanced-core, which are below the others when the thread per core is set to 2 and 4. We can see that that difference might be seen more drastic for the speedup, but when we move to the efficiency, this difference is barely noted. Also, this difference is only for the cases when we have more than one thread per core. What we conclude from comparing this configurations is that the performance of the SOLCTRA application is agnostic to the different combinations. The only exception will be for the cases when we have the balanced affinity, the granularity to cores and we are executing more than one thread per core. But that difference is so small that does not rise bigger conclusions.

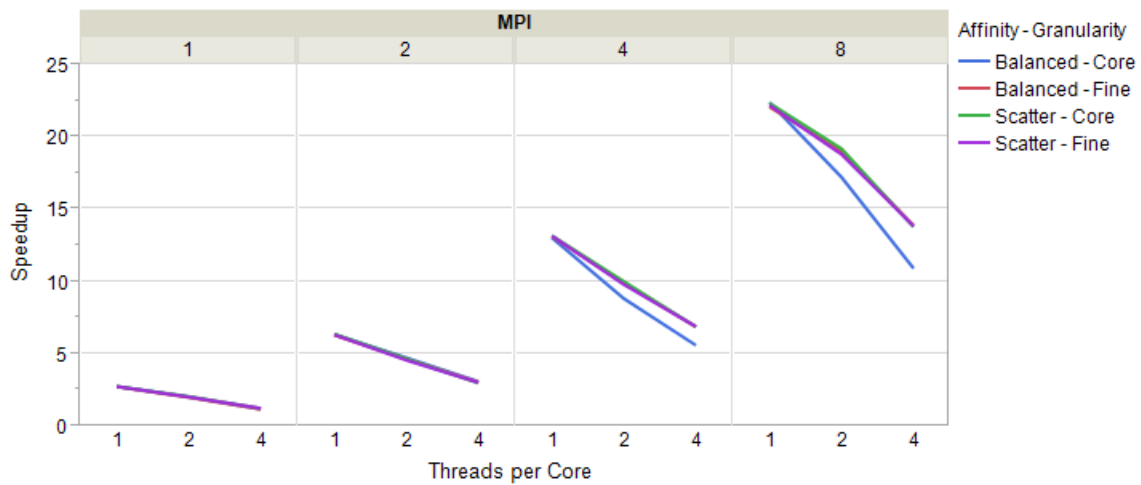


Figure 4.12: Average speedup contrasting the affinity and the granularity.

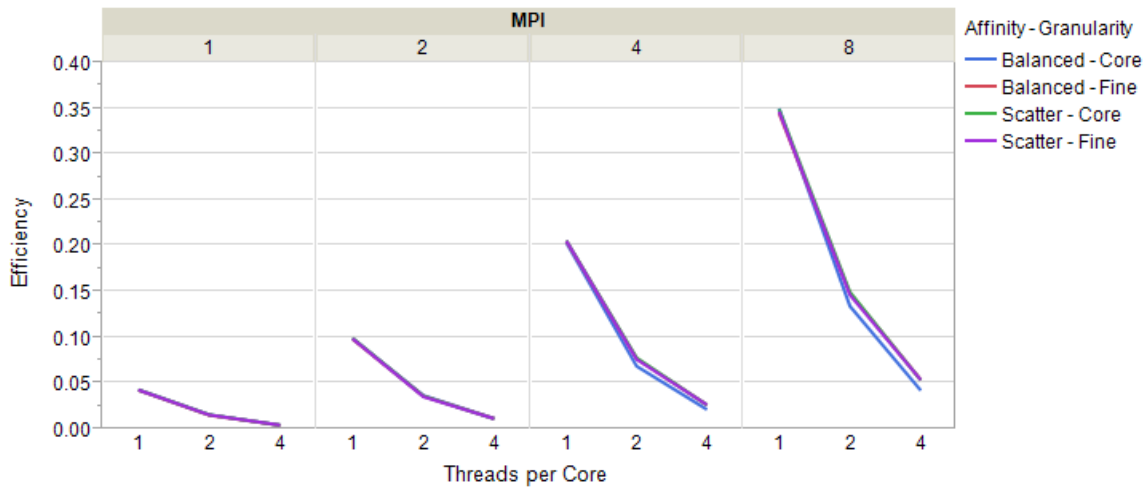


Figure 4.13: Average efficiency contrasting the affinity and the granularity.

4.3.3 Weak Scaling Experiments Results

On section 3.3.2 we showed the results for the experiments for the weak scaling on the Stampede supercomputer. Now, we are going to show the weak scaling results on the KNL.

We took the based configuration for the KNL selected at the end of section 4.3.2 and, as we did on the Stampede, we "played" with the different count of the MPI ranks and the OpenMP threads while keeping the size of the problem fixed. Same case as on Stampede, we changed those variables

in exponent of 2.

Differently to the experiments of Stampede (section 3.3.1), where the threads could run across sockets, these experiments on KNL are contained into one socket. This has an effect on the layout of the threads: when total number of threads passes the number of cores (64), the application will start to assign more than one thread per core, exploding the hyper-threading of the KNL.

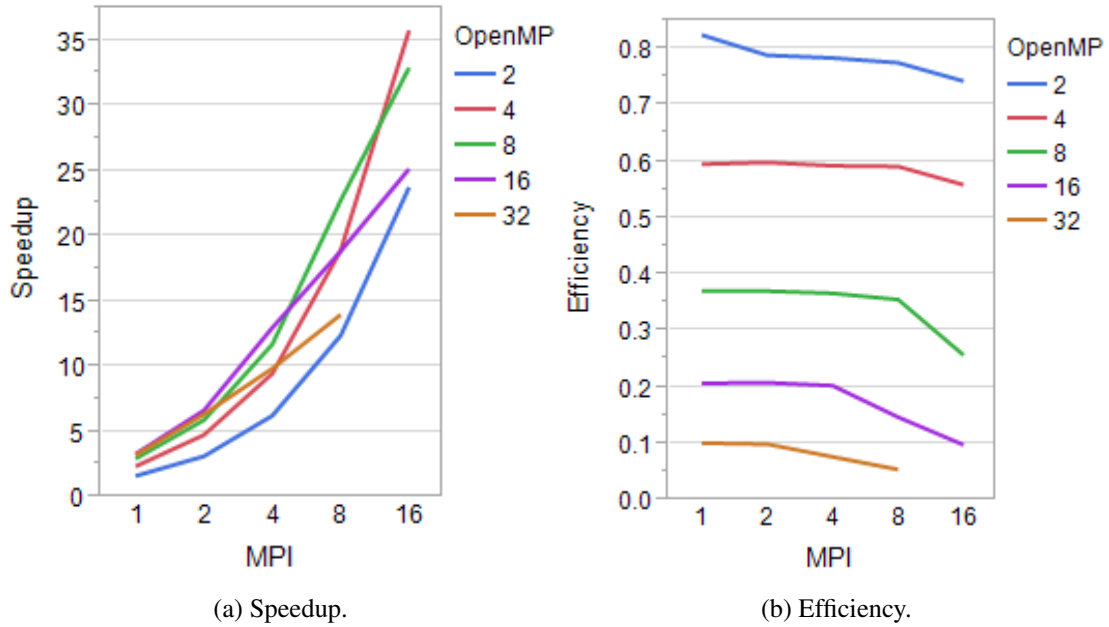


Figure 4.14: Experiment results of SOLCTRA on the KNL for the weak scaling.

In figure 4.14a we have the results of the speedup for these experiments. In that figure, we can see that all the lines follows an incremental curve, some more inclined than other. The line for OpenMP on 32 does not have a point on 16 MPI ranks because that case will generate an overbooking of the threads, since it will require 512 threads when we have only 256 per KNL.

In figure 3.15a we saw that for the Stampede we also have inclined curves. The main difference between the two architectures is that for Stampede the lines are clearly separated, with that separation following a thread. For the KNL, there is no trend on the separation of the different curves. Actually, for some curves, there are points where they get to cross each other.

For the efficiency on the KNL (figure 4.14b), all the lines are flatted until the worker configuration forces to have more than one thread per core. That is the reason why the line for 32 OpenMP threads is flat until 2 MPI ranks, then the line starts to be degraded. Similar cases are for the other lines.

This follows a similar performance behavior to the results on section 3.3.2 for the case when the threads have to work across sockets.

So, we can conclude that the SOLCTRA is weakly scalable on the KNL while the thread per core is contained to 1.

4.3.4 Strong Scaling Experiment Results

After analyzing the results of the weak scaling, now we are going to analyze the results for the strong scaling. For these experiments we followed a similar methodology of the previous sections on how the MPI and OpenMP were configured.

Also, for the case when the total of workers exceeded 64, we followed the same approach from section 4.3.3.

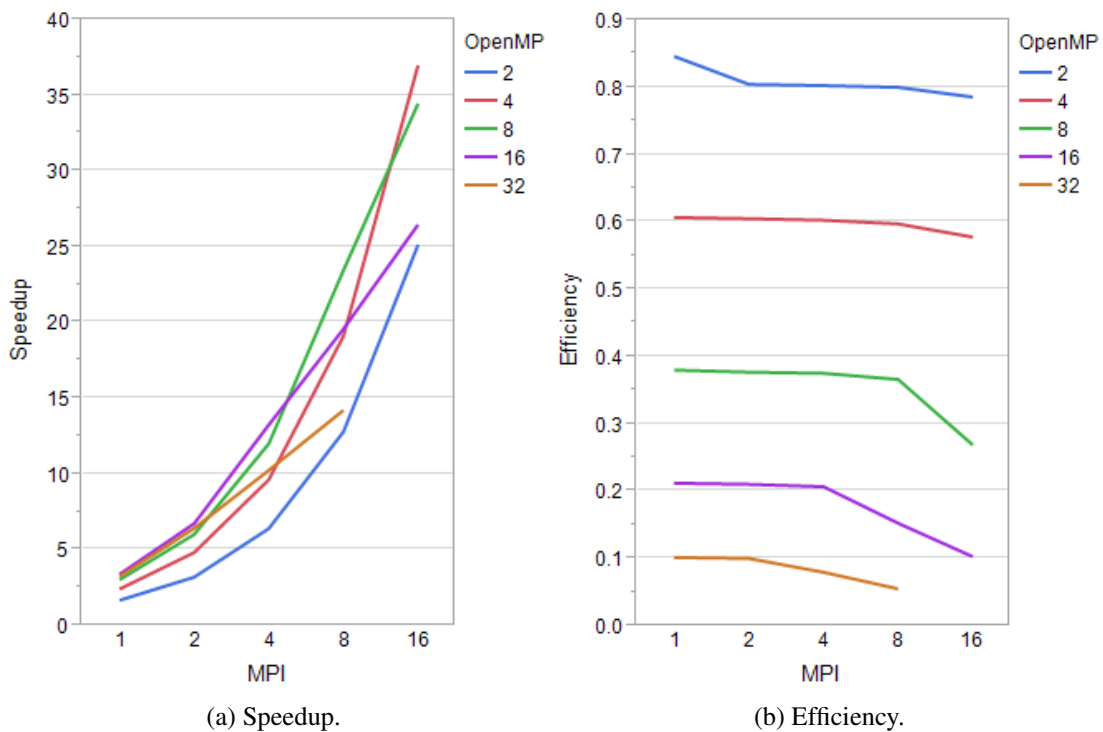


Figure 4.15: Experiment results of SOLCTRA on the KNL for the strong scaling.

In figure 4.15 we have the average of the speedup and the efficiency for the strong scaling results. We can see that those figures are also pretty similar to the figure in section 4.3.3. Here we have

a similar results of the obtained in chapter 3 which for the speedup and efficiency threw similar figures for the weak and the strong scaling.

Another outcome from this figure is that when we have 2 threads per MPI the efficiency is really good (near 0.8) in comparison with the other OpenMP configurations. This is the case when 2 threads are within a single tile (see section 4.1). So, when the OpenMP thread count is set to 4 or more, the communication between the threads have to beyond the tile, negatively affecting the performance of the application with a high drop-off for each OpenMP step.

So, even when we have some cases with an efficiency lower than expected, from these experiments we can conclude that the application is strongly scalable on the KNL while the thread per core is contained to 1.

4.3.5 Incrementing the Size of the Problem While Keeping the Workers Fixed Experiment Results

Similar to section 3.3.4, we executed an additional set of experiments for having the MPI and OpenMP fixed while changing the size of the problem. For these experiments we chose two combinations of MPI and OpenMP counts instead of one as we did for section 3.3.2. The first one was the combination of MPI=8 and OpenMP=8. We selected this combination because, even when it did not showed good efficiency on section 4.3.4 for KNL, it did for Stampede. The second combination is the one with MPI=16 and OpenMP=4. We selected this second combination because it was the one that showed better balance between efficiency and the threads on prior sections on KNL.

In figures 4.16 and 4.17 we have the graph for the speedup and the efficiency respectively. Because we are using different rank count, we are using the particles per rank instead the total of particles, so the x axis is relative to the combination.

In both figures, we can see that the two lines per combination follows the same trend. This trend has a slightly increment at the first four particles per rank, and it got stabilized for the graph remainder.

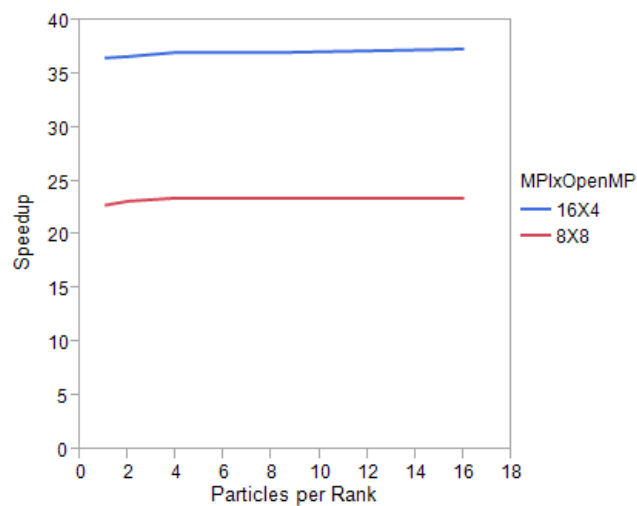


Figure 4.16: Average speedup of SOLCTRA on a KNL with worker count fixed.

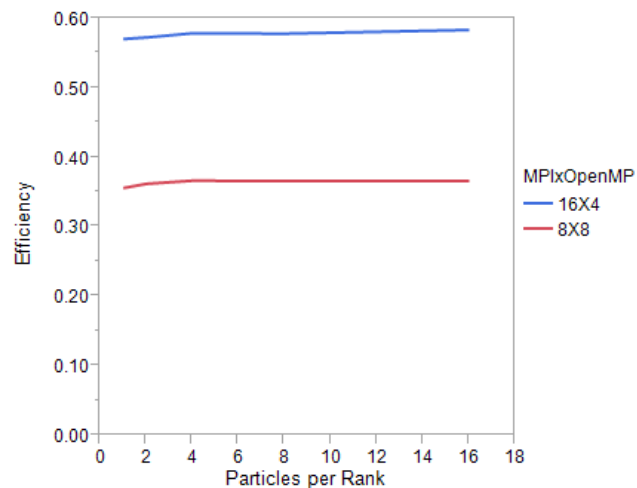


Figure 4.17: Average efficiency of SOLCTRA on a KNL with worker count fixed.

Conclusions and Future Work

This chapter serves as a summary of the findings in this dissertation and also as a path for those who want to continue exploring plasma physics applications on massively parallel architectures.

5.1 Conclusions

The following list presents the mayor conclusions of this work:

- **Plasma physics applications are good targets for parallelization:** In this work we added parallelism to the SOLCTRA application and we showed excellent results of that work.
- **The SOLCTRA is scalable on multiple types of architectures:** on this work we demonstrated that a plasma physics application, such as SOLCTRA, easily scales on both multicore and manycore architectures (Xeon processors and KNL processors). From the results of the experiments on chapters 3 and 4 we can conclude that this application is *strongly* and *weakly* scalable, based on the Amdahl's Law and Gustafson's Law.
- **The replicated resources on KNL's cores to support hyper-threading are not good enough for a CPU-bound application like SOLCTRA:** we saw on other publications that the hyper-threading on low end processors is not good for CPU-bound applications. The KNL has more resource replication than a regular processor, for example the 2 VPUs per core. We concluded that even with those resource improvements, the hyper-threading on the KNL is still not good for CPU-bound applications like the SOLCTRA.

- **Cross socket threading is not good for CPU-bound applications:** on this work we showed that for a CPU-bound application like SOLCTRA, when the threads have to be executed across sockets the performance suffers a high degradation that make that approach not worthy to follow.
- **The impact of exhausting synchronization of all the threads and data movements across the mesh:** The SOLCTRA application has a particularity: it has to sync all the threads and perform a set of reductions for every iteration. While on the Stampede this does not appear to be an issue, it does on the KNL. Our conclusion is that the 2D mesh implemented on the KNL for tile intercommunication is not good enough for this level of synchronization that SOLCTRA requires. Something that backup this is the fact that the efficiency is high when the OpenMP is set to 2, containing them within the same tile.
- **The impact of vectorization:** We demonstrated the potential that vectorization has to offer on the performance improvement of the application. Only on this feature, the KNL allowed an increment of slightly more than 8x of speedup.

5.2 Future Work

This section provides ideas for developing some proposals that stand upon the work presented in this dissertation:

- **Explore an increment of the size of the iterations in the magnetic field calculations:** In our conclusions we have that some of the reasons of not having good efficiency on KNL is because the mesh performance is not good enough for the level of synchronization required on SOLCTRA. Here we propose future work on how the speedup and efficiency would behave if we could increase the workload per iteration, reducing the ratio of required synchronizations per execution. One way of doing this for SOLCTRA would be by increasing the number of coils or by increasing the detail definition of each of the coils that we have.
- **Can a single thread use both VPU's on a single KNL core?:** The data type used on SOLCTRA is the type double, which as a length of 64 bits. On the KNL we have a vector units of 512bits wide. This means that we have a ratio of 8 SIMD operations. But, in our experiments we found that the difference in the performance of SOLCTRA with and without vectorization is a little bit higher than 8, when higher vectorization ratio is of 8. So, this rises the question: is this effect because there is another feature that allows a speedup a little bit

higher than 8? Or is it because one thread is able to use both VPU's on single thread? Because the results that we got are tied to 8 we cannot conclude something from our experiments, but this is something that will require more exploration.

- **Explore the performance on future Intel Xeon Platform:** The experiments we executed for this work were on one multicore architecture and a manycore architecture. The next generation of Intel Xeon family, codenamed Skylake (do not confuse with the already release desktop processor), will be in the middle of those architectures: around 30 high frequency cores, same vector wide instructions than KNL to mention some of the already released characteristics. Also, even when there is not public information about this, because core count will be higher than Stampede and frequency higher than the KNL's, we might expect a higher performance of the hardware on the intercore communication. If it is so, an experiment for future work will be to execute the SOLTRA in this new architecture and see if performance suffers due to inter-core communication in the same way that KNL, if it will not be affected or if it will be at some point in the middle.

Bibliography

- [1] Dr. Ron Barr. Engineering modeling: Mathematical and computer. <http://www.me.utexas.edu/~me302/classnotes/MODELING/sld003.htm>.
- [2] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Exploring the effects of hyper-threading on parallel simulation. In *2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 257–260, Oct 2006.
- [3] Chris Brady, Keith Bennett, Holger Schmitz, and Christopher Ridgers. *Users Manual for the EPOCH PIC codes*. University of Warwick, 4.3.4 edition, September 2015.
- [4] Holger Brunst and Bernd Mohr. Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampirng. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming, IWOMP'05/IWOMP'06*, pages 5–14, Berlin, Heidelberg, 2008. Springer-Verlag.
- [5] A. S. Chai. Error estimate of a fourth-order runge-kutta method with only one initial derivative evaluation. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, pages 467–471, New York, NY, USA, 1968. ACM.
- [6] Ad Astra Rocket Company. Vasimr. <http://www.adastrarocket.com/aarc/VASIMR>.
- [7] Intel®Corporation. Intel® xeon® processor 7210. https://ark.intel.com/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1_30-GHz-64-core.

- [8] Intel® Corporation. Intel® xeon processor e5530. http://ark.intel.com/products/37103/Intel-Xeon-Processor-E5530-8M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.
- [9] Intel® Corporation. Products (formerly knights landing). <http://ark.intel.com/products/codename/48999/Knights-Landing>.
- [10] B.D. Dudson, M.V. Umansky, X.Q. Xu, Snyder P.B., and H.R. Wilson. Compact expressions for the biot–savart fields of a filamentary segment. *Physics of Plasmas*, July 2002.
- [11] Ben Dudson. Bout++. <http://www-users.york.ac.uk/~bd512/bout/>.
- [12] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.
- [13] Steven P. Hanson, James D.; Hirshman. Compact expressions for the biot–savart fields of a filamentary segment. *Physics of Plasmas*, July 2002.
- [14] The UK Collaborative Computational Project in Plasma Physics. Ccp-plasma - codes. <http://www.ccpp.ac.uk/codes.html>.
- [15] ITCR. Tec hace la primera descarga de plasma en un dispositivo único en latinoamérica. <http://tecdigital.tec.ac.cr/servicios/hoyeneltec/?q=content/tec-hace-la-primera-descarga-de-plasma-en-un-dispositivo-unico-en-latinoame>
- [16] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, July 2016.
- [17] H. A. Watts L. F. Shampine. Comparing error estimators for runge-kutta methods. *Mathematics of Computation*, 25:445–455, July 1971.
- [18] Princeton Plasma Physics Laboratory. NESCOIL. <http://vmecwiki.pppl.wikispaces.net/NESCOIL>.
- [19] Princeton Plasma Physics Laboratory. Vmec. <http://vmecwiki.pppl.wikispaces.net/VMEC>.
- [20] The Top 500 list. Top 10 sites for november 2016. <https://www.top500.org/lists/2016/11/>.

- [21] Victor Mashayekhi, Jenwei Hsieh, Reza Rooholamini, Onur Celebioglu, Tau Leng, and Amina Saify. The performance impact of computational efficiency on hpc clusters with hyper-threading technology. *Parallel and Distributed Processing Symposium, International*, 15:250b, 2004.
- [22] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.
- [23] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.
- [24] NVIDIA®. What is gpu computing? <http://www.nvidia.com/object/what-is-gpu-computing.html>.
- [25] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.
- [26] N. H. Qun, Z. I. A. Khalib, M. N. Warip, M. E. Elobaid, R. Mostafijur, N. A. H. Zahri, and P. Saad. Hyper-threading technology: Not a good choice for speeding up cpu-bound code. In *2016 3rd International Conference on Electronic Design (ICED)*, pages 578–581, Aug 2016.
- [27] TACC. Current research on tacc systems. <https://www.tacc.utexas.edu/research-development/users>.
- [28] TACC. Stampede user guide. <https://portal.tacc.utexas.edu/user-guides/stampede>.
- [29] Yuan Tian, Chuang Lin, and Kangqiao Hu. The performance model of hyper-threading technology in intel nehalem microarchitecture. In *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, volume 3, pages V3–379–V3–383, Aug 2010.
- [30] V I Vargas, J Mora, J Asenjo, E Zamora, C Otárola, L Barillas, J Carvajal-Godínez, J González-Gómez, C Soto-Soto, and C Piedras. Constructing a small modular stellarator in latin america. *Journal of Physics: Conference Series*, 591(1):012016, 2015.

- [31] V I Vargas, J Mora, J Asenjo, E Zamora, C Otárola, L Barillas, J Carvajal-Godínez, J González-Gómez, C Soto-Soto, and C Piedras. Implementation of stellarator of costa rica 1 scr-1. *IEEE 26th Symposium on Fusion Engineering (SOFE)*, 2015.
- [32] Andrey Vladimirov, Ryo Asai, and Vadim Karpusenko. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. Colfax International, 2nd edition, May 2015.