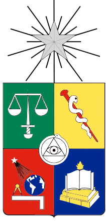

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PARALLEL METHODS FOR CLASSICAL AND DISORDERED SPIN MODELS

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS
MENCIÓN COMPUTACIÓN

CRISTÓBAL ALEJANDRO NAVARRO GUERRERO

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

PROFESOR CO-GUÍA:
FABRIZIO CANFORA TARTAGLIA
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
NELSON BALOIAN TATARYAN
RODRIGO SOTO BERTRAN
YOUJIN DENG

Este trabajo ha sido parcialmente financiado por CONICYT.

SANTIAGO DE CHILE
2015

Resumen

En las últimas décadas han crecido la cantidad de trabajos que buscan encontrar metodos eficientes que describan el comportamiento macroscópico de los *sistemas de spin*, a partir de una definición microscópica. Los resultados que se obtienen de estos sistemas no solo sirven a la comunidad física, sino también a otras áreas como dinámica molecular, redes sociales o problemas de optimización, entre otros. El hecho de que los sistemas de spin puedan explicar fenómenos de otras áreas ha generado un interés global en el tema. El problema es, sin embargo, que el costo computacional de los métodos involucrados llega a ser muy alto para fines prácticos. Por esto, es de gran interés estudiar como la computación paralela, combinada con nuevas estrategias algorítmicas, puede generar una mejora en velocidad y eficiencia sobre los metodos actuales.

En esta tesis se presentan dos contribuciones; **(1)** un algoritmo exacto multi-core distribuido de tipo *transfer matrix* y **(2)** un método Monte Carlo multi-GPU para la simulación del modelo 3D *Random Field Ising Model* (RFIM). La primera contribución toma ventaja de las relaciones jerárquicas encontradas en el espacio de configuraciones del problema para agruparlas en árboles de familias que se solucionan en paralelo. La segunda contribución extiende el método *Exchange Monte Carlo* como un algoritmo paralelo multi-GPU que incluye una fase de adaptación de temperaturas para mejorar la calidad de la simulación en las zonas de temperatura mas complejas de manera dinámica.

Los resultados muestran que el nuevo algoritmo de *transfer matrix* reduce el espacio de configuraciones desde $O(4^m)$ a $O(3^m)$ y logra un *fixed-size speedup* casi lineal con aproximadamente 90% de eficiencia al solucionar los problemas de mayor tamaño. Para el método multi-GPU Monte Carlo, se proponen dos niveles de paralelismo; local, que escala con GPUs mas rápidas y global, que escala con múltiples GPUs. El método logra una aceleración de entre uno y dos ordenes de magnitud respecto a una implementación de referencia en CPU, y su paralelismo escala con aproximadamente 99% de eficiencia. La estrategia adaptativa de distribución de temperaturas incrementa la taza de intercambio en las zonas que estaban mas comprometidas sin aumentar la taza en el resto de las zonas, generando una simulación mas rápida aun y de mejor calidad a que si se usara una distribución uniforme de temperaturas. Las contribuciones logradas han permitido obtener nuevos resultados para el área de la física, como el calculo de la matriz transferencia para el *kagome lattice* en $m = 9$ y la simulación del modelo *3D Random Field Ising Model* en $L = \{32, 64\}$.

Abstract

In the last decades a large amount of research has been devoted on finding efficient methods for studying *spin models* and obtain the macroscopic behavior of the system starting from a microscopic definition of it. The results obtained for some spin models have proven to be useful not only to the physics community, but also to other fields such as molecular dynamics, social networks and optimization problems, among others. The fact that these interaction models can explain phenomena from other fields has led to a global interest on the subject. The problem is however, that the cost of the computational methods involved, both exact and Monte Carlo, can become too expensive for practical purposes. For this, it is of interest to study how parallel computing, combined with new algorithmic strategies, can provide a performance improvement to the existing methods.

In this thesis we present two computational contributions; **(1)** an exact multi-core distributed transfer matrix algorithm and **(2)** a multi-GPU Monte Carlo method for the 3D *Random Field Ising Model*. The first contribution consists of a parallel multi-CPU transfer matrix algorithm that takes advantage of the hierarchical relations found in the configuration space and forms family trees of configurations that are solved in parallel. In the second contribution we propose a multi-GPU method for the simulation of the 3D *Random Field Ising Model*. The multi-GPU algorithm is an extension of the *Exchange Monte Carlo* method, since it adds a new strategy for choosing a set of temperatures that increases the exchange rate at the locations where the simulation is most compromised.

The results show that the new transfer matrix algorithm reduces the configuration space from $O(4^m)$ to $O(3^m)$ and runs 90% efficient for distributed multi-core CPUs, providing close to linear speedup. For the multi-GPU Monte Carlo method, the performance scales using two levels of parallelism; locally with faster GPUs and globally with multiple GPUs. Compared to a CPU implementation, the multi-GPU method runs at least an order of magnitude faster and scales with an efficiency of approximately 99% when moving from one to two GPUs. The new temperature distribution strategy provides higher exchange rate at the low temperature region, resulting in a better and faster simulation. The contributions of this thesis have allowed us to obtain new scientific results, such as the transfer matrix for the kagome strip at width $m = 9$ as well as physical observables for the *3D Random Field Ising Model* at $L = \{32, 64\}$.

Dedicated to my parents, brothers and María Belén.

Acknowledgements

Doing a Ph.D. has been an enriching experience in many aspects. I cannot help but thank the *Department of Computer Science* of *University of Chile* and *Centro de Estudios Científicos* for showing me what it means to be a researcher and for giving me the knowledge and tools that are required to become one. I also thank CONICYT, who is the entity that gives the economical support for doing a Ph.D. in Chile.

During a Ph.D., it is common to be asked why one chooses to do a Ph.D., or simply what is a Ph.D. If I remember well, every time I was asked these questions I gave a different answer, hoping that it would be the definitive one. Today, I strongly believe that being a researcher is more of an attitude than anything else; it is about feeling curiosity, love and passion for something. It reminds me of an old quote a friend from Valdivia once used: *"Choose a job you love, and you will never have to work a day in your life"*.

Through these years, I've met people who taught me many lessons, others that helped me when I needed most and others who shared with me really beautiful moments which I will always remember. In the following paragraphs I try to express how thankful I am to them.

The first two persons I want to thank are professors Nancy Hitschfeld and Fabrizio Canfora. It is hard to find the words to describe how thankful I am to both of them, from the conversations about life, their knowledge, to the help they provided whenever I needed them. Thanks to professor Gonzalo Navarro for giving such high quality lectures on algorithms and for joining as a co-advisor later on. I also thank professor Éric Tanter for being so close to us the students, and for his will to help me whenever I had a question or problem.

It is impossible for me to write an acknowledgement letter without giving thanks to Angélica Aguirre, Sandra Gáez and Francia Ormeño, whom we are all in debt for the kindness and help they have provided through all these years. To all my friends from the Ph.D. lab, thanks for all the good times we shared, from eating lunch together to the football matches of Saturday morning.

I thank my family, who live in Valdivia. My parents have supported me in doing a Ph.D. from the very beginning. I thank them for the strength they gave me when things were complicated and for all the life lessons given to me since I was a child. Last but not least, I want to thank my love, Maria Belén, for being with me in the good and difficult moments and for always reminding me the importance of the family.

Contents

Resumen i
Abstract ii

Introduction **1**

1 Parallel Computing Background **5**
1.1 Basic concepts 7
1.2 Performance measures 8
1.2.1 Work and Span 8
1.2.2 Speedup 9
1.2.3 Efficiency 12
1.2.4 FLOPS 12
1.2.5 Performance per Watt 13
1.2.6 Memory Bandwidth 13
1.3 Parallel Computing Models 14
1.3.1 Parallel Random Access Machine (PRAM) 14
1.3.2 Parallel Memory Hierarchy (PMH) 16
1.3.3 Bulk Synchronous Parallel (BSP) 17
1.3.4 LogP 18
1.4 Parallel programming models 19
1.4.1 Shared memory 19
1.4.2 Message passing 20
1.4.3 Implicit 20
1.4.4 Algorithmic skeletons 21
1.5 Architectures 22
1.5.1 Flynn's taxonomy 22
1.5.2 Memory architectures and organizations 23
1.5.3 Technical details of modern CPU and GPU architectures 24
1.5.4 The fundamental difference between CPU and GPU architectures 26
1.6 Strategy for designing a parallel algorithm 27
1.6.1 Partitioning 27
1.6.2 Communication 28
1.6.3 Agglomeration 28
1.6.4 Mapping 28
1.7 GPU Computing 29
1.7.1 The massive parallelism programming model 30
1.7.2 Thread managing and GPU concurrency 31

1.7.3 Technical considerations for a GPU implementation 32
1.8 Latest advances and open problems in GPU computing 33

2 Spin Models Background 35
2.1 Spin Systems 35
2.1.1 The partition function $Z( )$ 37
2.1.2 Ising Model 38
2.1.3 Potts model 38
2.1.4 Spin Glass: Edwards-Anderson model 39
2.1.5 Random Field model 39
2.2 Exact methods for computing $Z( )$ 40
2.2.1 Using the definition 40
2.2.2 Deletion-contraction 41
2.2.3 Transfer matrix technique 43
2.3 Monte Carlo methods 44
2.3.1 Computation of Averages 45
2.3.2 Markov Chain Monte Carlo (MCMC) 46
2.3.3 The Metropolis-Hastings Algorithms 47
2.3.4 Cluster Algorithms 48
2.3.5 The Worm Algorithm 49
2.3.6 Exchange Monte Carlo (*Parallel Tempering*) 51

3 Parallel Family Trees for Transfer Matrices in the Potts Model 54
3.1 Abstract 54
3.2 Introduction 55
3.3 Related work 56
3.4 Algorithm overview 58
3.4.1 Data structure 58
3.4.2 DC-based transfer matrix computation 59
3.4.3 Family trees strategy 61
3.5 Algorithm improvements 69
3.5.1 Serial and Parallel paths 69
3.5.2 Axial Symmetry 70
3.6 Implementation 71
3.7 Performance results 72
3.7.1 Multi-core results 72
3.7.2 Cluster results 75
3.7.3 Impact of DC on algorithm performance 77
3.7.4 Performance on wider strips 78
3.7.5 Comparison with related work 79
3.7.6 Dynamic scheduler and block size 80
3.7.7 Axial Symmetry 80
3.8 Validation 80
3.9 Discussion 83

4 Adaptive Multi-GPU Exchange Monte Carlo for the 3D Random Field Ising Model 85

4.1 Introduction . . . 85
4.2 Related Work . . . 87
4.3 Multi-GPU approach . . . 89
4.3.1 Parallelism in the Exchange Monte Carlo method . . . 89
4.3.2 Spin Level Parallelism . . . 90
4.3.3 Replica Level Parallelism . . . 92
4.4 Adaptive Temperatures . . . 93
4.5 Performance results . . . 94
4.5.1 Benchmark Plan . . . 94
4.5.2 Spin and Replica level Performance Results . . . 95
4.5.3 Multi-GPU Scaling . . . 96
4.5.4 Performance of The Adaptive Temperatures Technique . . . 96
4.6 Exchange Rates with Adaptive Temperatures . . . 97
4.7 Preliminary Physical Results . . . 98
4.8 Discussion . . . 99

Conclusions **100**

Bibliography **103**

Appendices **126**

A GPU-maps for triangular domains **127**
A.1 Introduction . . . 128
A.2 Related Work . . . 129
A.3 The blockwise triangular map . . . 130
A.3.1 Formulation . . . 130
A.3.2 Bounds on the improvement factor . . . 132
A.4 Implementation of LTM . . . 134
A.4.1 Choosing the best square root . . . 134
A.4.2 Implementing the other strategies . . . 135
A.5 Experimental results . . . 136
A.6 Discussion . . . 138

List of Tables

3.1 Catalan and root configurations 71
3.2 Executions times for wider strips. 78

4.1 Execution time for adaptive and uniform approaches 97

A.1 Hardware used for experiments. 137

List of Figures

1.1 The four possible curves for speedup. 10
1.2 Comparison of CPU and GPU performance. 13
1.3 PRAM model 15
1.4 The uniform parallel memory hierarchy tree. 16
1.5 Classic and blocked matrix strategies 17
1.6 Bulk Synchronous Parallel super-step 18
1.7 An example communication using the LogP model. 19
1.8 The UMA (*aka* SMP) and NUMA memory architectures. 24
1.9 A diagram and silicon image of the modern CPU architecture 24
1.10 A diagram and silicon image of the modern GPU architecture 25
1.11 A GPU streaming multiprocessor 25
1.12 Difference between CPU and GPU architectures 27
1.13 Foster's diagram of the design steps used in a parallelization process. 29
1.14 Massive parallelism programming model 31
1.15 The kernel function of a GPU 31

2.1 The square spin lattice in one, two and three dimensions. 36
2.2 Typical boundary conditions for spin lattices 36
2.3 Phase transition 37
2.4 Example of a square lattice 40
2.5 Computation of $Z()$ 41
2.6 Deletion contraction technique 42
2.7 Square and Kagome strip lattices 43
2.8 Spin flip in the Metropolis-Hastings algorithm 48
2.9 The cluster creation process for the Swendsen and Wang algorithm. 49
2.10 Closed paths formed by the worm algorithm 51
2.11 Quenched disorder and rough energy landscapes 52

3.1 Example data structure for a square strip lattice of width $m = 3$. 59
3.2 The configuration space for a square lattice of width $m = 3$. 59
3.3 Terminal configuration generated from DC 60
3.4 DC on external edges 62
3.5 An example of the perfect binary tree and subtrees for $m = 3$. 62
3.6 Building the solution from the sub-trees 63
3.7 Examples of regular simplexes drawn on the plane. 66
3.8 Fosters four step strategy 68

3.9 Serial and parallel paths. 69
3.10 The square and kagome lattices used for measuring performance. 72
3.11 Multi-core running time, speedup, efficiency and knee for the square strip test. 73
3.12 Multi-core running time, speedup, efficiency and knee for the kagome strip test. 74
3.13 Cluster running time, speedup, efficiency and the knee for the square strip test. 76
3.14 Cluster running time, speedup, efficiency and knee for the kagome strip test. 77
3.15 Comparison between PFT and CPM 79
3.16 Limiting curves on the complex q-plane 81
3.17 Reduced internal energy and specific heat 82

4.1 Energy landscape for disordered systems 86
4.2 Two levels of parallelization 90
4.3 A double 2D checkerboard 90
4.4 Mapping the space of computation to the problem domain 91
4.5 Multi-GPU replica level parallelism 93
4.6 The adaptive temperatures strategy 93
4.7 Spin and replica level performance 95
4.8 Multi-GPU speedup and efficiency 96
4.9 Evolution of exchanges and total exchange rates 97
4.10 Preliminary physical observables for the 3D Random Field with $h = 1$. . . . 99

A.1 The BB strategy is not the best choice for a *td-problem*. 128
A.2 The LTM strategy 131
A.3 Performance of square root methods 135
A.4 The RB and REC strategies. 136
A.5 Map performance 138

Introduction

This thesis is essentially a computer science research on a very attractive physical problem. When these two fields meet one usually says that the work belongs to ***computational physics****.*

Context and Motivation

The field of *Computational physics* studies the computational problems involved in physics research and it is in a constant search of new methods that are faster and more efficient than the actual ones. This research field is in fact important for science itself, because it eventually contributes at *pursuing new knowledge*.

The cost of a physical problem, measured in computing time, depends on how fast the number of computational operations grow as a function of the problem size. It is of great interest to know what is the minimum time required to solve a problem as well as what is the maximum time an algorithm can take in the worst case. Typically, these bounds are found by doing an asymptotic analysis on the problem, assuming that the constants involved will not dominate the overall cost once the problem has reached a considerable size. When these two bounds meet, one can say that the computational problem has been solved, since one has found an algorithm that performs the minimum number of operations required, leaving no room for a faster one.

In practice many physical problems have not been algorithmically solved, presenting a gap between its lower and upper bound. This gap allows the computer science community to research on the subject, usually by designing better algorithms that can lower the upper bound. But improving the upper bound of an algorithm is not an easy task at all, as there are cases where it seems not possible to go beyond certain complexity barriers. Such is the case of *spin systems*, more specifically the problem of computing the partition function $Z(\ )$[1] of a spin lattice, for which no worst case polynomial algorithm exists since it is an *NP-Hard* problem [285]. In these situations the best one can do is to find a new algorithm that is faster than a previous one, but will still be exponential in time. The *transfer matrix* technique in the Potts model is an example of a method that can compute $Z(\ )$ much faster than other exact methods, but is still exponential on the width of the lattice.

[1]The partition function $Z(\ )$ can be understood as the analytical expression of a spin lattice that encodes all of its physical information

The study of spin systems has been for decades an important topic of research for the physics community because the results obtained can explain how condensed matter systems behave according to their temperature. Some spin models have even caught the attention of other fields for being able to explain aspects of cellular growth [104], social networks [68] and optimization problems [300] among others. The importance of spin models lies in the possibility to obtain macroscopic behavior from a microscopically defined system, through the computation of $Z(\ )$ that encodes exact physical information about the system. However, the number of combinatorial operations required to compute $Z(\ )$ grows so fast that the algorithms soon encounter an intractable scenario in time and memory usage. This intractability limitation has been the main problem for exact algorithms and eventually led to the formulation of Monte Carlo methods, that instead of computing $Z(\ )$, simulate a finite system obtaining averaged physical results that are not exact but can still be accurate enough. While it is true that Monte Carlo methods have allowed the study of much greater and complex systems than the exact methods, one cannot ignore the high amount of floating point operations and memory accesses still involved in the simulation process, as well as the difficulty for reaching equilibrium in systems with quenched disorder, such as Spin glass and Random field models. Today, the simulation of large 3D disordered systems presents a great challenge for computer science both in Monte Carlo convergence quality and high performance computing.

Computational research on spin systems can be divided in two types; (1) to find faster exact algorithms and (2) to find faster and better quality Monte Carlo methods. Both scenarios have challenging problems and present advantages as well as disadvantages regarding exactness and speed. It is wise then to ask: *how can the actual methods for spin lattices benefit from new algorithmic strategies as well as from the latest computational technologies?* Fortunately, now it is possible to improve computational methods not only by finding a new algorithm with a lower upper bound, but also by proposing *parallel algorithms*. The massification of parallel architectures, both CPUs and GPUs, provide an opportunity to further improve the running time of exact and probabilistic algorithms. The CPU's flexible multicore architecture makes it a good candidate for exact methods, since the computation of the $Z(\ )$ requires irregular memory access patterns that can only be handled with efficient caches. On the other hand, GPU architectures can provide performance that can be an order of magnitude higher than CPUs for bandwidth and floating point intensive problems such as the Monte Carlo simulations. But in order to achieve such level of performance, GPU-based algorithms have to be carefully re-designed for its architecture, presenting a great challenge.

Parallel computing has become an active field in computer science and its research is usually devoted at finding new ways of parallelization, automatic or manual, that can scale efficiently on modern parallel processors [200]. What parallel computing seeks as a final goal is to understand what parallel computation means at a more fundamental level and eventually have a general parallel computing model that can immediately tell us what are the limits of parallelism for a given problem under a certain architecture. Much research has been done towards this direction and today there is a much more solid base of concepts and patterns that can be used in order to take full potential of the field. ***The possibility to improve exact and Monte Carlo methods by proposing new parallel algorithmic strategies constitutes the main motivation of this thesis project.***

The Problems and Goal

In this thesis we focus on two open problems:

1. In transfer matrix methods, the number of combinatorial possibilities to explore grows exponentially as a function of the lattice's width, presenting an intractable scenario at a very early stage. Although the transfer matrix method is more efficient than a direct computation of the partition function $Z(\ )$, faster exact methods are still necessary in order to extend the possibilities of physical research at wider sizes. Our case of interest is the problem of computing the generic (q, v) transfer matrix for strip lattices in the *Potts model*. The cost of transfer matrix methods in the Potts model is directly related to the size of the configuration space, which is $O(4^m)$ in size for the generic (q, v) case. Our research question is: *Is it possible to design a generic* (q, v) *transfer matrix algorithm that uses a reduced configuration space and is still highly parallel for multi-core CPUs?*
2. Monte Carlo simulations on large 3D systems with quenched disorder require a large amount of computation and must traverse adverse energy landscapes in the low temperature regime in order to reach equilibrium, making the simulation susceptible to reach incorrect results. Our case of interest is the problem of simulating the 3D *Random Field Ising Model* (RFIM). Our research question is: *Can parallel architectures such as GPUs, combined with Monte Carlo techniques based on replicas, improve the simulation process for 3D lattices with quenched disorder?*

The goal of this thesis is to propose high performance solutions for both problems by combining new algorithmic techniques with parallel strategies, resulting in scalable performance.

Thesis Overview

The thesis has been organized into four Chapters. The parallel computing and physical backgrounds required for this thesis can found in **Chapters 1 and 2**, respectively, and can be read in any particular order. The aim of these two chapters is to present basic concepts, notations, known methods and terminologies that are used throughout the thesis. Additionally, these chapters allow the contributions to be in context to each field. The rest of the thesis is devoted to the actual contributions, each one self-contained in a dedicated chapter:

- **Chapter 3 (Contribution 1)** it focuses on the transfer matrix contribution. We analyze a way of building the transfer matrix based on the deletion contraction technique and present a new method, named *Parallel Family Trees* (PFT), that takes advantage of the symmetries found between elements of the configuration space. The chapter starts with the definition of a strip lattice, explains the transfer matrix method and discusses related work on transfer matrix algorithms, pointing out the works that directly relate to the research. A complete section is devoted to explain in detail how

one can reduce the configuration space, from $O(4^m)$ to $O(3^m)$, as well as how high parallelism is achieved. We include performance results for the algorithm and present physical results for both the square and kagome strips, at different widths.

- **Chapter 4 (Contribution 2)** is about the Monte Carlo contribution for fast GPU-based simulation on disordered systems. We propose an adaptive multi-GPU method based on the *Exchange Monte Carlo* algorithm, for the 3D *Ising Random Field Model*. The chapter begins by presenting the problematic behind quenched disorder and how the idea of replica based methods can bypass this issue. Then we present our multi-GPU Exchange Monte Carlo method, explaining in detail how the two levels of parallelism cooperate in order to offer a scalable multi-GPU solution. We explain the adaptive strategy for increasing the exchange rate of replicas, based on recursive middle-point insertions of temperatures. Finally, performance results are presented, showing that the multi-GPU method scales efficiently for two GPUs and that the adaptative temperatures strategy is more convenient than the uniform distribution approach.

In the conclusions we discuss the impact, applicability and limits of the results obtained as well as comment on the future work.

Chapter 1

Parallel Computing Background

For some computational problems, sequential algorithms are not fast enough to provide a solution in a reasonable[1] amount of time. Problems such as these can be found in natural sciences [117, 245, 265] (Physics, Biology, Chemistry), information technologies [264] (IT), geospatial information systems [25, 157] (GIS), structural mechanics problems [26] and even abstract mathematical/computer science (CS) problems [191, 219, 225, 246], among other fields. In many cases, these problems can be solved within a reasonable amount of time with the use of a parallel algorithm, expanding the possibilities of research for the given field.

In the past, the only way to run parallel algorithms was by building a cluster of computers or by having exclusive access to a super-computer. The first attempt on building a parallel machine at human-scale came only when the silicon of conventional CPUs could not reach higher frequencies due to physical constraints. At that moment, the computer industry was forced for the first time to expand the chip's architecture by adding multiple *cores* that would work independently one to each other, thus increasing the performance through parallelism. From that point and on, the CPU evolved to the known multi-core CPU which is a flexible and parallel processor. Over the years, the computer science community noted that for problems with critical regions and complex memory patterns, the multi-core CPU algorithms worked efficiently. But there were other type of problems, very parallelizable, for which the multi-core CPU architecture was not performing as fast as required. These problems are known as *data-parallel* problems and they characterize for having a high number of sub-problems that grow as a function of the problem size. For these types of problems one can use massively parallel processors and achieve a higher performance.

The story of how massive parallel processors were born is an interesting one because it combines two fields that were thought to be unrelated; *computational science* and *video-game industry*. The constant need for solving larger scientific problems eventually led to the construction of *super-computers* for understanding phenomena such as galaxy formation, molecular dynamics and climate change, among many others. As the scientific community expanded, the need for high performance computers that could be cheaper, accessible and smaller became important. On the other hand, the video-game industry has the goal of

[1] The notion of *reasonable* varies for each scientific field.

achieving real-time photo-realistic graphics, with the major restriction of running their lighting and polygon algorithms on consumer-level computer hardware. The need of realistic video-games led to the invention of the graphics accelerator, which is a small parallel processor that handles millions of floating point computations per second. The two needs, combined together, gave birth to the modern GPU.

The high compute power of GPUs, combined with the flexible parallelism of multi-core CPUs, can produce over one Teraflops of performance in a workstation machine, making research on high performance methods accessible by the computer science community worldwide. A great portion of modern research on parallel computing is devoted to study the possibilities of GPU computing when applied to scientific problems, finding out that a considerable amount of speedup can be obtained with respect to a sequential CPU-based solution [24, 65], sometimes reaching over an order of magnitude of speedup[59, 179].

Some problems however, cannot have a parallel solution [106]. For example, the approximation of $\sqrt{x}$ using the Newton-Rhapson method [222] cannot be parallelized because each iteration depends on the value of the previous one; there is the issue of *time dependence.* Such problems do not benefit from parallelism at all and are best solved using an efficient sequential algorithm. On the other hand, there are problems that can be naturally split into many independent sub-problems; *e.g.,* matrix multiplication can be split into several independent multiply-add computations. Such problems are massively parallel, they are very common in computational physics and they are best solved using parallel computing. In some cases, these problems become so parallelizable that they receive the name *embarrassingly parallel*[2] or *pleasingly parallel* [195, 218].

One of the most important aspects of parallel computing is its close relation to the underlying hardware and programming models. Typical questions in the field are: *What type of problem I am dealing with? Should I use a CPU or a GPU? Is it a MIMD or SIMD architecture? It is a distributed or shared memory system? What should I use: OpenMP, MPI, CUDA or OpenCL? Why the performance is not what I had expected? Should I use a hierarchical partition? how can I design a parallel algorithm?.* The answers to these questions are indeed important when searching for a high performance solution and they lie in the areas of algorithms, computer architectures, computing models and programming models. GPU computing also brings up additional challenges such as manual cache usage, parallel memory access patterns, communication, thread mapping and synchronization, among others. These challenges are critical for implementing an efficient parallel algorithm.

This chapter is a comprehensive survey of basic and advanced topics that are often required as parallel computing background. The reader should become more confident in the fundamental and technical aspects of parallel CPU and GPU computing, with a clear idea of what types of problems are best suited for each architecture.

The sections are organized in the following way: fundamental concepts of parallel computing and theoretical background are presented first, such as basic definitions, performance

[2]The term *embarrassingly parallel* means that it would be embarrassing to not take advantage of such parallelization. In some cases, the term has been misunderstood as of being embarrassed to make such parallelization for being too straightforward; this meaning is unwanted. An alternative name is *pleasingly parallel*.

measures, computing models, programming models and architectures (from section 1.1 to section 1.5). Concepts of GPU computing start from section 1.6, and cover strategies for designing massive parallel algorithms, the massive parallelism programming model and its technical restrictions.

1.1 Basic concepts

The terms *concurrency* and *parallelism* are often debated by the computer science community and sometimes it has become unclear what the difference is between the two, leading to misunderstanding of very fundamental concepts. Both terms are frequently used in the field of HPC and their difference must be made clear before discussing more advanced concepts along the survey. The following definitions of concurrency and parallelism are consistent and considered correct [38];

Definition 1.1 ***Concurrency*** *is a property of a program (at design level) where two or more tasks can be in progress simultaneously.*

Definition 1.2 ***Parallelism*** *is a run-time property where two or more tasks are being executed simultaneously.*

There is a difference between *being in progress* and *being executed* since the first one does not necessarily involve being in execution. Let C and P be concurrency and parallelism, respectively, then $P \subset C$. In other words, parallelism requires concurrency, but concurrency does not require parallelism. A nice example where both concepts come into play is the operating system (OS); it is concurrent by design (performs multi-tasking so that many tasks are in progress at a given time) and depending on the number of physical processing units, these tasks can run parallel or not. With these concepts clear, now we can make a simple definition for parallel computing:

Definition 1.3 ***Parallel computing*** *is the act of solving a problem of size n by dividing its domain into $k \geq 2$ (with $k \in \mathbb{N}$) parts and solving them with p physical processors, simultaneously.*

Being able to identify the type of problem is essential in the formulation of a parallel algorithm. Let P_D be a problem with domain D. If P_D is parallelizable, then D can be decomposed into k sub-problems:

$$D = \mathrm{d}_1 + \mathrm{d}_2 + ... + \mathrm{d}_k = \sum_{\mathrm{i}=1}^{k} \mathrm{d}_\mathrm{i} \tag{1.1}$$

P_D is a **data-parallel problem** if D is composed of data elements and solving the problem requires applying a kernel function $f(...)$ to the whole domain:

$$f(D) = f(\mathrm{d}_1) + f(\mathrm{d}_2) + ... + f(\mathrm{d}_k) = \sum_{\mathrm{i}=1}^{k} f(\mathrm{d}_\mathrm{i}) \tag{1.2}$$

P_D is a **task-parallel problem** if D is composed of functions and solving the problem requires applying each function to a common stream of data S:

$$D(S) = \mathrm{d}_1(S) + \mathrm{d}_2(S) + ... + \mathrm{d}_k(S) = \sum_{\mathrm{i}=1}^{k} \mathrm{d}_\mathrm{i}(S) \tag{1.3}$$

Data-parallel problems are ideal candidates for the GPU since its architecture works best when all threads execute the same instructions but on different data. On the other hand, task-parallel problems are best suited for the CPU because its architecture allows different tasks to be executed with flexible memory access patterns. Identifying the amount of data-parallelism and task-parallelism in a problem is critical for achieving the best partition of the problem domain, which is in fact the first step when designing a parallel algorithm. It also provides useful information when choosing the best hardware for the implementation (CPU or GPU). Computational physics problems often classify as data-parallel, as the chosen problems for this thesis, thus they are good candidates for a parallelization with multi-core CPUs and GPUs.

1.2 Performance measures

Performance measures consist of a set of metrics that can be used for quantifying the quality of an algorithm. For sequential algorithms, *time* and *space* give a rich amount of information about the algorithm, and in many occasions it is sufficient. For parallel algorithms the scenario is a little more complicated. Apart from time and space, metrics such as *speedup* and *efficiency* are necessary for studying the quality of a parallel algorithm. Furthermore, when an algorithm cannot be completely parallelized, it is useful to have a theoretical estimate of the maximum speedup possible. In these cases, the laws of Amdahl and Gustafson become useful for such analysis. On the experimental side, metrics such as *memory bandwidth* and *floating point operations per second* (Flops) define the performance of a parallel architecture when running a parallel algorithm.

Given a problem of size n, the running time of a parallel algorithm, using p processors, is denoted:

$$T(n, p) \tag{1.4}$$

From the theoretical point of view, the metrics *work* and *span* define the basis for computing other metrics such as speedup and efficiency.

1.2.1 Work and Span

The quality of a parallel algorithm can be defined by two metrics as stated by Cormen *et al.* [61]; *work* and *span*. Both metrics are important because they give limits to parallel computing and introduce the notion of *work*. Parallel algorithms have the challenge of being fast, but also to generate the minimum amount of additional work from the sequential algorithm. By doing less additional work, they become more efficient.

Work is defined as the total time needed to execute a parallel algorithm using one processor; denoted as $T(n,1)$. *Span* is defined as the longest time needed to execute a parallel path of computation by one thread; denoted as $T(n,\infty)$. Span is the equivalent of measuring time when using an infinite amount of processors.

These two metrics provide lower bounds for $T(n,p)$. The *work law* states the first lower bound:

$$T(n,p) \geq \frac{T(n,1)}{p} \tag{1.5}$$

That is, the running time of a parallel algorithm must be at least $1/p$ of its *work*. With the *work law*, one can realize that parallel algorithms run faster when the work per processor is balanced.

The *span law* defines the second lower bound for $T(n,p)$:

$$T(n,p) \geq T(n,\infty) \tag{1.6}$$

This means that the time of a parallel algorithm cannot be lower than the span or the minimal amount of time needed by a processor in an infinite processor machine.

1.2.2 Speedup

One of the most important actions in parallel computing is to actually measure how fast can a parallel algorithm run with respect to the best sequential one. This measure is known as *speedup*.

For a problem of size n, the expression for speedup is:

$$S_p = \frac{T_s(n,1)}{T(n,p)} \tag{1.7}$$

where $T_s(n,1)$ is the time of the best sequential algorithm (*i.e.,* $T_s(n,1) \leq T(n,1)$) and $T(n,p)$ is the time of the parallel algorithm with p processors, both solving the same problem. Speedup is upper bounded when n is fixed because of the *work law* from equation (1.5):

$$S_p \leq p \tag{1.8}$$

If the speedup increases linearly as a function of p, then we speak of *linear speedup*. Linear speedup means that the overhead of the algorithm is always in the same proportion with its running time, for all p. In the particular case of $T(n,p) = T_s(n,1)/p$, we then speak of *ideal speedup* or *perfect linear speedup*. It is the maximum theoretical value of speedup a parallel algorithm can achieve when n is fixed. In practice, it is hard to achieve linear speedup let alone perfect linear speedup, because memory bottlenecks and overhead increase as a function of p. What we find in practice is that most programs achieve *sub-linear* speedup, that is, $T(n,p) \geq T_s(n,1)/p$. Figure 1.1 shows the four possible curves.

For the last three decades it has been debated whether *super-linear speedup* (*i.e.,* $S_p > p$) is possible or or not. Super-linear speedup is an important matter in parallel computing

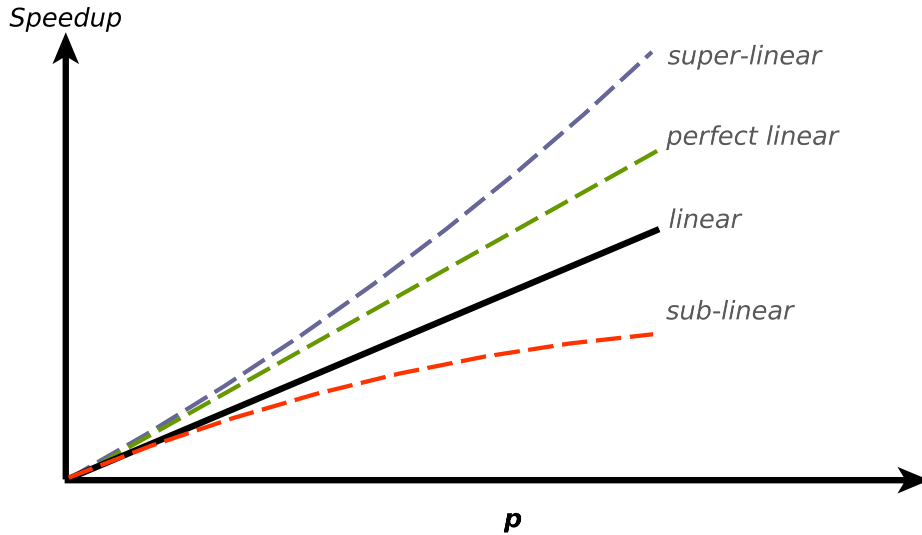

Figure 1.1: The four possible curves for speedup.

and proving its existence would benefit computer science, since parallel machines would be literally *more than the sum of their parts* (Gustafson's conclusion in [111]). Smith [251] and Faber *et al.* [81] state that it is not possible to achieve super-linear speedup and if such a parallel algorithm existed, then a single-core computation of the same algorithm would be no less than p times slower (leading to linear speedup again). On the opposite side, Parkinson's work [221] on parallel efficiency proposes that super-linear speedup is sometimes possible because the single processor has loop overhead. Gustafson supports super-linear speedup and considers a more general definition of S_p, one as the ratio of speeds ($speed = work/time$) [111] and not the ratio of times as in equation (1.7). Gustafson concludes that the definition of *work*, its assumption of being constant and the assumption of *fixed-size* speedup as the only model are the causes for thinking of the impossibility of super-linear speedup [113].

It is important to mention that there are three different models of speedup. (1) *Fixed-size speedup* is the one explained recently; fixes n and varies p. It is the most popular model of speedup. (2) *Scaled speedup* consists of varying n and p such that the problem size per processor remains constant. Lastly, (3) *fixed-time speedup* consists of varying n and p such that the amount of work per processor remains constant. Throughout this survey, *fixed-size speedup* is assumed by default. For the case of (2) and (3), speedup becomes a curve from the surface on (n, p).

If a problem cannot be completely parallelized (one of the causes for sub-linear speedup), a partial speedup expression is needed in the place of equation (1.7). Amdahl and Gustafson proposed each one an expression for computing partial speedup. They are known as the *laws of speedup*.

Amdahl's law

Let c be the fraction of a program that is parallel, $(1-c)$ the fraction that runs sequential and p the number of processors. Amdahl's law [12] states that for a fixed size problem the expected overall speedup is given by:

$$S(p) = \frac{1}{(1-c) + \frac{c}{p}} \tag{1.9}$$

If $p \approx \infty$, equation (1.9) becomes:

$$S(p) = \frac{1}{1-c} \tag{1.10}$$

That is, if a computer has a large number of processors (*i.e.,* a super-computer or a modern GPU), then the maximum speedup is limited by the sequential part of the algorithm (*e.g,* if $c = 4/5$ then the maximum speedup is 5x).

Amdahl's law is useful for algorithms that need to scale its performance as a function of the number of processors, fixing the problem size n. This type of scaling is known as *strong scaling.*

Gustafson's law

Gustafson's law [110] is another useful measure for theoretical performance analysis. This metric does not assume a fixed size of the problem as Amdahl's law did. Instead, it uses the fixed-time model where work per processor is kept constant when increasing p and n. In Gustafson's law, the time of a parallel program is composed of a sequential part s and a parallel part c executed by p processors.

$$T(p) = s + c \tag{1.11}$$

If the sequential time for all the computation is $s + cp$, then the speedup is:

$$S(p) = \frac{s+cp}{s+c} = \frac{s}{s+c} + \frac{cp}{s+c} \tag{1.12}$$

Defining α as the fraction of serial computation $\alpha = s/(s+c)$, then the parallel fraction is $1 - \alpha = c/(s+c)$. Finally, equation (1.12) becomes the *fixed-time speedup* $S(p)$:

$$S(p) = \alpha + p(1-\alpha) = p - \alpha(p-1) \tag{1.13}$$

Gustafson's law is important for expanding the knowledge in parallel computing and the definition of speedup. With Gustafson's law, the idea is to increase the work linearly as a function of p and n. Now the problem size is not fixed anymore, instead the work per processor is fixed. This type of scaling is also known as *weak scaling.* There are many applications where the size of the problem would actually increase if more computational power was available; weather prediction, computer graphics, Monte Carlo algorithms, particle simulations, etc. Fixing the problem size and measuring *time* vs p is mostly done for academic purposes. As the problem size gets larger, the parallel part p may grow faster than α.

While it is true that speedup might be one of the most important measures of parallel computing, there are also other metrics that provide additional information about the quality of a parallel algorithm, such as the efficiency.

1.2.3 Efficiency

If we divide expression (1.8) by p, we get:

$$E_p = \frac{S_p}{p} = \frac{T_s(n,1)}{pT(n,p)} \leq 1 \tag{1.14}$$

E_p is the efficiency of an algorithm using p processors and it tells how well the processors are being used. $E_p = 1$ is the maximum efficiency and means optimal usage of the computational resources. Maximum efficiency is difficult to achieve in an implemented solution (it is a consequence of the difficult to achieve perfect linear speedup). Today, efficiency has become as important as speedup, if not more, since it measures how well the hardware is used and it tells which implementations should have priority when competing for limited resources (cluster, supercomputer, workstation).

1.2.4 FLOPS

The FLOPS metric represents raw arithmetic performance and is measured as the number of floating point operations per second. Let F_h be the peak floating point performance of a known hardware and F_e the floating point performance measured for the implementation of a given algorithm, then F_c is defined as:

$$F_c = \frac{F_\mathrm{e}}{F_h} \tag{1.15}$$

F_c tells us the efficiency of the numerical computation relative to a given hardware. A value of $F_c = 1$ means maximum hardware usage for numerical computations.

In the year 2014, the fastest floating point performance reported was approximately 33.8 PFLOP/s by the *Tianhe-2* supercomputer located in the National Super Computer Center in Guangzhou, China. A list of the 500 most powerful super-computers in the world is kept updated each year at the site 'www.top500.org'. There is high enthusiasm for achieving for the first time the Exaflops scale. It is believed that in the following years, with the help of GPU-based hardware, which can be up to an order of magnitude faster than a CPU, the goal of Exaflops scale will be achieved. Figure 1.2 shows the performance between Nvidia's GPUs and Intel CPUs through the years. From the Figure one can see that the GPU performance is approximately five times higher for double precision (FP64) and almost to 10 times higher for single precision (FP32).

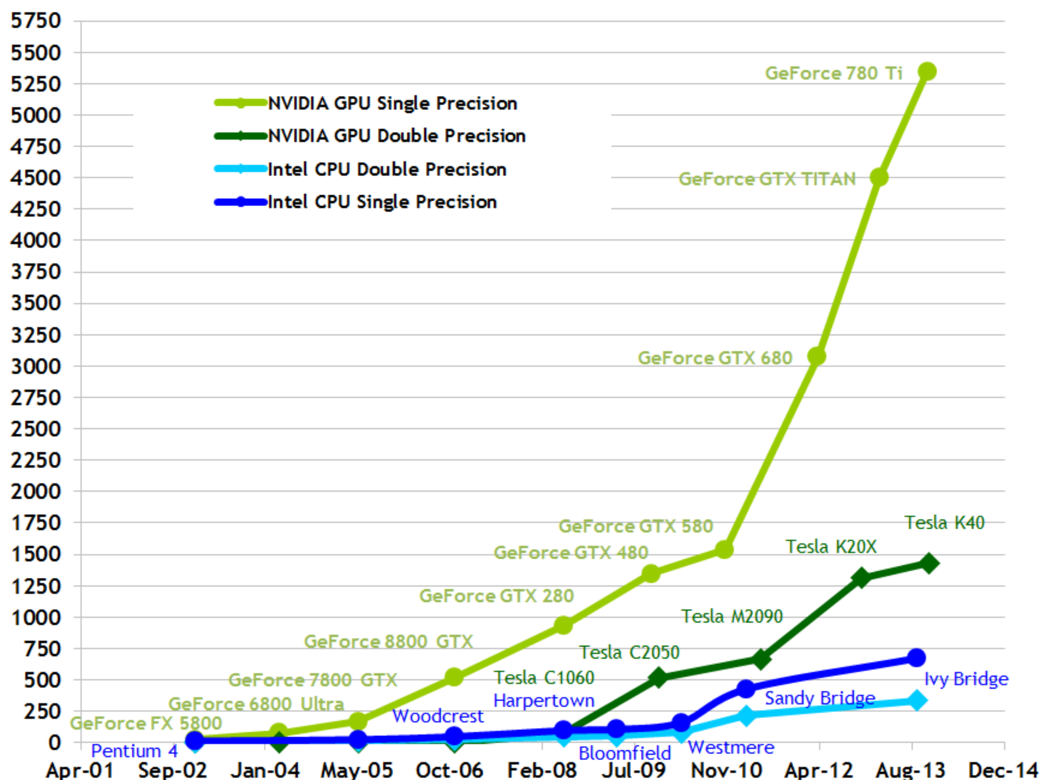

Figure 1.2: Comparison of CPU and GPU single precision floating point performance through the years. Plot taken from Nvidia's *CUDA C programming guide* [209].

1.2.5 Performance per Watt

In recent years, power consumption has become an important matter for sustainable technology. Today the notion of *performance per watt*[3] is one of the most important measures for choosing hardware and has been the subject of research [17]. The initiative to develop energy efficient hardware began as a way of doing HPC in a responsible manner. Latest CPU architectures such as Intel's Haswell CPUs and Nvidia's Maxwell GPUs have opted at improving the performance per Watt.

1.2.6 Memory Bandwidth

Memory Bandwidth is the rate at which data can be transferred between processors and main memory. It is usually measured as GB/s. The memory efficiency B_c of an implementation is computed by dividing the experimental bandwidth B_e by the maximum bandwidth B_h of the hardware:

$$B_c = \frac{B_e}{B_h} \tag{1.16}$$

[3]An updated list of the most energy efficient super computers is available at 'www.green500.org'.

A value of $B_c = 1$ means that the application is using the maximum memory bandwidth available on the hardware. Actual high-end CPUs have a memory bandwidth in the range $40GB/s \leq B_h \leq 80GB/s$ while high-end GPUs have a memory bandwidth in the range $200GB/s \leq B_h \leq 300GB/s$.

Achieving maximum bandwidth in the GPU sometimes can be much harder than in CPU. The main reason is because memory performance is problem-dependent. Data structures have to be aligned in simple patterns so that many chunks of data are read or written simultaneously for many threads. Irregular data accesses, non-compatible alignments and different data chunk sizes result in significant lower memory bandwidth. Latest GPU architectures such as *Fermi* and *Kepler* can mitigate this effect by using an L2 cache for global memory (see [63, 208] for more information on the GPU's L2 cache).

The performance measures presented in this section are all related in some way to the running time $T(n,p)$ of the parallel algorithm. Measuring the mean wall-clock time with a standard error below 5% is a good practice for obtaining an experimental value of $T(n,p)$. For the theoretical case, obtaining the analytical expression of $T(n,p)$ can be non-trivial because it will depend on *parallel computing model*.

1.3 Parallel Computing Models

Computing models are abstract computing machines that allow theoretical analysis on the cost of algorithms. These models simplify the computational scenario to a reduced set of parameters that define how much time a memory access or an arithmetic operation will cost. Theoretical analysis is fundamental for the process of researching new algorithms, since it can tell us which algorithm is asymptotically better. In the case of parallel computing, there are several models available, such as the *PRAM*, *PMH*, *Bulk parallel processing* and *LogP* models. The difference between one model and another basically resides on their definition of processor communication and memory access.

1.3.1 Parallel Random Access Machine (PRAM)

The *parallel random access machine*, or PRAM, was proposed by Fortune and Wyllie in 1978 [89]. It is inspired by the classic *random access machine* (RAM) and has been one of the most used models for parallel algorithm design and analysis.

In the 1990s, the PRAM model gained reputation as an unrealistic model for algorithm design and analysis because no computer could offer constant memory access times for simultaneous operations, let alone performance scalability. Implementations of PRAM-designed algorithms did not reflect the complexity the model was suggesting. However, in 2006, the model became relevant again with the introduction of general purpose GPU (GPGPU) computing APIs, although they are not pure PRAM machines.

In the PRAM model, there are p processors that operate synchronously over an unlimited memory completely visible for each processor (see Figure 1.3). The p parameter does not

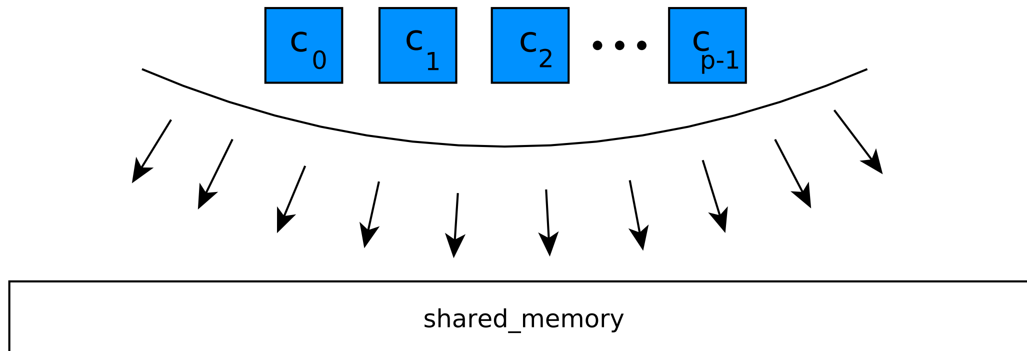

Figure 1.3: In the PRAM model each one of the cores has a complete view of the global memory.

need to be a constant number, it can also be defined as a function of the problem size n. Each r/w (read/write) operation costs $O(1)$. Different variations of the model exist in order to make it more realistic when modeling parallel algorithms. These variations specify whether memory access can be performed *concurrently* or *exclusively*. Four variants of the model exist.

EREW, or *Exclusive Read - Exclusive Write*, is a variant of PRAM where all read an write operations are performed exclusively in different places of memory for each processor. The EREW variation can be used when the problem is split into independent tasks, without requiring any sort of communication. For example, vector addition as well as matrix addition can be done with an EREW algorithm.

CREW, or *Concurrent Read - Exclusive Write*, is a variant of PRAM where processors can read from common sections of memory but always write to sections exclusive to one another. CREW algorithms are useful for problems based on tilings, where each site computation requires information from neighbor sites. Let k be the number of neighbors per site, then each site will perform at least k reads and one write operation. At the same time, each neighbor site will perform the same number of memory reads and writes. In the end, each site is read concurrently by k other sites but only modified once. This behavior is the main idea of a CREW algorithm. Algorithms for fluid dynamics, cellular automata, PDEs and N-body simulations are compatible with the CREW variation.

ERCW, or *Exclusive Read - Concurrent Write*, is a variant of PRAM where processors read from different exclusive sections of memory but write to shared locations. This variant is not as popular as the others because there are less situations one can model with the ERCW variation. Nevertheless, important results have been obtained for this variation. Mackenzie and Ramachandran proved that finding the maximum of n numbers has a lower bound of $\Omega(\sqrt{log\ n})$ under ERCW [183], while the problem is $\Theta(log\ n)$ under EREW/CREW.

CRCW, or *Concurrent Read - Concurrent Write*, is a variant of PRAM where processors can read and write from the same memory locations. Beame and Hastad have studied optimal solutions using CRCW algorithms [22]. Subramonian [257] presented an $O(log\ n)$ algorithm for computing the minimum spanning tree.

Concurrent writes are not trivial and must use one of the following protocols:

- *Common*: all processors write the same value;
- *Arbitrary*: only one write is successful, the others are not applied;
- *Priority*: priority values are given to each processor (*e.g.,* rank value), and the processor with highest priority will be the one to write;
- *Reduction*: all writes are reduced by an operator (add, multiply, OR, AND, XOR).

Over the last decades, Uzi Vishkin has been one of the main supporters of the PRAM model. He proposed an on-chip architecture based on PRAM [271] as well as the notion of explicit Multi-threading for achieving efficient implementations of PRAM algorithms [272].

1.3.2 Parallel Memory Hierarchy (PMH)

The *Parallel Memory Hierarchy* model, or PMH, was proposed in 1993 by Alpern *et al.* [10] and inspired by related works [3, 9] (HMM and UHM memory models). This model was proposed to deal with the inconsistency between theoretical and empirical performance of some PRAM algorithms, for assuming constant time memory accesses. Actual CPUs (such as Intel Xeon E5 series or AMD's Opteron 6000 series) have memory hierarchies composed of registers, L1, L2 and L3 caches. GPUs such as Nvidia GTX 680 or AMD's Radeon HD 7850 also have a memory hierarchy composed of registers, L1, L2 caches and the global memory. Indeed, the memory hierarchy should be considered when designing a parallel algorithm in order to match the theoretical complexity bounds.

The PMH model is defined by a hierarchical tree of memory modules. The leaves of the tree correspond to processors and the internal nodes represent memory modules. Modules closer to the processors are fast, but small, and modules far from the processors are slow, but larger. For the i-th level module, the following parameters are defined; s_i as the number of items per block (or block-size), n_i the number of blocks, l_i the latency and c_i is the child-count. In practice, it is easier to model an algorithm by using the uniform parallel memory hierarchy (UPMH) which is a simplified version of the PMH model. The UPMH model defines a complete τ-ary tree (see Figure 1.4).

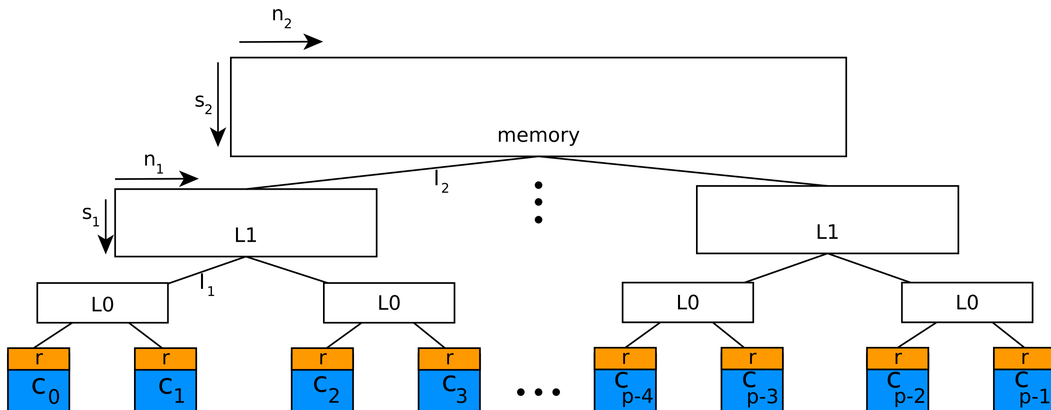

Figure 1.4: The uniform parallel memory hierarchy tree.

In UPMH, composite parameters are used, such as the aspect ratio $\alpha = n_i/s_i$, the packing factor $\rho = s_i/s_{i-1}$ and the branching factor τ which is the tree arity. Additionally, the UPMH model defines the transfer cost t_i as a function of the tree level; $t_i = f(i)$. Typical values of the transfer cost function are $f(i) = 1, i, \rho^i$. Function $f(i) = \rho^i$ is considered a realistic transfer cost function for modern architectures. Usually, the model is referred to as $\text{UPMH}_{\alpha,\rho,f(i),\tau}$ to indicate its four parameters. This model has proven to be more realistic than PRAM, but harder for analyzing algorithms.

Alpern *et al.* showed that an non-blocked matrix multiplication algorithm (*i.e.*, the basic matrix multiplication algorithm) can cost $\Omega(N^5/p)$ time instead of $O(N^3/p)$ [35] as in PRAM. In the same work, the authors prove that a parallel block-based matrix multiplication algorithm (see Figure 1.5) can indeed achieve the desired $O(N^3/p)$ upper bound by reusing the data from the fastest memory modules.

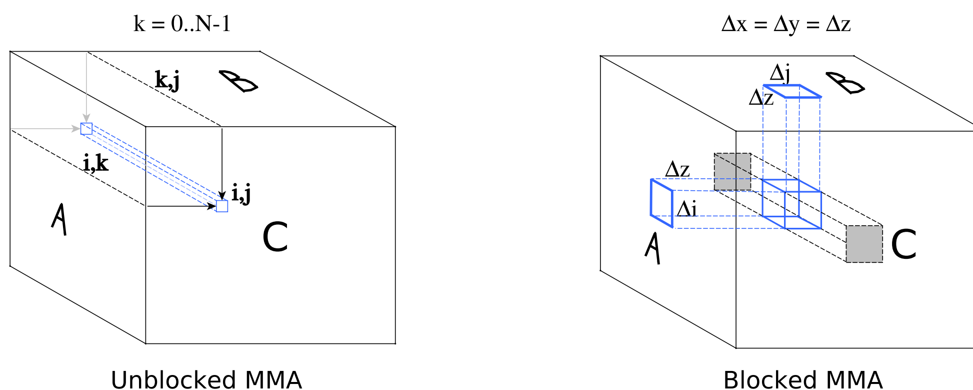

Figure 1.5: Classic algorithm processes sticks of computation as seen in the left side. The blocked version computes sub-cubes of the domain in parallel, taking advantage of locality.

The entire proof of the matrix multiplication algorithm for one processor can be found in the work of Alpern *et al.* [9]. The UPMH model can be considered a complement to other models such as PRAM or BSP.

1.3.3 Bulk Synchronous Parallel (BSP)

The *Bulk synchronous parallel*, or BSP, is a parallel computing model focused on communication, published in 1990 by Leslie Valiant [270]. Synchronization and communication are considered high priority in the cost equation. The model consists of a number of processors with fast local memory, connected through a network and capable of sending and receiving messages to and from any other processor. A BSP-based algorithm is composed of *super-steps* (see Figure 1.6).

A super-step is a parallel block of computation composed of three steps:

- Local computation: p processors perform up to L local computations;
- Global communication: Processors can send and receive data among them;
- Barrier synchronization: Wait for all other processors to reach the barrier.

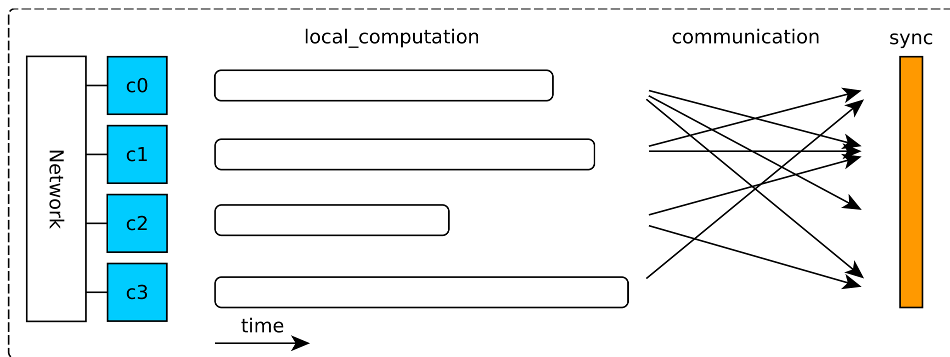

Figure 1.6: A representation of a super-step; processing, communication and a global synchronization barrier.

The cost c for a super-step using p processors is defined as:

$$c = max_{\mathrm{i}=1}^{p}(w_{\mathrm{i}}) + g\ max_{\mathrm{i}=1}^{p}(h_{\mathrm{i}}) + l \tag{1.17}$$

where w_{i} is the computation time of the i-th processor, h_{i} the number of messages used by the i-th processor, g is the capability of the network and l is the cost of the barrier synchronization. In practice, g and l are computed empirically and available for each architecture as lookup values. For an algorithm composed of S super-steps, the final cost is the sum of all the super-step costs:

$$C = \sum_{\mathrm{i}=1}^{S} c_{\mathrm{i}} \tag{1.18}$$

1.3.4 LogP

The LogP model was proposed in 1993 by Culler *et al.* [66]. Similar to BSP, it focuses on modeling the cost of communicating a set of distributed processors (*i.e.,* network of computers). In this model, local operations cost one unit of time but the network has latency and overhead. The following parameters are defined:

- *latency (L)*: the latency for communicating a message containing a word (or small number of words) from its source to its target processor;
- *overhead (o)*: the amount of time a processor spends in communication (sending or receiving). During this time, the processor cannot perform other operations;
- *gap (g)*: the minimum amount of time between successive messages in a given processor;
- *processors (P)*: the number of processors.

All parameters, except for the processor count (P), are measured in cycles. Figure 1.7 illustrates the model with an example of communication with one-word messages. The LogP model is similar to the BSP with the difference that BSP uses global barriers of synchronizations while LogP synchronizes by pairs of processors. Another difference is that LogP considers a message overhead when sending and receiving. Choosing the right model (BSP or LogP) depends if global or local synchronization barriers are predominant and if the com-

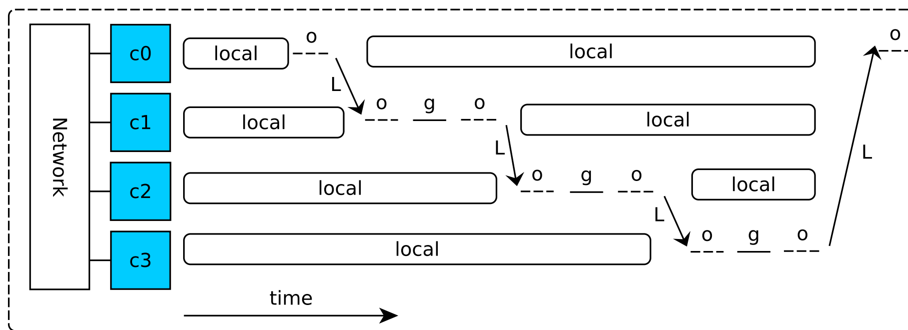

Figure 1.7: An example communication using the LogP model.

munication overhead is significant or not.

Parallel computing models are useful for analyzing the running time of a parallel algorithm as a function of n, p and other parameters specific to the chosen model. But there are also other important aspects to be considered related to the styles of parallel programming. These styles are well explained by the *parallel programming models*.

1.4 Parallel programming models

Parallel programming models focus on the expressiveness when programming parallel machines. For example, PRAM and UPMH use the *shared memory* model, while LogP and BSP use a *message passing* model. These two models are actually *parallel programming models*. A *parallel programming model*[4] is an abstraction of the programmable aspects of a computing model. While computing models from section 1.3 are useful for algorithm design and analysis (*i.e.,* computing time complexity), *parallel programming models* are useful for expressing the parallelism of the algorithm. In this section we cover four relevant parallel programming models.

1.4.1 Shared memory

In the shared memory model, threads can read and write asynchronously within a common memory. This programming model works naturally with the PRAM computing model and it is mostly useful for *multi-core* and GPU based solutions. A well known API for CPUs is the *Open Multiprocessing* interface or OpenMP [51] which is based on the Unix *pthreads* implementation [204]. In the case of GPUs, OpenCL [156] and CUDA [209] are the most common.

Many times, a shared memory parallel algorithm needs to manage non-deterministic behavior from multiple concurrent threads (the operating system thread scheduling is considered

[4]Some of the literature may treat the concept of *parallel programming model* as equal to *computing model*. In this survey we denote a difference between the two; thus the sections (1.3) and (1.4).

non-deterministic). When concurrent threads read and write on the same memory locations, one must supply an explicit synchronization and control mechanism such as *monitors [127], semaphores [77], atomic operations and mutexes (a binary semaphore)*. These control primitives allow threads to lock and work on shared resources without other threads interfering, making the algorithm consistent. In some scenarios the programmer must also be aware of the *shared memory consistency model*. These models define rules and the strategy used to maintain consistency on shared memory. A detailed explanation of consistency models is available in Adve *et al.* work [2].

For the case of GPUs, one can use atomic operations, synchronization barriers and memory fences [209].

1.4.2 Message passing

In a message passing programming model, or distributed model, processors communicate asynchronously or synchronously by sending and receiving messages containing words of data. In this model, emphasis is placed on communication and synchronization making distributed computing the main application for the model. Dijkstra introduced many new ideas for consistent concurrency on distributed systems based on exclusion mechanisms [74]. This programming model works naturally with the BSP and LogP models which were built with the same paradigm.

The standard interface for message passing is the *Message Passing Interface* or MPI [95]. MPI is used for handling communication in CPU distributed applications and is also used to distribute the work when using multiple GPUs.

1.4.3 Implicit

Implicit parallelism refers to compilers or high-level tools that are capable of achieving a degree of parallelism automatically from a sequential piece of source code. The advantage of implicit parallelism is that all the hard work is done by the tool or compiler, achieving practically the same performance as a manual parallelization. The disadvantage however is that it only works for simple problems such as *for* loops with independent iterations. Kim *et al.* [159] describe the structure of a compiler capable of implicit and explicit parallelism. In their work, the authors address the two main problems for achieving their goal; *(1) how to integrate the parallelizing preprocessor with the code generator* and *(2) when to generate explicit and when to generate implicit threads.*

Map-Reduce [67] is a well known implicit parallel programming tool (sometimes considered a programming model itself) and has been used for frameworks such as Hadoop with outstanding results at processing large data-sets over distributed systems. Functional languages such as Haskell or Racket also benefit from the parallel map-reduce model. In the 90's, High performance Fortran (HPF) [178] was a famous parallel implicit API. OpenMP can also be considered semi-implicit since its parallelism is based on hints given by the programmer. To-

day, *pH* (parallel Haskell) is probably the first fully implicit parallel programming language [206]. Automatic parallelization is hard to achieve for algorithms not based on simple loops and has become a research topic in the last twenty years [109, 168, 181, 237].

1.4.4 Algorithmic skeletons

Algorithm skeletons provide an important abstraction layer for implementing a parallel algorithm. With this abstraction, the programmer can now focus more on the strategy of the algorithm rather than on the technical problems regarding parallel programming. Algorithm skeletons, also known as parallelism patterns, were proposed by Cole in 1989 and published in 1991 [58]. This model is based on a set of available parallel computing patterns known as *skeletons* (implemented as higher order functions to receive other functions) that are available to use. The critical step when using algorithmic skeletons is to choose the right pattern for a given problem. The following patterns are some of the most important for parallel computing:

- Farm: or *parallel map*, is a master-slave pattern where a function $f()$ is replicated to many slaves so that slave s_{i} applies $f(x_{\mathrm{i}})$ to sub-problem x_{i};
- Pipeline: or *function decomposition*, is a staged pattern where $f()_1 - > f()_2 - > ... - > f()_n$ are parts of a larger logic that works as a pipeline. Each stage of the pipeline can work in parallel;
- Parallel tasks: In this pattern, $f()_1, f()_2, ..., f()_n$ are different tasks to be performed in parallel. These tasks can run completely independent or can include critical sections;
- Divide and Conquer: This is a recursive pattern where a problem A, a divide function $\mathrm{d} : A \rightarrow \{a_1, a_2, ..., a_k\}$ and a combine function $c : \{a_1, a_2, ..., a_k\},\ f() \rightarrow f(\{a_1, a_2, ..., a_k\})$ are passed as parameters for the skeleton. Then the skeleton applies a divide and conquer approach spanning parallel branches of computation as the recursion tree grows.

$$r(A, \mathrm{d}, c, f) = c(\{r(\mathrm{d}(A)_1, \mathrm{d}, c), r(\mathrm{d}(A)_2, \mathrm{d}, c), ..., r(\mathrm{d}(A)_k, \mathrm{d}, c)\}, f) \quad (1.19)$$

$$r(A', \mathrm{d}, c, f) = c(\mathrm{d}(A'), f) \quad (1.20)$$

The recursion stops when the smallest sub-problems $\mathrm{d}(A')$ are reached.

There are also basic skeleton patterns for managing *while* and *for* loops as well as conditional statements. The advantage of algorithmic skeletons is their ability to be combined or nested to make more complex patterns (because they are higher order functions). Their limitation is that the abstraction layers include an overhead cost in performance.

In the previous two sections we covered computing models and programming models which are useful for algorithm analysis and programming, respectively. It is critical however, when implementing a high performance solution, to know how the underlying architecture actually works.

1.5 Architectures

Computer architectures define the way processors work, communicate and how memory is organized; all in the context of a fully working computer (note, a working computer is an implementation of an architecture). Normally, a computer architecture is well described by one or two computing models. It is important to say that the goal of computer architectures is not to implement an existing computing model. In fact, it is the other way around; computing models try to model actual computer architectures. The final goal of a computer architecture is to make a computer run programs efficiently and as fast as possible. In the past, implementations achieved higher performance automatically because the hardware industry increased the processor's frequency. At that time there were not many changes regarding the architecture. Now, computer architectures have evolved into parallel machines because the single core clock speed has reached its limit in frequency[5] [236]. Today the most important architectures are the *multi-core* and *many-core*, represented by the CPU and GPU, respectively.

Unfortunately, sequential implementations will no longer run faster by just buying better hardware. They must be re-designed as a parallel algorithm that can scale its performance as more processors are available. Aspects such as the type of instruction/data streams, memory organization and processor communication indeed help for achieving a better implementation.

1.5.1 Flynn's taxonomy

Computer architectures can be classified by using Flynn's taxonomy [88]. Flynn realized that all architectures can be classified into four categories. This classification depends on two aspects; number of instructions and number of data streams that can be handled in parallel. He ended with four classifications.

SISD, or single instruction single data stream can only perform one instruction to one data stream at a time. There is no parallelism at all. Old single core CPUs of the 1950s, based on the original Von Neumann architecture, were all SISD types. Intel processors from *8086* to *80486* were also SISD.

SIMD, or single instruction multiple data streams can handle only one instruction but apply it to many data streams simultaneously. These architectures allow *data parallelism*, which is useful in science for applying a mathematical model to different parts of the problem domain. Vector computers in the 70's and 80's were the first to implement this architecture. Nowadays, GPUs are considered an evolved SIMD architecture because they work with many SIMD batches simultaneously. SIMD has also been supported on CPUs as instruction sets, such as MMX, 3DNow!, SSE and AVX. These instruction sets allow parallel integer or floating point operations over small arrays of data.

MISD, or multiple instruction single data stream can handle different tasks over the same stream of data. These architectures are not so common and often end up being

[5] Above 4.0 GHz of frequency, silicon transistors can become too hot for conventional cooling systems.

implemented for specific scenarios such as digital attack systems (*e.g.,* to destroy a data encryption) or fault tolerance systems and space flight controllers (NASA).

MIMD, or multiple instruction multiple data streams is the most flexible architecture. It can handle one different instruction for each data stream and can achieve any type of parallelism. However, the complexity of the physical implementation is high and often the overhead involved in handling such different tasks and data streams becomes a problem when trying to scale with the number of cores. Modern CPUs fall into this category (Intel, AMD and ARM *multi-cores*) and newer GPU architectures are partially implementing this architecture. The MIMD concept can be divided into SPMD (single program multiple data) and MPMD (multiple programs multiple data). SPMD occurs when a simple program is being executed in different processors. The key difference compared to SIMD is that in SPMD each processor can be at a different stage of the execution or at different paths of the code caused by conditional branching. MPMD occurs when different independent programs are being run on multiple processors.

1.5.2 Memory architectures and organizations

There are two forms of memory organization, shared and distributed. In distributed memory, each node has its own memory architecture and it is completely independent from other nodes. Communication is based on messages between nodes through a network. In a distributed memory scenario, the network plays a important role and its topology is different depending on the context. Some common topologies are *bus, star, ring, mesh, hypercube and tree.* Also, hybrid topologies are made based on the basic ones already mentioned.

In a shared memory organization, processors communicate through a common bank of global memory, not needing explicit messages as in a distributed memory scheme. Today, two architectures are mostly used; *UMA* and *NUMA*.

Uniform Memory Access **or UMA** consists of a shared memory in which the access time for any processor takes the same amount of time no matter the data location. UMA is also known as *Symmetric Multi-Processors* or SMP. The main disadvantage of UMA is the low scalability when increasing the number of processors. This occurs because of the single memory controller shared for all processors.

Non Uniform Memory Access **or NUMA** is an architecture where access time to shared memory depends on the location of data relative to the processor. This means that the memory that is closer to a processor is accessed much faster than memory closer to another processor (*i.e.,* cost is a function of distance). To take advantage of NUMA, the problem must be split into independent chunks of data, each one assigned to a unique CPU. Also, global *read-only* data is better replicated than shared. In practice, all NUMA architectures implement a hardware cache-coherence logic and become *cache-coherent NUMA* or ccNUMA.

One can find the SMP architecture in many desktop computers with dual core hardware and the NUMA architecture in modern multi-core machines with two or more CPU sockets (*e.g.,* AMD Opteron and Intel Xeon). Figure 1.8 shows the concepts of UMA and NUMA.

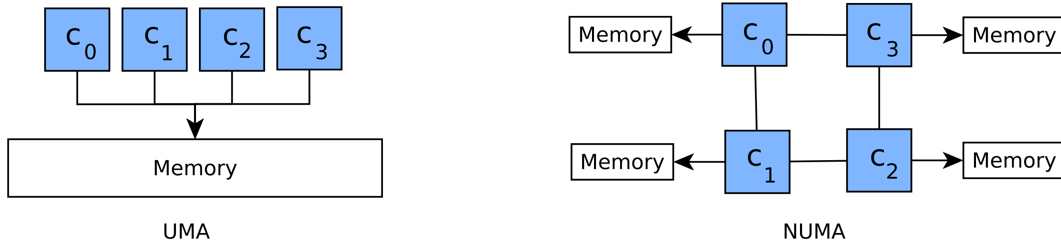

Figure 1.8: The UMA (*aka* SMP) and NUMA memory architectures.

Finally, shared and distributed memory architectures can also be mixed, leading to a *hybrid configuration* which is useful for MPI + OpenMP or MPI + GPGPU solutions.

1.5.3 Technical details of modern CPU and GPU architectures

The differences between *multi-core* and *many-core* architectures can be visualized in the schematics of modern CPUs and GPUs.

Actual high-end CPUs, such as the Xeon E5 2600, are built with many interconnections between the cores providing flexibility in communication (see Figure 1.9). Each core has a local L1 and L2 cache of 64KB and 256KB, respectively, and in the center of the chip there is a larger L3 cache of size 20MB, shared by all cores. The *Quick-Path Interconnect* or QPI section (known as *Hyper-transport* for AMD processors) of the chip implements part of the NUMA memory architecture. The PCI module handles communication with the PCI ports and finally the *Internal Memory Controller*, or IMC, handles the memory access to its section of RAM, completing the rest of the NUMA architecture.

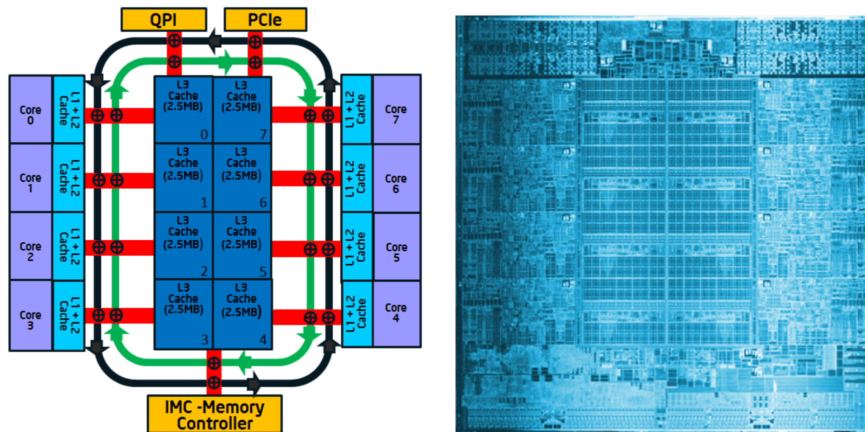

Figure 1.9: On the left, an Intel Xeon E5 2600 (2012) chip schematic. On the right, the actual chip.

On the other hand, modern GPUs such as the Tesla K20X have a completely different chip schematic that is oriented to massive parallelism. Figure 1.10 shows the schematic of an Nvidia Tesla K20X GPU as well as its actual chip. The cores of the GPU are grouped into SMX units, or *next generation streaming multiprocessors*. The most important aspects that characterize a GPU are inside the SMX units (see Figure 1.11).

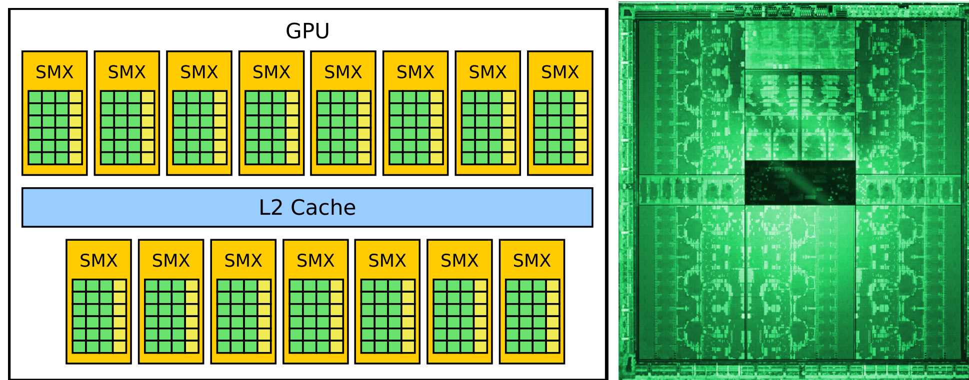

Figure 1.10: On the left, Nvidia's Tesla K20X GPU schematic. On the right, a picture of its chip.

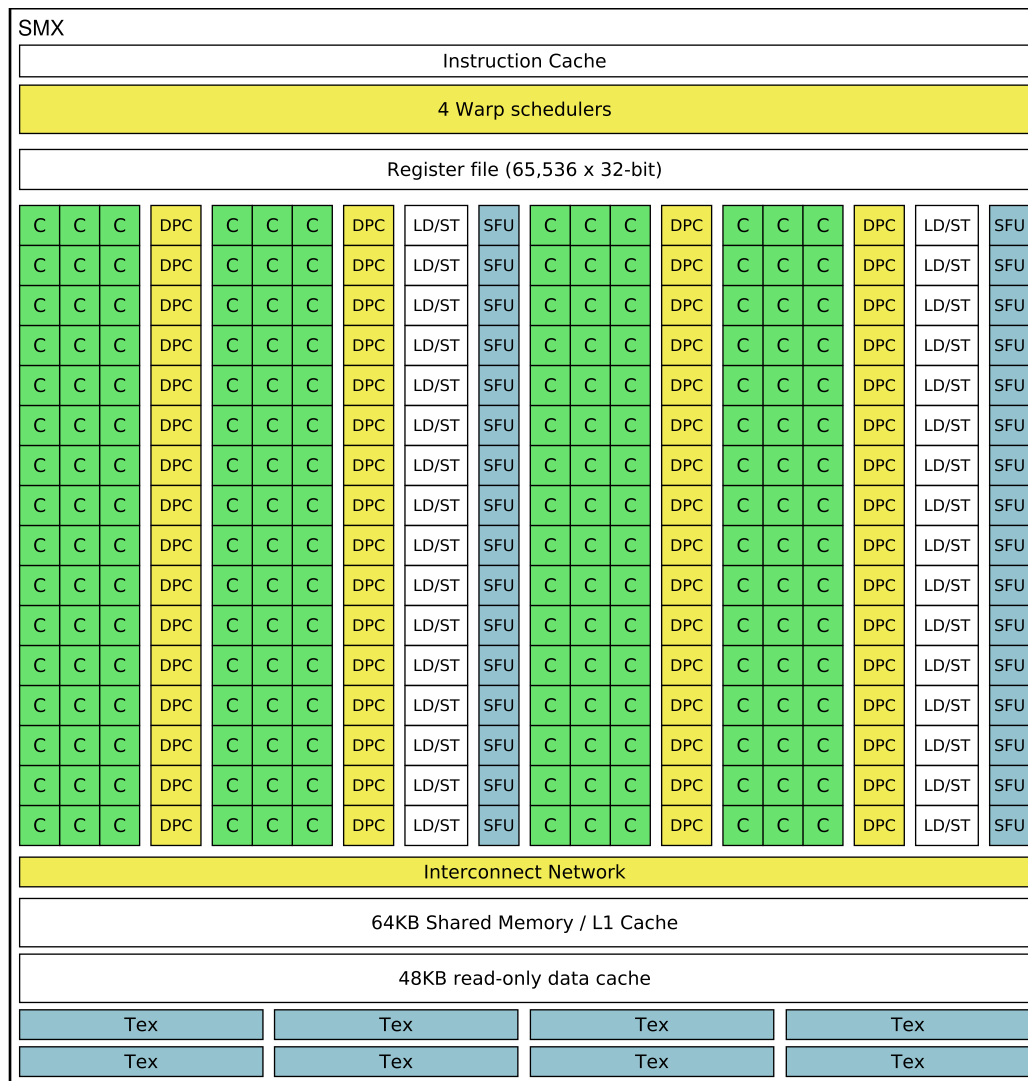

Figure 1.11: A diagram of a streaming multiprocessor next-generation (SMX). Image inspired from Nvidia's CUDA C programming guide [209].

A SMX is the smallest unit capable of performing parallel computing. The main difference between a low-end GPU and a high-end GPU of the same architecture is the number of SMX

units inside the chip. In the case of Tesla K20 GPUs, each SMX unit is composed of 192 cores (represented by the C boxes). Its architecture was built for a maximum of 15 SMX, giving a maximum of 2,880 cores. However in practice, some SMX are deactivated because of production issues.

The cores of a SMX are 32-bit units that can perform basic integer and single precision (FP32) floating point arithmetic. Additionally, there are 32 special function units or SFU that perform special mathematical operations such as *log, sqrt, sin and cos*, among others. Each SMX has also 64 double precision floating point units (represented as DPC boxes), known as FP64, and 32 LD/ST units (load / store) for writing and reading memory.

Numerical performance of GPUs is classified into two categories; FP32 and FP64 performance. The FP32 performance is always greater than FP64 performance. This is actually a problem for massive parallel architectures because they must spend chip surface on special units of computation for increasing FP64 performance. The Tesla K20X GPU can achieve close to 4TFlops of FP32 performance while only 1.1TFlops in FP64 mode.

Actual GPUs such as the Tesla K20 implement a four-level memory hierarchy; (1) registers, (2) L1 cache, (3) L2 cache and (4) global memory. All levels, except for the global memory, reside in the GPU chip. The L2 cache is automatic and it improves memory accesses on global memory. The L1 cache is manual, there is one per SMX, and it can be as fast as the registers. Kepler and Fermi based GPUs have L1 caches of size 64KB that are split into $16KB$ of programmable shared memory and $48KB$ of automatic cache, or *vice versa*.

1.5.4 The fundamental difference between CPU and GPU architectures

Modern CPUs have evolved towards parallel processing, implementing the MIMD architecture. Most of their die surface is reserved for control units and cache, leaving a small area for the numerical computations. The reason is, a CPU performs such different tasks that having advanced cache and control mechanisms is the only way to achieve an overall good performance.

On the other hand, the GPU has a SIMD-based architecture that can be well represented by the PRAM and UPMH models (sections 1.3.1 and 1.3.2, respectively). The main goal of a GPU architecture is to achieve high performance through massive parallelism. Contrary to the CPU, the die surface of the GPU is mostly occupied by ALUs and a minimal region is reserved for control and cache (see Figure 1.12). Efficient algorithms designed for GPUs have reported over $10\times$ speedup over CPU implementations [59, 179].

This difference in architecture has a direct consequence, the GPU is much more restrictive than the CPU but it is much more powerful if the solution is carefully designed for it. Latest GPU architectures such as Nvidia's Fermi and Kepler have added a significant degree of flexibility by incorporating a L2 cache for handling irregular memory accesses and by improving the performance of atomic operations. However, this flexibility is still far from the one found in CPUs.

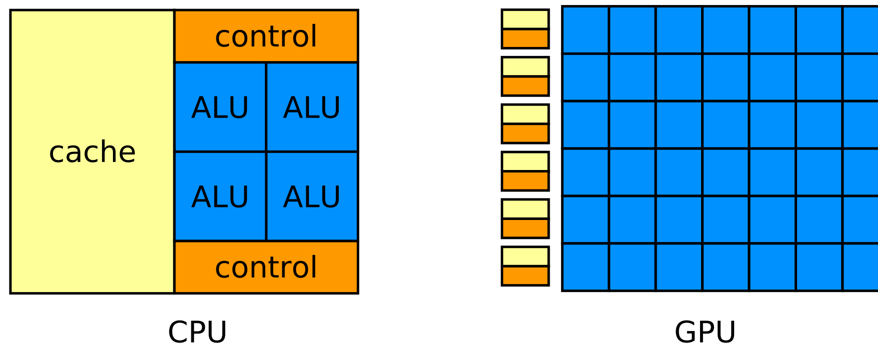

Figure 1.12: The GPU architecture differs from the one of the CPU because its layout is dedicated for placing many small cores, giving little space for control and cache units.

Indeed there is a trade-off between flexibility and computing power. Actual CPUs struggle to maintain a balance between computing power and general purpose functionality while GPUs aim at massive parallel arithmetic computations, introducing many restrictions. Some of these restrictions are overcome at the implementation phase while some others must be treated when the problem is being parallelized. It is always a good idea to follow a strategy for designing a parallel algorithm.

1.6 Strategy for designing a parallel algorithm

Designing a new algorithm is not a simple task. In fact, it is considered an art [163, 215] that involves a combination of mathematical background, creativity, discipline, passion and probably other unclassifiable abilities. In parallel computing the scenario is no different, there is no golden rule for designing perfect parallel algorithms.

However, there are some formal strategies that are frequently used for creating efficient parallel algorithms. Leighton and Thomson [175] have contributed considerably to the field by pointing out how data structures, architectures and algorithms relate when facing the act of implementing a parallel algorithm. In 1995, Foster [91] identified a four-step strategy that is present in many well designed parallel algorithms; *partitioning, communication, agglomeration and mapping* (see Figure 1.13).

1.6.1 Partitioning

The first step when designing a parallel algorithm is to split the problem into parallel sub-problems. In *partitioning*, the goal is to find the best possible partition; one that generates the highest amount of sub-problems (at this point, communication is not considered yet).

Identifying the ***domain type*** is critical for achieving a good partition of a problem. If the problem is *data-parallel*, then the data is partitioned and we speak of ***data parallelism***. On the other hand, if the problem is *task-parallel*, then the functionality is partitioned and

we speak of ***task-parallelism***. Most of the computational physics problems based on simulations are suitable for a *data-parallelism* approach, while problems such as parallel graph traverse, communication flows, traffic management, security and fault tolerance often fall into the *task-parallelism* approach.

1.6.2 Communication

After partitioning, communication is defined between the sub-problems (task or data type). There are two types of communication; *local communication* and *global communication*. In local communication, sub-problems communicate with neighbors using a certain geometric or logical pattern. Global communications involve broadcast, reductions or global variables. In this phase, all types of communication problems are handled; from race conditions handled by critical sections or atomic operations, to synchronization barriers to ensure that the strategy of computation is working up to this point.

1.6.3 Agglomeration

At this point, there is a chance that sub-problems may not generate enough work to become a thread of computation (given a computer architecture). This aspect is often known as the *granularity* of an algorithm [52]. A *fine-grained* algorithm divides the problem into a massive amount of small jobs, increasing parallelism as well as communication overhead. A *coarse-grained* algorithm divides the problem into less but larger jobs, reducing communication overhead as well as parallelism. *Agglomeration* seeks to find the best level of granularity by grouping sub-problems into larger ones. A parallel algorithm running on a *multi-core* CPU should produce larger agglomerations than the same algorithm designed for a GPU.

1.6.4 Mapping

Eventually, all agglomerations will need to be processed by the available cores of the computer. The distribution of agglomerations to the different cores is specified by the *mapping*. The *Mapping* step is the last one of Foster's strategy and consists of assigning agglomerations to processors with a certain pattern. The simplest pattern is the 1-to-1 geometric mapping between agglomerations and processors, that is, to assign agglomeration k_{i} to processor p_{i}. Higher complexity problems may require more elaborate mapping patterns in order to provide efficient performance.

Figure 1.13 illustrates all four steps using a data-partition based problem on a dual core architecture (c_0 and c_1).

Foster's strategy is well suited for computational physics because it handles data-parallel problems in a natural way. At the same time, Foster's strategy also works well for designing massive parallel GPU-based algorithms. In order to apply this strategy, it is necessary to know how the massive parallelism programming model works for mapping the computational

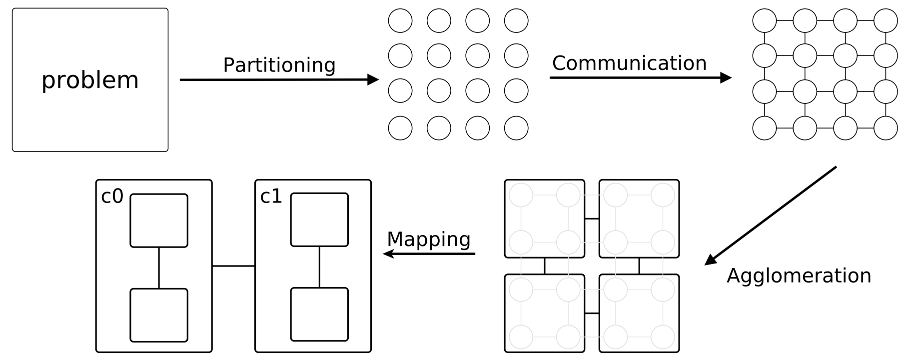

Figure 1.13: Foster's diagram of the design steps used in a parallelization process.

resources to a data-parallel problem and how to overcome the technical restrictions when programming the GPU.

1.7 GPU Computing

GPU computing is the utilization of the GPU as a general purpose unit for solving a given problem, unrestricted to the graphical context. It is also known by the acronym GPGPU coined by researcher Mark Harris, which means *General-Purpose computing on Graphics Processing Units*. The goal of GPU computing is to achieve the highest performance for data-parallel problems through a massive parallel algorithm that runs on the GPU.

GPU computing started as a research field for computer graphics (CG) in the early 2000s and gained high importance as a general purpose parallel processing technique [182]. In 2001, for the first time the graphics processing unit was built upon a programmable architecture, permitting programmable lighting [56, 154, 158], shadow [57] and geometry [243] effects to be computed and rendered in real-time. These effects were achieved using a high level shading language such as GLSL (OpenGL Shading Language) [190], HLSL (High-level Shading Language) [212] and CG (*C* for Graphics) [189]. At that time, the massive parallelism paradigm was already in the minds of the CG researchers who were designing per-vertex and per-fragment algorithms to work in a set of millions of primitives. As the years passed, the scientific community became interested in the power of GPUs and its low cost compared to other solutions (clusters, super-computers). However, adapting a scientific problem to a graphics environment was hard and challenging from the technical side. In the early days, the act of adapting different kinds of problems to the GPU was considered as *hacking the GPU*.

In 2002, McCool *et al.* published a paper detailing a meta-programming GPGPU language, named *Sh* [192]. In 2004, Buck *et al.* proposed *Brook for GPUs*, also known as *Brook-GPU* [39]. This was an extension of the C language that allowed general purpose programming on programmable GPUs. Both *Sh* and *Brook-GPU* played a fundamental role in expanding the idea of GPU computing by hiding the graphical context of shading languages.

In the year 2006 another general purpose GPU computing API was released. This time

by Nvidia and named CUDA (Compute Unified Device Architecture) [209]. Technically, the CUDA API is an extension of the C language and compiles general purpose code to be executed on the GPU (based on the shared memory programming model). The release of CUDA became an important milestone in the history of GPU computing because it was the first API that offered effective documentation for getting started in the field. The CUDA acronym refers to the general purpose architecture of Nvidia's GPUs [63], suitable for GPU computing. At the moment, only Nvidia GPUs can be programmed using CUDA.

In the year 2008, an open standard was released with the name of OpenCL (Open Computing Language), allowing the creation of multi-platform, massively parallel code [156]. Its programming model is similar to that of CUDA but uses different names for the same structures. The programming model behind CUDA and OpenCL is a key aspect for GPU computing because it defines several components that are essential for implementing a massively parallel algorithm.

1.7.1 The massive parallelism programming model

The programming models explained in section 1.4 are necessary but not sufficient for understanding the programming model of the GPU. There are important aspects regarding thread and memory organization that are relevant to the implementation of a GPU-based algorithm. This section covers these aspects.

The GPU programming model is characterized by its high level of parallelism, thus the name *Massive parallelism programming model*. This model is an abstract layer that lies on top of the GPU's architecture. It allows the design of massive parallel algorithms independent of how many physical processing units are available or how execution order of threads is scheduled.

The abstraction is achieved by the *space of computation*, defined as a discrete space where a massive amount of threads are organized. In CUDA, the space of computation is composed of a *grid*, *blocks* and *threads*. For OpenCL, it is *work-space*, *work-group* and *work-item*, respectively. A *grid* is a discrete k-dimensional (with $k = 1, 2, 3$) box type structure that defines the size and volume of the space of computation. Each element of the grid is a *block*. Blocks are smaller k'-dimensional (with $k' = 1, 2, 3$) structures identified by their coordinate relative to the grid. Each block contains many spatially organized *threads*. Finally, each thread has a coordinate relative to the block for which it belongs. This coordinate system characterizes the space of computation and serves to map the threads to the different locations of the problem. Figure 1.14 illustrates an example of two-dimensional space of computation. Each block has access to a small local memory, in CUDA it is known as the *shared memory* (in OpenCL it is known just as the *local memory*). In practical terms, the shared memory works as a manual cache. It is important to make good use of this fast memory in order to achieve peak performance of the GPU.

The programming work-flow of GPU computing is viewed as a *host-device* relationship between the CPU and GPU, respectively. A host program (*e.g.,* a C program) uploads the problem into the device (GPU memory), and then invokes a *kernel* (a function written to

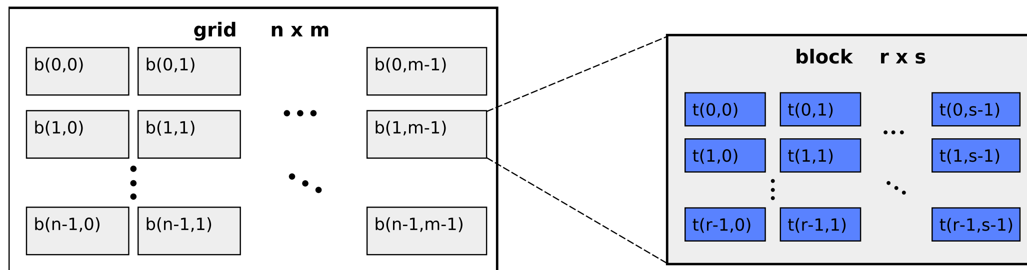

Figure 1.14: Massive parallelism programming model presented as a 2D model including grid, blocks and threads. Image inspired from Nvidia's CUDA C programming guide [209].

run on the GPU) passing as parameter the grid and block size. The host program can work in a synchronous or asynchronous manner, depending if the result from the GPU is needed for the next step of computation or not. When the kernel has finished in the GPU, the result data is copied back from device to host. Figure 1.15 summarizes the work-flow.

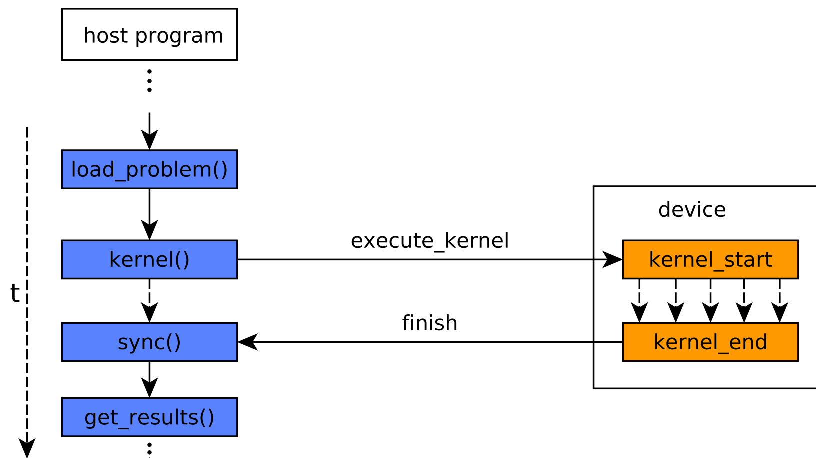

Figure 1.15: The GPU's main function, named *kernel*, is invoked from the CPU host code.

1.7.2 Thread managing and GPU concurrency

Actual GPUs manage threads in small groups that work in SIMD mode. For AMD GPUs, these groups are known as *wavefronts* and its size is 64 threads for their actual GCN (graphics core next) architecture. For Nvidia GPUs, these groups are known as *warps* and the actual architectures such as Fermi and Kepler work with a size of 32 threads. The OpenCL standard uses a more descriptive name; *SIMD width*. For simplicity reasons, we will refer to these groups as *warps*.

Both AMD's and Nvidia's GPUs support some degree of concurrency for handling the entire space of computation. Most of the time, there will be more threads than what can really be processed in parallel. While all threads are in progress (concurrency), only a subset are really working in parallel. The maximum number of parallel threads running on a GPU normally corresponds to the number of processing units. However, the maximum number of concurrent threads is much higher. For example, the Geforce GTX 580 GPU can process

up to 512 threads in parallel, but can handle up to 24,576 concurrent threads. For most problems, it is recommended to overflow the parallel computing capacity. The reason is that the GPU's thread scheduler is smart enough to switch idle warps (*i.e.,* warps that are waiting a memory access or a special function unit result, such as *sqrt()*) with new ones ready for computation. In other words, there is a small pipeline of numerical computation and memory accesses that the scheduler tries to maintain busy all the time.

1.7.3 Technical considerations for a GPU implementation

The GPU computing community frequently uses the terms *coalesced memory*, *thread coarsening*, *padding* and *branching*. These terms are critical technical considerations that must be taken into account in order to achieve the best performance on the GPU.

Coalesced memory refers to a desired scenario where consecutive threads access consecutive data chunks of 4, 8 or 16 bytes long. When this access pattern is achieved, memory bandwidth increases, making the implementation more efficient. In every other case, memory performance will suffer a penalty. Many algorithms require irregular access patterns with crossed relations between the chunks of data and threads. These algorithms are the hardest to optimize for the GPU and are considered great challenges in HPC research [215].

Thread coarsening is the act of reducing the fine-grained scheme used on a solution by increasing the work per thread. As a result, the amount of registers per block increases allowing to re-use more computations saved on the registers. Choosing the right amount of work per thread normally requires some experimental tuning.

Padding is the act of adjusting the problem size on each dimension, n_{d}, into one that is multiple of the block size; $n'_{\mathrm{d}} = \rho\lceil n_{\mathrm{d}}/\rho\rceil$ (where ρ is the number of threads per block per dimension) so that now the problem fits tightly in the grid (the block size per dimension is a multiple of the warp size). An important requirement is that the extra dummy data must not affect the original result. With padding, one can avoid putting conditional statements in the kernel that would lead to unnecessary branching.

Branching is an effect caused when conditional statements in the kernel code lead to sequential execution of the *if* and *else* parts. It has a negative impact in performance and should be avoided whenever possible. The reason why branching occurs is because all threads within a warp execute in a lock-step mode and will run completely in parallel only if they follow the same execution path in the kernel code (SIMD computation). If any conditional statement breaks the execution into two or more paths, then the paths are executed sequentially. Conditionals can be safely used if one can guarantee that the program will follow the same execution path for a whole warp. Additionally, tricks such as *clamp, min, max, module* and *bit-shifts* are hardware implemented, cause no branching and can be used to evade simple conditionals.

1.8 Latest advances and open problems in GPU computing

In the field of computational physics, $O(n)$ cost algorithms (the fast multi-pole expansion method [293, 294, 295]) have been implemented on GPU for n-body simulations. Single GPU implementations have been proposed for achieving high performance Potts model simulations [265, 266] even with biological applications [53]. Multi-GPU based implementations have also been proposed for the Potts model [166] and for n-body simulations [292]. In a multi-GPU scenario, two levels of parallelism are used; *distributed* and *local*. Distributed parallelism is in charge of doing a coarse grained partition of the problem, the mapping of sub-problems to computer nodes and the communication across the super-computer or cluster. Local parallelism is in charge of solving a sub-problem independently with a single GPU. Multi-GPU based algorithms have the advantage of computing solutions to large scale problems that cannot fit in a single machine's memory. The main challenge for multi-GPU methods is to achieve efficient distributed parallelism (*e.g.,* hiding data communication cost by overlapping communication with computation).

Cellular Automata are now being used as a model for fast parallel simulation of physical phenomena, traffic simulation and image segmentation, among others [83, 100, 153, 167]. In the field of computer graphics, new algorithms have been proposed for building kd-trees or oct-trees in GPU to achieve real-time ray-tracing [129, 150, 301] as well as real-time methods for 3D reconstruction and level set segmentation [102, 235]. The field of programming languages have contributed to parallel computing with high level parallel languages for the programmer (*i.e.,* to abstract the programmer so that the job of partitioning, communication, agglomeration and mapping is part of the compiler or framework [45, 259]). Tools for automatically converting CPU code into GPU code are now becoming popular [128] and useful for fields that use parallelism at a high level tool and not as a goal of their research. In a more theoretical level, a new GPU-based computational model has also been proposed; the K-model [41] which serves for analyzing GPU-based algorithms.

Architectural advances in parallel computing have focused on combining the best of the CPU and GPU worlds. Parallel GPU architectures are now making possible massive parallelism by using thousands of cores, but also with flexible work-flows, access patterns and efficient cache predictions. The latest GPU architectures have included *dynamic parallelism* [63]; a feature that consists of making it possible for the GPU to schedule additional work for itself by using a *command processor*, without needing to send data back and forth between host and device. This means that recursive hierarchical partition of the domain will be possible on the fly, without needing the CPU to control each step. Lastly, one of the most important revolutions in computer architecture is the introduction 3D memories such as the High Bandwidth Memory (HBM) for GPUs and the Hybrid Memory Cube (HMC) for CPUs [263]. A three-dimensional memory architecture can provide up to $15\times$ better performance than DDR3 memory, requiring 70% less energy per bit.

There are still open problems for GPU computing. Most of them exist because of the actual limitations of the massive parallelism model. In a parallel SIMD architecture, some data structures do not work so efficiently. Tree implementations have been implemented on

the GPU, with an acceptable efficiency, but data structures such as classic *dynamic arrays, heaps, hash tables and complex graphs* are not performance-friendly yet and need research for efficient GPU usage. Another problem is the fact that some sequential algorithms are so complex that porting them to a parallel version will lead to no improvement at all. In these cases, a complete redesign of the algorithm must be done. The last open problem we have identified is the difficulty of mapping the space of computation (*i.e.,* the grid of blocks) to different kinds of problem domains (*i.e.,* geometries). A naive space of computation can always build a bounding box around the domain and discard the non useful blocks of computation. Non Euclidean geometry is an interesting case, since finding an efficient map for each block of the grid to the fractal problem domain is not trivial. One way of solving this problem would be to find an efficient mapping function from the space of computation to the problem domain, or modify the problem domain so that it becomes an Euclidean box, but for the last approach data organization would be an issue to consider. In this thesis we did a research on the problem of mapping threads onto 2D triangular domains, where we found an efficient map that runs up to 18% faster than the bounding box method [198]. The research can be found in Appendix A.

Chapter 2

Spin Models Background

In this chapter we present the physical background for the thesis. The topics covered are essentially concepts related to statistical mechanics and spin models. The chapter includes the definition of a spin lattice, the definition of the partition function $Z(\ )$, the presentation of several spin models and an overview of the algorithmic approaches used to study these systems.

2.1 Spin Systems

A Spin system is an abstract representation of a ferromagnet. The first model was originally proposed by Wilhelm Lenz and was later studied by his student Ersnt Ising in 1925, which he solved for the one-dimensional case. Through the years, this model became known as the *Ising model* [133]. In 1944 the two-dimensional version was solved by Lars Onsager [213]. Over the following decades, extensions and generalizations were made to the model, giving birth to new spin models.

The study of spin systems starts with the definition of the spin lattice structure, which is essentially a graph that represents the system in question. The following is a technical definition of a spin lattice that is general for many spin models.

Definition 2.1 *A spin lattice is a graph $G(V, E)$, with $|V|$ spins $s_{\mathrm{i}} \in \{x_1, x_2, ..., x_q\}$ that sit on the vertices of the graph, and $|E|$ edges that represent the interactions $J_{\mathrm{i}j} \in [-1, 1]$ between neighboring spins.*

Figure 2.1 illustrates a square spin lattice in one, two and three dimensions. A spin s_{i} can take one of q possible spin states $\{x_1, x_2, ..., x_q\}$. The interactions $J_{\mathrm{i}j}$ usually take the values $\{1, -1\}$, which correspond to the ferromagnetic and anti-ferromagnetic cases, respectively. One of the most important aspects in spin systems is the *Hamiltonian*, denoted $\mathcal{H}$, which describes the energetic behavior of the lattice as a function of the temperature. A Hamiltonian

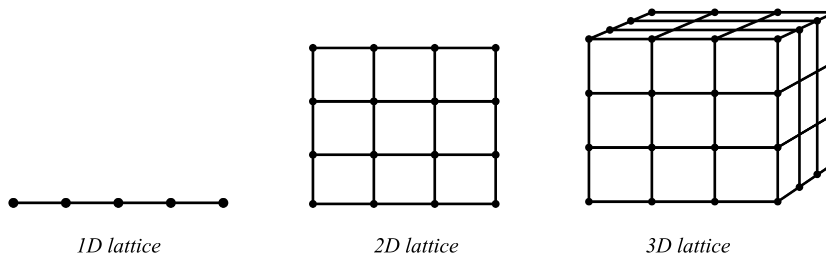

Figure 2.1: The square spin lattice in one, two and three dimensions.

usually has the following form:

$$\mathcal{H} = -\sum_{\mathrm{e} \in E} J_{\mathrm{ij}} f(s_{\mathrm{i}}, s_j) - \sum_{\mathrm{i}} h_{\mathrm{i}} s_{\mathrm{i}} \tag{2.1}$$

and depending on the model, some parameters get fixed or not. The first sum of the Hamiltonian carries all the energy contributions that are made for each pair of spins connected by an edge $\mathrm{e} \in E$. Function $f(x, y)$ (usually the product of spins or the Kronecker delta) acts on the pair of spins, returning a value that describes the energetic interaction for the given pair. The second sum is optional and corresponds to the contributions from external magnetic field $H = \{h_1, h_2, ..., h_N\}$ where s_{i} is affected by h_{i}.

Every spin lattice must define *boundary conditions* for handling the spins at the limit of the lattice. The most common ones are *free, cylindrical, band* and *periodic*. Figure 2.2 shows them for the case of the two-dimensional lattice.

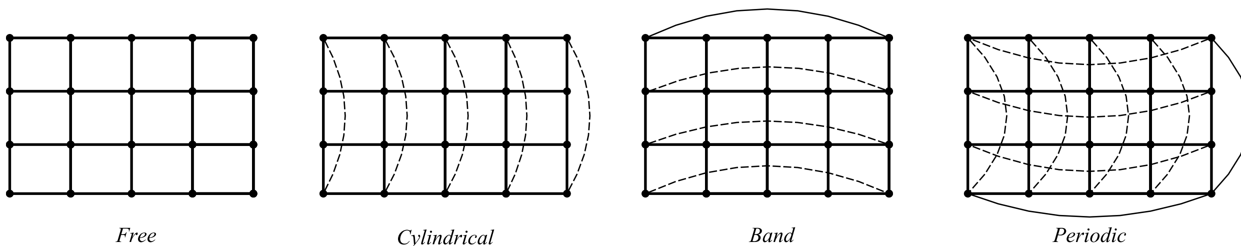

Figure 2.2: The free, cylindrical, band and periodic boundary conditions in two-dimensions.

The free boundary condition is nothing more than the default case of a graph. The *cylindrical* and *band* boundary conditions correspond to semi-periodic settings, that depending on the side chosen, will produce a different structure unless the lattice has equal height and width. The periodic boundary condition is where the lattice connects all its sides, becoming a torus. For simplicity, we will assume free boundary conditions as the default setting.

The Hamiltonian makes the spin lattice a dynamic system and its behavior is strongly dictated by the temperature T at which it is exposed. If the system is exposed enough time to a certain temperature T, it will eventually reach equilibrium. This minimum energy configuration is known as the *ground state* of the system, and there is one for each temperature value.

For several spin models, one can observe a phenomenon known as a *phase transition*, where the macroscopic properties of the system suffer an abrupt change around a singular

value of temperature, denoted T_c for *critical temperature*. From a more mathematical point of view, the phase transition of i-th order corresponds to the point where the i-th derivative of the free energy is not an analytical point. Figure 2.3 shows an example of how a phase transition would look for the case of the specific heat C as a function of the temperature.

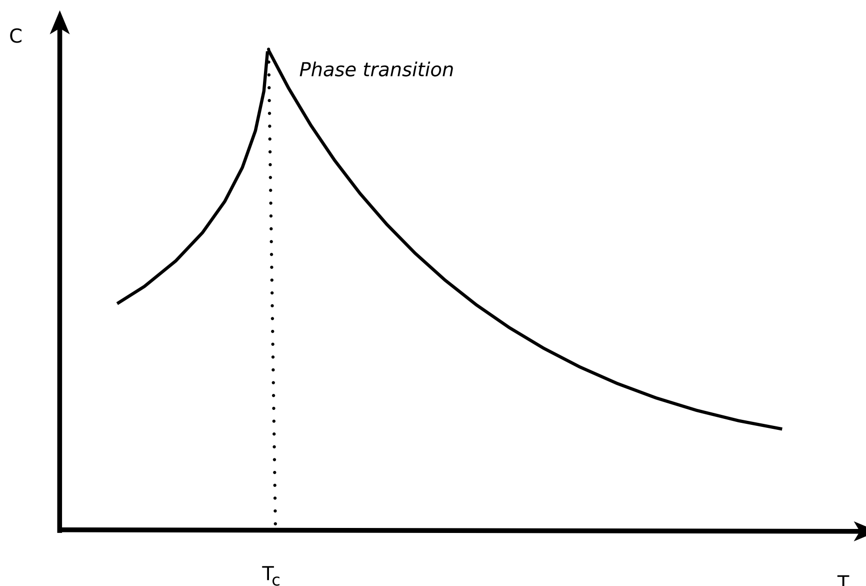

Figure 2.3: For the 3D Ising model, the system exhibits a phase transition at $T_c \approx 4.5$.

Phase transitions are important not only for the physics community, but in general for understanding how dynamical systems based on local interactions are capable of propagating long range correlations that diverge in the infinite volume limit. The notion of phase transition can be mapped to other dynamical systems such as cellular growth, social interactions, cellular automata, cellular growth or even computer networks [68, 104, 160, 300].

The study of phase transitions requires the ground states of the system, which for most of the cases is an NP-Hard problem [285]. The canonical way for studying phase transitions is through *statistical mechanics*; a theoretical framework that provides a way for obtaining the exact macroscopic properties of a system by computing the partition function $Z(\ )$.

2.1.1 The partition function $Z(\ )$

Let $G = (V, E)$ be a lattice, with $|V|$ spins $s_i \in [x_1..x_q]$. The partition function, denoted $Z(G, q, \beta)$, encodes all the physical information of the lattice into the expression

$$Z(G, q, \beta) = \sum_r e^{-\beta \mathcal{H}(\sigma_r)} \qquad (2.2)$$

where $\beta = \frac{1}{K_B T}$, K_B is the Boltzmann constant, T the temperature and $\mathcal{H}(\sigma_r)$ is the energy of the lattice at a given configuration σ_r[1], where $r = 1, 2, ..., q^{|V|}$. Once $Z(\ )$ is computed,

[1] A configuration σ_r can be seen as the graph G with a specific combination of spin values on its vertices.

one can obtain the free energy of the system:

$$F = -K_B T log(Z) \tag{2.3}$$

One can obtain many physical properties of the system such as specific heat, entropy and susceptibility by differentiating expression (2.3) with respect to the temperature and the magnetic field. The partition function can be used to study a wide range of phenomena from the lattice.

The aspect that makes one spin model different from another is the definition of the Hamiltonian $\mathcal{H}$, which describes how the interactions between neighboring spins occur. $\mathcal{H}$ must be defined in order to compute $Z(\ )$. The following subsections briefly describe the models that most relate to this thesis.

*The following Sections present several spin models and methods as part of the physical background for this thesis. Due to the high quantity, it is more practical to mention beforehand which ones will be actually used in this thesis. The spin models to keep in mind are **Potts** (Chapter 3) and **Ising, Random Field** (Chapter 4) ones. The exact methods to keep in mind for Chapter 3 are **deletion-contraction** and **transfer-matrix**, while the Monte Carlo methods for Chapter 4 are **Metropolis-Hastings** and **Exchange Monte Carlo**.*

2.1.2 Ising Model

The Ising model is the most popular model for spin lattices. The name was given after the physicist Ernst Ising who found that no phase transitions exist in the one-dimensional spin lattice [133]. The two-dimensional square lattice was solved analytically by Onsager in 1944 [213], for which phase transitions were found. In this model, the interaction energy $J_{\mathrm{i}j}$ is a constant and $s_{\mathrm{i}} = \pm 1$, that is $q = 2$. The interactions are described with the following Hamiltonian:

$$\mathcal{H} = -J \sum_{\langle \mathrm{i}, j \rangle} s_{\mathrm{i}} s_j \tag{2.4}$$

where the sum is over all neighboring sites in the graph. The model is still used today as a research model for the 3D case, which has not been solved, and for checking the correctness of new algorithms with their exact solution in 2D.

2.1.3 Potts model

The Potts model [226] is the generalization of the Ising model and it was named after Renfrey Potts, who described it in 1951. This model no longer assumes two possible spin values, instead it assumes q possible ones, *i.e.*, $s_{\mathrm{i}} = 1, 2, .., q$. The coupling constant J still remains as a constant, and the Hamiltonian is defined as

$$\mathcal{H} = -J \sum_{\langle \mathrm{i}, j \rangle} \delta(s_{\mathrm{i}}, s_j) \tag{2.5}$$

where the sum is over neighboring sites in the graph, s_i, s_j are the spin values of the neighbor sites and δ is the *Kronecker delta*:

$$\delta(s_\mathrm{i}, s_j) = \begin{cases} 1 & \text{if } s_\mathrm{i} = s_j \\ 0 & \text{if } s_\mathrm{i} \neq s_j \end{cases} \tag{2.6}$$

In this model no solution has been found yet. Modern research on Potts model is mostly devoted to study the 2D and 3D cases using exact algorithms as well as Monte Carlo simulations.

2.1.4 Spin Glass: Edwards-Anderson model

The *Spin glass* Edwards-Anderson model [80] introduces disorder to the system. The peculiar features of Spin Glass systems, which are found experimentally, can be described theoretically, using *quenched disorder*. From the practical point of view, this means that one must average[2] the free energy over all disorder realizations. In a *spin glass*, the interaction energy $J_{\mathrm{i}j}$ is no longer a constant for the whole lattice, but instead it follows a probability distribution resulting in specific values for each pair of neighbor sites. There are several variations of *spin glass* sub-models, but the Edwards-Anderson model is the most natural to follow since it defines the Hamiltonian as an extension to the Ising model:

$$\mathcal{H} = -\sum_{\langle \mathrm{i},j \rangle} J_{\mathrm{i}j}\delta(s_\mathrm{i}, s_j) \tag{2.7}$$

The original Hamiltonian does not use the Kronecker delta. However, by introducing it, the Hamiltonian still works as originally intended (*i.e.,* for Ising model), plus for any value of q. The name *spin glass* comes from the analogy to the spatial disorder of particles found in glass materials. Only the mean-field[3] version of spin glass has been solved analytically by using the concepts of *replica symmetry* [162] and *replica symmetry breaking* [194, 220]. Exact study of spin glasses with only local interactions is a problem for which no solution exists, and the computation of $Z(\ )$ is *NP-hard* for the exception of the 2D case with no magnetic field [15].

2.1.5 Random Field model

The *Random field* model [296] introduces an external random field to the system. The external field was actually considered in the original description of the Ising model, but it was usually assumed to be homogeneous. The case of a random external field is seen as a separate spin model because of its higher complexity. The Hamiltonian for the Ising Random Field model is given by

$$\mathcal{H} = -\sum_{\langle \mathrm{i},j \rangle} J s_\mathrm{i} s_j - h \sum_\mathrm{i} h_\mathrm{i} s_\mathrm{i} \tag{2.8}$$

[2] To average the logarithm of $Z(\ )$ and not $Z(\ )$

[3] The mean-field version assumes that each spin interacts with every other spin, *i.e.*, an all-with-all scenario

Similar to a spin glass, the random field presents *quenched disorder*, this time through the magnetic field $H = \{h_1, h_2, ..., h_n\}$ with random values $h_i = \pm 1$. The scaling factor h specifies the strength of the magnetic field.

The magnetic field acts against the classic ferromagnetic order of the spins. If the strength h of the field is sufficiently larger than J, then the system can become disordered even at the low temperature regime.

2.2 Exact methods for computing $Z(\ )$

An exact method is essentially an algorithm for computing the partition function $Z(\ )$ of the lattice in a given spin model. In general, exact methods have an exponential cost, with the exception of some spin models under special circumstances, such as the 2D Ising Spin Glass for planar graphs, where $Z(\ )$ can be computed in polynomial time [15]. More interesting scenarios, such as the 3D versions of the Ising, Spin Glass and Random Field models are all cases where the computation of $Z(\ )$ is NP-hard. For the case of the Potts model, the two-dimensional case is already NP-hard, thus an attractive model for doing research since it is not as intractable as the 3D cases of the other models. In the following subsections we describe three approaches for computing $Z(\ )$.

2.2.1 Using the definition

For illustration, we show how the computation of $Z(\ )$ is carried on for a small graph $G(V, E)$ in the Ising model, with $|V| = 4$ and $|E| = 4$, as shown in Figure 2.4.

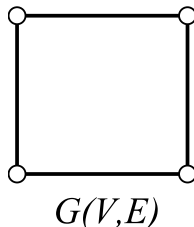

Figure 2.4: An example of a small square lattice in the Ising model, with $|V| = 4$ and $|E| = 4$.

The vertices of G take spin values $\{-1, 1\}$, which are visually represented as black or white colors, respectively. The edges of G correspond to the energy interactions, which in this case are ferromagnetic interactions, that is $J > 0$.

The main idea of the partition function is that it encodes the energy of every possible spin configuration, into a mathematical expression that can be treated later to obtain other physical properties. For this example, in order to compute $Z(\ )$, first one has to compute $\mathcal{H}$ for each one of the 2^4 possible spin configurations $\{\sigma_1, \sigma_2, ..., \sigma_{16}\}$ as shown in Figure 2.5.

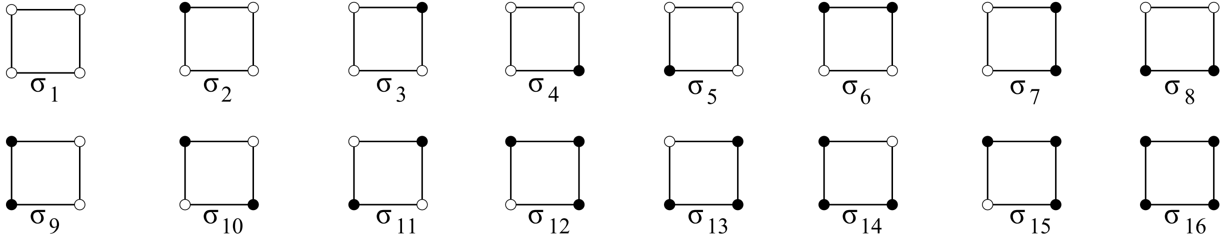

Figure 2.5: In order to compute $Z(\ )$, $\mathcal{H}$ must be computed for all spin configurations.

The Hamiltonians for σ_1, σ_2 and σ_{10} would be:

$$\mathcal{H}(\sigma_1) = -J - J - J - J = -4J$$
$$\mathcal{H}(\sigma_2) = +J + J - J - J = 0$$
$$\mathcal{H}(\sigma_{10}) = +J + J + J + J = 4J$$

The process is carried in the same way for the rest of the configurations. Once all the Hamiltonians are computed, the partition function becomes:

$$Z(G, q, \beta) = \sum_r \mathrm{e}^{-\beta\mathcal{H}(G_r)} = 2e^{4\beta J} + 2e^{-4\beta J} + 12 \tag{2.9}$$

For this example, the configurations $\{\sigma_1, \sigma_{10}, \sigma_{11}, \sigma_{16}\}$ are the only ones where $\mathcal{H}(\sigma_\mathrm{i}) \neq 0$. The configurations where $\mathcal{H}(\sigma_\mathrm{i}) = 0$ contribute only with a constant term.

The cost of the Hamiltonian itself is $\Theta(|E|)$, *i.e.*, linear in the number of edges, which is not a high cost in computing time. However, as the lattice becomes larger, the computation of $Z(\ )$ becomes rapidly intractable since the number of Hamiltonians required grows asymptotically as $\Theta(q^{|V|})$. For the case of disordered models the cost is even worse, because it requires to compute one partition function for each possible distribution of $J_{\mathrm{i}j}$ or h_i, depending if it is the Spin Glass or Random Field model, respectively. Therefore, exact analysis based on the computation of $Z(\ )$ is usually done for small lattices in the Ising or Potts models, where the computation is still tractable for some sizes of interest.

There are other exact approaches that, while still exponential, can be faster than the original definition.

2.2.2 Deletion-contraction

The *deletion-contraction* method [283], or DC method, was initially used to compute the Tutte polynomial [269] and was then extended to the Potts model after a relation of duality was found between the two (see [256, 282]). DC re-defines $Z(..)$ as the recursive expression

$$Z(G, q, v) = Z(G - \mathrm{e}, q, v) + vZ(G/\mathrm{e}, q, v) \tag{2.10}$$

$G - \mathrm{e}$ is the *deletion* operation, where edge $\{\mathrm{e}\}$ is removed, while G/e is the *contraction* operation, where the pair of vertices connected by $\{\mathrm{e}\}$ are merged into one and $\{\mathrm{e}\}$ is removed.

The auxiliary variable $v = e^{-\beta J} - 1$ makes $Z(..)$ a polynomial. There are three special cases where DC can perform a recursive step with linear cost:

$$Z(G,q,v) = \begin{cases} (q+v)Z(G/\mathrm{e},q,v); & \text{if } \{\mathrm{e}\} \text{ is a spike.} \\ (1+v)Z(G-\mathrm{e},q,v); & \text{if } \{\mathrm{e}\} \text{ is a loop.} \\ q^{|V|}; & \text{if } E = \{\emptyset\}. \end{cases} \tag{2.11}$$

The computational complexity of DC has a direct upper bound of $O(2^{|E|})$. When $|E| >> |V|$ a tighter bound is known based on the Fibonacci sequence complexity [283]; $O((\frac{1+\sqrt{5}}{2})^{|V|+|E|})$. In general, the time complexity of DC can be written as

$$T(G) = min\left(O(2^{|E|}), O\Big(\frac{1+\sqrt{5}}{2}\Big)^{|V|+|E|}\right) \tag{2.12}$$

Given a graph G, DC builds a recursion tree starting from any edge of G. Figure 2.6 shows how the process of computing $Z(\ )$ is carried for the lattice of Figure 2.4.

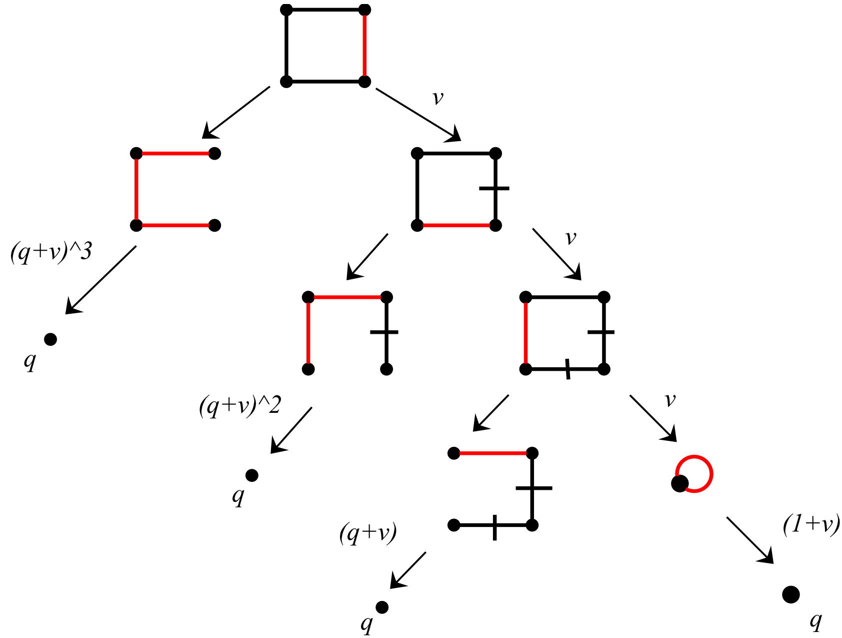

Figure 2.6: The recursion tree of deletion contraction technique. The method can take advantage of the graph structure.

The red edges represent the chosen edge for DC. The left arrows corresponds to a deletion, while the right ones correspond to contractions. It is convenient to represent a contracted edge with a crossed line instead of literally contracting it, this way one can always see which part of the original graph was contracted.

Once DC has finished, $Z(\ )$ is obtained by collecting the accumulated expressions found at the leaves of the recursion tree. Continuing with the example, the collection of terms leads to the following partition function.

$$Z(G,q,v) = q^4 + 4q^3v + 6q^2v^2 + qv^4 + 4qv^3 \tag{2.13}$$

The difference between the original definition of $Z(\ )$ and the DC method is that the former uses the *spin representation* approach, where the cost of the algorithm is combinatorial on the number of spin states and vertices, while the latter uses the *generic* (q, v) approach where the cost is combinatorial on the number of edges. By doing proper edge selection, one can increase the frequency of linear cases and improve the performance of DC significantly. There will still be cases however where DC is not the best choice, such as the case of long strips, or *strip lattices*. For this type of lattices, one can use a more efficient strategy that can take advantage of the repeating structure of the graph.

2.2.3 Transfer matrix technique

In the previous methods, the construction of $Z(\ )$ has been based on a combinatorial algorithm applied to the whole lattice. The advantage of that approach is that there is little restriction to the input graph as no assumptions are made about its structure. Nevertheless, there are many cases where the graph of interest is not random, neither too complex, but instead a lattice that repeats a pattern through its domain. There is a kind of lattice with repeating structure, called the *strip lattice*, for which the computation of $Z(\ )$ can be done more efficiently.

A *strip lattice* is defined as a bidimensional graph $G = (V, E)$ that repeats its pattern at least along one dimension. It can be built as the concatenation of graph layers $K_1, K_2, ..., K_n$ sharing their boundary vertices and edges. Figure 2.7 illustrates how the notion of strip lattice applies to the case of the square and kagome lattices. The transfer matrix, denoted

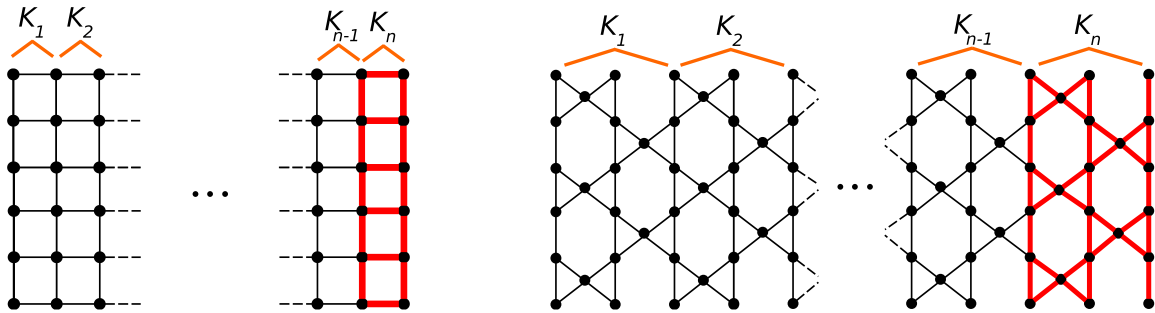

Figure 2.7: Two strip lattices; square and kagome, both with a width (vertical) of $m = 6$.

M, takes advantage of the repeating nature of the lattice, allowing the study of very long graphs. The dimension of M grows proportional to a combinatorial function $\Gamma(m)$ where m is the width of $G(V, E)$ and it represents *the different ways in which two layers can connect.* The set of configurations generated by the base corresponds to the *configuration space* of the problem. The rows of M, identified by $\sigma_{\rm i}$, correspond to all the possible initial conditions $K_{\rm i}$ can have, while the columns of M, identified by φ_j, correspond to all the possible ways $K_{\rm i}$ can end, after the local algorithm as been applied. The elements of M are denoted $f_{\sigma_{\rm i},\varphi_j}(q, v)$

and correspond to partial partition functions in (q, v):

$$M = \begin{vmatrix} f_{\sigma_1,\varphi_1}(q,v) & f_{\sigma_1,\varphi_2}(q,v) & \dots & f_{\sigma_1,\varphi_n}(q,v) \\ f_{\sigma_2,\varphi_1}(q,v) & f_{\sigma_2,\varphi_2}(q,v) & \dots & f_{\sigma_2,\varphi_n}(q,v) \\ \dots & \dots & \dots & \dots \\ f_{\sigma_{\Gamma(m)},\varphi_1}(q,v) & f_{\sigma_{\Gamma(m)},\varphi_2}(q,v) & \dots & f_{\sigma_{\Gamma(m)},\varphi_{\Gamma(m)}}(q,v) \end{vmatrix} \tag{2.14}$$

The computational cost of a transfer matrix method comes from two sources; (1) the size of $\Gamma(m)$ and (2) the cost of the local algorithm. The size of the configuration space is given by $\Gamma(m)$ and corresponds to the size of M. The local algorithm generates the partial partition functions $f_{\sigma_i,\varphi_j}(q, v)$ at each position of M. The upper-bound for the cost of computing M of a strip lattice G of width m follows the form:

$$T(G, m) = O(\Gamma(m)\Lambda(K)) \tag{2.15}$$

Where $\Lambda(K)$ is the cost of the local algorithm when applied to one block K of G. If the spin representation was chosen, then Γ function would depend on a different parameter, such as q. For the rest of the thesis we will assume that the transfer matrix method is based on the generic (q, v) representation. In Chapter 3 we specify what $\Gamma(m)$ and $\Lambda(K)$ are in more detail and how they end up for our contribution.

Once M is computed, we have that the partition function for a strip lattice of length n and width m is a vector $\vec{Z}$

$$\vec{Z}_n = M\vec{Z}_{n-1} = M^n\vec{Z}_0 \tag{2.16}$$

Where $\vec{Z}_0$ are the initial conditions. Once the expression is computed, the first component of $\vec{Z}_n$ becomes the partition function of G. The other elements of $\vec{Z}_n$ correspond to partition functions of the strip lattice but with different *initial conditions*; $\sigma_1, \sigma_2, ..., \sigma_{\Gamma(m)}$. The matrix M and the initial conditions $\vec{Z}_0$ can be stored analytically as string polynomials parameterized by (q, v) for multiple numerical evaluation.

In the case of an infinite length strip, the analysis can be done with the eigenvalues of M. In this case, the free energy per site becomes:

$$f = \frac{1}{n_K} ln\lambda_+ \tag{2.17}$$

where n_K is the number of non-shared vertices per block K and λ_+ is the dominant eigenvalue of M with nontrivial coefficient associated.

At this point, the background for the first contribution has been covered. The next Section contains the background about Monte Carlo methods which is only required for the second contribution. The reader may prefer to jump directly to Chapter 3 for the first contribution and then come back.

2.3 Monte Carlo methods

If the exactness requirement of a problem is relaxed, then one can come up with methods that run faster than the original exact ones and even reconsider some *NP-Hard* problems

that were intractable in the past.

A Monte Carlo method is, in its most basic definition, a randomized algorithm that computes a result within a deterministic amount of time. This result is no longer guaranteed to be exact since it has a probability of being incorrect, but in exchange it is significantly faster than its exact counterpart and brings the possibility to attempt intractable problems from a different point of view. The challenge is to make the Monte Carlo algorithm provide a result with a small error probability in a reasonable amount of time. In order to get a small error, one usually employs a large number of Monte Carlo steps until a certain tolerable error has been met for that specific problem.

Monte Carlo algorithms for spin systems are strongly based on the notion of measuring the thermal average of a physical observable from a system represented as a Markov Chain process. The following sub-sections present the main ideas that lead to the formulation of a Monte Carlo algorithm.

2.3.1 Computation of Averages

The definition of a Spin model Monte Carlo algorithm begins with the notion of measuring thermal averages, as explained by C. Gabriel [94].

The probability of a spin lattice to be on a specific configuration σ_r is defined as the Boltzmann factor.

$$P(\sigma_r) = \frac{e^{-\beta \mathcal{H}(\sigma_r)}}{Z} \tag{2.18}$$

Depending on the temperature, some configurations are more probable than others. Based on this fact, the thermal average of an observable A is expressed as

$$\langle A \rangle = \frac{1}{Z} \sum_{\sigma_r \in \{\sigma\}} e^{-\beta \mathcal{H}(\sigma_r)} A(\sigma_r) \tag{2.19}$$

where $\{\sigma\}$ denotes all possible configurations of the system and $\mathcal{H}(\sigma_r)$ is the Hamiltonian of the lattice at a specific configuration σ_r. The expression for the average can we re-written as

$$\langle A \rangle = \sum_{\sigma_r \in \{\sigma\}} A(\sigma_r) P(\sigma_r) \tag{2.20}$$

to express that the thermal average is just the sum of the measures of A at all possible configurations, with the probabilities acting as the weight factors.

The problem of computing an average using the form from expression (2.20) is that the number of terms grows as $O(q^{|V|})$ and the expression for the average soon becomes intractable for any computer. For illustration: a small system composed of $|V| = 299$ spins, with $q = 2$, has already $2^{299} \approx 10^{90}$ terms which is more than the number of particles in the observable universe ($\approx 10^{80}$). The approach to overcome this problem is to use the *central limit theorem*, which says that if random independent variables $x_1, x_2, ..., x_m$ are drawn from the same

distribution, then the arithmetic average

$$\mathcal{A} = \frac{1}{M}\sum_{m=1}^{M} A(x_m) \tag{2.21}$$

in the limit $M \to \infty$ will always be distributed according to a Gaussian distribution, no matter from which distribution the x_m where drawn. In addition, the variance of $\mathcal{A}$ becomes

$$\mathrm{Var}(\mathcal{A}) = \frac{\mathrm{Var}(A)}{M} \tag{2.22}$$

which allows the approximation of expression (2.20) to the average sample of random variables $\{x_1, x_2, ..., x_M\}$ in the form of

$$\langle A(x)\rangle = \frac{1}{M}\sum_{m=1}^{M} A(x_m) \pm \sqrt{\frac{\mathrm{Var}(A)}{M}} \tag{2.23}$$

with a distribution according to P_x. At this point, the question is how to produce the required random variables.

2.3.2 Markov Chain Monte Carlo (MCMC)

A Markov-Chain can produce random variables according to a given distribution P. Let $\{Y_1, Y_2, ..., Y_N\}$ be the possible states of a system, all exclusive to each other. A stochastic process is called a Markov Chain if the probability of the next event X_{t+1} only depends on the previous one X_t.

$$P(X_{t+1} = Y_j | X_t = Y_\mathrm{i}) = W_{\mathrm{i}j}(t) \tag{2.24}$$

$W_{\mathrm{i}j}^{(t)}$ is a matrix with the transition probabilities for all pairs $\{\mathrm{i}, j\}$. The matrix satisfies $W_{\mathrm{i}j} \geq 0$ and $\sum_{j=1}^{N} P_{\mathrm{i}j} = 1$. A Markov Chain is called *ergodic* if

$$\forall \mathrm{i}, j\ \exists K > 0 : (W^K)_{\mathrm{i}j} > 0 \tag{2.25}$$

which means that all states $\{Y_1, Y_2, ..., Y_N\}$ can eventually be reached, no matter what state the process began with. Ergodicity is an important property that is required for Monte Carlo methods in spin systems, because it guarantees that equilibrium can be reached through successive simulation steps independently of what initial conditions were chosen for the lattice.

The stochastic process of a spin lattice is modeled with an ensemble of Markov Chains, all sharing the same transition matrix $W_{\mathrm{i}j}$. Under this scheme, the states of the ensemble at time t follow a distribution $\pi^t(Y_\mathrm{i})$. The evolution of all states in the ensemble is achieved by applying $W_{\mathrm{i}j}$ to the distribution and obtaining a new distribution for the next time step. In order to compute appropriate averages, one requires the distribution to be an *invariant distribution*, also known as *stationary distribution*, that satisfies

$$\pi^\infty W_{\mathrm{i}j} = \pi^\infty \tag{2.26}$$

Repeated applications of W_{ij} to the distribution can eventually lead to a unique invariant distribution $P(Y_i)$. The question is what transition matrix W_{ij} must be used in order to achieve the invariant distribution. A sufficient condition is that the transition probabilities satisfy *detailed balance*

$$P(Y_i)W_{ij} = P(Y_j)W_{ji} \tag{2.27}$$

Detailed balance is sufficient for making sure that the process will converge and measurement of averages will eventually be possible in the thermodynamical equilibrium.

Algorithms based on the notion Markov chains are known as *Markov chain Monte Carlo* (MCMC) methods. A MCMC applies a randomized algorithm to simulate the behavior of the spins in the lattice, much like a dynamical system based on particles. In the end, the goal for a MCMC algorithm is to simulate the system towards the thermodynamic equilibrium. The simulation process involves modifying (also known as *flipping*) the spins of the system at discrete time steps, based on probabilistic rules that depend on the energy of the lattice and the given temperature. The evolution of the system favors the configurations with less energy over the ones with more energy. Eventually, after many simulation steps, the system will reach equilibrium; the stage where measurements can be made to get averages of the physical observables. The statistical error for the averaged measurement of a physical observable A is

$$\Delta\langle A\rangle = \sqrt{\frac{\sigma_A}{N}(1 + 2\tau_A)} \tag{2.28}$$

Where σ_A is the variance of A, N is the number of Monte Carlo steps and τ_A is the correlation time.

Today, many MCMC algorithms exist, some more sophisticated than others, but the most emblematic, simple and frequently used is the Metropolis-Hastings algorithm [123, 193].

2.3.3 The Metropolis-Hastings Algorithms

The Metropolis-Hastings algorithm is the most used MCMC algorithm for the simulation of spin models. The method works much like a Cellular Automata (CA) with a non-deterministic rule [132, 244]. The first notion of the Metropolis-Hastings algorithm was presented in the work of N. Metropolis *et. al.* in 1953 [193], which was then generalized by W.K Hastings in 1970 [123].

The Metropolis-Hastings algorithm in the context of spin models is known as the *heat bath method*, which is a special case of the algorithm. The method generates a sequence of states that are distributed according to the Gibbs distribution. It does so efficiently by what is called the importance sampling method, which generates the sequence of samples on regions where it is more probable to find the system according to the temperature T. Each step of the algorithm is computed by applying a local probabilistic rule $f(\ )$ to the spins of the lattice. The rule is local because it uses information just from the neighboring sites. The steps of the Metropolis-Hastings algorithm are:

1. Initialize the lattice with an arbitrary configuration.

2. Pick a spin s_i^t, and choose a trial state $s_i^{t+1} \neq s_i^t$ as the candidate state for step $t+1$.
3. Compute the energy difference $\Delta h = h_2 - h_1$, where h_1 and h_2 are the local Hamiltonians for s_i^t and s_i^{t+1}, respectively.
4. Accept s_i^{t+1} as the next spin state for s_i^t, with probability

$$P(s_i^t \Rightarrow s_i^{t+1}) = \begin{cases} 1 & , \Delta h \leq 0 \\ e^{\frac{-\Delta h}{\kappa T}} & , \Delta h > 0 \end{cases} \tag{2.29}$$

   If the new state is not accepted, then the original one is kept, *i.e.*, $s_i^{t+1} := s_i^t$.
5. Go to step 2.

Figure 2.8 illustrates an example of flipping a spin in the Ising model, forth and back, based on the energy of the neighboring spin states. .

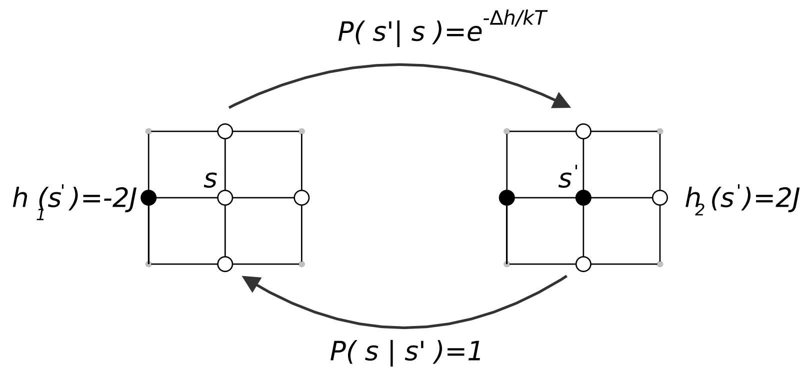

Figure 2.8: The process of flipping a spin, forth and back, with the Metropolis-Hastings algorithm.

For spin models with no disorder, such as the Ising and Potts model, the Metropolis-Hastings algorithm can provide a good quality simulation. However, near a phase transition the amount of simulation steps required to reach equilibrium increases approximately as L^2, with L being the linear size of the lattice. This effect is known as the *critical slow down* (CSD) and it has been one of the major problems for the Metropolis-Hastings algorithm [252, 253] and in general for simulating algorithms based on local spin flips. This problematic led to the proposal of new MCMC algorithms based known as the *cluster algorithms*.

2.3.4 Cluster Algorithms

In 1987 and 1989, two Monte Carlo algorithms were proposed, one by Swendsen and Wang [262] and the other by Wolff [287]. In both methods, the authors found that by flipping clusters of spins, one can accelerate the simulation process near the critical temperature, solving the CSD problem. The important change introduced with cluster methods is the notion of global spin updates, in contrast to the local spin updates strategy used in the Metropolis-Hastings algorithm.

A cluster algorithm works iteratively, at each iteration forming random clusters of spins and flipping them afterwards. The clusters are basically labels that are put on the spins,

indicating whom they belong to. The main difference between Swendsen and Wang's algorithm and Wolff's algorithm is that the former creates many clusters per iteration while the latter handles just one per iteration. The clustering phase of the algorithm is where most of the algorithmic complexity lies and where some ideas can be proposed. One approach that is used for forming the clusters is the *ants in the labyrinth* strategy [73], where ants are placed at random sites and each one begins the generation of a cluster by replicating to adjacent sites. The *ants in the labyrinth* strategy follows five steps:

1. Label all spins as *free*.
2. Place an ant at a random *free* site and label it as *taken* to begin a cluster.
3. Add *free* neighbor spins of the same state to the cluster, with probability $P = 1 - e^{-2J\beta}$
4. Label each successfully added spin as *taken* and spawn an ant at its location.
5. Each ant starts from step (3). If no ants were spawned, then go back to step (2).

Figure 2.9 illustrates how two clusters are created with the *ants in a labyrinth* strategy.

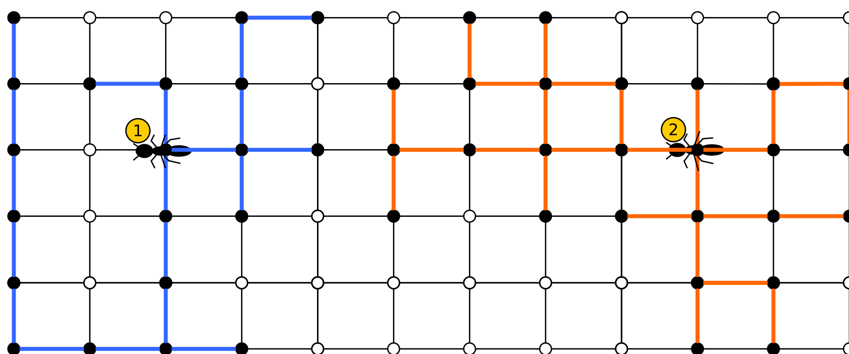

Figure 2.9: The cluster creation process for the Swendsen and Wang algorithm.

Cluster algorithms have proven to be significantly faster than the Metropolis-Hastings method for 2D lattices, specially near the critical temperature. The next MCMC algorithm is a different kind of method, that based on the random walks, can solve the CSD problem while still providing local updates on the spins.

2.3.5 The Worm Algorithm

The *Worm algorithm*, introduced by Prokof'ev and Svistunov in 2001 [230], is inspired by the random walk process, where closed paths of spins are updated by probabilistic movements of two end-points. The method solves the CSD problem and it is considered more flexible than the Cluster algorithms since it is applicable to a wide class of models; from classic ferromagnetic models [71] to quantum models [231]. Its autocorrelation time is also efficient; for example for the Ising model, autocorrelation time has found to be $\approx ln\ L$ for both 2D and 3D cases [94].

The formulation of the worm algorithm starts with the high-temperature representation

of Z. In the Ising model we have that

$$Z = \sum_{\sigma_r \in \{\sigma\}} e^{-\beta \mathcal{H}(\sigma_r)} = \sum_{\sigma_r \in \{\sigma\}} \prod_{\langle \mathrm{i},j \rangle} e^{K s_\mathrm{i} s_j} \tag{2.30}$$

The Ising high-temperature expansion specifies that $s_\mathrm{i} s_j = \pm 1$, thus allowing Z to be expressed in terms of hyperbolic functions

$$Z = \sum_{\sigma_r \in \{\sigma\}} \prod_{\langle \mathrm{i},j \rangle} e^{K s_\mathrm{i} s_j} = \sum_{\sigma_r \in \{\sigma\}} \prod_{\langle \mathrm{i},j \rangle} \left[\cosh(K) \Big(1 + \tanh(K) s_\mathrm{i} s_j \Big) \right] \tag{2.31}$$

which can be relocated in the expression, considering that the number of edges in a 2D lattice is $|V|$ and the bond states are $n_b = 0, 1$.

$$Z = \cosh(K)^{2|V|} \sum_{\{n_b\}} \tanh(K)^{\sum n_b} \sum_{\sigma_r \in \{\sigma\}} \prod_{\langle \mathrm{i},j \rangle} s_\mathrm{i}^{n_b} s_j^{n_b} \tag{2.32}$$

The sum of bond products from expression (2.32) can be re-formulated as the product of incident bonds per spin

$$\sum_{\sigma_r \in \{\sigma\}} \prod_{\langle \mathrm{i},j \rangle} s_\mathrm{i}^{n_b} s_j^{n_b} = \prod_{\mathrm{i}} \sum_{s_\mathrm{i}} s_\mathrm{i}^{p_\mathrm{i}} \tag{2.33}$$

where p_i is the sum of the values of the bonds incident to spin s_i. The sum $\sum_{s_\mathrm{i}} s_\mathrm{i}^{p_\mathrm{i}}$ just covers the case for each possible spin value; *i.e.*, $s_\mathrm{i} = \pm 1$. The only case where $\sum_{s_\mathrm{i}} s_\mathrm{i}^{p_\mathrm{i}} \neq 0$ is when p_i is *even*, which leads to a more simplified expression for Z

$$Z = 2^{|V|} \cosh(K)^{2|V|} \sum_{\{n_b\}} \tanh(K)^{\sum n_b} \tag{2.34}$$

In this new form we assume p_i is *even* and in order to satisfy this condition, the terms of the sum must necessarily correspond to *closed paths* made of bonds. The new form of Z in expression (2.34) has an important meaning because it says that the partition function can be expressed using a configuration space based only on closed paths. This concept is what leads to the formulation of the *worm algorithm*.

The worm algorithm specifies how to create closed paths dynamically in time by the use of two *end-points*, namely $\mathrm{i}_1, \mathrm{i}_2$, which act like bond-drawing pencils moving in the lattice domain based on probability functions. After successive movements, the bonds eventually form closed paths. The steps of the worm algorithm are:

1. Choose a random location in the lattice to place both $\mathrm{i}_{1,2}$.
2. (a) **If** $\mathrm{i}_1 = \mathrm{i}_2$ then with probability $p_0 \in [0, 1]$ move both end points $\mathrm{i}_1, \mathrm{i}_2$ to a new random location.
   (b) **Else**, go to step (3).
3. Pick an incident bond b of i_2 with probability $P(j \mid \langle \mathrm{i}_1, j \rangle) = \frac{1}{N_{nn}}$.
4. If bond value is $N_b = 0$, accept with probability $R = \tanh(K)$ moving i_2 towards b as well as increasing the bond value to $N_b = 1$. Else $N_b = 1$, thus accept with probability $R = 1$ moving i_2 towards b as well as decreasing the bond value to $N_b = 0$. Update averages and go back to step (2).

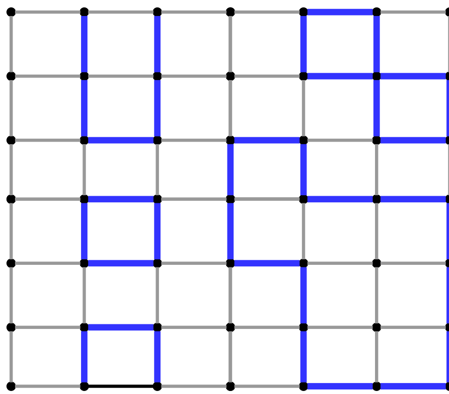

Figure 2.10: Closed paths formed by the worm algorithm in a lattice with periodic boundary conditions.

Figure 2.10 illustrates how worms are made randomly throughout the lattice.

It is important to clarify that in the Worm Algorithm, measurement of the averages is not done over the spin configuration as in the traditional way, but instead it is done progressively on the bonds as the closed paths are built. The formal description on how to get the averages can be found in the original article by Prokof'ev and Svistunov [230].

The Worm algorithm is considered efficient because it solves the CSD problem present in the Ising and Potts model, while still taking into account the local interactions between the spins, making it more flexible for other models. From the point of view of parallel computation it is not so efficient however, since the formation of worms does not provide enough parallelism for massively parallel architectures [69].

So far, we have presented Monte Carlo methods that can reach equilibrium on non-disordered spin models, some more efficiently than others. The following MCMC algorithm is different because it simulates many copies of the system in order to reach equilibrium in models with quenched disorder, such as the Spin Glass and Random Field ones.

2.3.6 Exchange Monte Carlo (*Parallel Tempering*)

Some spin models, such as the Spin Glass and Random Field, present *quenched disorder*, which is a type of disorder where one of the parameters of the Hamiltonian is no longer fixed, instead it is a distribution of random variables across the spin lattice. For the Spin Glass, it is the coupling constant, while for the Random Field, it is the magnetic field at each spin location. These systems, which are also called *hardly-relaxing*, are characterized for presenting an adverse energy landscape in the low-temperature regime that cannot be traveled easily by traditional MCMC algorithms. The reason is because at some point of the simulation the system will eventually become trapped at a local minimum, never approaching its ground state, as shown in Figure 2.11.

One way to overcome the local minimum problem is to introduce the notion of *replicas*, that is, keep more than one instance of the problem under simulation. The *Exchange Monte*

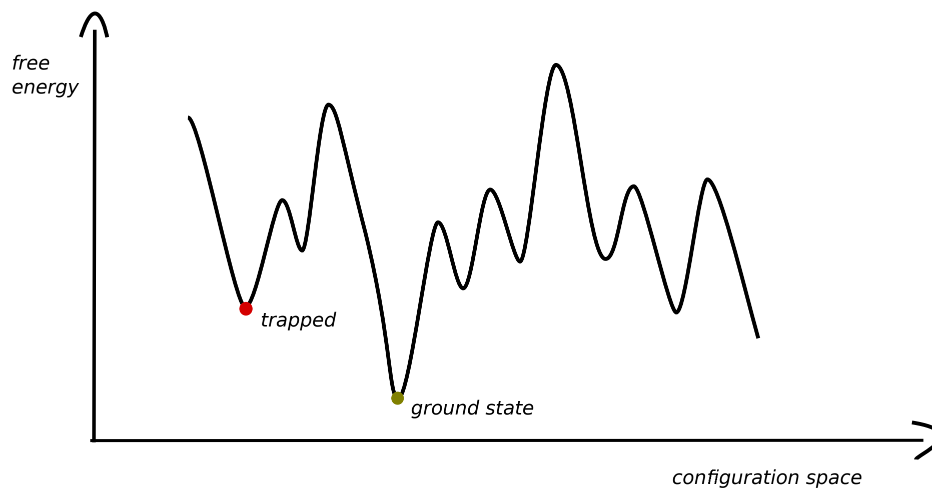

Figure 2.11: Quenched disorder leads to rough energy landscapes in the low temperature regime. Traditional algorithms trap the system at a local minimum.

Carlo algorithm, also known as *Parallel Tempering*, is an algorithm based on exchanging information between *replicas* of the system, all individually simulated. The idea of exchanging replicas for studying systems with quenched disorder was first introduced by Swendsen and Wang in 1986 [261] and then extended by Geyer in 1991 [97]. Hukushima and Nemoto presented in 1996 the full method as it is known today [130].

The *Exchange Monte Carlo* method is based on the principle that *a system simulated at the low-temperature regime can escape from a local minimum by exchanging its configuration with another system that has been simulated at a higher temperature*. The reason why high-temperatures configurations help the low temperature ones escape a local minimum is because the energy landscape at the high temperature is smoother and easier to explore, allowing the appearance of configurations that were needed, but were unlikely to appear at the low temperature regime. This act exchanging configurations between high T and low T systems can be described as *shaking the energy landscape* so that the red ball from Figure 2.11 can eventually fall into the green spot.

The steps of the *Exchange Monte Carlo* algorithm, for the case of the Ising Random Field model, are the following:

1. Choose R different temperatures $\{\beta_1, \beta_2, ..., \beta_R\}$ where $\beta = 1/T$.
2. Choose an arbitrary magnetic field placing random values at each spin location $H = \{h_1, h_2, h_3..., h_{|V|}\}$ with $h_\text{i} = rand(\pm 1)$. This disorder instance will be shared among all replicas.
3. Initialize a set X of R lattices or replicas, $X = \{X_1, X_2, ..., X_R\}$, with an arbitrary spin configuration and associate each one with the corresponding temperature, *i.e.*, $X_\text{i} \iff B_\text{i}$.
4. Simulate each replica simultaneously and independently in the Random Field Model, using the disorder instance for all replicas, using a standard MCMC algorithm such as Metropolis-Hastings.

5. Attempt to exchange two configurations X_m and X_{m+1}, with probability

$$W(X, \beta_m | X', \beta_{m+1}) = \begin{cases} 1 & \text{for } \Delta < 0 \\ e^{\Delta} & \text{for } \Delta > 0 \end{cases} \tag{2.35}$$

where $\Delta = (\beta_{m+1} - \beta_m)(\mathcal{H}(X) - \mathcal{H}(X'))$.

6. If the system is not equilibrated, go to step (4) and repeat the process.

The efficiency of the algorithm depends, in a great part, of the initial parameters chosen such as the temperatures and how separate they are one from another. Temperatures that are too separated will incur in an algorithm with very low exchange rate, which has a negative impact since trapped systems have less chance of escaping. On the other hand, temperatures that are too close can produce an unnecessary amount of exchanges, leading to a higher computational work and unstable equilibrium states for each replica. The low and high values of the temperatures are also important to consider, since the main idea of the algorithm is that the highest temperature is high enough to provide easy exploration of the energy landscape, in order to feed the low temperature ones with the required information. Last but not least, *parallel tempering* is potentially parallel, and the parallelization can be done at different levels of depth, making it an attractive topic of research for the high performance computing (HPC) community.

Chapter 3

Parallel Family Trees for Transfer Matrices in the Potts Model

3.1 Abstract

The computational cost of transfer matrix methods for the Potts model is related to the question *into how many ways can two layers of a lattice be connected?*. Answering the question leads to the generation of a combinatorial set of lattice configurations. This set defines the *configuration space* of the problem, and the smaller it is, the faster the transfer matrix can be computed. The configuration space of generic (q, v) transfer matrix methods for strips is in the order of the Catalan numbers, which grows asymptotically as $O(4^m)$ where m is the width of the strip. Other transfer matrix methods with a smaller configuration space indeed exist but they make assumptions on the temperature, number of spin states, or restrict the structure of the lattice. In this paper we propose a parallel algorithm that uses a sub-Catalan configuration space of $O(3^m)$ to build the generic (q, v) transfer matrix in a compressed form. The improvement is achieved by grouping the original set of Catalan configurations into a forest of family trees, in such a way that the solution to the problem is now computed by solving the root node of each family. As a result, the algorithm can run exponentially faster than the Catalan approach while still highly parallel. The resulting matrix is stored in a compressed form using $O(3^m \times 4^m)$ of space, making numerical evaluation and decompression to be faster than evaluating the matrix in its $O(4^m \times 4^m)$ uncompressed form. Experimental results for different sizes of strip lattices show evidence that the *parallel family trees (PFT)* strategy has an exponential advantage over the *Catalan Parallel Method (CPM)*, especially when dealing with dense transfer matrices. In terms of parallel performance, we report strong-scaling speedups of up to $5.7X$ when running on an 8-core shared memory machine and $28X$ for a 32-core cluster. The best balance of speedup and efficiency for the multi-core machine was achieved when using $p = 4$ processors, while for the cluster scenario it was in the range $p \in [8, 10]$. Because of the parallel capabilities of the algorithm, a large-scale execution of the parallel family trees strategy in a supercomputer could contribute to the study of wider strip lattices.

3.2 Introduction

The Potts model [227] has been widely used to study physical phenomena of *spin lattices* such as phase transitions [32] in the thermodynamical equilibrium. Lattices such as square, triangular, honeycomb and kagome are of high interest and are being studied frequently [46, 48, 49, 249]. When the number of possible spin states is set to $q = 2$, the Potts model becomes the classic Ising model [133], which was solved by Onsager [214] for the infinite-volume limit on a torus. For higher values of q the problem becomes much harder and no analytical solution has been found yet. Only at the critical temperature, the exact partition function of the Potts model on the square, triangular and honeycomb lattices have been obtained [289]. It is of interest to study the problem in the form of a strip lattice. Hopefully, the study of sufficiently wide strips could contribute at understanding the physical properties of such complex systems under different boundary conditions.

An effective technique for obtaining the partition function of *strip lattices* is to compute its transfer matrix, denoted M. The transfer matrix technique allows the study of strips that repeat their lattice structure along one of its dimensions. M can be computed symbolically or numerically (fully or partial) evaluated on (q, v). When there is enough disk space, we find that it is more convenient to compute M using polynomials on (q, v). Indeed, computing M with general (q, v) has an impact on performance and memory, but it gives the advantage that M will not have to be re-computed many times when doing numerical sweeps for q and v. Another advantage is that from the general (q, v) transfer matrix one can generate many partially evaluated instances of the transfer matrix that can be used later for numerical sweeps on the remaining parameter. For limited computational resources, generating M partially or fully evaluated is a practical choice.

If the strip lattice represents an infinite band, then analysis can be performed by computing the eigenvalues of M. If the strip lattice is finite, then an initial condition vector $\vec{Z}_0$ is needed. In that case, boundary conditions have to be specified. Typical boundary conditions are free, periodic, cylindrical and cyclic. M and $\vec{Z}_0$ together form a partition function vector $\vec{Z}_n$ based on the following recursion:

$$\vec{Z}(n) = M\vec{Z}_{n-1} = \vec{Z} = M^n\vec{Z}_0 \tag{3.1}$$

Computing the powers of M^n is done in a numerical context, otherwise memory usage would become intractable. When M^n is computed, only the first element of $\vec{Z}_n$ becomes the partition function of the strip lattice, because it uses the original initial conditions from $\vec{Z}_0$, while the other elements use different initial conditions. The choice of the initial vector $\vec{Z}_0$ is important, since it cannot be orthogonal to the eigenvector corresponding to the first eigenvalue of M.

This work focuses on the process of building M, which is an *NP-hard* problem [285] where exponential cost algorithms are involved in the process, with the width m as the exponent. There are different approaches for building M: (1) In the *spin representation* approach, an integer value is chosen for q and the transfer matrix M is obtained by combining the different spin configurations in the graph layer. Under this approach, the size of M becomes $q^{|V|} \times q^{|V|}$, where $|V|$ is the number of spins in the layer of the strip. A more detailed explanation on the spin representation approach is available in the first of the six works by Salas, Sokal and Jacobsen series of papers [241]. (2) One can also obtain M as a product of sparse matrices of

asymptotic size $O(4^m)$ [33, 137], one per edge and practically linear in the number of edges, where M is not constructed explicitly but only its action on a given vector of states. (3) Alternatively one can compute M with a generic (q, v) method where the configuration space grows proportional to the Catalan numbers [50] or asymptotically as $O(4^m)$, leading to a matrix of size $O(4^m \times 4^m)$. Indeed there are other strategies that can achieve smaller transfer matrices [23, 98, 242], but they assume special properties for the lattice, work only for finite graphs or need to fix the values of v and/or q in order to take any advantage. We believe it is worth studying what are the possibilities for algorithmic improvements in the generic (q, v) Catalan based approach since it is a general method applicable to any planar strip.

In the light of these aspects just mentioned, we ask ***question 1:*** *Is there a generic* (q, v) *method that can compute the transfer matrix for any planar strip lattice, using a sub-Catalan configuration space?*. From our research we have found that: *a hierarchical symmetry exists among elements of the configuration space that define the transfer matrix.* This symmetry is revealed when first applying deletion-contraction to certain edges of the strip layer. If this symmetry is used so that the configuration space is re-organized as a forest of hierarchical families, then a parallel computation only on the root nodes is sufficient for generating a compressed transfer matrix. When exploiting this symmetry, the configuration space is reduced from $O(4^m)$ to $O(3^m)$, which is an improvement to the actual bound on general transfer matrix methods for strips. This result allows us to answer positively to *question 1.*

With the evolution of computer architectures towards a higher amount of cores [30, 76], parallel computing is not anymore limited to clusters or super-computing; workstations can also provide high performance for solving physical problems [200]. It is in this last category where most of the scientific community lies, therefore parallel implementations for multi-core machines are the ones to have the largest impact on the community. Considering how technology is changing, we ask ***question 2:*** *Can transfer matrix methods work in parallel for modern multi-core architectures and scale their performance efficiently as more processors are used?*. Given the amount of data-parallelism on the number of root nodes, the performance of the algorithm scales efficiently as more processors are used. Results on a multi-core 8-core machine show a speedup of $5.7X$ is achieved when using $p = 8$ processors, and an efficiency of 95% is achieved when using $p = 4$. Results on a 32-core cluster confirm that the implementation can scale in a distributed scenario, achieving a speedup of $28X$ when using $p = 32$ processors and an efficiency of over 90% for the full range $p \in [1, 32]$ when dealing with large square strips. We can also confirm that a compressed transfer matrix not only saves data space in comparison to the original one, but it is also faster to load considering that it must be first evaluated for any practical usage. In the case of cluster performance, a dynamic scheduler is mandatory in order to bypass potential *performance valleys* that are caused by the combination of unbalanced work and a static scheduler. Again, this result allows a positive answer for *question 2.*

3.3 Related work

The transfer matrix methods were introduced by Derrida *et. al.* in 1980 [72] as an approach to study percolation and phenomenological re-normalization. In 1982, Baxter used transfer ma-

trix techniques in his seminal works as a tool for solving statistical mechanics problems [19]. Salas, Sokal and Jacobsen have greatly contributed with a series of results, plus an additional unnumbered one that follows the same line, in which they study the physics of square and triangular strip lattices through the transfer matrix technique [140, 141, 142, 143, 240, 241, 242]. In those works, the authors use different types of algorithmic optimizations for the construction of M based on the symmetries available. Different scenarios are considered along the works, such as the zero temperature (chromatic polynomial) case, ferromagnetic and antiferromagnetic cases, and different boundary conditions such as free, periodic, cylindrical and a special boundary condition that consists of adding two extra vertices on the sides of the strip. Some of the contributions made in these works include the use of non-nearest neighbors partitions for $v = -1$, sparse matrix factorization, algebraic input from the representation of the Temperley-Lieb algebra, symmetries for different boundary conditions and the computation of the limiting curves or partition function zeroes for the different boundary conditions up to $m \leq 13$. State of the art works on the square lattice normally study strips in the range $3 \leq m \leq 13$. For the case of the square lattice with free boundary conditions, Salas *et. al.* achieved $m = 12$ using $v = -1$ [240]. It should be noted that if $v \neq -1$ and free boundary conditions are used, then the configuration space is the one proportional to the Catalan numbers and the problem becomes computationally harder to handle. The problem of the matrix size has also been improved by algebraic techniques [98] in the spin representation, reducing the matrix size when working with $q = 2$ and $q = 3$. The authors studied the square and triangular strips with layers of up to $r = 11$ spins, which is equivalent to a square strip of width $m \approx 5$. Jacobsen *et. al.* have studied the q-state Potts model for $q = 4cos^2(\pi/a)$ being a Beraha number with $a > 2$ and integer [141]. In the work, the authors study strips of widths in the range $m \in [2, 6]$. The relevance of their work is that they manage to compute the partition function using the RSOS representation. Álvarez *et. al.* [11] have reported exact results for the kagome strip of width $m = 5$ using the generic (q, v) Catalan based transfer matrix technique. In contrast to these related works, we are interested in exploring a general (q, v) method that can allow the study of strips in the state of the art range for free boundary conditions using generic (q, v). For simplicity, we will restrict our physical results just to the computation and validation of the limiting curves using free boundary conditions in order to stay within the scope of our work, but not restrict the proposed strategy to these conditions.

More general methods for computing the exact partition function of a lattice have also been proposed [23, 121, 248]. Bedini *et. al.* [23] proposed a transfer matrix method for computing the partition function of arbitrary graphs using a tree-decomposed transfer matrix technique. For arbitrary graphs, they mean any type of finite graph; *i.e.*, random or regular planar/non-planar graphs. In their work, the authors obtain a sub-exponential complexity when processing random planar graphs. Their algorithm is considered the best so far for arbitrary graphs and the authors manage to achieve results for regular lattices of up to 18×18 sites. If the tree-decomposed transfer matrix method is applied to a strip, the configuration space to explore becomes the same as the traditional transfer matrix methods for strips, *i.e.*, the tree-width becomes the width of the strip and the cost is proportional to the Catalan number of the tree-width. The work is closely related to another result by Jacobsen in which large regular lattices of up to 20×21 sites were studied [137] by using a sparse transfer matrix method based on the product of sparse matrices, of dimension 3^m for $v = -1$ and 4^m for $v \neq -1$. The work of Haggard *et. al.* [115] is considered to have the best implementation

of a deletion-contraction technique for the computation of the Tutte polynomial for any arbitrary graph (the Tutte polynomial is the dual of the partition function [282]). Their algorithm reduces the computation tree in the presence of loops, multi-edges, cycles and biconnected graphs (as one-step reductions). By using a cache, some computations can be reused (*i.e.*, sub-graphs that are isomorphic to the ones stored in the cache do not need to be computed again). An alternative algorithm to Haggard *et. al.* was proposed by Björklund *et. al.* [29] which achieves exponential time only in the number of vertices; $O(2^n n^{O(1)})$ with $n = |V|$. Asymptotically their method is better than deletion-contraction considering that many interesting lattices have more edges than vertices. However, Haggard *et. al.* [115] have stated that the memory usage of Björklund's method is too high for practical use. These techniques, which are more general than the ones from the beginning of this section, cannot be directly compared against the classic transfer matrix approach, nevertheless they still needed to be mentioned as part of the related work background. General techniques compute the transfer matrix efficiently for arbitrary graphs, but do not take advantage of the regular graph structure when it is available. On the other hand, classic transfer matrix methods for strips indeed take advantage of the regular graph structure but for arbitrary graphs are not so efficient because for each layer there is a new non-sparse transfer matrix to be computed. Both strategies play an important role in the study of spin lattices. In our case, we focus on strips with regular graph structure, therefore our approach should be considered as a classic transfer matrix method.

Research on transfer matrices for strip lattices in the Potts model have not reported experimental results on the parallel performance, except for a prior work of the authors [199] that consists of a parallel method for computing general (q, v) transfer matrices using the Catalan approach, which will be named the *Catalan Parallel Method (CPM)* for the ease of referencing it later on. The CPM method was successfully used to study new widths of the kagome strip [11] with generic (q, v). The present work is a substantial improvement from CPM.

3.4 Algorithm overview

3.4.1 Data structure

Since the graph is a strip lattice, only layer $K_n(V', E')$ of the graph $G(V, E)$ is explicitly needed. The following naming scheme is now introduced for distinguishing two types of boundary vertices in the layer: *shared vertices* and *external vertices*. For convention, *shared vertices* are indexed top-down from 0 to $m - 1$ and correspond to the left-most ones of K_n, which are being shared with layer K_{n-1}. *External vertices* are the right-most ones of K_n and are indexed bottom-up from $|V'| - m$ to $|V'| - 1$. Figure 3.1 illustrates the data structure for an square strip of $m = 3$.

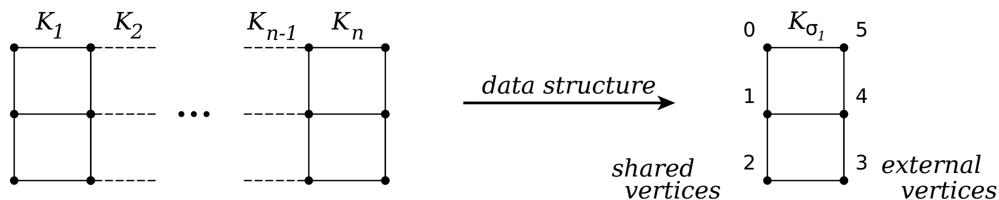

Figure 3.1: Example data structure for a square strip lattice of width $m = 3$.

3.4.2 DC-based transfer matrix computation

When using (q, v) polynomials, the configuration space of generic q transfer matrix methods turns out to be the set of all *non-crossing partitions* on a sequence of m serially connected vertices. The size of this configuration space is defined by the Catalan numbers:

$$\Gamma(m) = C_m = \frac{1}{m+1}\binom{2m}{m} = \frac{(2m)!}{(m+1)!m!} = \prod_{k=2}^{m} \frac{m+k}{k} \tag{3.2}$$

We will first explain how the transfer matrix can be built from partial DC repetitions and then proceed to the *parallel family trees* strategy.

At this point we introduce two terminologies that are important for the rest of the section; *initial configurations* and *terminal configurations*. These configurations define a combinatorial sequence of identifications[1] on the *external* and *shared vertices* of layer K_n. *Initial configurations*, denoted σ_i with $\mathrm{i} \in [0..C_m - 1]$, define a combinatorial sequence of identifications just on the *external vertices* of K_n. The *terminal configurations*, denoted φ_j with $j \in [0..C_m - 1]$, define a combinatorial sequence of identifications just on the *shared vertices* of K_n. Initial configurations generate terminal ones, through the DC method.

The case of σ_1 is the basic case and matches K_n. That is, σ_1 is the *initial configuration* where no identifications are applied to the *external vertices* of K_n. It is equivalent as saying that σ_1 is the empty partition of the Catalan set. Similarly, φ_1 corresponds to the base case where no *shared vertices* are identified. In other words, φ_1 is the empty configuration for the Catalan set on the *shared vertices* of K_n. For illustration, Figure 3.2 shows the configuration space for the square lattice of width $m = 3$:

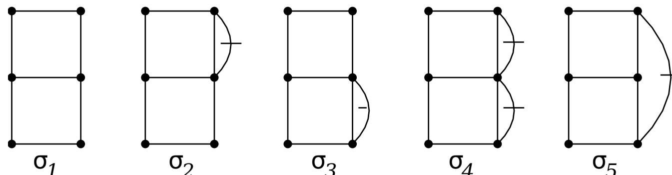

Figure 3.2: The configuration space for a square lattice of width $m = 3$.

In order to compute the transfer matrix M (row by row), one must apply C_m partial DCs, each time to a different *initial configuration* σ_i. Each one of the C_m partial DC applications

[1]For identification we mean a pair of vertices that actually represent a single vertex (they are identified). Graphically, it is represented by a crossed curved connecting the pair of vertices.

generates a row of M in the form of partial partition functions on (q, v), distributed into a maximum of C_m *terminal configurations*. By *partial DC* we mean to perform DC on the layer, with the corresponding *initial configuration* σ_{i} applied, but stopping the recursion branches whenever they meet an edge that connects two *shared vertices*. The stop condition on the recursion branches is needed otherwise one would be processing vertices and edges of the next layer of the strip, breaking the idea of a transfer matrix. For the example of Figure 3.1 with $m = 3$, the partial DC is applied to σ_1, σ_2, σ_3, σ_4 and σ_5 from Figure 3.2.

An example of a partial DC for the example of $m = 3$ is illustrated in Figure 3.3 for the case when computing the first row. The process is analogous for the other four rows of M (*i.e.*, σ_2, σ_3, σ_4 and σ_5).

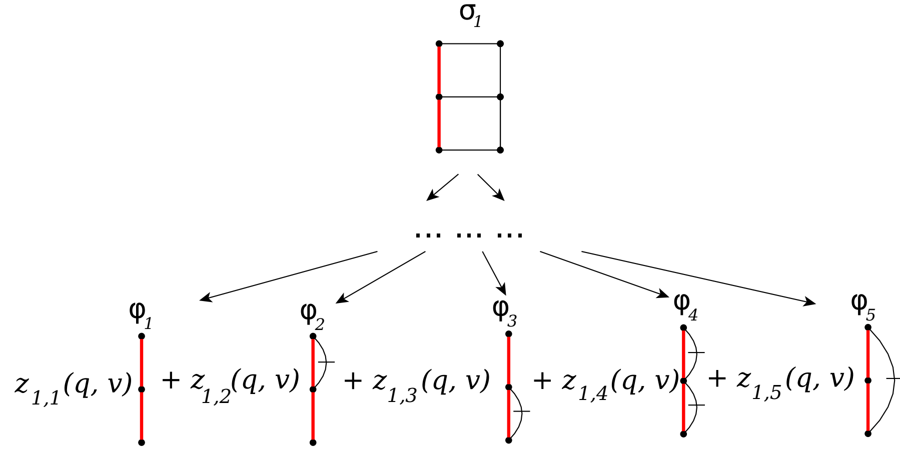

Figure 3.3: Terminal configurations generated from a partial DC on a square strip of width $m = 3$.

Once a recursion branch has been stopped, partial partition functions $z_{\mathrm{i},j}(q, v)$ appear associated to remanents of the graph layer. Remanents are parts of the graph layer that cannot be computed (*i.e*, edges connecting *shared vertices*) and they match one of the C_m possible *terminal configurations* that can exist. For some *initial configurations*, not all *terminal configurations* may be generated from a single DC, but only a subset of them.

A *terminal configuration* φ_j contains a unique sequence of planar identifications on the *shared vertices* that is useful to differentiate one from another. We use the term *key* to denote such sequences since they allow fast search and modification in a hash table. Proper construction of *keys* are achieved by using a simple algebra that defines how multiple identifications on *shared vertices* are combined. A key of n identifications is denoted as $\Pi = \pi_{x_1,y_1} + \pi_{x_2,y_2} + ... + \pi_{x_n,y_n}$. The following properties hold true for keys:

$$\pi_{a,b} = \pi_{b,a} \tag{3.3}$$

$$\pi_{a,b} + \pi_{c,\mathrm{d}} = \pi_{c,\mathrm{d}} + \pi_{a,b} \tag{3.4}$$

$$\pi_{a,b} + \pi_{b,c} = \pi_{a,b,c} \tag{3.5}$$

Properties (3.3) and (3.4) allow the application of a lexicographical order on the keys, while property (3.5) allows them to be combined through transitivity. There are important differences when comparing this algebra to the partition algebras studied by Halverson and Ram [116], specially because the former is much simpler and defines operations on a single layer

of points, while the latter defines a different set of operations for a partition monoid that is represented as a graph of two layers of points. Nevertheless, we can still find a relation with the number of partitions in the case of the planar sub-monoid P_k, which is C_{2k} for two layers of length k, and the number of *keys* for a single layer of length m, which is C_m.

Using Stirling's approximation, we have that $C_m \approx \frac{4^m}{m^{3/2}\sqrt{\pi}}$, which is consistent with the upper bound:

$$C_m = \frac{1}{m+1}\binom{2m}{m} \leq \binom{2m}{m} \leq 4^m \tag{3.6}$$

Dutton and Brigham proved in 1986 that the Stirling approximation of the Catalan numbers is in fact already a valid upper bound [78]. In addition, they obtain tighter lower and upper bounds for the Catalan numbers. The cost of the DC-based transfer matrix method is the product of the cost of the partial DC and the size of the configuration space C_m.

So far, the worst case running time of the algorithm for computing M is:

$$T(K_n(V', E'), m) = O\Big(\Gamma(m) \cdot DC(K_n))\Big) = O\Big(4^m \cdot min\Big(2^{|E'|}, \Big(\frac{1+\sqrt{5}}{2}\Big)^{|V'|+|E'|}\Big)\Big) \tag{3.7}$$

In the following sub-section, we show how a finer analysis can lead to a smaller configuration space of $\Gamma(m) = O(3^m)$ for computing a compressed transfer matrix M.

3.4.3 Family trees strategy

It is possible to reduce the Catalan configuration space by exploiting a symmetry present in the *deletion-contraction* (DC) method, resulting in an exponentially faster algorithm. Basically, the idea is the following: *if the DC procedure is forced to act first on certain external edges of the layer, and act later on the rest of the graph, then symmetries appear between nodes of the recursion tree and other initial configurations.* Exploiting such symmetry allows one to group many Catalan configurations into families of configurations, where a single DC procedure applied to the root node of a family contributes to the solution of the whole family.

Forcing DC to start on the external edges results in a recursion tree composed of two phases; (1) a perfect binary tree (PBT) of height $h = m - 1 - b$ and (2) several sub-trees t_j with $j \in [1..2^h]$ (see Figure 3.4).

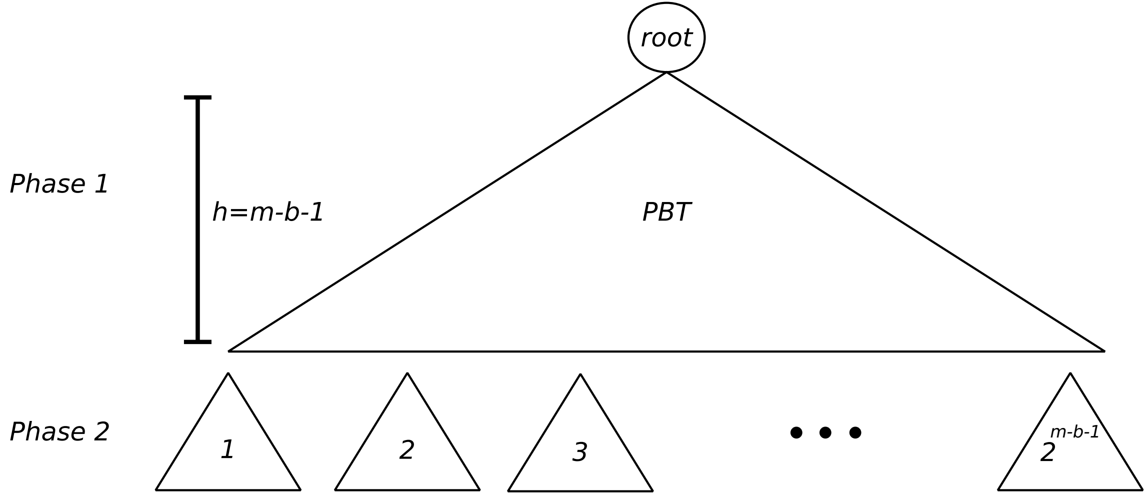

Figure 3.4: When DC is forced to start on the external edges, the recursion is divided into two phases.

Variable b is the number of external edges that sit in between an identification $\pi_{\mathrm{i}j}$ where at least one of its vertices is i or j. These b edges are left for phase (2) because they do not produce the symmetries needed for the *family trees strategy*. Each node of the PBT of phase (1) that comes from a contraction produces a unique algebraic symmetry to one of the configurations found in the original Catalan set. The configuration of a contracted node from the recursion tree is denoted χ_{i} and the symmetric correspondence is $\chi_{\mathrm{i}} \longleftrightarrow \sigma_{\mathrm{i}}$. All χ_{i} configurations that share the same PBT, together form a *family tree*. Following the example of the square strip with $m = 3$, its configuration space would be grouped into two *family trees* (see Figure 3.5); $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ and $\{\sigma_5\}$, being σ_1 and σ_5 their root configurations, respectively.

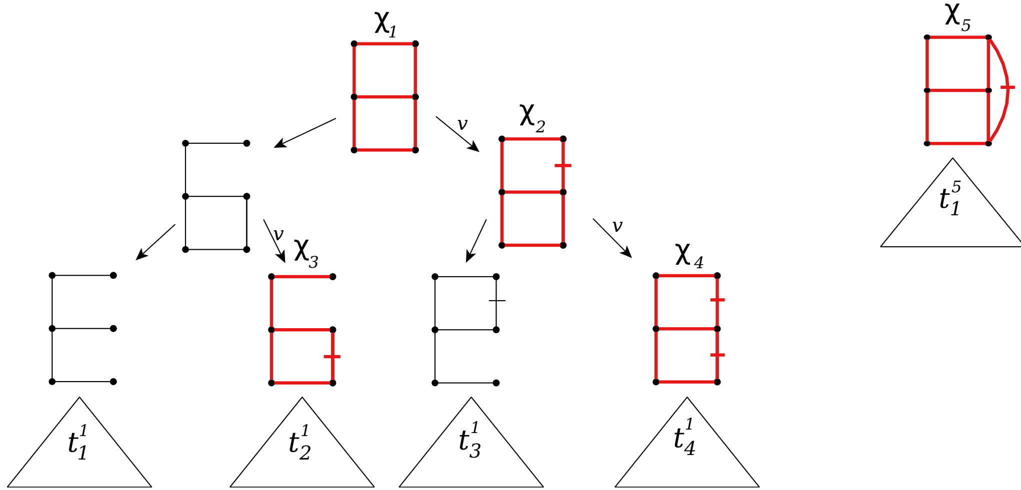

Figure 3.5: An example of the perfect binary tree and subtrees for $m = 3$.

The solution of a configuration, namely $\langle\sigma_{\mathrm{i}}\rangle$, is defined in terms of its symmetric χ_{i} found in the PBT:

$$\langle\sigma_{\mathrm{i}}\rangle = (1+v)^c \sum_{k=0}^{2^{\mathrm{d}}-1} v^{b(k)} \langle\chi_{\mathrm{i}}^k\rangle \tag{3.8}$$

Variable d denotes the number of deletions (*i.e.*, holes in the external layer) and variable c the

contractions accumulated along its path, both starting from the root. The $(1+v)^c$ coefficient corresponds to the expression for the c loops that are present in the external layer of $\sigma_{\rm i}$, but are missing in $\chi_{\rm i}$. For the example of the square strip of width $m = 3$, $c = 0, 1, 1, 2, 0$ for $\chi_1, \chi_2, \chi_3, \chi_4, \chi_5$, respectively. Function $b(k)$ counts the number of non-zero bits of k and the expression $\chi_{\rm i}^k$ is the application of the binary mask k just on the holes of $\chi_{\rm i}$. The mask works as follows: if bit $k_j = 1$, with $j \in [0..{\rm d}-1]$, then the j-th hole is filled with an edge, otherwise it is left as a hole.

When ${\rm d} = 0$, $\chi_{\rm i}$ represents exactly the starting point of an eventual solution $\langle\sigma_{\rm i}\rangle$, algebraically symmetric in $(1+v)^c$. When ${\rm d} > 0$, $\chi_{\rm i}$ is no longer the starting point of $\langle\sigma_{\rm i}\rangle$, but instead it is the left-most node in an eventual recursion tree of the solution $\langle\sigma_{\rm i}\rangle$, at level d. In order to compute $\langle\sigma_{\rm i}\rangle$, $2^{\rm d}-1$ variations of $\chi_{\rm i}$ are needed to build the missing steps and eventually reach $\sigma_{\rm i}$ in a bottom-up way. An important property of the variations of $\chi_{\rm i}$ is that they actually correspond to other family members within the PBT that will be eventually solved too. This means that there is no need to compute these variations, instead one has to make the correct relations between the different family members. We propose a hash map of the type $(\chi_{\rm i}, r[])$ so that for each $\chi_{\rm i}$, represented by its unique key, there is an array of related configurations $r[]$ that need $\langle\chi_{\rm i}\rangle$. Each time a contracted configuration is reached in the PBT, equation (3.8) is applied and $2^{\rm d}-1$ relations are inserted in the hash map. Figure 3.6 illustrates the example of the strip of width $m = 3$ when processing χ_3; it needs χ_4 in order to build the solution $\langle\sigma_3\rangle$.

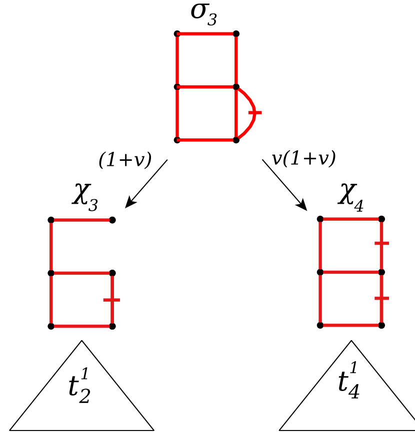

Figure 3.6: An illustration of how χ_3, with ${\rm d} = 1$, builds the solution of σ_3 bottom up with the help of χ_4.

The solution for each family member $\langle\chi_{\rm i}\rangle$ can be written in terms of the solutions of the 2^h subtrees. A convenient way for storing the solution for a whole family is to write a system of equations, using a linear combination of the 2^h sub-trees. A v^c coefficient is included, where c is the amount of contractions found in the path from the familiar to the sub-tree. For the example of the strip of $m = 3$, the solution for the family of σ_1 is:

$$\langle\sigma_1\rangle = \langle\chi_1\rangle = \langle t_1^1\rangle + v\langle t_2^1\rangle + v\langle t_3^1\rangle + v^2\langle t_4^1\rangle, \tag{3.9}$$

$$\langle\sigma_2\rangle = (1+v)\langle\chi_2\rangle = (1+v)[\langle t_3^1\rangle + v\langle t_4^1\rangle] \tag{3.10}$$

$$\langle\sigma_3\rangle = (1+v)[\langle\chi_3\rangle + v\langle\chi_4\rangle] = (1+v)[\langle t_2^1\rangle + v\langle t_4^1\rangle] \tag{3.11}$$

$$\langle\sigma_4\rangle = (1+v)^2\langle\chi_4\rangle = (1+v)^2\langle t_4^1\rangle \tag{3.12}$$

Note how $\langle \sigma_3 \rangle$ includes $\langle \chi_4 \rangle$, as shown in Figure 3.6. The solution for the family of σ_5 is

$$\langle \sigma_5 \rangle = \langle \chi_5 \rangle = \langle t_1^5 \rangle \tag{3.13}$$

These equations, plus the solutions of the sub-trees, conform the compressed transfer matrix for the example strip of width $m = 3$. It is important to mention that the sub-trees are stored only once and the system of equations use indices to the sub-trees.

Given how DC works, identification can only occur on pairs of vertices that are neighbors. This aspect of DC allows us to establish a formal definition for a family.

Definition 3.1 *A family is a set of configurations in which for any chosen pair $\sigma_{\rm i}$ and σ_j of the set, the difference of their corresponding keys $\Pi^{\rm i}$ and Π^j is $\Pi^{{\rm i}-j} = \pi_{x_1,x_1+1} + \pi_{x_2,x_2+1} + ... + \pi_{x_n,x_n+1}$.*

In other words, the difference between $\sigma_{\rm i}$ and σ_j must only consist of identifications of length $l = 1$. Configurations that differ at least by one identification of length $l > 1$ belong to a different family. Each family is identified by its root configuration, therefore it is important to know which configurations are root and which are not.

Definition 3.2 *A root configuration is an instance of K_n where its key $\Pi = \pi_{x_1,y_1} + \pi_{x_2,y_2} + ... + \pi_{x_n,y_n}$ satisfies $|x_{\rm i} - y_{\rm i}| > 1$ for ${\rm i} \in [1..n]$.*

That is, a root configuration is one that does not have identifications of length $l = 1$. The number of root configurations will be denoted Δ_m as a function of the width m. We formulate the following expression for Δ_m, based on Definition 3.2 and using the *inclusion-exclusion* principle:

$$\Delta_m = \sum_{k=0}^{m-1} (-1)^k \binom{m-1}{k} C_{m-k} \tag{3.14}$$

Theorem 3.3 *The amount of root configurations is upper bounded as $\Delta_m = O(3^m)$.*

Proof. Using (3.6) into (3.14) leads to the following bound:

$$\Delta_m = \sum_{k=0}^{m-1} (-1)^k \binom{m-1}{k} C_{m-k} \leq \sum_{k=0}^{m-1} \binom{m-1}{k} (-1)^k 4^{m-k} = 4 \sum_{k=0}^{m-1} \binom{m-1}{k} (-1)^k 4^{m-1-k} \tag{3.15}$$

$$= 4(4-1)^{m-1} \tag{3.16}$$

$$= O(3^m) \tag{3.17}$$

Step 3.16 is obtained by using the Binomial formula with $x = 4$ and $y = -1$. □

The number of root configurations Δ_m corresponds to the number of *non-crossing non-nearest-neighbor partitions* (*nc-nnn*). The number of *nc-nnn* can also be counted with the Motzkin number evaluated at $m - 1$; $\Delta_m = M_{m-1}$, where M_m is

$$M_m = \sum_{j=0}^{\lfloor m/2 \rfloor} \binom{m}{2j} C_j \tag{3.18}$$

The asymptotic number of *nc-nnn* partitions has been previously studied by Chang *et. al.* in [47] by using the asymptotic behavior of M_m:

$$M_m = \frac{3^{3/2}}{2\sqrt{\pi}\ m^{3/2}} 3^m \Big[1 + O(m^{-1})\Big] \tag{3.19}$$

Although the asymptotic bound was already obtained in two earlier works [47, 242] in the context of *nc-nnn partitions*, the proof of Theorem 3.3 still remains interesting as a short and alternative way to establish the $O(3^m)$ upper bound coming from an inclusion-exclusion formulation that has not considered the Motzkin numbers.

Upper bound for relating k-hole familiars

Counting the amount of family relations within a DC procedure allows one to precise an upper bound on the number of accesses made to the hash map. For each DC application, the cost of relating family members is defined as:

$$g(h) = \sum_{k=0}^{h-1} c(k,h) r(k) \tag{3.20}$$

Where $r(k) = 2^k - 1$ is the cost of performing the relations for a k-hole configuration. Function $c(k,h)$ counts the number of k-hole configurations, which is a subset of the total number of familiars. Since familiars can only be contracted nodes within the PBT, the size of a family is 2^{h-1}. A direct upper bound can be computed assuming the worst case for $r(k)$:

$$g(h) < (2^m - 1) \sum_{k=0}^{h-1} c(k,h) \leq (2^m - 1)2^h < 4^m = O(4^m) \tag{3.21}$$

A tighter upper bound is possible when $c(k,h)$ is analyzed more carefully. The following pattern can be found when counting the number of k-hole configurations.

$$c(0,h) = h \tag{3.22}$$

$$c(1,h) = 1 + 2 + ... + h - 1 \tag{3.23}$$

$$c(2,h) = (1) + (1+2) + ... + (1+2+3... + h - 2) \tag{3.24}$$

$$c(3,h) = \big[(1)\big] + \big[(1) + (1+2)\big] + ... + \big[(1) + (1+2) + ... + (1+2+3+...+h-3)\big] \tag{3.25}$$

The recursion for $c(k,h)$ is:

$$c(k,h) = \sum_{\mathrm{i}=0}^{h-k} c'(k-1,\mathrm{i}), \quad 1 \leq k \leq h-1 \ \ \& \ c(0,h) = h \tag{3.26}$$

$$c'(k,h) = \sum_{\mathrm{i}=0}^{h} c'(k-1,\mathrm{i}), \quad 1 \leq k \leq h-1 \ \ \& \ c'(0,h) = h \tag{3.27}$$

Function $c(k, h)$ is equivalent to counting the number of k-faces in a regular $(h-1)$-simplex [64]. A regular $(h-1)$-simplex is a $(h-1)$-dimensional polytope that is the convex hull of h vertices in a regular spatial distribution. A regular simplex can also be seen as the generalization of the notion of a triangle or a tetrahedron, for an arbitrary dimension. A regular $(h-1)$-simplex can be drawn in the plane by placing h vertices inscribed in a circle, with all pairs connected (see Figure 3.7).

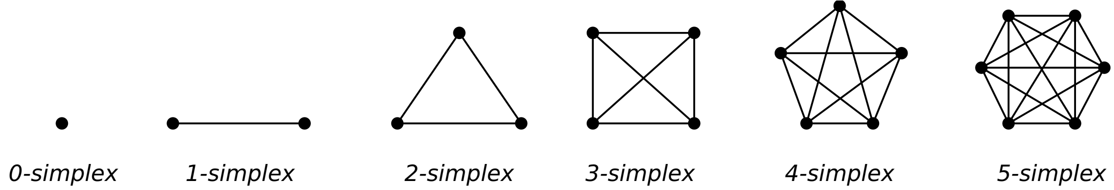

Figure 3.7: Examples of regular simplexes drawn on the plane.

The number of k-faces in a $(h-1)$-simplex [125] is defined as:

$$c(k, h) = \binom{h}{k+1} \tag{3.28}$$

Using (3.28) in (3.20), we have that

$$g(h) = \sum_{k=0}^{h-1} \binom{h}{k+1} (2^k - 1) \tag{3.29}$$

Theorem 3.4 *The cost of relating all configurations within a PBT is upper bounded as* $g(m-1) = \frac{1}{6}(3^m - 3 \cdot 2^m + 3) = O(3^m)$.

Proof. For simplicity, we will assume that every DC application processes the default initial configuration. This configuration is the one that spans the largest family, hence the worst case where $b = 0$, that is $h = m - 1$.

$$g(h) \leq g(m-1) = \sum_{k=0}^{m-2} \binom{m-1}{k+1} (2^k - 1) = \sum_{k=0}^{m-2} \binom{m-1}{k+1} 2^k - \sum_{k=0}^{m-2} \binom{m-1}{k+1} \tag{3.30}$$

Both summations obey the following form:

$$\sum_{k=0}^{m-2} \binom{m-1}{k+1} a^k = \frac{1}{a} \sum_{k=1}^{m-1} \binom{m-1}{k} a^k = \frac{1}{a} \left(-1 + \sum_{k=0}^{m-1} \binom{m-1}{k} a^k \right) \tag{3.31}$$

Using the Binomial theorem for the summation, we get

$$\frac{1}{a} \left(-1 + \sum_{k=0}^{m-1} \binom{m-1}{k} a^k \right) = \frac{(a+1)^{m-1} - 1}{a} \tag{3.32}$$

Using $a = 2$ and $a = 1$ leads to the first and second terms of Eq. (3.30)

$$g(h) \leq g(m-1) = \frac{3^{m-1} - 1}{3} - \frac{2^{m-1} - 1}{2} = \frac{1}{6}(3^m - 3 \cdot 2^m + 3) = O(3^m) \tag{3.33}$$

□

Running time of the family trees strategy

The asymptotic sequential running time of the family trees algorithm applied to the layer $K_n(V', E')$ of a strip lattice is:

$$T(K_n(V',E'),m) = \Delta_m\Big(DC + g(m-1)\Big) \tag{3.34}$$

$$= O\Big(3^m\Big(min\Big(2^{|E'|}, \Big(\frac{1+\sqrt{5}}{2}\Big)^{|V'|+|E'|}\Big) + 3^m\Big)\Big) \tag{3.35}$$

The extra cost provided by $g(m-1)$ does not incur in too much extra computation compared to the cost of DC itself, where the amount of edges of $K_n(V', E')$ must at least double the amount of edges in the boundary, that is $E' \geq 2(m-1)$. Additionally, $g(m-1)$ is considering the worst case for each root configuration where $h = m-1$. In practice, all configurations, except for the default one, will have $h < m-b-1$ with $b > 0$.

Parallel family trees

By default, the algorithm does not know the Δ_m different root configurations except for σ_1 which is given as part of the input of the strip lattice and is the one that triggers the computation. Under this scheme, the configuration space would have to be explored incrementally, each time adding a sub-set of configurations from the *terminal configurations* found from a DC application. This is indeed a problem for parallelization because the data-parallel elements are being discovered sequentially, limiting the efficiency and scalability of a parallel computation. In order to solve this problem, we use a recursive generator $g(A[\,][\,], s, H, S)$, that with the help of a hash table H, generates all the Δ_m configurations before hand and stores them in an array S. $A[\,][\,]$ is an auxiliary array that stores the intermediate auxiliary subsequences and s is the accumulated sequence of identifications. Before the first call to $g(A[\,][\,], s, H, S)$, $A = [[0, 1, 2, ..., m-1]]$, s is null and H as well as S are empty. $g(A[\,][\,], s, H, S)$ is defined as:

```
g(A[][] ,s,H,S){
    if(!add_sequence(s,H,S))
        return;
    for(int k=0; k<A.size(); k++){
        for(int j=2; j<A[k].size(); j++){
            for(int i=0; i<j-1; i++){
                if(can_identify(A[k],i,j)){
                    cA = copy(A);
                    cs = copy(s);
                    identify(cA,i,j,k,cs);
                    divide(cA,i,j,k);
                    g(cA,cs,H,S);
}}}}}
```

Basically, $g(..)$ performs a recursive partition of the domain A. If $|j - \mathrm{i}| \leq 3$ then no further identifications can be carried on, otherwise the identification would be of length $l = 1$ and the generated configuration would not be a *root configuration*. Similarly, for the top and bottom parts if $|j - \mathrm{i}| \leq 2$ then no more identifications are possible. Each time a new identification i, j is added, the resulting configuration is checked in the hash table. If it is a new configuration, then it is added, else it is discarded as well as all further recursion computations continuing from that point. By using this approach we ensure that redundant recursion branches are never computed. Once $g(..)$ has finished, S becomes the array of all possible configurations and H the hash that maps configurations to indices.

Parallel family trees are achieved by first generating all root configurations with $g(..)$, followed by the parallel computation of p family trees simultaneously, using p processors and a total of Δ_m/p family trees per processor. The initial *key* needed by each processor p_{i} is obtained by reading in parallel from $S[p_{\mathrm{i}}]$, assuming the PRAM-CREW model. Once the *key* is obtained, it is applied to the *external vertices* of its own local copy of the base layer σ_1. Foster's four-step strategy [91] describes the design process of a parallel algorithm; *partitioning, communication, agglomeration, mapping*. The design steps for the parallel family trees is illustrated in Figure 3.8.

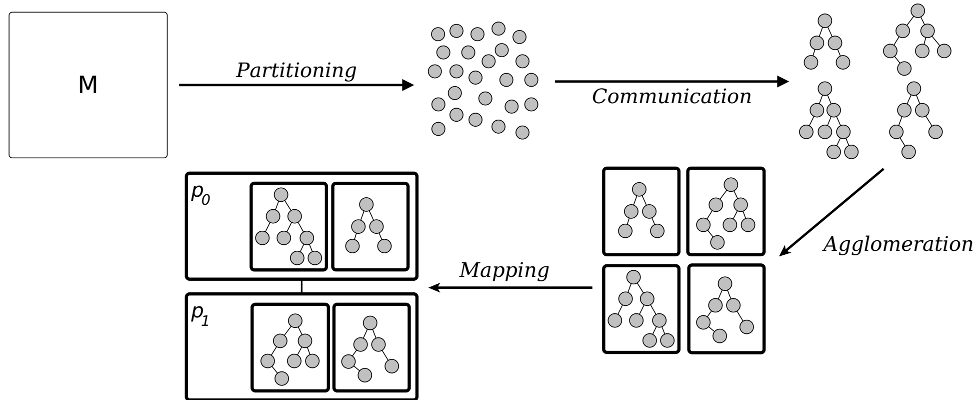

Figure 3.8: Foster's four step strategy for achieving parallel family trees, for two processors.

The work for each processor p_{i} is divided in the following steps: (1) pick one root configuration *key* from $S[]$, (2) apply it to its local copy of the K_{σ_1} layer, (3) perform the DC procedure, (4) write the results into non-volatile memory, *i.e.,* sub-tree results as well as the linear equations into disk, and (5) go to step (1) if there are still root configurations remaining. For step (3), familiars of a root configuration are detected at runtime within the PBT by computing its *key*, each time the recursion comes from a contraction. When the beginning of a sub-tree is reached, no more familiars are guaranteed to be found on what is left of the recursion, therefore the algorithm can proceed to compute the whole sub-tree without needing to check for the existence of familiars. The solution of a sub-tree t_{i} is a vector of expressions $z_{\mathrm{i},j}(q, v)$ that associates a j index to a *terminal configuration* φ_j within the sub-tree t_{i}. The hash-map H from the generator becomes useful for searching with average cost $O(1)$ the index j of a terminal configuration φ_j. Also, H ensures that all vectors are consistent with the order established in the generator and in the transfer matrix.

The 2^{m-1} sub-tree vectors and the coefficients for the set of equations provide the solution for a whole family. Both of these results are saved locally for each processor. This output for-

mat based on sub-trees and coefficients makes the matrix compressed in the same proportion of the improvement in the running time.

The asymptotic running time for the parallel family trees algorithm using p processors is:

$$T(K_n(V', E'), m) = O\Big(\frac{3^m}{p}\Big[DC(K_n) + g(k, m)\Big]\Big) \tag{3.36}$$

$$= O\Big(\frac{3^m}{p}\Big[min\Big(2^{|E'|}, \Big(\frac{1+\sqrt{5}}{2}\Big)^{|V'|+|E'|}\Big) + 3^m\Big]\Big) \tag{3.37}$$

Further computations for achieving physical results require decompression of the matrix, leading to a matrix of Catalan dimensions again. In practice, large symbolic matrices need first to be evaluated before doing any analysis. If the numerical evaluation is performed before decompressing the matrix, then the process is much faster than first decompressing and then evaluating, even faster than evaluating an uncompressed transfer matrix on (q, v). Numerical evaluation has the potential to be exponentially faster as a consequence of the parallel family trees compression, which is in the same order of the running time improvement.

The analysis of the algorithm has been made for the case of free boundary conditions but it is not restricted to it. For different boundary conditions such as cylindrical, full periodic or cyclic, the parallel family trees can be still applied following the same principle, while taking advantage of additional symmetries like the dihedral group in the cylindrical case. The rest of the paper assumes free boundary conditions unless we explicitly mention the contrary.

For the case of a finite strip, the initial conditions vector $\vec{Z}_1$ is computed by applying DC to each one of the C_m *terminal configurations*:

$$\vec{Z}_1 = (DC(\varphi_1), DC(\varphi_2), ..., DC(\varphi_{C_m})) \tag{3.38}$$

The computation of $\vec{Z}_1$ has very little impact on the overall cost of the algorithm and practically costs $O(mC_m)$ in time because a terminal configuration contains mostly *spikes* and/or *loops*, which are linear in cost for DC.

3.5 Algorithm improvements

3.5.1 Serial and Parallel paths

The DC contraction procedure can be improved for graphs that present *serial* or *parallel* paths between two endpoints v_a and v_b, as shown in Figure 3.9.

Figure 3.9: Serial and parallel paths.

A ***serial path***, denoted s, is a set of edges $e_1, e_2, ..., e_n$ that serially connects $n-1$ vertices between v_a and v_b. It is possible to process a serial path of n edges in one recursion step by using the following expression;

$$Z(K,q,v) = \left[\frac{(q+v)^n - v^n}{q}\right] Z(K_{-s},q,v) + v^n Z(K_{/s},q,v) \tag{3.39}$$

A ***parallel path*** p is a set of edges $e_1, e_2, ..., e_n$ that reduntandly connect v_a and v_b. It is possible to process a parallel path of n edges in one recursion step by using the following expression;

$$Z(K,q,v) = Z(K_{-p},q,v) + \left[(1+v)^n - 1\right] Z(K_{/p},q,v) \tag{3.40}$$

3.5.2 Axial Symmetry

One practical optimization is to detect the lattice's reflection symmetry when computing the root configurations as well as the Catalan configurations. When detecting reflection symmetry, the size of the configuration space is decreased for all symmetric pairs of *configurations*, no matter if it is *initial, terminal or root.* As the width of the strip lattice increases, the number of symmetric states increases too, leading to configuration spaces almost half the size of the original. We establish reflection symmetry between two *configurations* φ_a and φ_b with keys $\pi_{a_1,...,a_n}$ and $\pi_{b_1,...,b_n}$ respectively in the following way:

$$\pi_{a_1,...,a_n} = \pi_{b_1,...,b_n} \Leftrightarrow a_{\mathrm{i}} = (m-1) - b_{n-\mathrm{i}+1} \tag{3.41}$$

Exploiting this symmetry results in a matrix size C^s_m:

$$C^s_m = \frac{C_m}{2} + \frac{m!}{2\lfloor\frac{m}{2}\rfloor!} \tag{3.42}$$

For large values of m, $C^s_m \approx \frac{C_m}{2}$.

For the case of *root configurations*, Chang *et. al.* [47] proved that the number of non-crossing non nearest-neighbor partitions under reflection symmetry, which we denote Δ^s_m, is:

$$\Delta^s_m = \frac{1}{2}M_{m-1} + \frac{(m'-1)!}{2}\sum_{j=0}^{\lfloor m'/2\rfloor}\frac{m'-j}{(j!)^2(m'-2j)!} \tag{3.43}$$

where $m' = \left\lfloor\frac{m+1}{2}\right\rfloor$. The expression was also obtained by Salas and Sokal [242] for studying the square lattice symmetries when $v=-1$. When $m \to \infty$ we have:

$$\Delta^s_m \sim \frac{\sqrt{3}}{4\sqrt{\pi}\ m^{-3/2}} 3^m \left[1 + O(m^{-1})\right] \tag{3.44}$$

Table (3.1) shows how the amount of Catalan and root configurations increase for non-symmetric and symmetric lattices up to $m = 14$. If cylindrical boundary conditions are used, then the reflection symmetry can be replaced by the symmetry of the dihedral group which further reduces the size of the matrix. For this manuscript we limit our work to the case of free boundary conditions.

Table 3.1: Number of Catalan and root configurations under non-symmetric and symmetric cases.

m	C_m	C_m^s	Δ_m	Δ_m^s
1	1	1	1	1
2	2	2	1	1
3	5	4	2	2
4	14	10	4	3
5	42	26	9	7
6	132	76	21	13
7	429	232	51	32
8	1430	750	127	70
9	4862	2494	323	179
10	16796	8524	835	435
11	58786	29624	2188	1142
12	208012	104468	5798	2947
13	742900	372308	15511	7889
14	2674440	1338936	41835	21051

3.6 Implementation

We tried two implementations for the parallel family trees parallel algorithm; one using OpenMP [51] and the other one using MPI [90]. We observed that the MPI implementation achieved better performance in the multi-core scenario and allows parallel computation in a distributed scenario. For this, we decided to continue the research with the MPI implementation for both multi-core and distributed scenarios. Basic mathematical operations on symbolic expressions are handled through the GiNaC C++ library [18]. Parallel execution of the algorithm receives two parameters; the number of processors p and the block size B, which is the amount of consecutive jobs per process. When the parallelization is unbalanced, the value of B plays an important role for efficiently distributing work to all processors. In our implementation we make each process to generate its own H lookup table and S array. This small sacrifice in memory leads to better performance than if H and S were shared among all processes. There are mainly three reasons why the replication approach is better than the sharing approach: (1) caches will not have to deal with consistency of shared data, (2) there is no sending/receiving of data structures and (3) the allocation of the replicated data is correctly placed on memory modules when working under a NUMA architecture. The last claim is true because on NUMA systems memory allocations on a given process are automatically placed in its fastest location according to the NUMA topology between memory and CPU cores. It is responsibility of the OS (or make manual mapping) to stick the process to the same processor throughout the entire computation.

The implementation writes each row to a persistent secondary memory (*i.e.,* HDD or SSD) as soon as it is computed. Each processor does this with its own file, therefore the matrix is fragmented into p files. In practice, a fragmented matrix is not a problem at all, because numerical evaluation is needed before using the matrix in its full form. Furthermore, a fragmented matrix allows parallel numerical evaluation.

3.7 Performance results

We have realized performance tests for the parallel transfer matrix method implemented with MPI for both shared and distributed memory scenarios. The experimental design consists of measuring the main performance metrics (*i.e.,* running time, speedup, efficiency, knee) of the implementation by computing the compressed transfer matrix several times, each time varying the number of processors p. We also compute the improvement factor with respect to previous work [199]. The experiments are divided into two categories; (1) multi-core and (2) cluster. For each case, we measure performance with two strip lattices; (1) *square* and (2) *kagome*, respectively (see Figure 3.10).

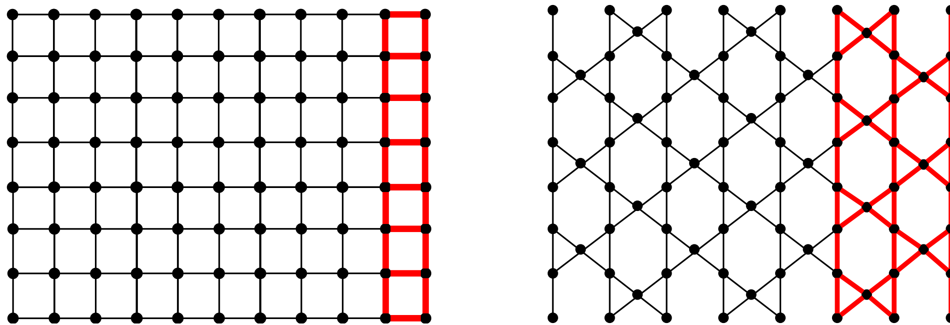

Figure 3.10: The square and kagome lattices used for measuring performance.

Explicit algebraic expressions for the sparse-matrix factorization of M for all the Archimedean lattices (which include the square and kagome lattices) have been computed by Jacobsen [138], on finite lattice regions of up to $|E| = 882$ edges. The approach taken by the sparse-matrix differs from the standard transfer matrix technique, since the former processes a whole finite lattice region, using one sparse matrix computation per edge, while the latter computes a dense TM for each different graph layer of width m.

Note: PFT refers to the actual *Parallel Family Trees* strategy and PCM to the *Parallel Catalan Method* from [199].

3.7.1 Multi-core results

The machine used for the multi-core performance tests has an 8-core CPU AMD FX-8350 at 4.0 GHz, 8GB of RAM and uses the *openMPI* implementation of the MPI standard [90].

Square strip lattice test

For the square lattice, we measure performance for 9 different strip widths in the range $m \in [2, 10]$. For each width, we measure 8 average execution times, one for each value of $p \in [1, 8]$. As a whole, we perform a total of 72 average measurements for the square test. The standard error for each average execution time is below 5%. Different block sizes where tested, giving no significant difference on performance. For this reason, we kept a block size

of $B = 1$. The other performance measures include speedup, efficiency and the *knee*[2] [79]. In this case we took advantage of the reflection symmetry for all sizes of m.

Figure 3.11 shows all four performance measures for the square lattice. From the results,

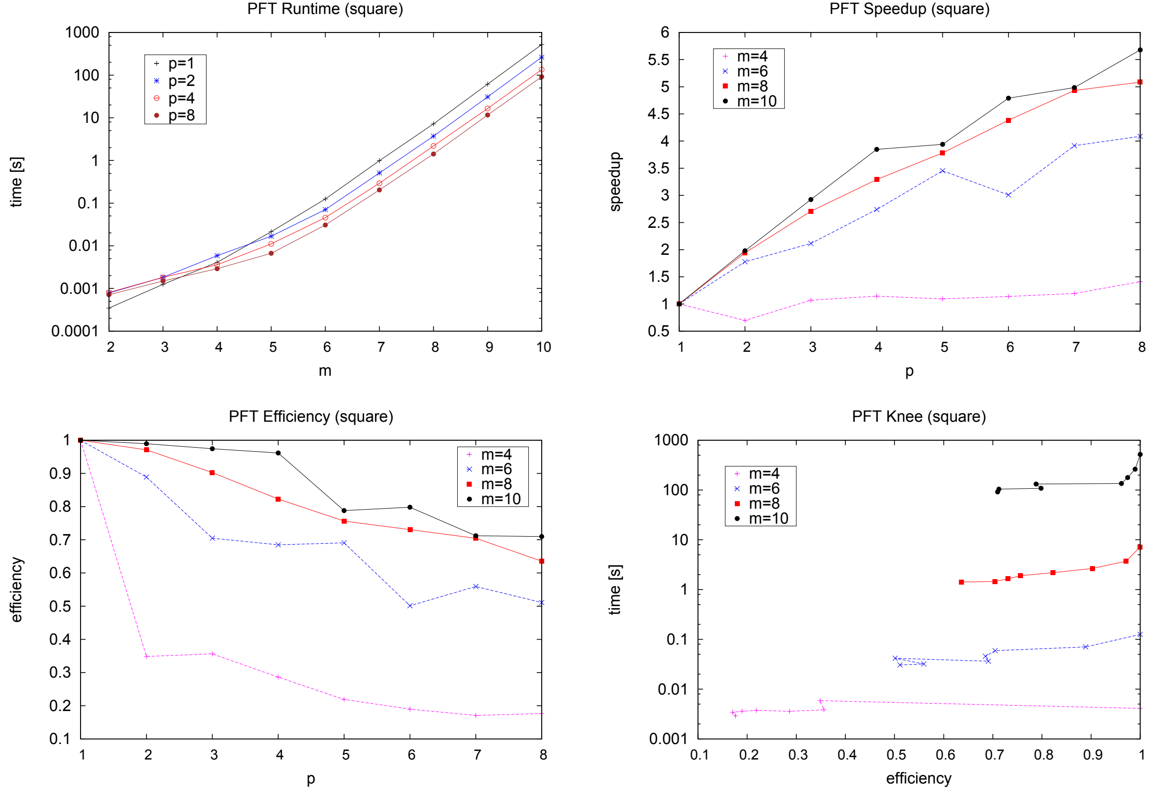

Figure 3.11: Multi-core running time, speedup, efficiency and knee for the square strip test.

we observe that the running time grows at an exponential rate which is compatible with the upper bound in (3.37), assuming that the cost of DC had a little impact on the algorithm. Indeed it is possible for DC to have a little impact, considering that algorithmic improvements are linear and they occur with more or less frequency depending on the edge selection order [115] and the lattice structure. For the speedup, there is improved performance for every value of p as long as $m > 4$. For $m \leq 4$, the problem is not large enough to justify parallel computation, hence the overhead from MPI makes the implementation perform poorly and sometimes even worse than the sequential version. The plot of the execution times confirms this behavior since the curves cross each other for in the transition from $m = 3$ to $m = 4$. The maximum speedup obtained was 5.7 when using $p = 8$ processors. From the lower left plot we can see that efficiency decreases as p increases, which is expected in every parallel implementation. What is important is that for large enough problems (*i.e.*, $m > 6$), efficiency is over 62% for all p. For the case of $p = 4$, we report at least 95% of efficiency, which is close to perfect linear speedup. For $m \leq 6$, the implementation is not so efficient because the amount of computation involved is not enough to keep all cores working at full capacity. The *knee* is useful for finding the optimal value of p for a balance between efficiency and computing time. It is called knee because the hint for the optimal value of p is located in

[2]In the knee, point counting is in reverse order.

the knee of the curve (thought as a leg), that is, its lower right part. In order to know the value of p suggested by the knee, one has to count the position of the closest point to the knee region, in reverse order. Our results of the knee for $m > 6$ show that the best balance of performance and efficiency is achieved with $p = 4$ (for $m \leq 6$, the knee is not effective since there was no speedup in the first place). In other words, while $p = 8$ is faster, it is not as efficient as with $p = 4$.

Kagome strip lattice test

For the test of the kagome lattice, we used 6 different strip widths in the range $m \in [2, 7]$. For each width, we measured 8 average execution times, one for each value of $p \in [1, 8]$. As a whole, we performed a total of 48 measurements for the kagome test. The standard error for each average execution time is below 5%. Additional performance measures such as speedup, efficiency and knee have also been computed. Different values of block size were tested, achieving noticeable differences on performance as B changed. We found by experimentation that $B = 1$ makes the work assignment slightly more balanced. In this test we can only use lattice axial symmetry for $m = 2, 4, 6, 8, \ldots$. For this reason we decided to run the whole kagome benchmark without axial symmetry in order to maintain a coherence between odd and even values of m.

Figure 3.12 shows the performance results for the kagome strip test. From the results

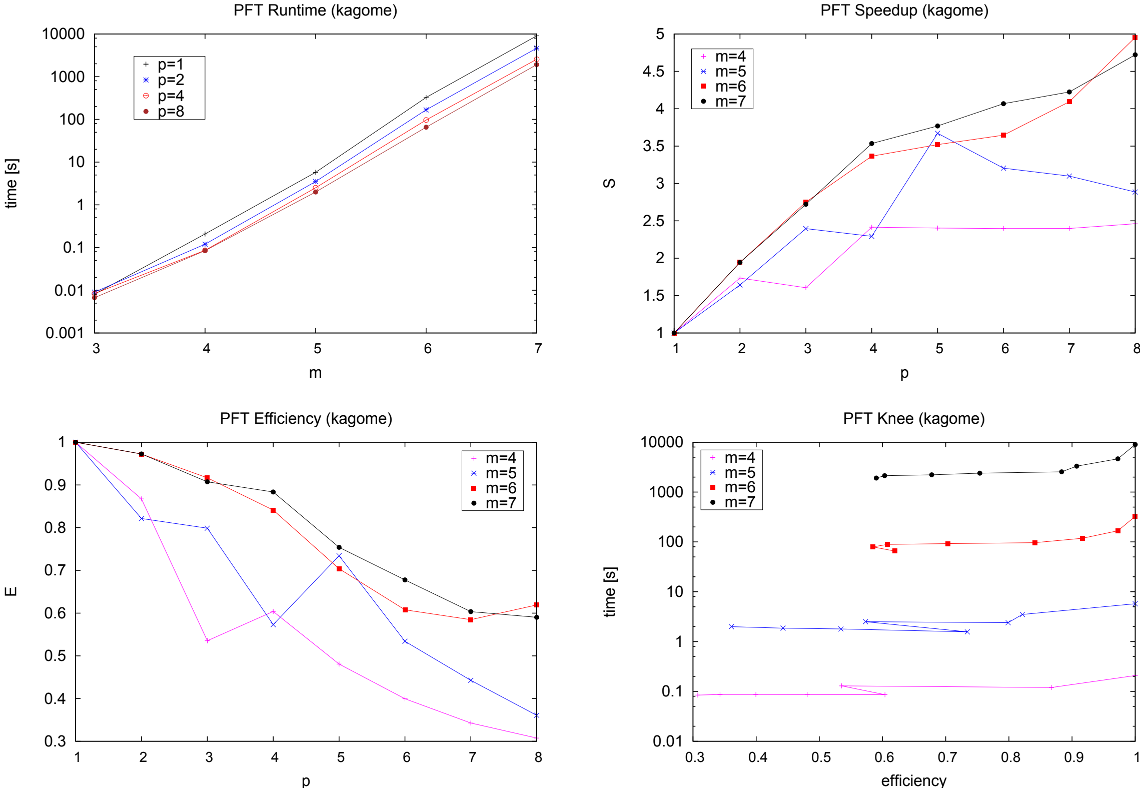

Figure 3.12: Multi-core running time, speedup, efficiency and knee for the kagome strip test.

we have that the parallel performance is still scalable even for dense layers; the maximum

speedup is over 4.7 for $p = 8$ on the largest problems. When $m > 5$, the efficiency of the parallel implementation is approximately over 60% for all values of p. In this test the knee is harder to identify, however for the largest problems one can see a small curve that suggests $p = 4$ which is in fact 90% efficient when solving large problems.

3.7.2 Cluster results

The cluster used for the tests has a total four nodes; each one with 32GB RAM and two quad-core processors Xeon 5500 2.26 GHz. The full systems offers a total of 32 processing cores and 128GB RAM. The network is Ethernet gigabit centralized and the implementation of MPI is *openMPI*.

Square results

For the test of the square strip lattice in the cluster environment, we tested 9 different strip widths in the range $m \in [2, 10]$. For each width, we measure 32 average execution times, one for each value of $p \in [1, 32]$. This process is repeated for both static and dynamic scheduling. The standard error for each average execution time is below 5%. For the dynamic scheduler we have chosen a block size value of $B = 1$. This value of B produces the highest amount of communication between the worker processes and the scheduler, hence the most dynamic scenario. Advantage of axial symmetry has also been taken.

Figure 3.13 shows the performance measures of the running time, speedup, efficiency and the *knee* [79] for the cluster environment. Note that for each color (size), the solid and dashed lines represent static and dynamic scheduling, respectively.

From the results we observe that the reduction of the running time becomes effective starting from problems of size $m \geq 6$. Speedup has an overall linear behavior for the full range $p \in [1, 32]$ which tells good scalability. Interestingly, near $p = 4$ there is a region of *super-linear speedup* [284] that occurs only for sizes $m = 6, 8$. For $p > 10$, super-linear speedup vanishes for all problem sizes. In the cluster environment, the behavior between static (solid lines) and dynamic scheduling (dashed lines) is notorious; the former behaves irregularly producing several *performance valleys*, while the latter behaves regularly, gives higher performance and produces close to zero *performance valleys*. The maximum speedup achieved is approximately $28X$ for $p = 32$, being superior in the dynamic case by a small margin. The efficiency of the parallel algorithm stays above 90% for the largest case of $m = 10$. Again, dynamic scheduler proves to be much more efficient than the static one when $m > 6$, and overall the algorithm is over 70% efficient for large enough problems, that is $m \geq 8$. The knee suggests that $p \in [8, 10]$ gives the best balance of running time and efficiency whenever $m \geq 8$.

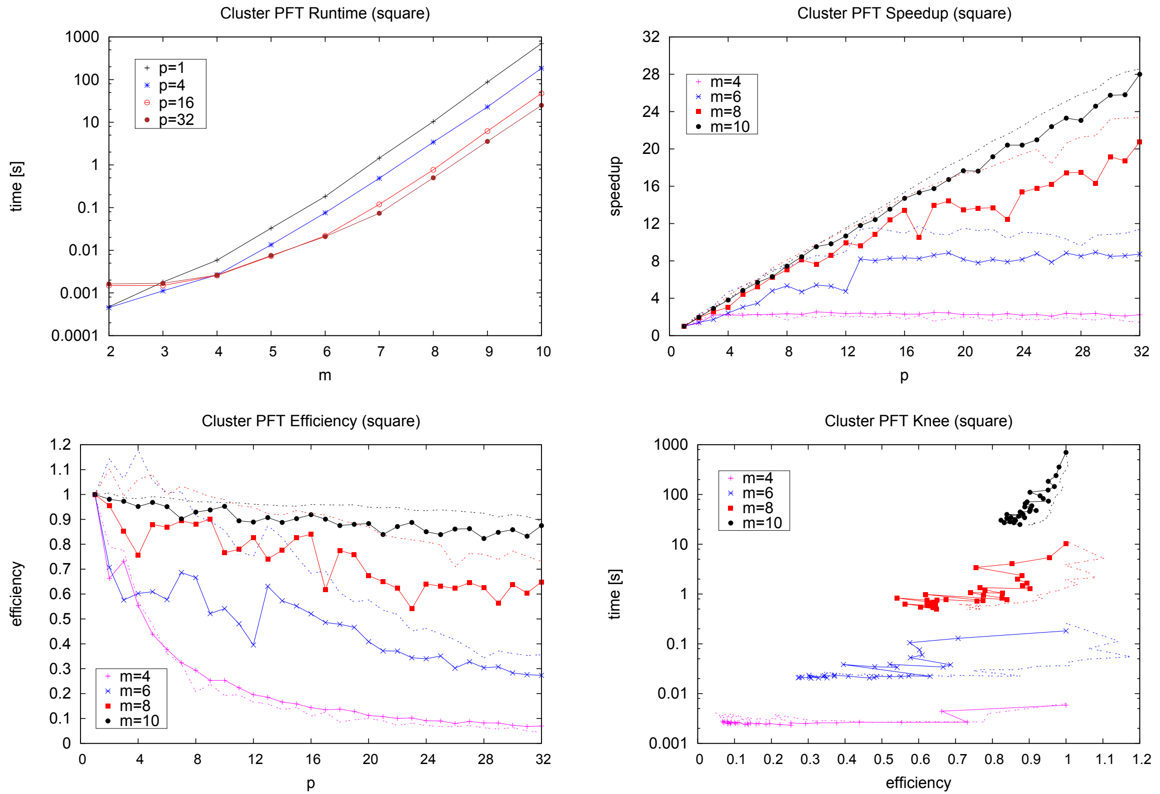

Figure 3.13: Cluster running time, speedup, efficiency and the knee for the square strip test.

Kagome results

For the test of the kagome strip lattice in the cluster environment, we tested 5 different strip widths in the range $m \in [3, 7]$. For each width, we measure 32 average execution times, one for each value of $p \in [1, 32]$. This process is repeated for both static and dynamic scheduling. The standard error for each average execution time is below 5%. For the dynamic scheduler we have chosen a block size value of $B = 1$, same as in the square cluster test.

Figure 3.13 shows the performance measures of running time, speedup, efficiency and the *knee* [79] for the cluster environment. Note that for each color (size), the solid and dashed lines represent static and dynamic scheduling, respectively.

The results show that the reduction of the running time becomes effective in a cluster as long as $m \geq 6$. In this case, speedup is closer to a logarithmic curve rather than a linear one. It is interesting to note that speedup gets stuck at specific values for sizes $m = 4, 5, 6$. The reason why is because the size of the configuration space is not large enough for cluster execution; $\Delta_m \leq 32$ for $m = 4, 5, 6$. In fact, the values of p where speedup starts to get stuck actually match the values found for $\Delta_4, \Delta_5, \Delta_6$ in Table 3.1. This phenomenon is totally normal in cluster or supercomputer environments, where the amount of work needed to reach full system occupancy is not always provided by the problem input. In order for speedup to take off, the configuration space must be equal or greater than the amount of processors available in the system.

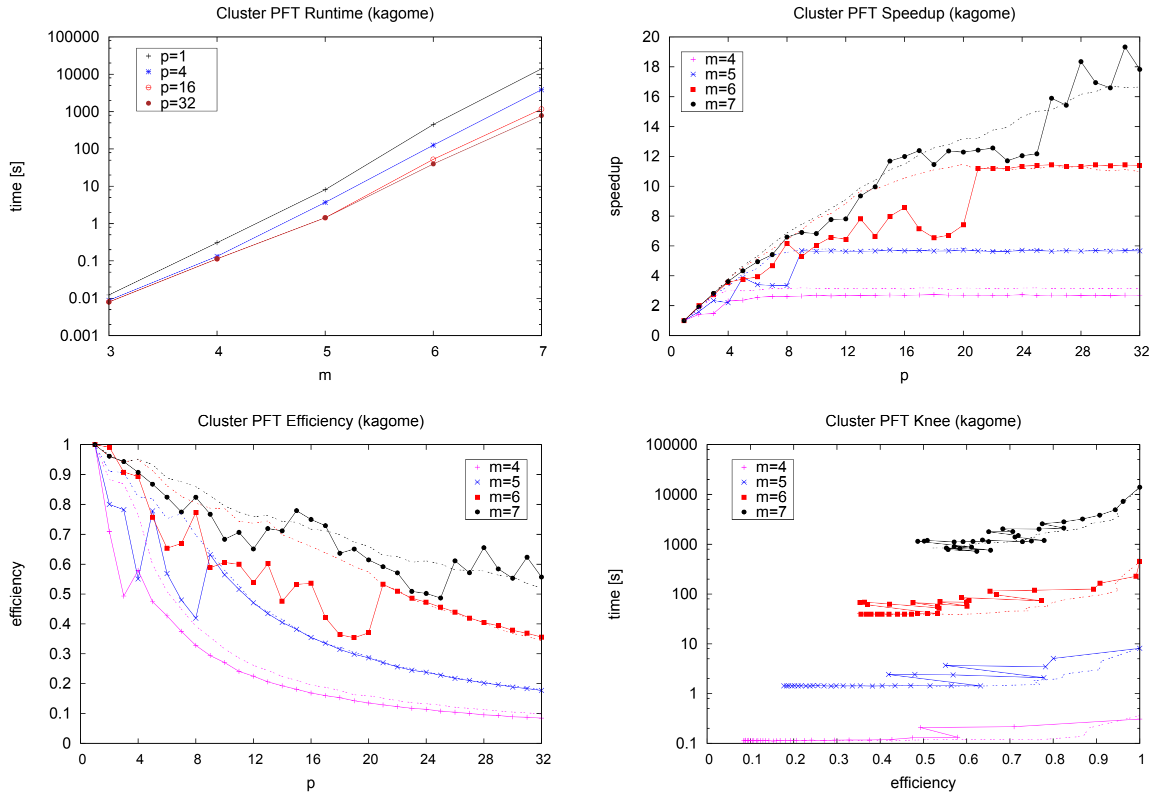

Figure 3.14: Cluster running time, speedup, efficiency and knee for the kagome strip test.

There is a notorious difference in performance between static and dynamic scheduling. With dynamic scheduling, the *performance valleys* are practically non-existent, giving a much more stable parallel performance for the full range of p. Efficiency is not as good as in the square test; the largest problem is solved with an efficiency over 55%, while the others reach below 50% at some point of p. Dynamic scheduling proves to be in average more efficient than static scheduling, by-passing the *performance valleys*. The Knee curve suggests a value $p \approx 8$ for a good balance between running time and efficiency.

3.7.3 Impact of DC on algorithm performance

We observed from the results that the running time of PFT applied to the kagome strip is slower than in the square strip. DC may cost too much in layers with a dense number of edges if optimizations do not occur too frequently. For the square lattice layer, we can write the DC worst case cost as $O(2^{2m}) - O(opt) = O(4^m - opt)$ which is one of the fastest cases we can find, and optimizations, namely $O(opt)$, appear without too much effort. If we multiply this cost by the configuration space we have that the upper bound for the time to compute the transfer matrix of the square strip is $O(3^m \times (4^m - opt)) = O(12^m - 3^m \cdot opt)$, which is a notorious improvement with respect to the $O(16^m)$ bound with the standard Catalan technique, even if no DC optimizations occur. Now for the kagome we can write the DC worst case cost as $O(2^{6m}) - O(opt) = O(64^m - opt)$ which would cost $O(3^m \times (64^m - opt)) = O(192^m - 3^m \cdot opt)$ in time when computing the matrix. For dense layers the performance depends on how good

the optimizations are and how frequently one can make them appear for a specific strip type. In our case the optimizations for kagome did not occur as frequent as in the square case because we programmed the heuristics in a very general way, nevertheless the method still managed to perform at least two times faster than the Catalan approach. It should be possible to make DC become more aware of the kagome structure and make it to generate the maximum number of optimization opportunities, as mentioned in the work of Haggard et. al. [115].

3.7.4 Performance on wider strips

We ran the PFT method to compute general (q, v) transfer matrices on square strips at $m = \{11, 12, 13\}$ and kagome strips at $m = \{8, 9\}$, using free boundary conditions and all the 32 processors we had available. Table 3.2 presents the results.

Table 3.2: Executions times for wider strips.

m	lattice	time
8	Kagome	$\sim$ 11 12 hours
9	Kagome	$\sim$ 3 months
11	Square	$\sim$ 5.5 mins
12	Square	$\sim$ 46 mins
13	Square	$\sim$ 6.7 hours

The parallel performance was not included in the performance plots because it would have required excessive amount of time to benchmark for all values of p, specially for $p = 1$ where the computation is sequential. For the kagome strip we consider that we have reached the limit of tractability and wider kagome strips would become intractable[3] with our hardware resources. For the square strip, we believe it is still possible to go further with our hardware resources, possibly up to $m = 14$, or in the best scenario up to $m = 15$ before reaching intractability. Moreover, if cylindrical boundary conditions are used, then it should be possible to go further beyond by using the symmetry of the dihedral group.

An important aspect of having a parallel solution is that if enough processors are used, that is $p = \Delta_m$, then the time for computing the transfer matrix becomes proportional to the depth of the largest directed-acyclic graph (DAG) of computation, which would correspond to the time required to solve the deepest family. The DAG concept allows one to know what to expect when having more processors (*i.e.*, a supercomputer) and gives insights on the limits of computation regarding parallelism. If we apply the DAG concept to our results, we have that the time needed to compute the TM for the square strip would have been less than 5 seconds for $m = 11$ using $p = 1142$ processors, less than 10 seconds for $m = 12$ using $p = 2947$ processors and less than 5 minutes for $m = 13$ using $p = 7889$ processors. Analogous for kagome; the time needed to compute the TM would have been between $2 \sim 3$ hours for $m = 8$ using $p = 70$ processors and ~ 1 week for $m = 9$ using $p = 323$ processors. As we

[3]We consider that a problem becomes intractable when the time it takes to be solved is in the order of years for a given computer. It is possible that a faster computer can handle the problem, making it tractable.

mentioned earlier, DC heuristics that are aware of the kagome structure should improve the performance further.

3.7.5 Comparison with related work

In this subsection we compare the *Parallel Family Trees* (PFT) strategy against the *Catalan Parallel Method* (CPM) [199] by using the improvement factor of the following metrics: (1) running time, (2) matrix evaluation time and (3) matrix space. Figure 3.15 shows the results.

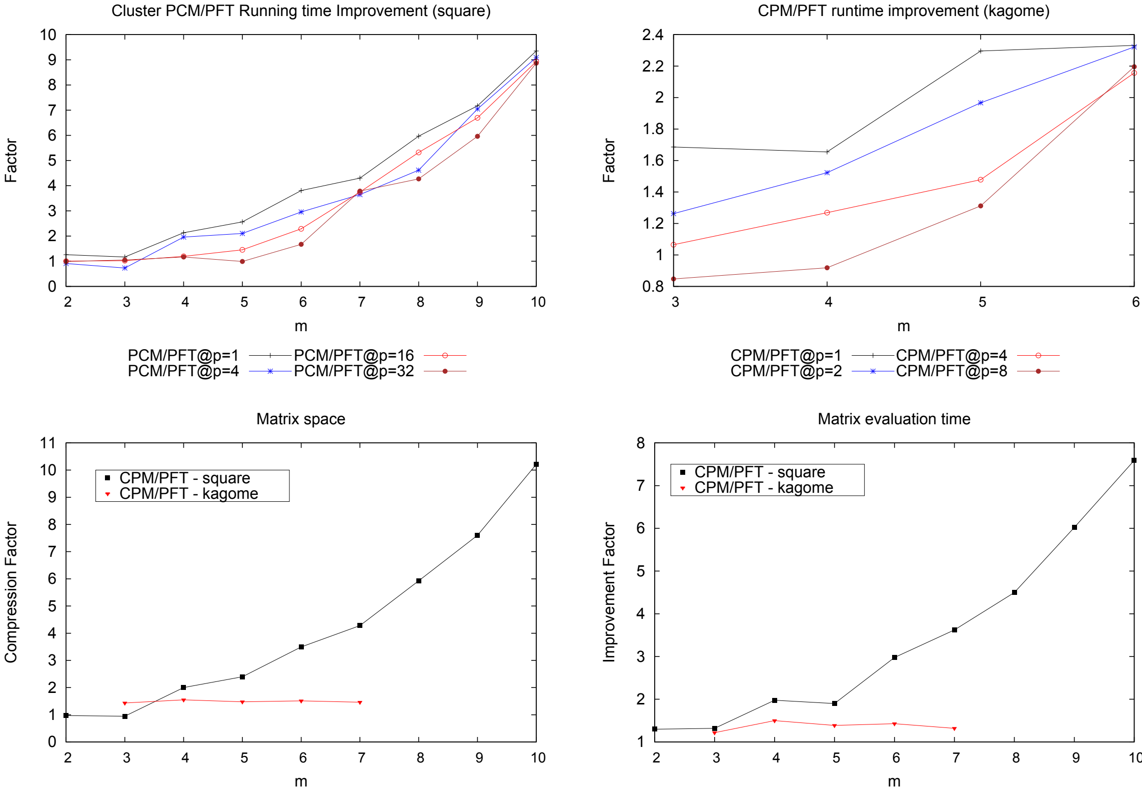

Figure 3.15: Comparison between *Parallel Family Trees (PFT)* and the *Catalan Parallel Method (CPM)*.

The first aspect to note from the running time results is that there is an non-linear improvement with respect to CPM that is independent of the amount of processors used. This improvement corresponds to the asymptotic reduction from $O(4^m)$ to $O(3^m)$ in configuration space. The improvement is less clear in the kagome strip test, but we expect that it should manifest when exploring larger sizes of m or when using better heuristics for the DC optimizations. For the space metric, we observe that the size of the compressed matrices is indeed smaller than in the CPM case. Moreover, for the square strip the amount of compression increases non-linearly as we expected from the theoretical bound. For the kagome test, the compression factor stabilizes at approximately 1.5. We believe that the reason why kagome compression stays fixed is because the kagome matrix is more sparse than in the square case, making the method to group zero-elements instead of large polynomials, reducing the

compression factor from the maximum possible if the matrix was dense. For the results of Matrix evaluation, we observe that evaluation and decompression on a PFT-matrix is faster than just evaluation on a CPM-matrix. The improvement seems to be a consequence of the compression factor achieved previously, since the behavior is similar.

3.7.6 Dynamic scheduler and block size

The role of the block size under dynamic scheduling can be viewed as the amount of *staticness* induced to the program. A value of $B = 1$ means a fully dynamic scheduler, while a value of $B = \lceil n/p \rceil$ means a fully static scheduler. Given that the dynamic scheduler of our implementation communicates via 1-byte messages, it is safe to use B as long as the network is sufficiently fast and dedicated to the cluster, like in our case. In a limited and shared network environment, one could consider exploring the range $1 < B < \lceil n/p \rceil$ until a good local minimum is found.

3.7.7 Axial Symmetry

When using axial symmetry, we observed an extra improvement in performance of up to $2X$ for the largest values of m. This improvement applies to both sequential and parallel execution. The size of the transfer matrix is improved under axial symmetry, in the best cases we achieved almost half the dimension of the original matrix, which in practice translates into up to 1/4 of the space of the original non-symmetric matrix. Lattices as the kagome will only have certain values of m where it is axial symmetric. In the other cases, one must perform a non-symmetric computation.

3.8 Validation

In this section we present some physical results we have computed for different widths of the square strip using free boundary conditions, as a way to validate the correctness of the *parallel family trees* method by comparing the curves with the ones from related works.

The first set of results are shown in Figure 3.16. In the graphics we present the limiting curves on the complex q-plane for different values of the temperature-like parameter; $v = \{-1.0, -0.5, -0.1\}$, at different strip widths in the range $m \in [2, 8]$. The case $v = -1$ corresponds to the chromatic polynomial and the crossings with the real-q axis can be interpreted as the real values of q for which no q-coloring exists. The curves were obtained by using the *direct-search approach* method which consists of scanning the complex domain in small discrete steps, and checking on each discrete location the condition $|\lambda_1| = |\lambda_2|$ where λ_1 and λ_2 are the first and second dominant eigenvalues, respectively. If the condition is true, then the pair (x, y) is a point of the curve, where x and y are the real and imaginary parts of q, respectively. Due to numerical precision limits, we allowed %1 of numerical error for accepting the condition $|\lambda_1| = |\lambda_2|$. For the case of $v = -0.5$ we allowed up to %4 of error

for drawing the limiting curve at size $m = 8$. The curves for $v = -1$ agree with the ones

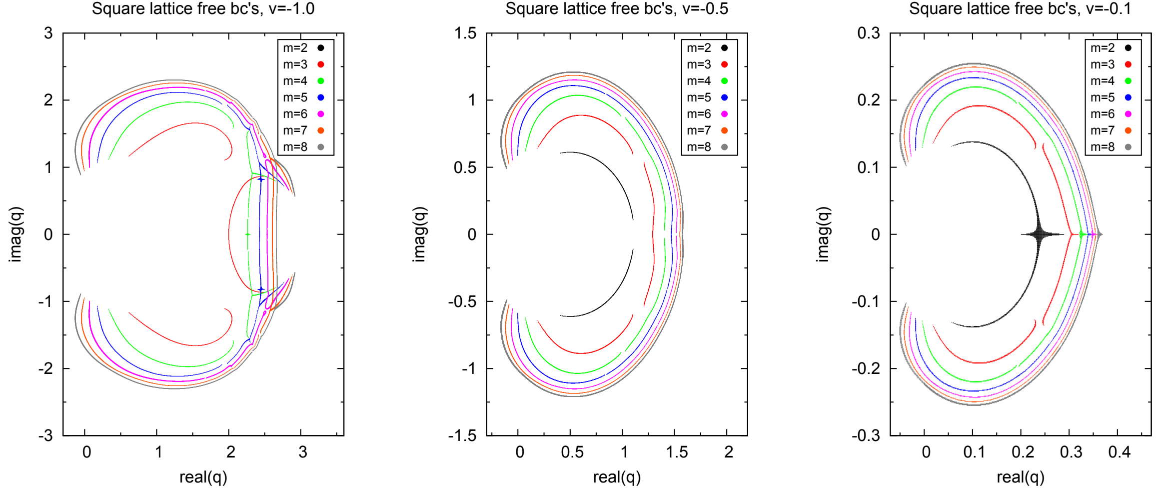

Figure 3.16: Limiting curves on the complex $q-$plane for $v = \{-1.0, -0.5, -0.1\}$. In each graphic there are seven limiting curves with different colors, each one corresponding to a different strip width.

presented by Salas *et. al.* in Figure 21 of ref. [241]. The curves for $v = -0.5$ and $v = -0.1$, although grouped in a different way, agree with the result obtained by Chang *et. al.* from Figures 2, 3, 4 of ref. [47]. Limiting curves for $6 \le m \le 8$ did not appear in the cited work.

For the next set of physical results we are interested in fixing the q parameter at values $q = \{2, 3, 4\}$ and compute the dimensionless reduced internal energy E_r as well as the reduced function C_H of the specific heat C, for different strip widths in the range $m \in [2, 8]$. The dimensionless reduced internal energy is defined as

$$E_r = -\frac{E}{J} = (v+1)\frac{\partial f}{\partial v} \tag{3.45}$$

where f is the free energy density as defined in equation (2.17), J the coupling constant which is $J > 0$ for the *ferromagnetic* case ($0 < v < \infty$) and $J < 0$ for the *antiferromagnetic* case ($-1 < v < 0$). The specific heat is defined as

$$C = \frac{\partial E}{\partial T} = k_B K^2 (v+1)\left[\frac{\partial f}{\partial v} + (v+1)\frac{\partial^2 f}{\partial v^2}\right] \tag{3.46}$$

and C_H uses the reduced form

$$C_H = \frac{C}{k_B K} \tag{3.47}$$

The results are presented in Figure 3.17, where each row presents the results for a given q value. The curves for $2 \le m \le 5$ agree with the ones presented by Chang *et. al.* [47]. Results for $6 \le m \le 8$ did not appear in the cited work.

Although the computation of new physical curves for wider strips is indeed possible, it would require more time with our resources, or a much larger cluster than ours for faster results. Nevertheless, our present results already show that with the *PFT* strategy known results are obtained faster than with CPM. We would like to remind the reader that the focus of this work is on the algorithmic improvements and the possibilities to compute the general (q, v) transfer matrix for strips, using a configuration space that is asymptotically $O(3^m)$.

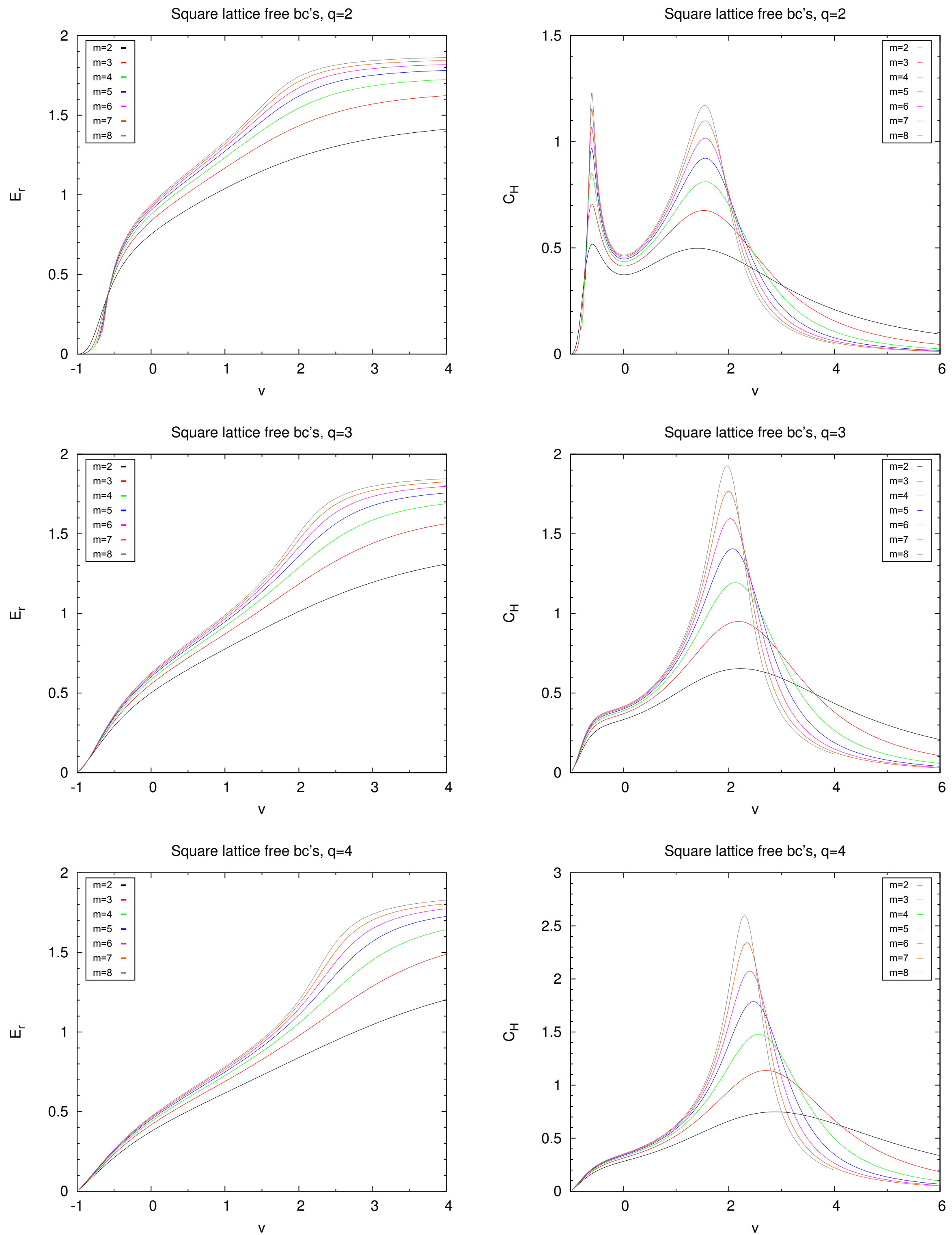

Figure 3.17: Plots for reduced internal energy E_r and reduced specific heat C_H for $q = \{2, 3, 4\}$.

3.9 Discussion

We have presented a parallel strategy for computing the general (q, v) transfer matrix of strip lattices in the Potts model. Our main result is the asymptotic reduction of the configuration space, from $O(4^m)$ to $O(3^m)$, by re-organizing the problem domain as *parallel family trees (PFT)*. Using this strategy, the transfer matrix can now be computed by just processing the root configurations, which are $O(3^m)$ in number. Computation of the family trees can be performed completely in parallel because family trees are independent from each other, and the configuration space is generated *a priori*, removing any potential time-dependence. We have compared the experimental results of PFT show evidence of the exponential advantage over the *Catalan Parallel Method (CPM)* [199], both in sequential and parallel execution.

The resulting matrix of PFT is a compressed structure based on systems of linear equations. Numerical evaluation on the matrix, including decompression time, is actually faster than numerical evaluation using the CPM method, by a factor that is proportional to the improvement we measured for running time. Therefore, it is not only faster to generate the matrix using PFT, but it is also faster to use it later for extracting the physical information.

Multi-core results have shown that PFT benefits from shared-memory parallelism, achieving a maximum of 5.7X of speedup for the square strip test when using $p = 8$ processors. At $p = 4$, the efficiency of the implementation is still over 95%, which is worth mentioning. By plotting the *knee* curve, we have managed to confirm that choosing $p = 4$ is in fact a wise decision for a balance of speed and efficiency. In the Multi-core scenario, a dynamic scheduler did not produce a beneficial change in performance, therefore static scheduling still remains convenient.

For the cluster results, we achieved up to $28X$ of speedup using $p = 32$ for the square strip tests, with an efficiency above 90% for a strip of width $m = 10$ (largest one). For the kagome strip test, efficiency stayed above 55% for a strip of $m \geq 7$ and the maximum value of speedup reached was close to $20X$ when using $p = 31$. A small *super-linear speedup* region emerged near $p = 4$ when solving square strips of sizes $m = 6, 8$, giving an efficiency of up to 120%. We believe that this is just a particular fortunate event, possibly produced by the reduction of cache misses, which is caused when partitioned data fits entirely in cache. In general, we do not expect *super-linear* behavior since we are measuring *fixed-size speedup* which is upper bounded as $S_p \leq p$ [112]. The knee curve suggests that $p \in [8, 10]$ produces a good balance between speed and efficiency. An important result in cluster execution is that dynamic scheduling is mandatory in order to achieve a performance curve that will not fall into *performance valleys*, as static scheduling did. On average, dynamic scheduling achieves considerable higher performance than static scheduling.

One of the goals of this work was to present an algorithmic improvement that is implicitly parallel and scalable. For this, we introduced a preprocessing step that generates all possible *root configurations* and *Catalan configurations*, which are critical for processing the family trees in parallel. This step takes a small amount of time compared to the whole problem. Other technical improvements had been introduced, some of them being already known in the literature [115]; (1) fast computation of serial and parallel paths of the graph, (2) exploiting axial symmetry, (3) a set of algebra rules for making consistent keys in all leaf nodes and

(4) a hash table for accessing column values of the transfer matrix. In particular, when taking advantage of axial symmetry, the implementation achieved extra improvement of up to $2X$ in performance, using almost a quarter of the matrix space used in a non-symmetric computation.

In order to achieve a scalable parallel implementation, some small data structures were replicated among processors while some other data structures per processor were created within the corresponding worker process context, not in any master process. This allocation strategy results in faster cache performance and brings up the possibility to scale better under NUMA architectures. It is not a problem to store the matrix fragmented into many files as long as the matrix is in its symbolic form. In practice, it is first necessary to evaluate the matrix on q and v before doing any further numerical analysis. Therefore, the fragmented parts can be evaluated at runtime as they become read. This evaluation can also be done in parallel.

The only technical restriction of the *parallel family trees* strategy in order to work is that vertices of the left and right boundaries of the layer need to be connected sequentially. This restriction is not a problem, because any planar strip lattice can be rotated so that the restriction is satisfied. Additionally, PFT allows any graph structure along the vertical direction, that is, one can study strips where its K_i layer is composed by a sequence of different tiles.

In the kagome tests, the performance results were not as good as we expected, because the number of edges in the layer is much higher than in the square case, making DC to take a considerable amount of time for each configuration. We believe that the dependence of DC on the number of edges in the layer is a sensible aspect for the PFT algorithm, and an extrapolation of this situation would suggest that the largest Archimedean lattices could be much harder to the point of being intractable. However, it is important to consider that DC can significantly improve its performance if the heuristics are improved so that they choose the best sequence of edges based on the connectivity of the graph layer [115]. These heuristics, combined with the linear-cost optimizations, can make the PFT method more resistant to the number of edges in the layer. Furthermore, if more processors are used to the point that $p = \Delta_m$, then the time for computing the TM will be much lower than in our case with $p = 32$, and will correspond to the time taken to solve the deepest DAG of computation. For this reason, we expect that an execution on a large cluster or supercomputer could allow the computation of transfer matrices of strips wider than what has been reached before.

In the next Chapter, we present the second and last contribution of the thesis, which consists of a multi-GPU Monte Carlo method for disordered systems.

Chapter 4

Adaptive Multi-GPU Exchange Monte Carlo for the 3D Random Field Ising Model

The study of disordered spin systems through Monte Carlo simulations has proven to be a hard task due to the adverse energy landscape present at the low temperature regime, making it difficult for the simulation to escape from a local minimum. Replica based algorithms such as the *Exchange Monte Carlo* (also known as *parallel tempering*) method have proven to be effective at overcoming this problem, reaching equilibrium on disordered spin systems such as the Spin Glass or Random Field models, by exchanging information between replicas of neighbor temperatures. In this Chapter we present a *multi-GPU Exchange Monte Carlo* method designed for the simulation of the 3D Random Field Model. The implementation is based on a two-level parallelization scheme that allows the method to scale its performance in the presence of faster and GPUs as well as multiple GPUs. In addition, we modified the original algorithm by adapting the set of temperatures according to the exchange rate observed from short trial runs, leading to an increased exchange rate at zones where the exchange process is sporadic. Experimental results show that parallel tempering is an ideal strategy for being implemented on the GPU, and runs between one to two orders of magnitude with respect to a single-core CPU version, with multi-GPU scaling being approximately 99% efficient. The results obtained extend the possibilities of simulation to sizes of $L = 32, 64$ for a workstation with two GPUs.

4.1 Introduction

Monte Carlo methods have become a convenient strategy for simulating finite size spin lattices towards equilibrium and to perform average measurements of physical observables. Classic spin models such as Ising [55, 133] and Potts [226] are usually simulated with the Metropolis-Hastings algorithm [123, 193], cluster [14, 262, 287] or worm algorithm [230], with the last two being more efficient reaching equilibrium near the critical temperature T_c [253].

For systems with *quenched disorder* such as the Spin Glass and the Random Field Ising Model (RFIM) we can find that classic algorithms are not efficient anymore in the low temperature regime. In this work we are interested in studying the 3D RFIM, which introduces a disordered magnetic field $\{h_1, h_2, \cdots, h_n\}$ of strength h, to the Hamiltonian:

$$\mathcal{H} = -\sum_{\langle \mathrm{i},j \rangle} J s_{\mathrm{i}} s_j - h \sum_{\mathrm{i}} h_{\mathrm{i}} s_{\mathrm{i}} \tag{4.1}$$

The reason why classic MCMC algorithms fail is because systems with quenched disorder present an adverse energy landscape, making the simulation with a classic MCMC algorithm to easily become trapped in a local minimum, thus never reaching the ground state of the system. Figure 4.1 illustrates the problem. The problem occurs at low temperatures, *i.e.*,

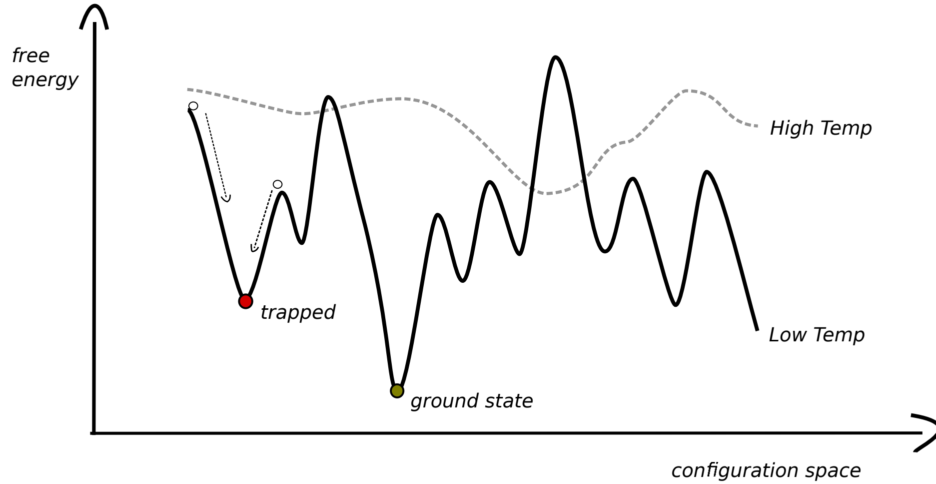

Figure 4.1: At low temperature, classic algorithms fail to reach the ground state for systems with quenched disorder.

$T \leq T_c$, which is a required zone to explore if one is studying phase transitions. Instead of simulating the system with one instance of the lattice, as in classic algorithms, one can simulate R replicas at different temperatures and exchange information among them, to eventually overcome the local minimum problem [130, 261]. By exchanging information between replicas from time to time, information from the high temperature regime arrives to the replicas at low temperature, *shaking* the system and providing the opportunity to escape the local minimum. The algorithm based on this principle is known as the *Exchange Monte Carlo* method, also as *Parallel Tempering*, and it is one of the most used algorithms for simulating systems with quenched disorder. The notion of exchanging replicas for studying systems with quenched disorder was first introduced by Swendsen and Wang in 1986 [261] and then extended by Geyer in 1991 [97]. Hukushima and Nemoto presented in 1996 the full method as it is known today [130].

The replica based approach has been used to simulate 2D and 3D spin glasses [80, 130, 151, 211, 233]. While it is true that replica based methods overcome the main difficulty of the Monte Carlo simulation in disordered systems, the computational cost is still considered a problem, since it requires at least $\Omega(R \cdot L^{\mathrm{d}})$ per time step, with L being the linear size of the lattice and d the number of dimensions. Furthermore, the number of simulation steps required to reach the ground state is in the order of millions, and increases as the lattice

gets larger. In this work we are interested in the particular case of simulating the 3D Ising Random Field model at sizes $L = \{8, 16, 32, 64\}$ using the Exchange Monte Carlo method adapted to massively parallel architectures such as the GPU.

The fast expansion of *massively parallel architectures* [200] provides an opportunity to further improve the running time of data-parallel algorithms [6, 7, 184]; *Parallel Tempering* (PT) in this case. The GPU architecture, being a *massively parallel architecture*, can easily perform an order of magnitude faster than a CPU. Moreover, the GPU is more energy efficient and cheaper than classic clusters and supercomputers based on CPU hardware. But in order to achieve such level of performance, GPU-based algorithms need to be carefully designed and implemented, presenting a great challenge on the computational side. This computational challenge is what motivates our work.

In this work, we propose a multi-GPU method for modern GPU architectures for the simulation of the *3D Random Field Ising Model*. The implementation uses two levels of parallelism; (1) *spin parallelism* that scales in the presence of faster GPUs, and (2) *replica parallelism* that scales in the presence of multiple GPUs. Both levels, when combined together, provide a substantial boost in performance that allows the study of problems that were too large in the past for a conventional CPU implementation, such as $L = \{8, 16, 32, 64\}$. In addition to the parallelization strategy, we also propose a new temperature selection scheme, based on recursive insertion of points, to improve the exchange rate at the zones where exchange is often less frequent. Physical results have been included for the 3D random field model at sizes $L = \{8, 16, 32, 64\}$.

The rest of the Chapter is organized as follows: Section 4.2 presents the related work regarding parallel implementations of the Exchange Monte Carlo method. In Section 4.3 we point out the levels of parallelism present in the Exchange Monte Carlo method, and explain the multi-GPU method in its two levels. Section 4.4 presents the *Adaptive Temperature* strategy we use for choosing the temperature distribution and show how it can improve the results of the simulation as well as reduce simulation time. In Section 4.5 we show the experimental performance results, which consist of comparison against a CPU implementation, performance scaling under different GPUs as well as under one and two GPUs, and results on the improvement provided by the adaptation technique, compared to a simulation without the approach. Section 4.6 we present the exchange rate of the adaptive strategy and compare it to other homogeneous approaches. In Section 4.7 we present some physical results on the 3D Ising Random Field, for sizes $L = \{8, 16, 32, 64\}$. Finally, in Section 4.8 we discuss our results and conclude our work.

4.2 Related Work

Several works have shown the benefits of GPU-based implementations of MCMC algorithms for spin systems. The Metropolis-Hastings algorithm has been efficiently re-designed as a GPU algorithm for both 2D and 3D lattices [31, 86, 180, 228]. The parallelization strategy is usually based on the *checkerboard* decomposition of the problem domain, where black and white spins are simulated in a two-step parallel computation. Although the checkerboard

method violates *detailed balance*, it still obeys the *global balance* condition which is sufficient to ensure convergence of the stochastic process. M. Weigel proposed the *double checkerboard* strategy (see Figure 4.3), that takes advantage of the GPU's shared memory [278, 279, 280] for doing partial Metropolis sweeps entirely in cache. In the work of Lulli *et. al.*, the authors propose to reorganize the lattice in alternating slices in order to achieve efficient memory access patterns.

For cluster algorithms, recent works have proposed single and multiple GPU implementations, for both Ising and Potts models [164, 165, 166, 275]. For the case of the Swendsen and Wang algorithm, which is a multi-cluster one, some use a parallel labeling strategy based on the work of Hawick *et. al.* [124]. The cluster work of Weigel uses an approach based on self-labeling with hierarchical sewing and label relaxation [275]. A study on the parallelism of the Worm algorithm has been reported by Delgado et al. [70]. The authors conclude that an efficient GPU parallelization is indeed hard because very few worms stay alive at a given time.

Two GPU-based implementations of the Exchange Monte Carlo method have been proposed. The first one was proposed by Weigel for 2D Spin Glasses [279], in which the author treats all the replicas of the system as one large lattice, therefore additional replicas in practice turn out to be additional thread blocks. The second work is by Ye Fang *et. al* [82] and they propose a fast multi-GPU implementation for studying the 3D Spin Glass. In their work, the authors propose to keep the replicas in shared memory instead of global memory. This modification provides a performance memory accesses that are an order of magnitude faster than global memory ones, but limits the lattice size to the size of the shared memory, which for todays GPUs it means 3D lattices of size $L \leq 16$. Katzgraber *et. al.* proposed a method for improving the temperature set in the Exchange Monte Carlo method [152]. The strategy is based on keeping a histogram record of the number of round trips of each replica (*i.e.*, the number of times a replica travels from T_{min} to T_{max} and vice versa), and improving the locations where this value is the lowest. Another strategy was presented by Bittner *et. al.*, where they propose a method for obtaining a good set of temperatures and also they propose to set the number of lattice sweeps according to the auto-correlation time observed [27].

To the best of our knowledge, there is still room for additional improvements regarding GPU implementations for the Exchange Monte Carlo method, such as using concurrent kernel launches, extending the double checkerboard strategy to 3D, optimal 3D thread blocks, global-memory multi-GPU partitions, and low level optimizations for the case of the 3D Random Field model, among others. In relation to choosing the temperature set, it is still possible to explore different adaptive strategies based on recursive algorithms. In the next section we present our parallel implementation of the Exchange Monte Carlo method as well as our strategy for choosing an efficient temperature set.

4.3 Multi-GPU approach

For a multi-GPU approach, we analyze the *Exchange Monte Carlo* to find out how many levels of parallelism exist.

4.3.1 Parallelism in the Exchange Monte Carlo method

The *Exchange Monte Carlo* method is an algorithm for simulating systems with *quenched disorder*. The notion of exchanging replicas was first introduced by Swendsen and Wang in 1986 [261], later extended by Geyer in 1991 [97]. In 1996, Hukushima and Nemoto formulated the algorithm as is it known today [130]. The algorithm has become widely known for its efficiency at simulating Spin Glasses, and for its simplicity in its definition. Due to its general definition, the algorithm can be applied with no difficulties to other models different from the spin glass model such as the Random Field Ising Model (RFIM), which is the model of study in this work.

The algorithm works with M *replicas* $\mathcal{X} = \{X_1, X_2, ..., X_M\}$ of the system with each one at a different temperature. The main steps for one disorder realization of our parallel *Exchange Monte Carlo* algorithm, for the case of the Ising Random Field model, are the following:

1. Choose M different temperatures $\{\beta_1, \beta_2, ..., \beta_M\}$ where $\beta = 1/T$.
2. Choose an arbitrary random magnetic field $H = \{h_1, h_2, h_3..., h_{|V|}\}$ with $h_{\mathrm{i}} = rand(\pm 1)$. This instance H of disorder is used for the entire simulation by all M replicas.
3. Set an arbitrary spin configuration to each one of the M replicas and assign the corresponding temperature, *i.e.*, $X_{\mathrm{i}} \leftarrow \beta_{\mathrm{i}}$.
4. [**Parallel**] Simulate each replica simultaneously and independently in the Random Field Model for p parallel tempering moves, using H for all replicas and a highly parallel MCMC algorithm such as Metropolis-Hastings. At each parallel tempering move, exchange the *odd* ***xor*** *even* configurations X_{i} with their next neighbor $X_{\mathrm{i}+1}$, with probability
$$W(X_{\mathrm{i}}, \beta_{\mathrm{i}} | X_{\mathrm{i}+1}, \beta_{\mathrm{i}+1}) = \begin{cases} 1 & \text{for } \Delta < 0 \\ e^{\Delta} & \text{for } \Delta > 0 \end{cases} \tag{4.2}$$
where $\Delta = (\beta_{\mathrm{i}+1} - \beta_{\mathrm{i}})(\mathcal{H}(X_{\mathrm{i}+1}) - \mathcal{H}(X_{\mathrm{i}}))$. Choosing odd or even depends if the j-th exchange is odd or even, respectively.

The algorithm itself is inherently *data-parallel* for step (4) and provides a sufficient number of data elements for a GPU implementation. In fact, there are two levels of parallelism that can be exploited; (1) *spin parallelism* and (2) *replica level parallelism*. In *spin parallelism* the challenge is to come up with a classic MCMC method that can take full advantage of the GPU parallel power. For this, we use a GPU-based Metropolis-Hastings implementation optimized for 3D lattices. For (2), the problem seems *pleasingly parallel* for a multi-GPU implementation, however special care must be put at the exchange phase, since there is a potential memory bottleneck due to the distributed memory for a multi-GPU approach. Figure 4.2 illustrates the two levels of parallelism and their organization.

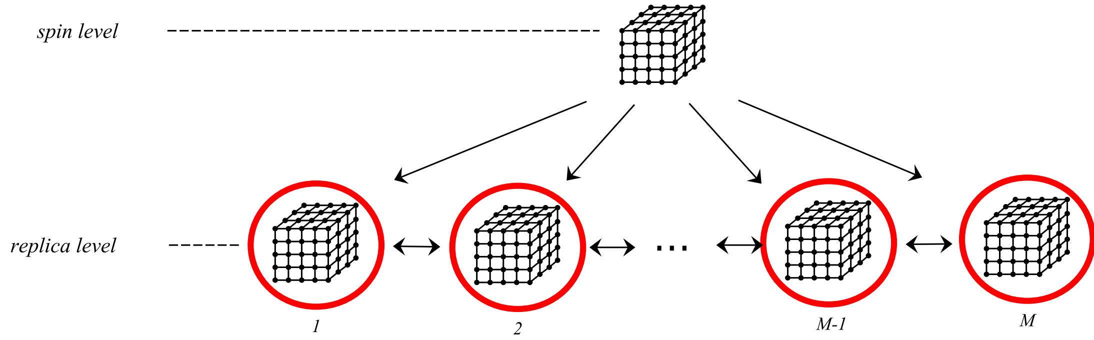

Figure 4.2: The two levels of parallelization available in the exchange Monte Carlo method.

4.3.2 Spin Level Parallelism

Spin parallelism corresponds to the parallel simulation of the spins of a single replica and we propose to handle it by using a single CUDA kernel based on the *double checkerboard* idea, proposed by M. Weigel [279, 280]. A *double checkerboard* approach allows the efficient simulation of spin systems using coalesced memory accesses, as well as the option to choose multiple partial sweeps in the same kernel at a much higher performance because of the GPU's shared memory. In the original works by Weigel, the optimizations are only described and implemented for the 2D Ising Spin Glass [279, 280], but the author mentions that the ideas can be extended to 3D by using a more elaborate thread indexing scheme. The *spin parallelization* implementation of this work is a *double 3D checkerboard* and corresponds to the extension mentioned by the author.

A *double checkerboard* is a *two-fold* Metropolis-Hastings simulation strategy that is composed of several fine grained checkerboards organized into one coarse checkerboard. The case of 2D is illustrated in Figure 4.3.

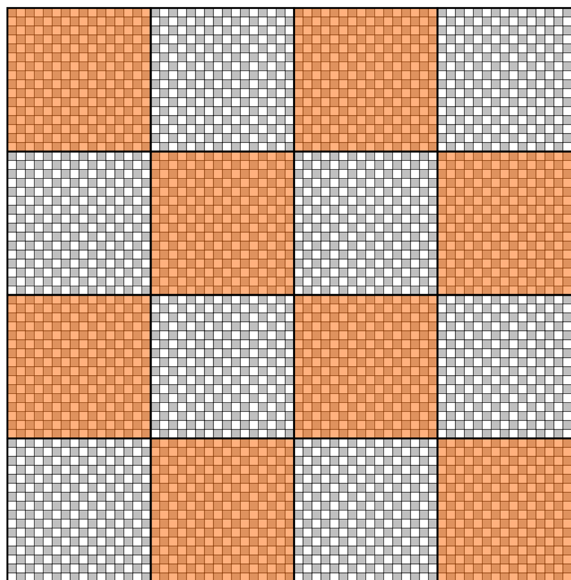

Figure 4.3: A double 2D checkerboard of $D \times D = 4 \times 4$ tiles, each one containing $T \times T = 16 \times 16$ spins.

The double checkerboard method starts by simulating all spins from the orange tiles of $T \times T$ spins, doing a local two-step *black-white* simulation on the fine grained checkerboard of $T \times T$. The tile sub-checkerboard is loaded entirely in GPU shared memory, including a halo of spins that reside on the neighbor tiles of the opposite color. After all orange tiles are finished, the same is done for the white tiles. Two kernel executions are needed to fully synchronize all threads when changing tile color. While it is true that only $L^3/4$ spins are being simulated at a given time, its advantage is that the memory access pattern is fully coalesced and the spin flip is performed in shared memory.

In order to create a *double 3D checkerboard*, we convert both the fine and coarse grained checkerboards to 3D. For this, we use the fact that a 3D checkerboard can be generated by using alternated 2D checkerboard layers stacked over a third dimension. For a given point $p = (x, y, z)$ in 3D discrete space, its alternation value $A = \{0, 1\}$ is defined as

$$A = (p_x + p_y + p_z) \mod 2 \tag{4.3}$$

Indeed one could build a *space of computation* of the size of the whole lattice, launch the kernel and compute the value of A using expression (4.3), but this approach would be inefficient because $3L^3/4$ threads would remain idle waiting for its turn in the checkerboard process. Instead, we use a space of computation composed of $L^3/4$ threads; $T^3/2$ threads per block and $D^3/2$ blocks, where D is the number of tiles per dimension. Figure 4.4 illustrates a space of computation of $L^3/4$ threads being sufficient for handling a double 3D checkerboard.

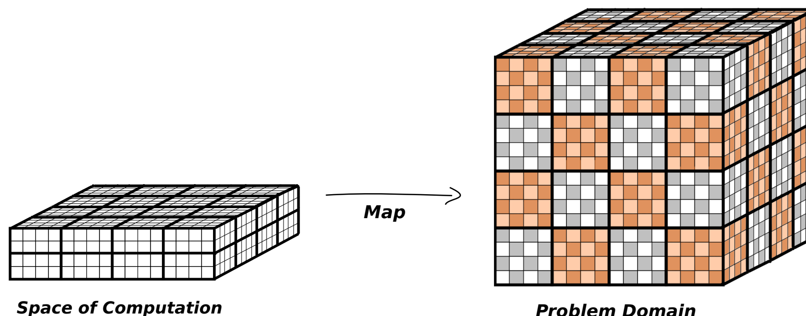

Figure 4.4: A space of computation of size $L^3/4$ threads is necessary and sufficient for simulating the spins in parallel using a double 3D checkerboard approach.

For the halos it is important to consider what percentage of the spins loaded into the GPU shared memory will actually be halo spins. Halo spins are more expensive to load into shared memory than internal tile spins due to the unaligned memory access pattern, therefore one would want the minimum number of halo spins in the shared memory. Considering that actual GPU architectures have a constant warp of threads w and require a constant block volume V to be specified, finding the minimum halo is equivalent to solve the optimization problem of minimizing the surface of a closed box with dimensions $w \times x \times y$ and constant volume V. The objective function to minimize is the surface expression

$$S = 2(wy + xy + wx) \tag{4.4}$$

subject to $V = wxy$, from where we can rewrite S in one variable. Setting the derivative of S to zero leads to

$$\frac{\partial S}{\partial x} = 2(w - V/x^2) = 0 \tag{4.5}$$

where we finally get the solution: $x = y = \sqrt{V/w}$. Considering that the number of spins inside a tile is double the number of threads (because of the checkerboard approach) and that the maximum number of threads in a block is $B_{max} = 1024$ for actual GPUs, we have that $V = 2|B_{max}|$. With this, $x = y = 8$ and the optimal block to use is $B_{opt} = (32, 4, 8)$.

4.3.3 Replica Level Parallelism

Replica level parallelism is based on the combination of concurrent kernel execution from modern GPUs and coarse parallelism from the multi-GPU computing. In modern GPU architectures, one can launch multiple kernels in the same GPU and let the driver scheduler handle the physical resources to execute these kernels concurrently for that GPU. Starting from the Kepler GPU architecture, it is possible to launch up to 32 kernels concurrently on a single GPU. The idea is to divide the M replicas into k available GPUs and simulate $m = \lceil M/k \rceil$ replicas concurrently for each GPU. It is possible that $m > 32$, but it is not a problem because the GPU can handle the exceeding kernels automatically with very small overhead, by using an internal execution queue. With the new approach, the new number of replicas becomes

$$M' = D \cdot m \leq M + D \tag{4.6}$$

where D is the number of GPUs used in the multi-GPU computation. By using M' replicas instead of M, we guarantee a balanced parallel computation for all GPUs and at the same time the extra replicas help to produce a better result. One assumes that $D \leq M$.

In a multi-GPU approach, the shared memory scenario transforms into a distributed scenario that must be handled using global indexing of the local memory regions. For each GPU, there is a region of memory allocated for the m replicas. For any GPU D_{i}, the global index for its *left-most* replica is $D_{\mathrm{i}}^L = m \cdot \mathrm{i}$ while the global index for its *right-most* replica is $D_{\mathrm{i}}^R = m \cdot \mathrm{i} + m - 1 = m(\mathrm{i}+1) - 1$.

Replica exchanges that occur in the same GPU are efficient since the swap can just be an exchange of pointers. In the limit cases at D_{i}^L and D_{i}^R, the task is not local to a single GPU anymore, since the exchange process would need to access replicas $D_{\mathrm{i}-1}^R$ and $D_{\mathrm{i}+1}^L$, both which reside in different GPUs. Because of this special case, the pointer approach is not a robust implementation technique for a multi-GPU approach, neither explicit spin exchanges because it would require several memory transfers from one GPU to another. The solution to this problem is to swap temperatures, which is totally equivalent for the result of the simulation. One just needs to keep track of which replica has a given temperature and know their neighbors. For this we use two index arrays, trs and rts, for *temperatures replica sorted* and *replica temperature sorted*, respectively. Figure 4.5 illustrates the whole multi-GPU approach.

The temperature swap approach is an efficient technique for single as well as for clusters of GPU-based workstations, since for each exchange only two floating point values (*i.e.*, $2 \times 32 bits$) need to be swapped. This part of the algorithm results in little overhead compared to the simulation time, thus we opted to implement it on the CPU side.

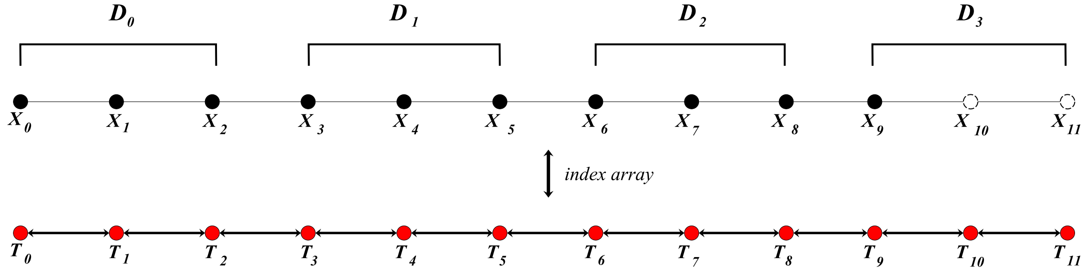

Figure 4.5: The multi-GPU version of replica level parallelism combined with temperature swapping.

4.4 Adaptive Temperatures

The random field, as any other system with quenched disorder, becomes difficult to study as L increases. For $L \geq 64$ the selection of parameters is already a complex task, since a small change on one can lead to a bad quality simulation.

One important parameter for simulation is the selection of temperatures and the distance among them. In general, for the low-temperature regime and near the transition point T_c, the replicas need to be placed at temperatures much closer compared to the high temperature regime in order to achieve an acceptable exchange rate. If these temperature requirements are not met, the simulation may suffer from exchange bottlenecks, preventing the information to travel from one side to other. Indeed one can decide to simulate with a dense number of temperatures for the whole range, but this strategy tends to be inefficient because the simulation does much more work than it should. Based on this facts, we propose to use an adaptive method that builds, incrementally, a good distribution of temperatures based on the exchange rates needed at each region. Two immediate advantages of adding temperatures is that (1) the initial adaptation phase costs significantly less compared to an approach that adapts the final number of temperatures from the beginning, and (2) we do not generate exchange holes since we just insert temperatures.

The idea is to start with M replicas, equally distributed from T_{low} to T_{high}. The adaptive method performs an arbitrary number of trial simulations to measure the exchange rate between each consecutive pair of replicas. After each trial simulation, an array of exchange rates is obtained and put in a min-heap. For the a intervals with lowest exchange rate (with a chosen arbitrarily), new replicas are introduced using its middle value as the new temperature.

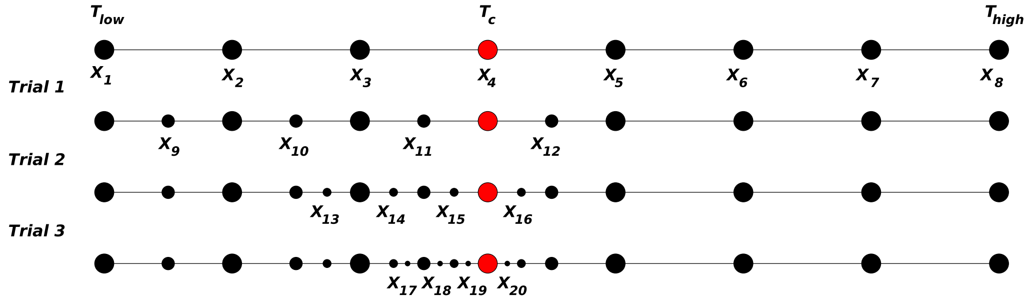

Figure 4.6: The adaptive temperature strategy choosing the lowest four exchange rates for three trial runs.

The trial simulations use multi-GPU computation. Given that the number of trial runs and the number of intervals to split is given a priori, the memory pool can be allocated before hand and equally distributed among all GPUs. After each trial run, the new i-th new replica is assigned to the j-th GPU with $j = \mathrm{i} \mod k$, where k is the number of GPUs. The order in which new replicas are assigned to each GPU actually does not affect the performance neither the result of the simulation.

4.5 Performance results

In this section we present the performance results of our multi-GPU implementation (which we have named *trueke* for *exchange* in Spanish, which is *trueque*) and compare them against other GPU and CPU implementations. The purpose of including a comparison against a sequential CPU implementation is not to claim very high speedups, but rather to provide a simple reference point for future comparisons by the community.

4.5.1 Benchmark Plan

Four performance metrics, averaged over N repetitions, are used to measure the parallel performance of *trueke* (Note, in this section T means running time, not temperature):

1. Benchmark the **Spin-level Performance** by computing the average time of a spin flip:
$$\langle T_{spin} \rangle = \frac{1}{L^3 N \cdot w} \sum_{\mathrm{i}=1}^{N} T_w \tag{4.7}$$
where w corresponds to the number of sweeps chosen for the Metropolis simulation.
2. Benchmark the **Replica-level Performance** by computing the average time of a parallel tempering realization:
$$\langle T_{rep} \rangle = \frac{1}{N \cdot x} \sum_{\mathrm{i}=1}^{N} T_x \tag{4.8}$$
where x is the number of exchange steps chosen for the simulation.
3. Benchmark the **Multi-GPU Performance Scaling** by measuring, based on the single GPU and multi-GPU running times t_1, t_g, the fixed-size speedup S_{GPU} and efficiency E_{GPU}
$$\begin{aligned} S_{GPU} &= T_1 / T_g \\ E_{GPU} &= S_{GPU} / g \end{aligned} \tag{4.9}$$
at different problem sizes when using two ($g = 2$) GPUs.
4. Benchmark the **Adaptive Temperatures Performance** by computing the average time of an adapted parallel tempering realization:
$$\langle T_{\mathrm{adapt}} \rangle = \frac{1}{N} \left(\sum_{\mathrm{i}=1}^{N} T_k + \sum_{\mathrm{i}=1}^{N} T_x \right) \tag{4.10}$$

where T_k is the time for doing k trials.

The number of repetitions (*i.e.*, N) for computing the performance averages range between $[10, 40]$ (*i.e.*, less repetitions are required for benchmarking large problem sizes), which are sufficient to provide a standard error of less than 1%.

The workstation used for all benchmarks (including the comparison implementations) is equipped with two 8-core Intel Xeon CPU E5-2640-V3 (Haswell), 128GB of RAM and two Nvidia Tesla K40 each one with 12GB.

4.5.2 Spin and Replica level Performance Results

The first two benchmark results are compared, for reference, against a cache-aligned CPU implementation of the 3D Ising Random Field running both sequential and multi-core. Additionally, we include the Spin-level performance of Weigel's GPU implementation for the 2D Ising running on our system, as a reference of how close or far trueke's spin flip performance is compared to Weigel's highly optimized code for 2D Ising model. The L values used in Weigel's implementation, which simulate L^2 spins, have been adapted to the form $L' = \sqrt[3]{L^2}$ so that the input sizes can be compared against the 3D ones, in the number of total spins.

Figure 4.7 presents the results of spin flip time and the average parallel tempering time.

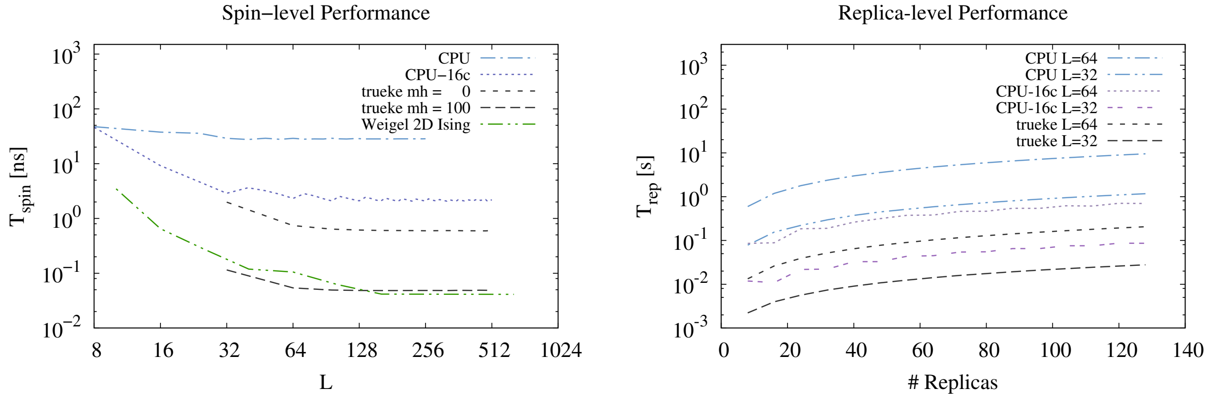

Figure 4.7: On the left, spin-level performance. On the right, replica-level performance.

On the spin level performance results (left plot of Figure 4.7), we observe that *trueke* with 100 multi-hit updates (*i.e.*, $mh = 100$) is over two orders of magnitude faster than the sequential CPU implementation and over one order of magnitude faster than the multi-core CPU implementation using 16 cores. It is worth noting that *trueke* is almost as fast as Weigel's highly optimized GPU Metropolis implementation for the 2D Ising model which uses $mh = 100$ too. If no multi-hit updates are used, then the performance of trueke decreases as expected, becoming $6\times \sim 7\times$ faster than the multi-core implementation. It is important to consider that the comparison has been done using just one GPU for trueke while using two CPU sockets (8 cores each) for the CPU implementation. Normalizing the results to one silicon chip, one would obtain that for spin-level performance the GPU performs approximately one order of magnitude faster than a multi-core CPU. In order to obtain good quality

physical results, the 3D Ising Random Field model must be simulated with $mh = 0$. For this reason, the rest of the results do not include the case when $mh = 100$.

The replica performance results (right plot of Figure 4.7) shows that the multi-GPU implementation outperforms the CPU implementation practically in the same orders of magnitude as in the spin level performance result with $mh = 0$. This result puts in manifest the fact that the exchange phase has little impact on the replica level parallelism, indicating that multi-GPU performance should scale efficiently.

4.5.3 Multi-GPU Scaling

Figure 4.8 presents the speedup and efficiency of *trueke* for computing one 3D Ising Random Field simulation using two Tesla K40 GPUs.

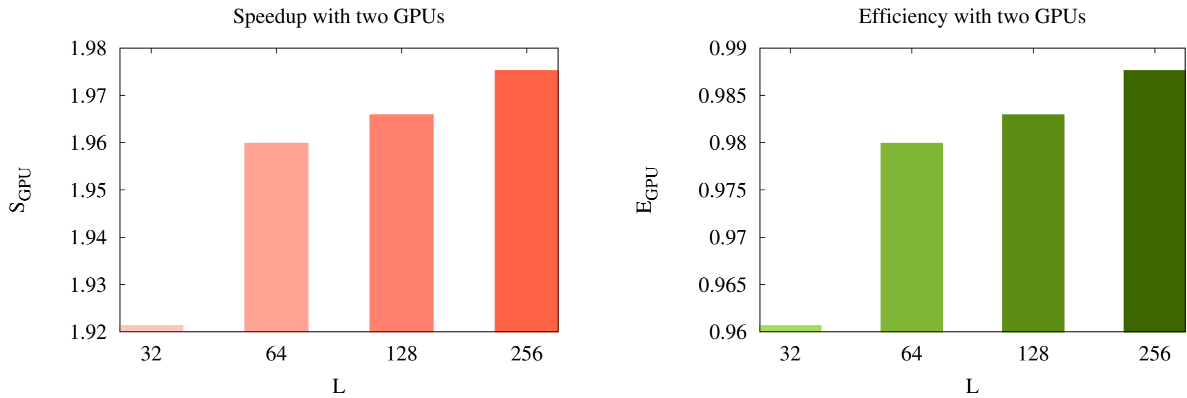

Figure 4.8: Multi-GPU speedup and efficiency for different lattice sizes.

From the results, we observe that as the problem gets larger, the replica level speedup improves almost to $S_{GPUs=2} = 2$ which is the perfect linear speedup. The efficiency plot shows that the replica-level parallel efficiency approaches to $E_{GPUs=w} = 1$ as the lattice becomes larger, indicating that replica-level parallelism does not degrade when increasing L. This favorable behavior can be explained in part by the fact that the work of the exchange phase grows at least as $W_{ex} = \Omega(M) + \Omega(ML^3)$ where the $\Omega(M)$ term is the sequential work for exchanging the M replicas, while the $\Omega(ML^3)$ term is the parallel work for computing the energy at each replica, which is done with a parallel GPU reduction, in $O(log(L^3))$ time for each replica. It is clear that the amount of parallel work grows faster than the sequential work, therefore the parallel efficiency of the whole method should be higher as L, M and the number of GPUs increase.

4.5.4 Performance of The Adaptive Temperatures Technique

Table 4.1 presents the running times of the *adaptive temperatures* technique compared to dense and spare homogeneous techniques for $L = 32, 64$. The simulation parameters used for

all simulations were 100 disorder realizations, each one with 2000 parallel tempering steps and 10 Metropolis sweeps. For $L = 32$, the adaptation phase used 10 trial runs with 2 insertions at each trial. For $L = 64$, the adaptation used 32 trial runs, with 3 insertions at each trial. The trial runs also use 2000 parallel tempering steps with 10 Metropolis sweeps.

Table 4.1: Executions times, in seconds, for adaptive and uniform approaches.

L	sparse-sim	dense-sim	adapt-trials	adapt-sim	adapt (trials + sim)
32	670.25	1417.04	41.84	1167	1209.49
64	10386.75	26697.17	1689.46	19994.70	21684.15

From the results, it is observed that a full adaptive simulation, including its adaptation time, is more convenient than a dense simulation with no adaptation from the point of view of performance. The sparse technique is the fastest one because it simply uses less replicas, but as it will be shown in the next Section, it is not a useful approach for a disordered system starting from $L \geq 64$, neither the dense one, because the exchange rate becomes compromised at the low temperature regime.

4.6 Exchange Rates with Adaptive Temperatures

Results on the exchange rate of the adaptive strategy is presented by plotting the evolution of the average, minimum and maximum exchange rates through the trial runs. Also a comparison of the exchange rates between a dense homogeneous set, the adaptive set and a sparse homogeneous set is presented. The simulation parameters were the same as the ones used to get the performance results of Table 4.1. Figure 4.9 presents the results.

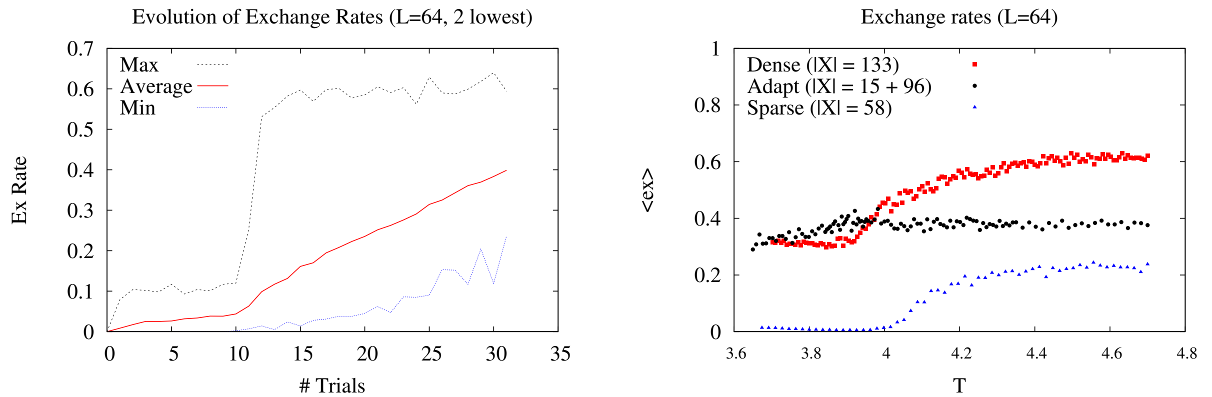

Figure 4.9: On the left, the evolution of average, min and max exchanges through the trial runs. On the right, the exchange rates for the whole temperature range.

From the left plot we observe that as more replicas are added, the average and minimum exchange rates increase and tend to get closer, while the maximum exchange rate stabilizes after starting from trial number 10. For the right plot, it is clearly shown that the dense approach, although is an homogeneous approach, does not generate an homogeneous exchange

rate value for the whole temperature range. For the sparse approach the scenario is even worse because at the low temperature regime there almost no exchanges. On the other hand, the adaptive method generates an exchange rate that is almost homogeneous for the entire range, which is preferred in order to have more control of the simulation. The numbers on the labels indicate the number of replicas used, and it can be seen that the adaptive method uses less replicas than the dense homogeneous approach, therefore it runs faster.

4.7 Preliminary Physical Results

For the correctness of the whole algorithm, including the exchange phase, we ran simulations in the 3D Random Field Ising Model with field strength $h = 1$. The observables have been measured using 5000 parallel tempering steps, 10 sweeps at each step, 1 measurement at each parallel tempering step and using the adaptive temperatures technique.

Average observables are computed as $[\langle A \rangle]$, where [..] corresponds to the average over different disorder realizations and $\langle A \rangle$ denotes the thermal average for a single random field configuration. The magnetization $\langle |M| \rangle$ is defined as

$$[\langle |M| \rangle] = \left[\left\langle \left| \frac{1}{V} \sum_{\mathrm{i}=1}^{L^3} s_{\mathrm{i}} \right| \right\rangle \right] \tag{4.11}$$

The specific heat is

$$[C] = \frac{L^3}{T^2} \Big[(\langle E^2 \rangle - \langle E \rangle^2) \Big] \tag{4.12}$$

where E is the average energy per site. The Binder factor is an average at the disorder realization level and it is defined as:

$$[g] = \frac{1}{2} \Big(3 - \frac{[\langle M^4 \rangle]}{[\langle M^2 \rangle]^2} \Big) \tag{4.13}$$

and the correlation length is

$$[\xi] = \Big[\sqrt{\frac{\langle M^2 \rangle}{\langle F \rangle} - 1} \Big] \tag{4.14}$$

with:

$$F = \frac{1}{3L^3} \Big(F_1 + F_2 \Big) \tag{4.15}$$

$$F_1 = \Big(\sum_{\mathrm{i}}^{L^3} h_{\mathrm{i}} \cos(K \cdot \mathrm{i}_x) \Big)^2 + \Big(\sum_{\mathrm{i}}^{L^3} h_{\mathrm{i}} \cos(K \cdot \mathrm{i}_y) \Big)^2 + \Big(\sum_{\mathrm{i}}^{L^3} h_{\mathrm{i}} \cos(K \cdot \mathrm{i}_z) \Big)^2 \tag{4.16}$$

$$F_2 = \Big(\sum_{\mathrm{i}}^{L^3} h_{\mathrm{i}} \sin(K \cdot \mathrm{i}_x) \Big)^2 + \Big(\sum_{\mathrm{i}}^{L^3} h_{\mathrm{i}} \sin(K \cdot \mathrm{i}_y) \Big)^2 + \Big(\sum_{\mathrm{i}}^{L^3} h_{\mathrm{i}} \sin(K \cdot \mathrm{i}_z) \Big)^2 \tag{4.17}$$

where $K = 2\pi/L$, $h_{\mathrm{i}} = \{-1, 1\}$ is the disordered magnetic field value at spin s_{i} and $\{\mathrm{i}_x, \mathrm{i}_y, \mathrm{i}_z\}$ correspond to the spatial coordinates of a given spin s_{i} in the lattice. For visual clarity,

we only included the error bars of the largest size studied, *i.e.*, $L = 64$, nevertheless it is worth mentioning that the error bars for $L = 8, 16, 32$ were even smaller than the ones for $L = 64$. The results are presented in Figure 4.10 and confirm the transition-like behavior at $3.8 \leq T_c \leq 3.9$, or $0.2564 \leq \beta_c \leq 0.2631$, as shown by Fytas and Malakis in their phase diagram when $h = 1$ [93]. The results presented in this section are preliminary, with up to

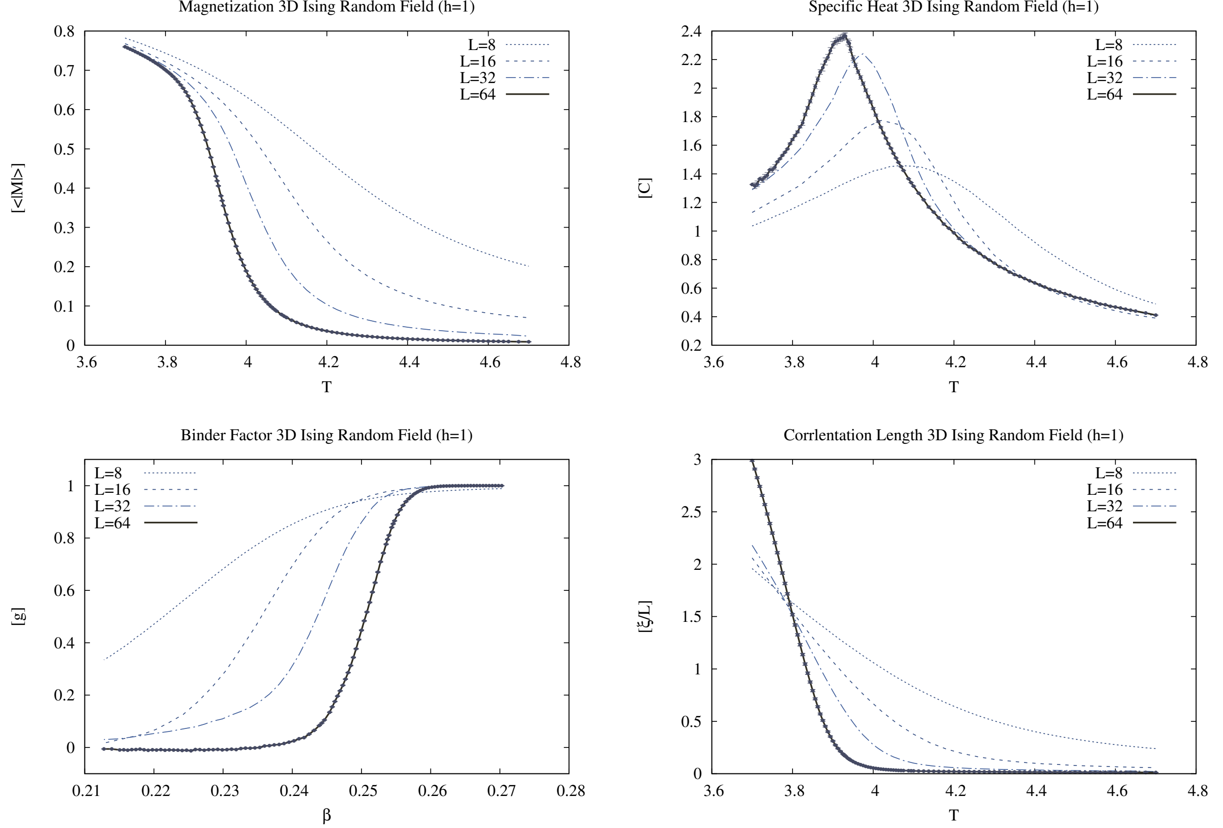

Figure 4.10: Preliminary physical observables for the 3D Random Field with $h = 1$.

2000 disorder realizations for $L = 64$ (less for $L < 64$), each doing 5000 exchange Monte Carlo steps with 10 Metropolis-Hastings sweeps between exchanges. A physical paper devoted to the physical results will be prepared for the future.

4.8 Discussion

We presented a multi-GPU implementation of the *Exchange Monte Carlo* method, named *trueke*, for the simulation of the 3D Ising Random Field model. The parallelization strategy is organized in two levels; (1) *spin-level parallelism*, which scales in the presence of more cores per GPU, and (2) the *replica-level parallelism* that scales in the presence of additional GPUs. The spin-level parallelism is up to two orders of magnitude faster than its CPU counterpart when using multi-hit updates, and between one and two orders of magnitude faster when not using multi-hit updates. The parallel scaling of the method improved as the problem size got larger, in part because the amount of parallel work increases faster than the sequential

work (*i.e.*, exchange phase). This behavior is indeed favorable for multi-GPU computation, where *trueke* achieved approximately up to 99% of efficiency when using two GPUs. Due to hardware limitations, we could not go beyond two GPUs, which would have been ideal for having a better picture of how the performance would scale in large systems. Nevertheless, we intend to put *trueke* available to the community in the near future, so that it can be benchmarked in systems with a high number of GPUs.

The adaptive strategy for selecting the temperature set was based on the idea of inserting new temperatures in between the lowest exchange rates found by an arbitrary number of trial runs. As a result, the simulation used more replicas at the locations were exchange rates were originally low, and less replicas were the exchange rate was already good, such as in the high temperature regime. The adaptive strategy outperformed any homogeneous approach, since these last ones had to deal with an over-population of replicas at places that actually did not require more temperature points, resulting in extra computational cost and slower performance, and an under-population of replicas at the low temperature regime near T_c, generating low exchange rates. The adaptive method works better when a small number of points are added at each trial run, *i.e.*, between one and five insertions at each trial run, because this way is more unlikely to misplace an insertion. Compared to the method by Katzgraber *et. al.* [152], our adaptive method differs since it feedbacks from the local exchanges of pairs of temperatures, always lifting the minimum values observed by inserting new temperatures, while in the work of Katzgraber *et. al.* they feedback by counting the number of times a replica travels the whole temperature range and based on this information they move the temperature set. In the method by Bittner *et. al* [27], they vary the number of Metropolis sweeps based on the auto-correlation times in order to avoid two replicas getting trapped exchanging together, which they identify as a problem for Katzgraber method. Our approach of inserting new temperatures provides the advantage that it does not compromise the rest of the temperature range and the technique by itself is general since it is just based on improving the lowest k minimum exchange rates observed at each trial run. For the near future, we intend to do a formal comparison of all three implementations of the methods.

The main reason for choosing multi-GPU computing at the replica-level, and not at the realization level (fully independent parallelism) as one would naturally choose, is mostly because the latter strategy is not prepared for the study of large lattice systems, which would be of great interest for the near future. From our experience with the 3D RFIM, the number of replicas needed to keep at least $35\% \sim 45\%$ of exchange rate grew very fast as L increased. Thus, distributing the replicas dynamically across several GPUs extends the possibilities of studying larger disordered lattices.

The multi-GPU approach proposed, alias *trueke*, has allowed us to study th 3D Ising Random Field model at size $L = 64$ for which its results can become physical contributions to the field. It is expected that eventually, as more GPUs are used, larger lattices could be studied.

Conclusions

This thesis has addressed the problem of the expensive computations involved in the study of spin systems when using both exact and Monte Carlo approaches. The main goal of this thesis, which was to elaborate novel parallel methods for each type of approach, was achieved and the two research questions formulated at the beginning were answered positively. The computational concepts and research developed in this thesis has contributed to the field of parallel computing and computational physics in the form of scientific publications.

For the exact case, the contribution was a new parallel algorithm for computing the general (q, v) transfer matrix of strip lattices in the Potts model. The algorithm achieves an important reduction in the asymptotic size of the configuration space, from $O(4^m)$ to $O(3^m)$, by re-organizing the problem domain as *parallel family trees (PFT)*. With this re-organization, the transfer matrix can now be computed by just processing the root configurations, which are $O(3^m)$ in number. Experimental performance results have shown evidence that PFT has an exponential advantage over the *Catalan Parallel Method (CPM)* [199]. Moreover, it is not only faster to generate the matrix using PFT, but it is also faster to use it later for extracting the physical information since numerical evaluation is done at the compressed stage. In terms of parallel performance scaling, PFT achieves $28X$ of speedup using $p = 32$ for the square strip tests, with an efficiency above 90% for a strip of width $m \geq 10$. For the kagome strip test, efficiency stayed above 55% for a strip of $m \geq 7$ and the maximum value of speedup reached was close to $20X$ when using $p = 31$. The best balance between speedup and efficiency was found at $p \in \{8, 9, 10\}$. If more processors are used to the point that $p = \Delta_m$, then the time for computing the TM would only be the time to solve the deepest *dag* of computation. For this reason, we expect that an execution on a large cluster or supercomputer could allow the computation of transfer matrices of strips wider than what has been reached before. Part of the Chapter of this work was published in the proceedings of the *15-th IEEE International Conference on High Performance Computing and Communications* (HPCC 2013) [199], and the full Chapter was published in the *Computer Physics Communications* Journal [197]

For the Monte Carlo case, the contribution was a massively parallel multi-GPU method, alias *trueke*, for the simulation of 3D lattices in the Random Field Ising Model. The method is based on the Exchange Monte Carlo algorithm, but re-designed to have two levels of parallelism; *spin-level parallelism* and *replica-level parallelism*. Spin-level parallelism scales in the presence of more powerful GPUs, while replica-level parallelism scales in the precense of multiple GPUs. When both levels of parallelism are combined, the result is a highly efficient parallel method that is between one and two orders of magnitude faster than

the equivalent sequential CPU implementation. The multi-GPU performance scaling using two GPUs achieved almost perfect linear speedup, with approximately 99% of efficiency. The fact that the multi-GPU efficiency becomes better as the size increases puts in evidence that as the problem gets larger, the amount of parallel work grows faster than the amount of sequential work, favorating a GPU-cluster scenario. In addition to the paralelism, we also proposed a new adaptative algorithmic strategy that inserts temperatures at the places that suffered from low exchange rates. As a result, the adaptative algorithm allowed to compute physical results as good as if using a very dense approach, while having a much smaller computational cost. With the resources available, it was possible to compute preliminary physical results for $L = \{8, 16, 32, 64\}$, but with more GPUs and enough time, it is possible to aim for larger sizes such as $L = \{128, 256\}$. In the near future we expect to obtain definitive physical observables for the 3D Random Field Ising Model at $L = 64$ and hopefully $L = 128$. For this, one must test many simulation parameters in order to find a good set that will make the simulation reach equilibrium properly. This research of this Chapter will be submitted to a comptutational physics journal in the near future.

Two additional works of this thesis became contributions as well. The parallel computing background was published as a survey paper in the Journal *Communications in Computational Physics* [200] and the work presented in the Appendix about GPU maps in triangular domains was published in the proceedings of *16-th IEEE International Conference on High Performance Computing and Communications* (HPCC 2014) [198].

Future work

We believe it would be interesting to study how the efficiency of GPU computing is affected when working on particle-based domains that have different geometries, such as triangular structures, spheres, pyramids and also fractal geometries. In such scenarios GPU computing needs efficient mapping strategies, different from the standard map $f(x) = x$, in order to minimize the number of unnnecessary threads and still preserve spatial thread locality. Improvements on this field would have inmmediate applications to science and engineering, including the possibility of extending the results of Chapter 4 for different geometries. An initial research was done during the time of this thesis, about GPU maps (available in Appendix A), where we found that GPU performance can increase up to 18% when using a dedicated map function for handling GPU threads in 2D triangular structures. The study of GPU maps for complex domains is a fresh and modern research line in the field of GPU computing and can provide interesting results for the field of parallel algorithms and parallel architectures.

Bibliography

[1] Scott Aaronson. Np-complete problems and physical reality. *SIGACT News*, 36:2005, 2005.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[3] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 305–314, New York, NY, USA, 1987. ACM.

[4] A.-K. Cheik Ahamed and F. Magoulès. Fast sparse matrix-vector multiplication on graphics processing unit for finite element analysis. In *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC 2012), Liverpool, UK, June 25–27, 2012*, pages 1307–1314. **IEEE Computer Society**, 2012.

[5] A.-K. Cheik Ahamed and F. Magoulès. Iterative methods for sparse linear systems on graphics processing unit. In *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC 2012), Liverpool, UK, June 25–27, 2012*, pages 836–842. **IEEE Computer Society**, 2012.

[6] A.-K. Cheik Ahamed and F. Magoulès. Iterative Krylov methods for gravity problems on graphics processing unit. In *Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), Kingston, London, UK, September 2nd-4th, 2013*, pages 16–20. **IEEE Computer Society**, 2013.

[7] A.-K. Cheik Ahamed and F. Magoulès. Schwarz method with two-sided transmission conditions for the gravity equations on graphics processing unit. In *Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), Kingston, London, UK, September 2nd-4th, 2013*, pages 105–109. **IEEE Computer Society**, 2013.

[8] Dorit Aharonov, Itai Arad, Elad Eban, and Zeph Landau. Polynomial quantum algorithms for additive approximations of the Potts model and other points of the tutte plane, 2007.

[9] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994. 10.1007/BF01185206.

[10] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *In Proc. Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.

[11] P.D. Alvarez, F. Canfora, S.A. Reyes, and S. Riquelme. Potts model on recursive lattices: some new exact results. *The European Physical Journal B*, 85(3), 2012.

[12] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[13] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Fast collision culling in large-scale environments using GPU mapping function. In *EGPGV*, pages 71–80, 2012.

[14] Clive F. Baillie and Paul D. Coddington. Comparison of cluster algorithms for two-dimensional Potts models. *Phys. Rev. B*, 43:10617–10621, May 1991.

[15] F Barahona. On the computational complexity of ising spin glass models. *Journal of Physics A: Mathematical and General*, 15(10):3241, 1982.

[16] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[17] Luiz André Barroso. The price of performance. *Queue*, 3(7):48–53, September 2005.

[18] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the ginac framework for symbolic computation within the c++ programming language. *Journal of Symbolic Computation*, 33(1):1 – 12, 2002.

[19] R.J. Baxter. *Exactly solved models in statistical mechanics*. Academic Press, 1982.

[20] Rodney J. Baxter. *Exactly Solved Models in Statistical Mechanics*. Dover Publications, January 2008.

[21] Carter Bays. Cellular automata in triangular, pentagonal and hexagonal tessellations. In Robert A. Meyers, editor, *Computational Complexity*, pages 434–442. Springer New York, 2012.

[22] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the crcw pram. In *In Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 25–27. ACM, 1987.

[23] Andrea Bedini and Jesper Lykke Jacobsen. A tree-decomposed transfer matrix for computing exact Potts model partition functions for arbitrary graphs, with applications to planar graph colourings. *Journal of Physics A: Mathematical and Theoretical*, 43(38):385001, 2010.

[24] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *J. Comput. Phys.*, 231(7):2825–2839, April 2012.

[25] Adrien Bernhardt, Andre Maximo, Luiz Velho, Houssam Hnaidi, and Marie-Paule Cani. Real-time terrain modeling using cpu-GPU coupled computation. In *Proceedings of the 2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, SIBGRAPI '11, pages 64–71, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Z. Bittnar, J. Kruis, J. Němeček, B. Patzák, and D. Rypl. Parallel and distributed computations for structural mechanics: a review. *Civil and structural engineering computing: 2001*, pages 211–233, 2001.

[27] Elmar Bittner, Andreas Nußbaumer, and Wolfhard Janke. Make life simple: Unleash the full power of the parallel tempering algorithm. *Phys. Rev. Lett.*, 101:130603, Sep 2008.

[28] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Computing the tutte polynomial in vertex-exponential time. *CoRR*, abs/0711.2585, 2007.

[29] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Computing the tutte polynomial in vertex-exponential time. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 677–686, 2008.

[30] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[31] Benjamin Block, Peter Virnau, and Tobias Preis. Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2d ising model. *Computer Physics Communications*, 181(9):1549 – 1556, 2010.

[32] H. W. J. Blöte and R. H. Swendsen. First-order phase transitions and the three-state Potts model. *Phys. Rev. Lett.*, 43:799–802, Sep 1979.

[33] H.W.J Blte and M.P Nightingale. Critical behaviour of the two-dimensional Potts model with a continuous number of states; a finite size scaling analysis. *Physica A: Statistical Mechanics and its Applications*, 112(3):405 – 465, 1982.

[34] Attila Boer. Monte Carlo simulation of the two-dimensional Potts model using nonextensive statistics. *Physica A.*, 390:4203–4209, 2011.

[35] Larry Carter Bowen Alpern. The ram model considered harmful towards a science of performance programming. Technical report, IBM Watson research center, 1994.

[36] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(11):1222 –1239, nov 2001.

[37] Yuri Boykov, Olga Veksler, and Ramin Zabih. A new algorithm for energy minimization with discontinuities. In *Proceedings of the Second International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, EMMCVPR '99, pages 205–220, London, UK, 1999. Springer-Verlag.

[38] Clay P. Breshears. *The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, 2009.

[39] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[40] Moler C. *Matrix Computation on Distributed Memory Multiprocessors.* Heath, Michael T. Hypercube Multiprocessors (Society for Industrial and Applied Mathematics, Philadelphia)., 1986.

[41] Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. K-model: A new computational model for stream processors. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, pages 239–246, Washington, DC, USA, 2010. IEEE Computer Society.

[42] C. Castaneda-Marroquen, C.B. Navarrete, A. Ortega, M. Alfonseca, and E. Anguiano. Parallel metropolis-montecarlo simulation for potts model using an adaptable network topology based on dynamic graph partitioning. In *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on*, pages 89 –96, july 2008.

[43] John L. Casti. *Would-Be Worlds: How Simulation Is Changing the Frontiers of Science.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1998.

[44] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 104–111, New York, NY, USA, 2008. ACM.

[45] Bradford L. Chamberlain. Chapel (cray inc. hpcs language). In *Encyclopedia of Parallel Computing*, pages 249–256. Springer, 2011.

[46] Shu-Chiuan Chang, Jesper Lykke Jacobsen, Jess Salas, and Robert Shrock. Exact Potts model partition functions for strips of the triangular lattice. *Physica A*, 286(1-2):59, 2002.

[47] Shu-Chiuan Chang, Jess Salas, and Robert Shrock. Exact Potts model partition functions for strips of the square lattice. *Journal of Statistical Physics*, 107(5-6):1207–1253, 2002.

[48] Shu-Chiuan Chang, Jess Salas, and Robert Shrock. Exact Potts model partition functions on wider arbitrary-length strips of the square lattice. *Journal of Statistical Physics*, 107(5/6):1207–1253, 2002.

[49] Shu-Chiuan Chang and Robert Shrock. Exact Potts model partition functions on strips of the honeycomb lattice. *Physica A: Statistical Mechanics and its Applications*, 296(1-2):48, 2000.

[50] Shu-Chiuan Chang and Robert Shrock. Structure of the partition function and transfer matrices for the Potts model in a magnetic field on lattice strips. *Journal of Statistical Physics*, 137(4):667–699, 2009.

[51] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[52] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. *SIGARCH Comput. Archit. News*, 18(3a):239–248, May 1990.

[53] Nan Chen, James A. Glazier, Jesús A. Izaguirre, and Mark S. Alber. A parallel implementation of the cellular potts model for simulation of cell-based morphogenesis. *Computer Physics Communications*, 176(11-12):670–681, 2007.

[54] S. Chen, Alan M. Ferrenberg, and D. P. Landau. Monte Carlo simulation of phase transitions in a two-dimensional random-bond Potts model. *Phys. Rev. E*, 52:1377–1386, Aug 1995.

[55] Barry A. Cipra. An introduction to the Ising model. *Am. Math. Monthly*, 94:937–959, December 1987.

[56] Florent Cohen, Philippe Decaudin, and Fabrice Neyret. GPU-based lighting and shadowing of complex natural scenes. In *Siggraph'04 Conf. DVD-ROM (Poster)*, August 2004. Los Angeles, USA.

[57] Mark Colbert and Jaroslav Křivánek. Real-time dynamic shadows for image-based lighting. In *ShaderX 7 - Advanced Rendering Technicques*. Charles River Media, 2009.

[58] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[59] Aleksandar Colic, Hari Kalva, and Borko Furht. Exploring nvidia-cuda for video coding. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, pages 13–22, New York, NY, USA, 2010. ACM.

[60] Matthew Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15(1):1–40, 2004.

[61] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[62] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[63] Nvidia Corporation. *Kepler Whitepaper for the GK110 architecture*, 2012.

[64] Harold Scott Macdonald Coxeter. *Regular polytopes*. Courier Dover Publications, 1973.

[65] Eliana Scheihing Cristobal A. Navarro, Nancy Hitschfeld-Kahler. A GPU-based method for generating quasi-delaunay triangulations based on edge-flips. In *Proceedings of the 8th International on Computer Graphics, Theory and Applications*, GRAPP 2013, pages 27–34, February 2013.

[66] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[67] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[68] Aurelien Decelle, Florent Krzakala, Cristopher Moore, and Lenka Zdeborová. Asymptotic analysis of the stochastic block model for modular networks and its algorithmic applications. *Phys. Rev. E*, 84:066106, Dec 2011.

[69] Ydalia Delgado. A GPU-accelerated worm algorithm. Technical report, Karl-Franzens University Graz, February 2011.

[70] Ydalia Delgado. A GPU-accelerated worm algorithm. Technical report, University of Grazz, Graz, Austria, 2011.

[71] Youjin Deng, Timothy M. Garoni, and Alan D. Sokal. Dynamic critical behavior of the worm algorithm for the ising model. *Phys. Rev. Lett.*, 99:110601, Sep 2007.

[72] B Derrida and J Vannimenus. Transfer-matrix approach to percolation and phenomenological renormalization. *Journal de Physique Lettres*, 41(20):473–476, 1980.

[73] R Dewar and C K Harris. Parallel computation of cluster properties: application to 2d percolation. *Journal of Physics A: Mathematical and General*, 20(4):985, 1987.

[74] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[75] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[76] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[77] Neil Dunstan. Semaphores for fair scheduling monitor conditions. *SIGOPS Oper. Syst. Rev.*, 25(3):27–31, May 1991.

[78] R D Dutton and R C Brigham. Computationally efficient bounds for the catalan numbers. *Eur. J. Comb.*, 7(3):211–213, July 1986.

[79] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Computers*, 38(3):408–423, 1989.

[80] S F Edwards and P W Anderson. Theory of spin glasses. *Journal of Physics F: Metal Physics*, 5(5):965, 1975.

[81] V Faber, O M Lubeck, and A B White, Jr. Superlinear speedup of an efficient sequential algorithm is not possible. *Parallel Comput.*, 3(3):259–260, July 1986.

[82] Ye Fang, Sheng Feng, Ka-Ming Tam, Zhifeng Yun, Juana Moreno, J. Ramanujam, and Mark Jarrell. Parallel tempering simulation of the three-dimensional edwardsanderson model with compact asynchronous multispin coding on GPU. *Computer Physics Communications*, 185(10):2467 – 2478, 2014.

[83] Néstor Ferrando, M. A. Goslvez, Joaqun Cerd, Rafael Gadea Girons, and K. Sato. Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. *Computer Physics Communications*, pages 628–640, 2011.

[84] Alan M. Ferrenberg and Robert H. Swendsen. New Monte Carlo technique for studying phase transitions. *Phys. Rev. Lett.*, 61:2635–2638, Dec 1988.

[85] Ezequiel E Ferrero, Juan Pablo De Francesco, Nicols Wolovick, and Sergio A Cannas. q-state potts model metastability study using optimized GPU-based Monte Carlo algorithms. *Arxiv preprint arXiv11010876*, page 26, 2011.

[86] Ezequiel E Ferrero, Juan Pablo De Francesco, Nicols Wolovick, and Sergio A Cannas. q-state Potts model metastability study using optimized GPU-based Monte Carlo algorithms. *Computer Physics Communications*, 183(8):15781587, 2012.

[87] Ezequiel E. Ferrero, Juan Pablo De Francesco, Nicols Wolovick, and Sergio A. Cannas. q-state Potts model metastability study using optimized GPU-based Monte Carlo algorithms. *Computer Physics Communications*, 183(8):1578 – 1587, 2012.

[88] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[89] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[90] Message P Forum. Mpi: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[91] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[92] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!*

Morgan Kaufmann, May 1994.

[93] N. G. Fytas and A. Malakis. Phase diagram of the 3d bimodal random-field ising model. *The European Physical Journal B*, 61(1):111–120, 2008.

[94] Christine Gabriel. *Dynamical properties of the Worm Algorithm*. PhD thesis, Technischen Universität Graz, August 2002.

[95] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[96] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

[97] C. Geyer. Markov Chain Monte Carlo maximum likelihood. In *Proceedings of the 23rd Symposium on the Interface*, pages 156–163, 1991.

[98] M Ghaemi and G. A. Parsafar. Size reduction of the transfer matrix of two-dimensional Ising and Potts models. *2*, 4, 2003.

[99] Stéphane Gobron, Hervé Bonafos, and Daniel Mestre. GPU accelerated computation and visualization of hexagonal cellular automata. In *Proceedings of the 8th international conference on Cellular Automata for Reseach and Industry*, ACRI '08, pages 512–521, Berlin, Heidelberg, 2008. Springer-Verlag.

[100] Stéphane Gobron, Arzu Çöltekin, Hervé Bonafos, and Daniel Thalmann. GPGPU computation and visualization of three-dimensional cellular automata. *The Visual Computer*, 27(1):67–81, 2011.

[101] Stéphane Gobron, François Devillard, and Bernard Heit. Retina simulation using cellular automata and GPU programming. *Mach. Vision Appl.*, 18(6):331–342, November 2007.

[102] Stphane Gobron, Clment Marx, Junghyun Ahn, and Daniel Thalmann. Real-time textured volume reconstruction using virtual and real video cameras. In *proceedings of the Computer Graphics International 2010 conference*, 2010.

[103] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUterasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[104] F. Graner and J. A. Glazier. Simulation of biological cell sorting using a two-dimensional extended Potts model. *Physical Review Letters*, 69:2013–2016, September 1992.

[105] F. Graner and J. A. Glazier. Simulation of biological cell sorting using a two-dimensional extended Potts model. *Physical review letters*, 69(13):2013–2016, September 1992.

[106] Raymond Greenlaw, James H. Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, USA, April 1995.

[107] G S Grest. Monte Carlo study of the antiferromagnetic Potts model on frustrated lattices. *Journal of Physics A: Mathematical and General*, 14(6):L217, 1981.

[108] The Khronos Group. *OpenGL Shading Language specification*, 2013.

[109] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. *Int. J. Parallel Program.*, 28(6):537–562, December 2000.

[110] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[111] John L. Gustafson. Fixed time, tiered memory, and superlinear speedup. In *In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, 1990.

[112] John L Gustafson. Fixed time, tiered memory, and superlinear speedup. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 1255–1260, 1990.

[113] John L. Gustafson. The consequences of fixed time performance measurement. In *Proceedings of the 25th Hawaii International Conference on Systems Sciences, IEEE Computer Society*, 1992.

[114] Fred Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 2328 of *Lecture Notes in Computer Science*, pages 418–436. Springer Berlin / Heidelberg, 2006.

[115] Gary Haggard, David J. Pearce, and Gordon Royle. Computing tutte polynomials. *ACM Trans. Math. Softw.*, 37:24:1–24:17, September 2010.

[116] Tom Halverson and Arun Ram. Partition algebras. *European Journal of Combinatorics*, 26(6):869–921, 2005.

[117] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence. In *SC*, 2009.

[118] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New

York, NY, USA, 2009. ACM.

[119] Takahiro Harada. Real-time rigid body simulation on GPUs. In Hubert Nguyen, editor, *GPU Gems 3*, pages 611–632. Addison-Wesley, 2008.

[120] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[121] A. K. Hartmann. Partition function of two- and three-dimensional Potts ferromagnets for arbitrary values of q>0. *Phys.rev.lett.*, 94:050601, 2005.

[122] Martin Hasenbusch. Finite size scaling study of lattice models in the three-dimensional ising universality class. *Phys. Rev. B*, 82:174433, Nov 2010.

[123] W. K. Hastings. Monte Carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[124] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.*, 36(12):655–678, December 2010.

[125] Martin Henk, Jürgen Richter-Gebert, and Günter M. Ziegler. Basic properties of convex polytopes. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of discrete and computational geometry*, pages 243–270. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[126] Peter Hilton and Jean Pedersen. Catalan numbers, their generalization, and their uses. *Math. Intelligencer*, 13(2):64–75, 1991.

[127] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[128] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between cpu and GPU. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[129] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[130] Koji Hukushima and Koji Nemoto. Exchange Monte Carlo method and application to spin glass simulations. *Journal of the Physical Society of Japan*, 65(6):1604–1608, 1996.

[131] Loretta Ichim, Ecaterina Oltean, and Radu Dobrescu. Fractal evaluation of a discrete model for simulation of avascular tumor growth. In *Proceedings of the 10th WSEAS International Conference on Automatic Control, Modelling & Simulation*, ACMOS'08,

pages 90–95, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[132] Andrew Ilachinski. *Cellular Automata: A Discrete Universe*. World Scientific Pub Co Inc, 1st edition, 2001.

[133] E Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift Fr Physik*, 31(1):253–258, 1925.

[134] Lubomir Ivanov. The n-body problem throughout the computer science curriculum. *J. Comput. Sci. Coll.*, 22(6):43–52, June 2007.

[135] O Ivasishin, S Shevchenko, N Vasiliev, and S Semiatin. A 3-d monte-carlo (Potts) model for recrystallization and grain growth in polycrystalline materials. *Materials Science and Engineering A*, 433(1-2):216–232, 2006.

[136] D Luebke J Tran, D Jordan. New challenges for cellular automata simulation on the GPU. Technical Report MSU-CSE-00-2, Virginia University, 2003.

[137] Jesper Lykke Jacobsen. Bulk, surface and corner free-energy series for the chromatic polynomial on the square and triangular lattices. *Journal of Physics A: Mathematical and Theoretical*, 43(31):315002, 2010.

[138] Jesper Lykke Jacobsen. High-precision percolation thresholds and Potts-model critical manifolds from graph polynomials. *Journal of Physics A: Mathematical and Theoretical*, 47(13):135001, 2014.

[139] J.L. Jacobsen, J.-F. Richard, and J. Salas. Complex-temperature phase diagram of Potts and rsos models. *Nuclear Physics B*, 743(3):153–206, 2006.

[140] J.L. Jacobsen and J. Salas. Transfer matrices and partition-function zeros for antiferromagnetic Potts models. II. extended results for square-lattice chromatic polynomial. *Journal of Statistical Physics*, 104(3-4):701–723, 2001.

[141] J.L. Jacobsen and J. Salas. Transfer matrices and partition-function zeros for antiferromagnetic Potts models : IV. chromatic polynomial with cyclic boundary conditions. *Journal of Statistical Physics*, 122(4):705–760, 2006.

[142] J.L. Jacobsen and J. Salas. Phase diagram of the chromatic polynomial on a torus. *Nuclear Physics B*, 783(3):238–296, 2007.

[143] J.L. Jacobsen, J. Salas, and A.D. Sokal. Transfer matrices and partition-function zeros for antiferromagnetic Potts models. III. triangular-lattice chromatic polynomial. *Journal of Statistical Physics*, 112(5-6):921–1017, 2003.

[144] Wolfhard Janke and Ramon Villanova. Monte Carlo study of 8-state Potts model on 2d random lattices. *Nuclear Physics B*, 47:641, 1996.

[145] P. Jimnez, F. Thomas, and C. Torras. 3d collision detection: A survey. *Computers and*

Graphics, 25:269–285, 2000.

[146] Sicilia F. Judice, Bruno Barcellos S. Coutinho, and Gilson A. Giraldi. Lattice methods for fluid animation in games. *Comput. Entertain.*, 7(4):56:1–56:29, January 2010.

[147] Jin Hyuk Jung and Dianne P. OLeary. Exploiting structure of symmetric or triangular matrices on a GPU. Technical report, University of Maryland, 2008.

[148] K. and Rummukainen. Multicanonical cluster algorithm and the two-dimensional 7-state Potts model. *Nuclear Physics B*, 390(3):621 – 636, 1993.

[149] Alain S Karma and Michael J Nolan. Monte Carlo investigation of the one-dimensional dynamical Potts model. *Physical Review B*, 27(3):1922–1924, 1983.

[150] Sriram Kashyap, Rhushabh Goradia, Parag Chaudhuri, and Sharat Chandran. Implicit surface octrees for ray tracing point models. In *Proceedings of the Seventh Indian Conference on Computer Vision, Graphics and Image Processing*, ICVGIP '10, pages 227–234, New York, NY, USA, 2010. ACM.

[151] Helmut G Katzgraber, Mathias Körner, and AP Young. Universality in three-dimensional Ising spin glasses: A Monte Carlo study. *Physical Review B*, 73(22):224432, 2006.

[152] Helmut G Katzgraber, Simon Trebst, David A Huse, and Matthias Troyer. Feedback-optimized parallel tempering Monte Carlo. *Journal of Statistical Mechanics: Theory and Experiment*, 2006(03):P03018, 2006.

[153] Claude Kauffmann and Nicolas Piche. Seeded nd medical image segmentation by cellular automaton on GPU. *Int. J. Computer Assisted Radiology and Surgery*, 5(3):251–262, 2010.

[154] Jan Kautz, Wolfgang Heidrich, and Hans-Peter Seidel. Real-time bump map synthesis. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, pages 109–114, New York, NY, USA, 2001. ACM.

[155] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, Tools. Society for Industrial and Applied Mathematics, 2011.

[156] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[157] David B. Kidner, Philip J. Rallings, and J. Andrew Ware. Parallel processing for terrain analysis in GIS: Visibility as a case study. *Geoinformatica*, 1(2):183–207, August 1997.

[158] M.J. Kilgard. A practical and robust bump-mapping technique for todays GPUs. Technical report, Nvidia, 2000.

[159] Seon Wook Kim and Rudolf Eigenmann. The structure of a compiler for explicit and implicit parallelism. In *Proceedings of the 14th international conference on Languages*

and compilers for parallel computing, LCPC'01, pages 336–351, Berlin, Heidelberg, 2003. Springer-Verlag.

[160] W. Kinzel. Phase transitions of cellular automata. *Zeitschrift fr Physik B Condensed Matter*, 58(3):229–244, 1985.

[161] Peter Kipfer. LCP algorithms for collision detection using CUDA. In Hubert Nguyen, editor, *GPUGems 3*, pages 723–739. Addison-Wesley, 2007.

[162] S. Kirkpatrick and D. Sherrington. Infinite range models of spin-glasses. *Physical Review B*, 17:4384–4403, 1988.

[163] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, December 1974.

[164] Yukihiro Komura and Yutaka Okabe. GPU-based single-cluster algorithm for the simulation of the ising model. *J. Comput. Phys.*, 231(4):1209–1215, February 2012.

[165] Yukihiro Komura and Yutaka Okabe. Gpu-based Swendsen-wang multi-cluster algorithm for the simulation of two-dimensional classical spin systems. *Computer Physics Communications*, 183(6):1155–1161, 2012.

[166] Yukihiro Komura and Yutaka Okabe. Multi-GPU-based SwendsenWang multi-cluster algorithm for the simulation of two-dimensional q-state Potts model. *Computer Physics Communications*, 184(1):40 – 44, 2013.

[167] Pavol Korek, Luk Sekanina, and Otto Fuk. Cellular automata based traffic simulation accelerated on GPU. In *Proceedings of the 17th International Conference on Soft Computing (MENDEL2011)*, pages 395–402. Institute of Automation and Computer Science FME BUT, 2011.

[168] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, June 2007.

[169] Jaroslav Křivánek. Perceptually driven point sample rendering. In *Proceedings of Workshop 2002*, volume A, pages 194–195. Czech Technical University in Prague, 2002.

[170] Jaroslav Křivánek, Jiří Žára, and Kadi Bouatouch. Fast depth of field rendering with surface splatting. In *Computer Graphics International, CGI 2003. Proceedings*, pages 196 – 201, july 2003.

[171] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 55–55, New York, NY, USA, 2001. ACM.

[172] G De las Cuevas, W Dr, M Van den Nest, and M A Martin-Delgado. Quantum algorithms for classical lattice models. *New Journal of Physics*, 13(9):093021, 2011.

[173] G De las Cuevas, W Dr, M Van den Nest, and M A Martin-Delgado. Quantum algorithms for classical lattice models. *New Journal of Physics*, 13(9):093021, 2011.

[174] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. cpu myth: an evaluation of throughput computing on cpu and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[175] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[176] Qi Li, Vojislav Kecman, and Raied Salman. A chunking method for euclidean distance matrix calculation on large dataset using multi-GPU. In *Proceedings of the 2010 Ninth International Conference on Machine Learning and Applications*, ICMLA '10, pages 208–213, Washington, DC, USA, 2010. IEEE Computer Society.

[177] Qingquan Liu, Youjin Deng, and Timothy M. Garoni. Worm Monte Carlo study of the honeycomb-lattice loop model. *Nuclear Physics. B*, 846(2):283–315, 2011.

[178] D. Loveman. High performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[179] Peter Lu, Hidekazu Oki, Catherine Frey, Gregory Chamitoff, Leroy Chiao, Edward Fincke, C. Foale, Sandra Magnus, William McArthur, Daniel Tani, Peggy Whitson, Jeffrey Williams, William Meyer, Ronald Sicker, Brion Au, Mark Christiansen, Andrew Schofield, and David Weitz. Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the international space station. *Journal of Real-Time Image Processing*, 5:179–193, 2010. 10.1007/s11554-009-0133-1.

[180] M. Lulli, M. Bernaschi, and G. Parisi. Highly optimized simulations on single- and multi-gpu systems of the 3d ising spin glass model. *Computer Physics Communications*, pages –, 2015.

[181] Xiaosong Ma, Jiangtian Li, and N.F. Samatova. Automatic parallelization of scripting languages: Toward transparent desktop parallel computing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007.

[182] M. Macedonia. The GPU enters computing's mainstream. *Computer*, 36(10):106–108, 2003.

[183] Philip D. Mackenzie and Vijaya Ramachandran. ERCW PRAMs and optical communication. In *in Proceedings of the European Conference on Parallel Processing, EUROPAR 96*, pages 293–302, 1996.

[184] F. Magoulès, A.-K. Cheik Ahamed, and R. Putanowicz. Auto-tuned krylov methods on cluster of graphics processing unit. *International Journal of Computer Mathematics*, (in press), 2014.

[185] Bruce D. Malamud and Donald L. Turcotte. Cellular-automata models applied to natural hazards. *Computing in Science and Engg.*, 2(3):42–51, May 2000.

[186] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano. Implementations of parallel computation of euclidean distance map in multicore processors and GPUs. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 120–127, 2010.

[187] Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano. A GPU implementation of computing euclidean distance map with efficient memory access. In *Proceedings of the 2011 Second International Conference on Networking and Computing*, ICNC '11, pages 68–76, Washington, DC, USA, 2011. IEEE Computer Society.

[188] E. Marinari and G. Parisi. Simulated tempering: A new Monte Carlo scheme. *EPL (Europhysics Letters)*, 19(6):451, 1992.

[189] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, July 2003.

[190] Ricardo Marroquim and André Maximo. Introduction to GPU programming with glsl. In *Proceedings of the 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, SIBGRAPI-TUTORIALS '09, pages 3–16, Washington, DC, USA, 2009. IEEE Computer Society.

[191] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. In Michael Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 729–743. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0032070.

[192] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[193] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087, 1953.

[194] M. Mezard, G. Parisi, N. Sourlas, G. Toulouse, and M. Virasoro. Replica Symmetry-Breaking and the Nature of the Spin-Glass Phase. *J. Physique*, 45:843, 1984.

[195] A.S. Mikhayhu. *Embarrassingly Parallel*. Tempor, 2012.

[196] Pascal Monceau. Critical behavior of the ferromagnetic q-state Potts model in fractal dimensions: Monte Carlo simulations on Sierpinski and Menger fractal structures. *Phys. Rev. B*, 74:094416, Sep 2006.

[197] Cristobal A. Navarro, Fabrizio Canfora, Nancy Hitschfeld, and Gonzalo Navarro. Parallel family trees for transfer matrices in the potts model. *Computer Physics Communications*, 187(0):55 – 71, 2015.

[198] Cristobal A. Navarro and Nancy Hitschfeld. GPU maps for the space of computation in triangular domain problems. In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*, pages 375–382, 2014.

[199] Cristobal A. Navarro, Nancy Hitschfeld, and Fabrizio Canfora. Multi-core computation of transfer matrices for strip lattices in the potts model. In *15th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*, pages 125–134, 2013.

[200] Cristobal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Commun. Comput. Phys.*, 15:285–329, 2014.

[201] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[202] M.E.J. Newman and G.T. Barkema. *Monte Carlo methods in statistical physics*. Clarendon Press, 1999.

[203] Hubert Nguyen. *GPU gems 3*. Addison-Wesley Professional, first edition, 2007.

[204] Bradford Nichols, Dick Buttlar, and Jacqueline P. Farrell. *Pthreads Programming*. O'Reilly, 101 Morris Street, Sebastopol, CA 95472, 1998.

[205] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[206] Rishiyur Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann, May 2001.

[207] inc. Technical Information Center North American Aviation. *The N-body Problem: A Survey [of International Literature, 1940-1960*. North American Aviation, Space and Information Systems Division, 1963.

[208] Nvidia. *Fermi Compute Architecture Whitepaper*.

[209] Nvidia-Corporation. *Nvidia CUDA C Programming Guide*, 2012.

[210] Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[211] Andrew T Ogielski and Ingo Morgenstern. Critical behavior of three-dimensional Ising spin-glass model. *Physical Review Letters*, 54(9):928, 1985.

[212] Michael Oneppo. Hlsl shader model 4.0. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 112–152, New York, NY, USA, 2007. ACM.

[213] Lars Onsager. Crystal statistics. i. a two-dimensional model with an order-disorder transition. *Phys. Rev.*, 65:117–149, Feb 1944.

[214] Lars Onsager. The effects of shape on the interaction of colloidal particles. *Annals of the New York Academy of Sciences*, 51(4):627–659, 1949.

[215] Stan Openshaw and Ian Turton. *High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists, and Engineers.* Routledge, New York, NY, 10001, 1999.

[216] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[217] Simon Pabst, Artur Koch, and Wolfgang Straßer. Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *Computer Graphics Forum*, 29(5):1605–1612, 2010.

[218] David A. Padua, editor. *Encyclopedia of Parallel Computing*, volume 4. Springer, 2011.

[219] Michele Pagani and Paolo Tranquilli. Parallel reduction in resource lambda-calculus. In *APLAS*, pages 226–242, 2009.

[220] G. Parisi. Infinite number of order parameters for spin-glasses. *Phys. Rev. Lett.*, 43:1754–1756, Dec 1979.

[221] D. Parkinson. Parallel efficiency can be greater than unity. *Parallel Computing*, 3(3):261 – 262, 1986.

[222] Howard A. Peelle. To teach Newton's square root algorithm. *SIGAPL APL Quote Quad*, 5(4):48–50, December 1974.

[223] Sriram Pemmaraju and Steven Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica ®*. Cambridge University Press, New York, NY, USA, 2003.

[224] M. Picco. Weak randomness for large q-state Potts models in two dimensions. *Phys. Rev. Lett.*, 79:2998–3001, Oct 1997.

[225] V. P. Plagianakos, N. K. Nousis, and M. N. Vrahatis. Locating and computing in parallel all the simple roots of special functions using pvm. *J. Comput. Appl. Math.*, 133(1-2):545–554, August 2001.

[226] R B Potts. Some generalized order-disorder transformations. *Cambridge Philos. Soc. Math. Proc.*, 48:106–109, 1952.

[227] Renfrey B. Potts. Some generalized order-disorder transformation. In *Transformations,*

Proceedings of the Cambridge Philosophical Society, volume 48, pages 106–109, 1952.

[228] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. GPU accelerated Monte Carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468 – 4477, 2009.

[229] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. GPU accelerated Monte Carlo simulation of the 2d and 3d ising model. *J. Comput. Phys.*, 228(12):4468–4477, July 2009.

[230] Nikolay Prokof'ev and Boris Svistunov. Worm Algorithms for Classical Statistical Models. *Physical Review Letters*, 87(16):160601+, 2001.

[231] N.V Prokof'ev, B.V Svistunov, and I.S Tupitsyn. worm algorithm in quantum Monte Carlo simulations. *Physics Letters A*, 238(45):253 – 257, 1998.

[232] D. Raabe. Scaling Monte Carlo kinetics of the Potts model using rate theory. *Acta Materialia*, 48:1617–1628, 2000.

[233] H Rieger, L Santen, U Blasum, M Diehl, M Jünger, and G Rinaldi. The critical exponents of the two-dimensional Ising spin glass revisited: exact ground-state calculations and Monte Carlo simulations. *Journal of Physics A: Mathematical and General*, 29(14):3939, 1996.

[234] Florian Ries, Tommaso De Marco, Matteo Zivieri, and Roberto Guerrieri. Triangular matrix inversion on graphics processing unit. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 9:1–9:10, New York, NY, USA, 2009. ACM.

[235] Mike Roberts, Jeff Packer, Mario Costa Sousa, and Joseph Ross Mitchell. A work-efficient GPU algorithm for level set segmentation. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 123–132, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[236] P. E. Ross. Why cpu frequency stalled. *IEEE Spectr.*, 45(4):72–72, April 2008.

[237] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 72–83, 1999.

[238] Stefan Rybacki, Jan Himmelspach, and Adelinde M. Uhrmacher. Experiments with single core, multi-core, and GPU based computation of cellular automata. In *Proceedings of the 2009 First International Conference on Advances in System Simulation*, SIMUL '09, pages 62–67, Washington, DC, USA, 2009. IEEE Computer Society.

[239] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08,

pages 73–82, New York, NY, USA, 2008. ACM.

[240] Jesús Salas and Alan D. Sokal. Transfer matrices and partition-function zeros for antiferromagnetic Potts models. V. Further results for the square-lattice chromatic polynomial. *J. Stat. Phys.*, 135(2):279–373, 2009.

[241] Jess Salas and AlanD. Sokal. Transfer matrices and partition-function zeros for antiferromagnetic Potts models. I. General theory and square-lattice chromatic polynomial. *Journal of Statistical Physics*, 104(3-4):609–699, 2001.

[242] Jess Salas and AlanD. Sokal. Transfer matrices and partition-function zeros for antiferromagnetic Potts models VI. square lattice with extra-vertex boundary conditions. *Journal of Statistical Physics*, 144(5):1028–1122, 2011.

[243] Pedro V. Sander and Jason L. Mitchell. Progressive buffers: view-dependent geometry and texture lod rendering. In *Proceedings of the third Eurographics symposium on Geometry processing*, SGP '05, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.

[244] Joel L. Schiff. *Cellular Automata: A Discrete View of the World*. Wiley-Interscience, 2008.

[245] Angela Di Serio and María Blanca Ibáñez. Evaluation of a nearest-neighbor load balancing strategy for parallel molecular simulations in mpi environment. In *PVM/MPI*, pages 226–233, 2002.

[246] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[247] Takashi Shimokawabe, Takayuki Aoki, Chiashi Muroi, Junichi Ishida, Kohei Kawano, Toshio Endo, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. An 80-fold speedup, 15.0 tflops full GPU acceleration of non-hydrostatic weather model asuca production code. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[248] Robert Shrock. Exact Potts model partition functions on ladder graphs. *Physica A: Statistical Mechanics and its Applications*, 283(3-4):73, 2000.

[249] Robert Shrock and Shan-Ho Tsai. Exact partition functions for Potts antiferromagnets on cyclic lattice strips. *Physica A*, 275:27, 1999.

[250] Robert Shrock and Yan Xu. Exact results on Potts model partition functions in a generalized external field and weighted-set graph colorings. *Journal of Statistical Physics*, page 39, 2010.

[251] Justin R. Smith. *The design and analysis of parallel algorithms*. Oxford University Press, Inc., New York, NY, USA, 1993.

[252] A. D. Sokal. Monte Carlo methods in statistical mechanics: Foundations and new algorithms. Lecture notes, 1989.

[253] A. D. Sokal. Overcoming critical slowing-down: Where do we stand 23 years after Swendsen and Wang? In *104th Statistical Mechanics Conference*. Rutgers University, 2010.

[254] Alan D. Sokal. Bounds on the complex zeros of (di)chromatic polynomials and Potts-model partition functions. *Comb. Probab. Comput.*, 10:41–77, January 2001.

[255] Alan D. Sokal. Bounds on the complex zeros of (di)chromatic polynomials and Potts-model partition functions. *Comb. Probab. Comput.*, 10:41–77, January 2001.

[256] Alan D Sokal. The multivariate tutte polynomial (alias Potts model) for graphs and matroids. *Surveys in Combinatorics*, 327:173–226, 2005.

[257] Ramesh Subramonian. An o(log n) time common CRCW PRAM algorithm for minimum spanning tree. Technical Report UCB/CSD-92-673, EECS Department, University of California, Berkeley, Mar 1992.

[258] Martin Suess, Tughrul Arslan, Didier Keymeulen, Adrian Stoica, Ahmet T. Erdogan, and David Merodio, editors. *NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009, San Francisco, California, USA, July 29 - August 1, 2009*. IEEE Computer Society, 2009.

[259] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1):4:1–4:11, February 2009.

[260] R. H. Swendsen and J. S. Wang. Nonuniversal, critical dynamics in Monte Carlo simulations. *Phys. Rev. Lett.*, 58:86, 1987.

[261] Robert H. Swendsen and Jian-Sheng Wang. Replica Monte Carlo simulation of spin-glasses. *Phys. Rev. Lett.*, 57:2607–2609, Nov 1986.

[262] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in Monte Carlo simulations. *Phys. Rev. Lett.*, 58:86–88, Jan 1987.

[263] Noboru Tanabe, Nobuhiro Hori, Boonyasitpichai Nuttapon, and Hironori Nakajo. Preliminary evaluations for hybrid memory cube with gather functions using FPGA. *IPSJ SIG Notes*, 2012(6):1–10, 2012-03-19.

[264] David Taniar, Clement H. C. Leung, Wenny Rahayu, and Sushant Goel. *High-Performance Parallel Database Processing and Grid Databases*. Wiley Series on Parallel and Distributed Computing, 2008.

[265] Jose Juan Tapia and Roshan DSouza. Data-parallel algorithms for large-scale real-time simulation of the cellular Potts model on graphics processing units. In *2009 IEEE International Conference on Systems Man and Cybernetics*, pages 1411–1418. IEEE,

2009.

[266] José Juan Tapia and Roshan D'Souza. Parallelizing the cellular potts model on graphics processing units. *Computer Physics Communications*, 182(4):857–865, 2011.

[267] W. Tinney, V. Brandwajn, and S. Chan. Sparse Vector Methods. *IEEE Transactions on Power Apparatus and Systems*, PAS-104(2):295–301, February 1985.

[268] Pawel Topa and Pawel Mlocek. GpGPU implementation of cellular automata model of water flow. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, pages 630–639, Berlin, Heidelberg, 2012. Springer-Verlag.

[269] W. T. Tutte. A contribution to the theory of chromatic polynomials. *J. Math*, 6:80–91, 1954.

[270] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[271] Uzi Vishkin. A pram-on-chip vision (invited abstract). In *SPIRE*, page 260, 2000.

[272] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA*, pages 140–151, 1998.

[273] J. von Neumann. The general and logical theory of automata. In *Cerebral Mechanisms in Behaviour*. Wiley, 1951.

[274] Jian-Sheng Wang. Binary tree summation Monte Carlo simulation for Potts models. *Physica A: Statistical Mechanics and its Applications*, 321(1-2):9, 2002.

[275] M. Weigel. Connected-component identification and cluster update on graphics processing units. *Phys. Rev. E*, 84:036709, 2011.

[276] Martin Weigel. Simulating spin models on GPU. *Computer Physics Communications*, 2010.

[277] Martin Weigel. Performance potential for simulating spin models on GPU. *Arxiv preprint arXiv:1101.1427*, 2011.

[278] Martin Weigel. Simulating spin models on GPU. *Computer Physics Communications*, 182(9):1833–1836, 2011.

[279] Martin Weigel. Performance potential for simulating spin models on GPU. *Journal of Computational Physics*, 231(8):3064–3082, 2012.

[280] Martin Weigel. Simulating spin models on GPU: A tour. *International Journal of Modern Physics C*, 23(08), 2012.

[281] Martin Weigel and Des Johnston. Frustration effects in antiferromagnets on planar

random graphs. *Physical Review B*, 76(5):054408, 2007.

[282] D.J.A. Welsh and C Merino. The Potts model and the tutte polynomial. *J. Math. Phys.*, 43:1127–1149, 1 2000.

[283] Herbert S. Wilf. *Algorithms and Complexity*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2002.

[284] Barry Wilkinson and C Michael Allen. *Parallel programming, page 7*. Prentice hall New Jersey, 1999.

[285] Gerhard J. Woeginger. Exact algorithms for np-hard problems: a survey. In Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, editors, *Combinatorial optimization - Eureka, you shrink!*, pages 185–207. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[286] U. Wolff. Collective Monte Carlo updating for spin systems. *Physical Review Letters*, 62:361–364, 1989.

[287] Ulli Wolff. Collective Monte Carlo updating for spin systems. *Phys. Rev. Lett.*, 62:361–364, Jan 1989.

[288] Stephen Wolfram. *A new kind of science*. Wolfram Media Inc., Champaign, Ilinois, US, United States, 2002.

[289] F. Y. Wu. The potts model. *Rev. Mod. Phys.*, 54(1):235–268, January 1982.

[290] F. Y. Wu. The Potts model. *Reviews of Modern Physics*, 54(1):235–268, January 1982.

[291] Zhi Ying, Xinhua Lin, Simon Chong-Wee See, and Minglu Li. Gpu-accelerated dna distance matrix computation. In *Proceedings of the 2011 Sixth Annual ChinaGrid Conference*, CHINAGRID '11, pages 42–47, Washington, DC, USA, 2011. IEEE Computer Society.

[292] Rio Yokota, Lorena Barba, Tetsu Narumi, and Kenji Yasuoka. Scaling fast multipole methods up to 4000 GPUs. In *Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?*, ATIP '12, pages 9:1–9:6, Singapore, Singapore, 2012. A*STAR Computational Resource Centre.

[293] Rio Yokota and Lorena A. Barba. Fast n-body simulations on GPUs. *CoRR*, abs/1108.5815, 2011.

[294] Rio Yokota and Lorena A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *CoRR*, abs/1106.2176, 2011.

[295] Rio Yokota and Lorena A. Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science and Engineering*, 14(3):30–39, 2012.

[296] A. P. Young. *Spin Glasses and Random Fields*, chapter Theory of the Random Field Ising Model. World Scientific, 1997.

[297] Tjalling J. Ypma. Historical development of the Newton-Raphson method. *SIAM Rev.*, 37(4):531–551, December 1995.

[298] Qiang Yu, Michael Nosonovsky, and Sven K. Esche. On the accuracy of Monte Carlo Potts models for grain growth. *J. Comp. Methods in Sci. and Eng.*, 8:227–243, December 2008.

[299] Shuichi Yukita. Cellular automata in non-euclidean spaces. In *Proceedings of the 7th WSEAS International Conference on Mathematical Methods and Computational Techniques In Electrical Engineering*, MMACTE'05, pages 200–207, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[300] Lenka Zdeborová and Florent Krzakala. Phase transitions in the coloring of random graphs. *CoRR*, abs/0704.1269, 2007.

[301] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008.

Appendices

Appendix A

GPU-maps for triangular domains

There is a stage in the GPU computing pipeline where a grid of thread-blocks, or *space of computation*, is mapped to the problem domain. Normally, the space of computation is a k-dimensional bounding box (BB) that covers a k-dimensional problem. Threads that fall inside the problem domain perform computations and threads that fall outside are discarded, all happening at runtime. For problems with non-square geometry, this approach makes the space of computation larger than what is necessary, wasting many threads. Our case of interest are the two-dimensional triangular domain problems, alias *td-problems*, where almost half of the space of computation is unnecessary when using the BB approach. Problems such as the *Euclidean distance map* or *collision detection* are *td-problems* and they appear frequently as part of a larger computational problem. In this work, we study several mapping functions and their contribution to a better space of computation by reducing the number of unnecessary threads. We compare the performance of four existing mapping strategies; the *bounding box* (BB), the *upper-triangular mapping* (UTM), the *rectangular box* (RB) and the *recursive partition* (REC). In addition, we propose a *map* $g(\lambda)$, that maps any λ block to a unique location (i, j) in the triangular domain. The mapping is based on the properties of the lower triangular matrix and works in *block space*. The theoretical improvement I obtained from using $g(\lambda)$ is upper bounded as $I < 2$ and the number of unnecessary blocks is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Experimental results using different Nvidia Kepler GPUs show that for computing the Euclidean distance matrix $g(\lambda)$ achieves an improvement of up to 18% over the basic bounding box (BB) strategy, runs faster than UTM and REC strategies and it is almost as fast as RB. Performance results on shared memory 3D collision detection show that $g(\lambda)$ is the fastest map of all, and the only one capable of surpassing the brute force (BB) approach by a margin of up to 7%. These results help us realize that one of the main advantages of $g(\lambda)$ is the fact that it uses *block space mapping*, where coordinate values are small in magnitude and thread organization is not compromised, making the map stable in performance under different memory access patterns.

A.1 Introduction

GPU computing is without question a well established research area [4, 5, 200, 205, 216], since the release of general purpose computing platforms such as CUDA [209] and OpenCL [156]. In the CUDA GPU programming model there are three constructs that allow the execution of highly parallel algorithms; (1) thread, (2) block and (3) grid. Threads are the smallest elements and they are in charge of executing the instructions of the GPU kernel. A block is an intermediate structure that contains a set of threads organized in an Euclidean space. Blocks provide fast shared memory access as well as synchronization for all of its threads. The grid is the largest construct of the GPU and it is in charge of keeping all blocks together, spatially organized for the whole execution of the kernel. These three constructs play an important role when mapping the execution resources to the problem domain, and are also necessary for the GPU to schedule and distribute the work properly among its clusters of processing cores. OpenCL chooses different names for these constructs; (1) work-element, (2) work-group and (3) work-space, respectively.

For every GPU application, there is a stage in which the grid, or *space of computation*, is mapped to a problem domain for an eventual computation later on. This map can be defined as a function $f(x) : R^k \rightarrow R^p$ that transforms each k-dimensional point $x = (x_1, x_2, ..., x_k)$ of the grid to a unique p-dimensional point of the problem domain. In other words, $f(x)$ *maps the space of computation to the problem domain*. When the problem domain is simple in shape, rectangular or square grids are good choices because the bounding box perfectly matches the domain. Rectangular or square grids are the most used ones and they are characterized for using the bounding box strategy (BB) map, where $f(x) = x$. Other problems however do not match a box shaped domain because they have a different geometry. For instance, some 2D problems have a triangular shaped domain. We call these type of problems *triangular-domain-problems* or simply *td-problems*. Building a square grid for a *td-problem* is not the best choice because it generates unnecessary thread-blocks that would need to execute if-else conditionals to discard themselves, leading to a performance penalty. The scenario is illustrated in Figure A.1, where the unnecessary blocks are painted with a sad red face.

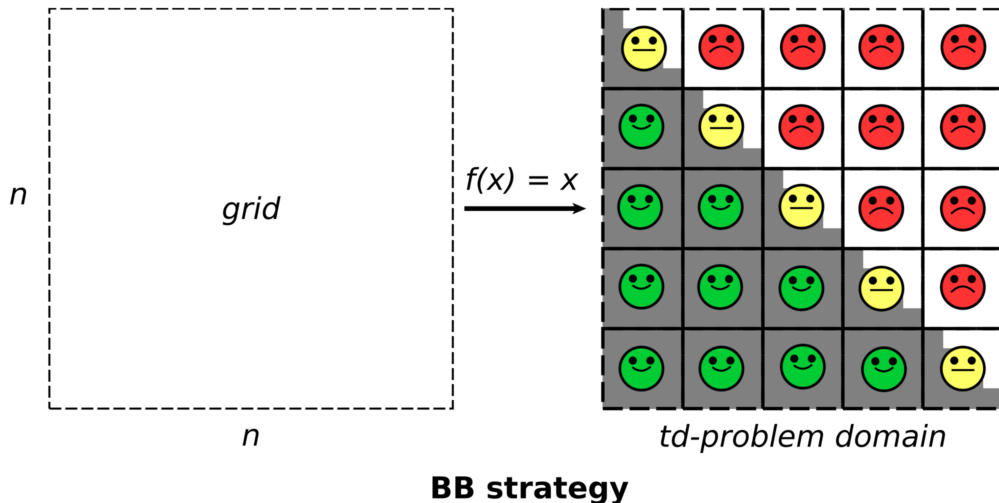

Figure A.1: The BB strategy is not the best choice for a *td-problem*.

In this work we address the lack of comparisons between different existing mapping strategies for *td-problems*, presenting performance results running the same tests under the same hardware. The idea is to establish a common benchmark for all mapping strategies, using three different GPUs, and measure their performance under three cases: (1) dummy kernel, (2) Euclidean distance matrix and (3) collision detection. We have chosen these three problems because they represent different scenarios; (1) very small kernel, (2) global memory patterns and (3) shared memory patterns. In addition to this benchmark, we present a new map $g(\lambda)$, that works in *block space* and uses the *lower-triangular matrix* properties. Map $g(\lambda)$ basically makes three contributions; (1) simpler implementation than the other strategies, (2) greater square root range than in UTM [13] and (3) uncompromised thread organization per block.

The rest of the Appendix is organized as follows: Section A.2 presents related work on GPU maps. Section A.3 covers the formulation of $g(\lambda)$. Details about its implementation and how we chose the fastest square root function are in Section A.4. In Section A.5 we present the performance results for all existing strategies using three different Nvidia Kepler GPUs; (1) GTX 765M, (2) GTX 680 and (3) Tesla K40. Finally, in Section A.6 we conclude by discussing the main points of our work.

A.2 Related Work

The problem of improving the space of computation for *td-problems* is indeed important because any contribution on the matter will eventually, by consequence, benefit every problem of this class. Many computational problems are in fact *td-problems*; Euclidean distance maps (EDM) [176, 186, 187], collision detection [13], graph traversal through adjacency matrices [155], Cellular Automata (e.g John Conway's Game of life [96]) in triangular domains, matrix inversion [234], LU/Cholesky decomposition [114], among others.

In the field of distance maps, Ying *et. al.* have proposed a GPU implementation for parallel computation of DNA sequence distances [291] which is based on EDM. In their work, the authors mention that the problem domain is indeed symmetric and they do realize that only the upper or lower triangular part of the interaction matrix requires computation. Li *et. al.* [176] have also worked on GPU-based EDMs on large data and have also identified the symmetry involved in the computation. However, in both works there is no mention of the mapping of the space of computation.

Jung *et. al.* [147] proposed in 2008 packed data structures for representing triangular and symmetric matrices with applications to LU and Cholesky decomposition [114]. The strategy is based on building a *rectangular box strategy* (RB) for accessing and storing a triangular matrix (upper or lower). Data structures become practically half the size with respect to classical methods based on the full matrix. Originally, this strategy is aimed at the memory structure of the matrix, but it can also be applied analogously to the space of computation.

In 2009, Ries *et. al.* contributed with a parallel GPU method for the triangular matrix inversion [234]. The authors identify that the space of computation indeed can be improved

by using a *recursive partition* (REC) of the grid, based on a *divide and conquer* strategy.

In 2012, Q. Avril *et. al.* proposed a GPU mapping function for collision detection based on the properties of the *upper-triangular map* [13]. The map, namely *UTM*, is a function $f(k) \rightarrow (a, b)$, where k is the linear index of a thread t_k and the pair (a, b) is a unique two-dimensional coordinate in the upper triangular matrix. The authors mention that for the square root computation they use Carmack's and Lomont's fast square root approximation (based on the Newton-Raphson approximation algorithm [222]) for speeding up the mapping function. Approximation errors are fixed by using two conditionals statements inside the kernel. Their square root implementation $sqrt(x)$ is accurate for $x \in [0, 100M]$, which according to their map function, would refer to problems in the range $N \in [0, 3000]$.

Up to date, there has not been a dedicated comparison of the different strategies proposed for improving the space of computation in *td-problems*. In the best case we can find a comparison between Avril's map and the BB strategy [13] where the authors report a speedup of almost $2X$, however the filtering is done by thread ID instead of by block coordinates, making the BB map slower.

The triangular map can still be improved to work for problem sizes of $N < 30000$ and also to be able to use the GPU shared memory, through a blockwise formulation.

A.3 The blockwise triangular map

A.3.1 Formulation

It is important first of all to distinguish between two types of mappings that can be performed on the GPU; (1) threadwise mapping and (2) blockwise mapping. Threadwise mapping is where each thread uses its own unique index as the parameter for the mapping function, just as in the UTM map [13]. In blockwise mapping threads use their block index to map all of them to a specific location, followed by a local shift according to the relative position in the block. We have chosen to continue with blockwise mapping and its advantages are explained later on.

Let N be the problem size (assuming data-parallel elements), A a *td-problem* of size $N(N+1)/2$, $n = \lceil N/\rho \rceil$ the number of blocks needed to cover the data along one dimension and ρ the number of threads per block per dimension, or *dimensional block-size* (for simplicity, we assume all dimensions are the same). A BB strategy would simply build a square grid, namely G_{BB}, of nxn blocks and put conditional instructions to cancel the computations outside the problem domain. A finer analysis tells that $n(n+1)/2$ blocks are sufficient to

cover the problem domain of A:

$$A = \begin{vmatrix} 0 & & & & \\ 1 & 2 & & & \\ 3 & 4 & 5 & & \\ ... & ... & ... & ... & \\ \frac{n(n-1)}{2} & \frac{n(n-1)}{2}+1 & ... & ... & \frac{n(n+1)}{2}-1 \end{vmatrix} \tag{A.1}$$

Note : for the rest of the paper we will refer to our mapping approach as **LTM** for **lower triangular mapping**, or simply $g(\lambda)$.

The idea is to build a two-dimensional balanced grid G_{LTM} that covers the lower-triangular matrix, just using $n(n+1)/2$ blocks. By balanced grid, we mean $n' = \lceil\sqrt{n(n+1)/2}\rceil$ blocks per dimension. Figure A.2 illustrates G_{LTM} and how it is smaller than G_{BB} from Figure A.1, while still providing the necessary number of blocks to cover the problem domain. The

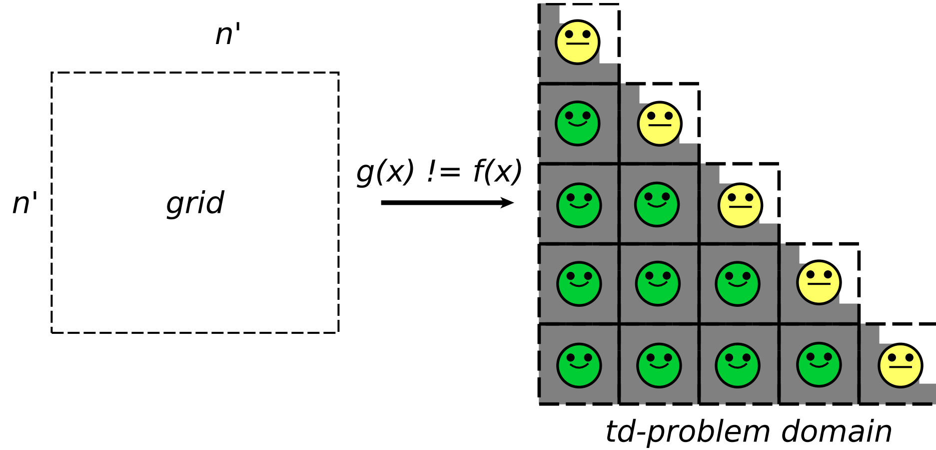

Figure A.2: The LTM strategy uses just the necessary number of blocks to cover the problem domain.

result is a reduction from $n(n-1)/2 \in \mathcal{O}(n^2)$ to $n/2 \in \mathcal{O}(n)$ unnecessary blocks.

The next step is to formulate $g(\lambda) = (\mathrm{i}, j)$ where (i, j) are the coordinates in problem space and λ is the index of block $B_{x,y}$ computed as $\lambda = x + yn'$ in *block space.*

Theorem A.1 *For any block $B_{x,y}$ with $\lambda = x + yn'$, its mapping function $g(\lambda)$ is:*

$$g(\lambda) = (\mathrm{i}, j) = \left(\left\lfloor\sqrt{\frac{1}{4} + 2\lambda} - \frac{1}{2}\right\rfloor, \lambda - \mathrm{i}(\mathrm{i}+1)/2\right) \tag{A.2}$$

. The λ index can be expressed as a triangular number with an unknown summation limit x.

$$\lambda = \sum_{k=1}^{x} k \tag{A.3}$$

The index of the far left block that lies on the same row of the λ-th block corresponds to the sum in the range $[1, \mathrm{i}]$. Similarly, the index of the far left block of the $(\mathrm{i}+1)$-th row is a sum

in the range $[1, \mathrm{i}+1]$. For all λ indices under the same row i, the range of the summation lies in a range $[1, \mathrm{i}+\varepsilon]$, where $\varepsilon < 1$:

$$\sum_{k=1}^{\mathrm{i}} k \leq \lambda = \sum_{k=1}^{\mathrm{i}+\varepsilon} k < \sum_{k=1}^{\mathrm{i}+1} k \tag{A.4}$$

Therefore $x \in \mathbb{R}$ and $\mathrm{i} = \lfloor x \rfloor$. Since $\sum_{k=1}^{x} k = x(x+1)/2$, x is found by just solving a second order equation with coefficients $a = 1$, $b = 1$ and $c = -2$:

$$x^2 + x - 2\lambda = 0 \tag{A.5}$$

The solution of interest is:

$$x = \frac{\sqrt{1+8\lambda} - 1}{2} = \sqrt{1/4 + 2\lambda} - 1/2 \tag{A.6}$$

And the i coordinate of the λ-th block is computed as:

$$\mathrm{i} = \lfloor x \rfloor = \left\lfloor \sqrt{1/4 + 2\lambda} - 1/2 \right\rfloor \tag{A.7}$$

Coordinate j is the distance from the λ-th block to the left most block in the same row:

$$j = \lambda - \frac{\mathrm{i}(\mathrm{i}+1)}{2} \tag{A.8}$$

$\square$

If the diagonal is not needed, then $g(\lambda)$ becomes:

$$g(\lambda) = (\mathrm{i}, j) = \left(\left\lfloor \sqrt{\frac{1}{4} + 2\lambda} + \frac{1}{2} \right\rfloor, \lambda - \mathrm{i}(\mathrm{i}+1)/2 \right) \tag{A.9}$$

When comparing LTM and its closest counterpart UTM [13], we identify three improvements: (1) $g(\lambda)$ uses fewer floating point operations than in UTM since it uses the lower-triangular mapping. (2) LTM maps blocks and not threads as in UTM. Since $g(\lambda)$ is a map of blocks and the number of blocks is $n = N/\rho$, the square root gives smaller approximation errors, allowing accurate computation for larger values of N. (3) Thread organization is not compromised in LTM, while in UTM it is.

A.3.2 Bounds on the improvement factor

The reduction of unnecessary blocks from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ may suggest that the improvement could reach in the best case a $2X$ factor. For this to be possible, one would need to measure just the mapping time, so that necessary and unnecessary blocks do the same jobs, and assume that the mapping function $g(\lambda)$ is as cheap as in the BB strategy. In the following analysis we analyze the improvement factor considering a more realistic scenario where $g(\lambda)$ has a higher cost than the BB map, due to the square root computation involved.

The LTM strategy depends strongly on the square root which in theory is asymptotically $O(M(n))$ [297] where $M(n)$ is the cost of multiplying two numbers of n digits. Considering that real numbers are represented by a finite number of digits (*i.e.,* floating point numbers with a maximum of m digits), then all basic operations cost a fixed amount of time, leading to a constant cost $M(m) = C_s \in \mathcal{O}(1)$. All other computations are elemental arithmetic operations and will be taken as an additional cost of $C_a \in \mathcal{O}(1)$. The LTM strategy has a cost of $\tau = C_s + C_a = \mathcal{O}(1)$ for each mapped thread. On the other hand, the BB strategy checks for each thread if $B_y <= B_x$ in order to continue or not, also leading to a constant cost of $\beta \in \mathcal{O}(1)$.

For this particular case, asymptotic analysis does not give precise information about the improvement factor, since both the LTM and BB strategies lie in the same complexity order (*i.e.,* $n(n+1)/2 \in \mathcal{O}(n^2)$) therefore the improvement should be a constant factor. We proceed to a finer non-asymptotic analysis in order to find the constants involved in the improvement factor.

Let $|G_{BB}|$ and $|G_{LTM}|$ be the number of blocks for the BB and LTM strategies, respectively, and ρ the number of threads per block per dimension as mentioned earlier. It is indeed evident that β is cheaper than τ, therefore $\tau = k\beta$ with a constant $k \geq 1$. The improvement factor I can be obtained by dividing the total cost of BB by LTM across their entire grids:

$$I = \frac{\beta|G_{BB}|\rho^2}{\tau|G_{LTM}|\rho^2} = \frac{\beta n^2}{\tau n(n+1)/2} = \frac{2\beta n^2}{\tau n^2 + \tau n} \tag{A.10}$$

As shown in (A.10), the improvement does not depend on the threads, but instead, on the blocks. For large n we have:

$$I = \frac{2\beta n^2}{\tau n^2 + \tau n} \approx \frac{2\beta}{\tau} \tag{A.11}$$

If $\tau \geq 2\beta$, performance is equal to or worse than that of BB. A real improvement is achieved when:

$$\beta \leq \tau < 2\beta \tag{A.12}$$

By using the relation $\tau = k\beta$ in (A.11) we get that:

$$I \approx 2/k \tag{A.13}$$

Since $k > 1$, the range of I is:

$$0 < I < 2 \tag{A.14}$$

Any value in the range $1 < I < 2$ means an improvement in performance and a value in the range $0 < I < 1$ will mean a slowdown respect to the BB strategy. Constant k can be interpreted as the cost and overhead of the mapping function. A value of $k \approx 1$ means that the maximum possible improvement is achieved; $I_{max} \approx 2$, under large n. In practice, a value of $k \approx 1$ is too optimistic and would not occur. Our hypothesis is that actual hardware could give a value in the range $1.5 \leq k < 2.0$ which would correspond to $1.00 < I \leq 1.33$. Any value of $k \geq 2$ will lead to no improvement at all, resulting in slower performance than the BB strategy. It is important to put emphasis on the fact that C_a (arithmetic operations) will not have much room for optimization as C_s. Therefore, getting the maximum possible value of I will finally depend on how fast the square root can be.

A.4 Implementation of LTM

In this section we choose the best of three square root implementations to be used in the LTM map. We also explain in general terms the idea behind the other mapping strategies we have also implemented for later comparison.

A.4.1 Choosing the best square root

The performance of the LTM strategy depends strongly on how fast the computation of index i is. More precisely, the computation of the square root as mentioned earlier. We made three versions of the LTM map, using different square root implementations, and computed the improvement factor with respect to the BB strategy. The first one, named *LTM-X*, uses the default $sqrtf(x)$ function from CUDA and it is the simplest one.

The second implementation of LTM, named *LTM-N*, computes the square root by using three iterations of the Newton-Raphson method [222, 297]. We use the implementation of Carmack and Lomont. This implementation has proved to be effective for applications that allow small errors. The initial value used is the magic number '0x5f3759df' (this initial guess became known when 'Id Software' released Quake 3 source code back in the year 2005). We have found that adding a constant of $\varepsilon = 10^{-4}$ to the result of the square root can fix approximation errors in the range $N \in [0, 30720]$.

The third implementation, named *LTM-R*, uses the hardware implemented reciprocal square root, $rsqrtf(x)$:

$$\sqrt{x} = \frac{x}{\sqrt{x}} = x \cdot rsqrtf(x) \tag{A.15}$$

In terms of simplicity, *LTM-R* is similar to *LTM-X*, the only difference is that it adds $\varepsilon = 10^{-4}$ at the end to fix approximation errors, just like in *LTM-N*.

We measured the improvement factor of each implementation with respect to the BB strategy by running a dummy kernel that computes the i, j indices and writes the sum i+j to a constant location in memory. It is necessary to perform at least one memory access otherwise the compiler can optimize the code removing part of the cost of mapping. Figure A.3 shows the improvement factor as $I = BB/LTM$ using the three different implementations, running on three different Nvidia Kepler GPUs; GTX 680, GTX 765M and Tesla K40. We have included the BB map for reference, as an horizontal black line at $I = 1$. On the right side of Figure A.3 we compare the number of unnecessary blocks between BB and LTM. From the results, we observe that *LTM-X* is slower than BB when running on the GTX 680 and GTX 765M, achieving $I_{680} \approx 0.87$ and $I_{765M} \approx 0.83$, respectively. For the same two GPUs, *LTM-N* achieves an improvement of $I_{680} \approx 1.025$ and $I_{765M} \approx 0.96$ which is practically the performance of BB. On the other hand, *LTM-R* achieves improvements of $I_{680} \approx 1.15$ and $I_{765M} \approx 1.1$. From these results, we observe that using the inverse square root is the best option for the GTX 680 and GTX 765M. For the case of the Tesla K40, we observe that all three implementations achieve an improvement of $I_{K40} \approx 1.08$, allowing to choose *LTM-X* without any performance penalty.

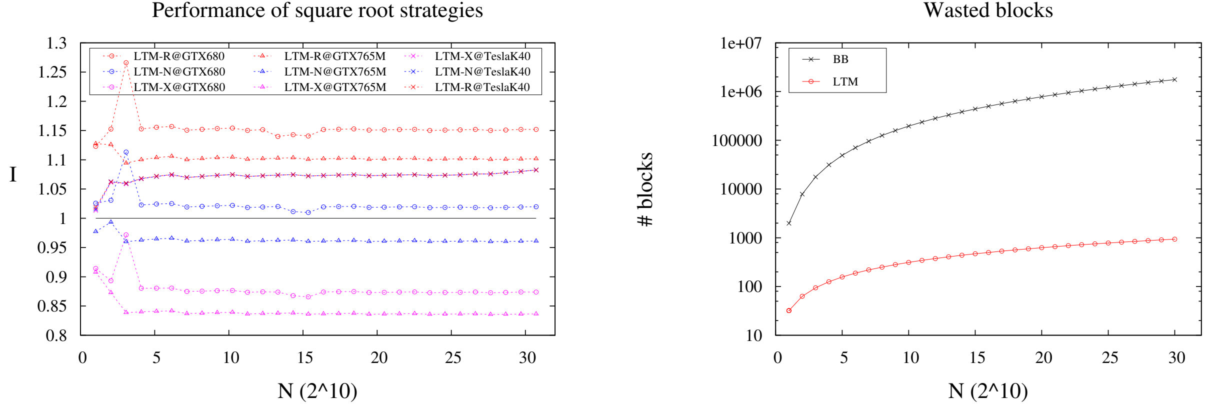

Figure A.3: On the left, the performance of the different square root strategies and on the right the number of unnecessary blocks using $\rho = 16$.

A.4.2 Implementing the other strategies

We have implemented the other mapping strategies; BB, RB, REC and UTM following the details provided by the authors [13, 147, 234]. To each implementation we added the following restriction: the map cannot use any memory that grows as a function of N. This means no auxiliary array such as lookup tables are allowed; constants are allowed though. We have put this restriction in order to guarantee that GPU memory is dedicated to the application problem.

For the *bounding box* (BB) strategy, blocks above the diagonal are discarded at runtime, without needing to compute a thread coordinate. This is done by checking if $B_x > B_y$ for every thread. For the rest of the threads, the coordinate is computed. The condition i $> j$ is still performed to discard threads on blocks where $B_x = B_y$. This implementation of BB is faster than computing the thread coordinate first and filtering afterwards.

The *rectangular box* (RB) strategy is based on the work of Jung *et. al.* [147]. This method takes the sub-triangular portion of the threads where $t_x > N/2$, rotates it CCW and places it above the diagonal to form a rectangular grid (see Figure A.4). The original work was actually a memory packing technique aimed at the data structures and not a GPU map, however the main idea of the strategy is suitable as a mapping function. We remove the lookup texture used in the original implementation and perform the coordinate mapping arithmetically at runtime. All threads below the diagonal just need to map to i $= t_y - 1$, while j remains the same. Threads in or above the diagonal map to i $= N - t_y - 1$ and $j = N -$ i $- 1$. This mapping is for even values of N. The case of odd N is the same idea but partitioning at $\lfloor N/2 \rfloor$. A remarkable feature of the RB map is that the number of unnecessary blocks is asymptotically $\mathcal{O}(1)$.

The *recursive partition* (REC) strategy was proposed for the GPU [234] for matrix inversion. In this method, the size of the problem is defined as $N = m2^k$ where k and m are positive integers and m is a multiple of ρ (the block-size). The idea is to do a binary *bottom-up* recursion of k levels, similar to *merge-sort* (see Figure A.4). At each level, a grid of blocks is launched for parallel execution, a total of k times. This method requires an

additional pass for computing the blocks at the diagonal. More details of how the grid is built and how blocks are distributed are well explained in [234]. In the original work, the mapping of blocks to their respective locations at each level is achieved by using a lookup table stored in constant memory. In this work, we do the mapping at runtime as in RB.

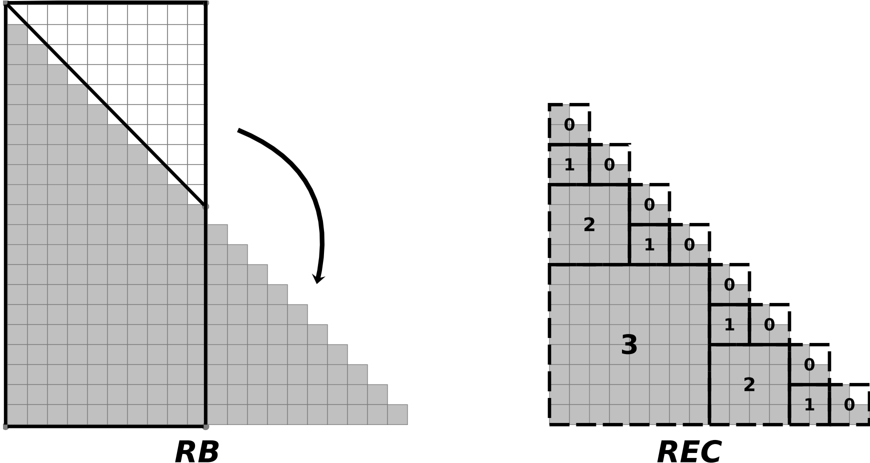

Figure A.4: The RB and REC strategies.

The *upper-triangular mapping* (UTM) was proposed by Avril *et. al.* [13] for performing efficient collision detection on the GPU. This method is very similar to LTM; given a problem size N and a thread index k, its unique pair (a, b) is computed as $a = \lfloor \frac{-(2n+1)+\sqrt{4n^2-4n-8k+1}}{-2} \rfloor$ and $b = (a+1) + k - \frac{(a-1)(2n-a)}{2}$. The UTM strategy uses the idea of mapping to the upper-triangular matrix without the diagonal. Mapping to the upper-triangular matrix still allows solving lower-triangular shaped domains, and *vice-versa* via transposition. An important difference between UTM and LTM is that UTM uses linear thread organization and the map works in *linear thread space*, where very large numbers need to be computed in the square root. On the other hand LTM uses a two-dimensional thread organization and the map works in *linear block space*, making the square root to work with smaller numbers than what UTM uses.

A.5 Experimental results

Our experimental design consists of measuring the performance of LTM and compare it against all existing strategies; *bounding box* (BB), *upper-triangular mapping* (UTM) [13], *rectangular box* (RB) [147] and the *recursive partition* (REC) [234]. We checked that the outputs for each strategy were always correct. Three tests are performed to each strategy; (1) the dummy kernel, (2) EDM and (3) Collision detection. Test (1) just writes the (i, j) coordinate into a fixed memory location. The purpose of the dummy kernel is to measure just the cost of the strategy and not the cost of the application problem. Test (2) consists of computing the Euclidean distance matrix (EDM) using four features. The purpose is to measure what is the performance of all strategies when solving a global memory based problem. Test (3) consists of performing collision detection of N spheres with random radius

inside a unit box. The goal of this last test is to measure the performance of the kernels using a shared memory approach.

The reason why we have chosen these tests is because they are simple enough to study their performance from a GPU map perspective and use different memory access paradigms such as global memory and shared memory. Based on these arguments, we expect that the performance results obtained by the three tests will give insights on what would be the behavior for more complex problems that fall into one of the two memory access paradigms. Furthermore, we have decided to run the tests on three different GPUs in order to obtain metrics general enough that can provide performance patterns. The details of the hardware and the maximum number of simultaneous blocks (*sblocks*) for each GPU is listed in table A.1. Performance results for the dummy kernel, EDM and collision detection in 1D/3D are

Table A.1: Hardware used for experiments.

Component	Description
CPU	Intel(R) Core(TM) i7-3770K @ 3.50GHz
RAM	32GB DDR3 1333MHZ
GPU_1	Geforce GTX 680 (GK104, 2GB, 1536 cores, sblocks = 128)
GPU_2	Geforce GTX 765M (GK106, 2GB, 768 cores, sblocks = 80)
GPU_3	Tesla K40 (GK110, 12GB, 2880 cores, , sblocks = 240)
API	CUDA 5.5

presented in Figure A.5. On each graphic we plot the performance of all four strategies as different dashed line colors, while the point symbol corresponds to the different GPU chosen for that test. The performance of each mapping strategy is given in terms of its improvement factor I with respect to the BB strategy (*i.e.,* the black and solid horizontal line fixed at $I = 1$). Values that are located above the horizontal line represent actual improvement, while curves that fall below the horizontal line represent a slowdown with respect to the BB strategy. For the dummy kernel test we observe that the RB strategy is the fastest one achieving up to 33% of improvement with respect to BB when running on the GTX 765M. LTM comes in the second place achieving a stable improvement of 18% when running on the GTX 680. The REC and UTM strategies performed slower than BB for the whole range of N. We note that this test running on the Tesla K40 does not provide any clear performance difference between the mapping strategies once $N > 15000$, since they all converge to a 7% of improvement with respect to BB.

For the EDM test, we observe that RB is again the fastest map achieving an improvement of up to 28% with respect to BB when running on the Tesla K40 GPU. In second place comes LTM with a stable improvement of 18% and third the REC map with an improvement that reaches up to 12% for the largest N. The performance of the UTM strategy is lower than BB and unstable for all GPUs. At this point, we have to consider that UTM was designed to work in the range $N \in [0, 3000]$, where it actually does perform better than BB offering up to 4% of improvement over BB. For this test, performance differences did manifest for all GPUs, even on the Tesla K40.

For the collision detection test we have included two cases; 3D and 1D collision detection. From all the map strategies, only LTM manages to perform better than BB, offering an

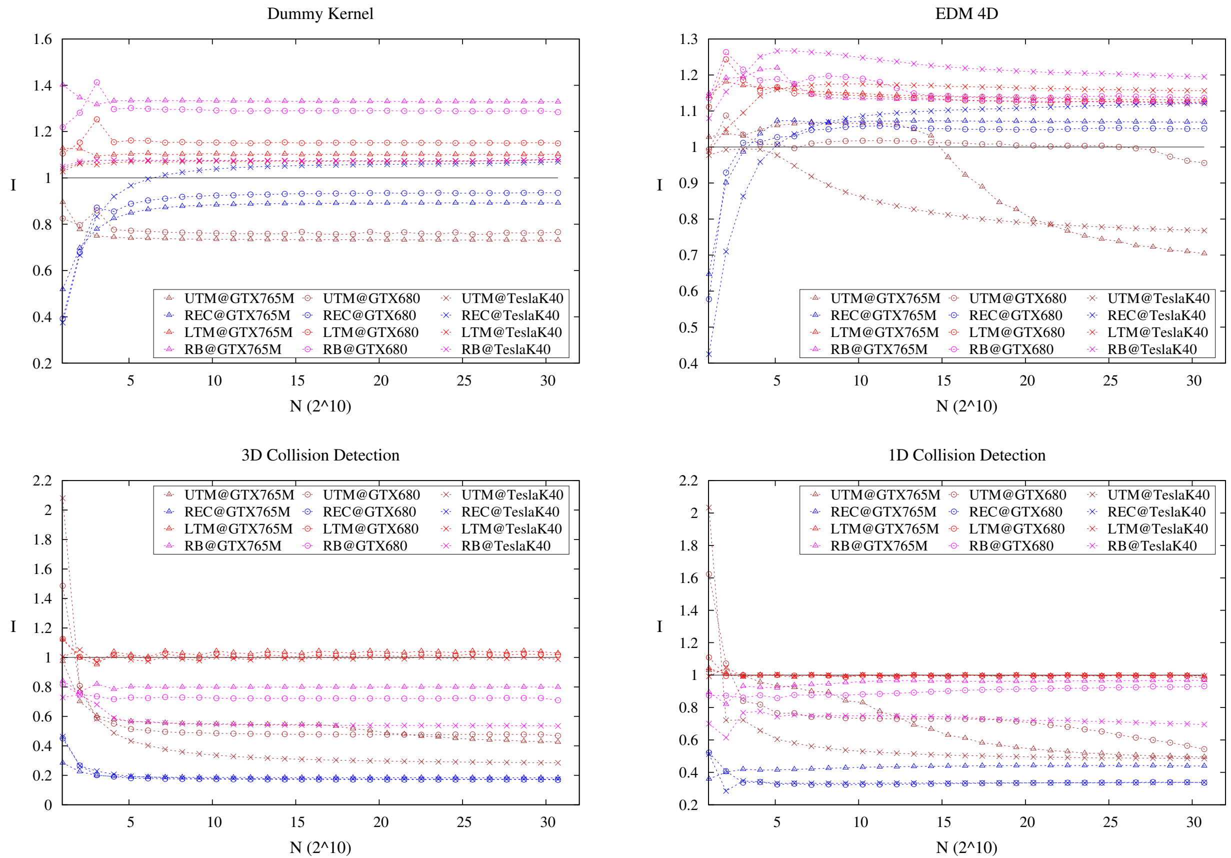

Figure A.5: Improvement factors for the dummy kernel, Euclidean distance matrix and collision detection.

improvement of up to 7%. Indeed, the performance scenario changes drastically in the presence of a different memory access pattern such as shared memory; the RB strategy, which was the best in global memory, now performs slower than BB. The case is similar with the REC map which now performs much slower. It is important to mention that the UTM map is the only strategy that cannot use a 2D shared memory pattern because the mapping works in *linear thread space*. At low N, UTM achieves a 100% of improvement because of the different memory approach used. However as N grows, its performance is over passed by the rest of the strategies that use shared memory. For the case of 1D collision detection, results are not so beneficial for the mapping strategies and in the case of LTM performance is in the limit of not becoming an improvement. The 1D results show that in low dimensions the benefits of a mapping strategy can be not as good as in high dimensions.

A.6 Discussion

We have studied the benefits of GPU maps for triangular domain problems, alias *td-problems*. These maps allow the use of a smaller space of computation compared to the bounding box strategy (BB) in order to solve a problem. By using a smaller space of computation, the number of unnecessary threads is reduced and the GPU kernel can execute faster. Our proposed GPU map, namely LTM for *lower triangular map*, uses a $g(\lambda)$ function to map the

space of computation to any *td-problem*. The main advantage of LTM is that it works in *block space*, not affecting thread organization and allowing problem sizes to be at least in the range of $N \in [0, 30720]$, which is ten times the range achieved in the upper triangular map (UTM) [13]. It is important to mention that the problem size must be large enough to generate more blocks than the number of blocks the GPU can handle in parallel. Assuming a warp of 32 threads, a value of $N \geq 800$ already generates more than double the number of blocks in parallel for any of the GPUs we used. For smaller values of N, the BB strategy will still remain useful.

When comparing all mapping strategies under the same performance tests, we found that the rectangular box (RB) and LTM are the best mapping strategies for a global memory based problem such as EDM, achieving 28% and 18% of improvement, respectively. The recursive partition (REC) strategy managed to provide up to 12% of improvement for the global memory problem. The UTM strategy achieved less performance than the BB method and could not provide exact results when $N > 3000$. For shared memory collision detection we found that no strategy but LTM managed to perform better than BB, by 7%. The reason why LTM could still provide an improvement under a different memory access pattern is because LTM maps in *block space* thus it does not compromise thread organization. The benefits of *block space mapping* are better manifested under shared memory algorithms or thread coarsening techniques [210].

The implementation of the LTM strategy is extremely short in code and totally detached from the problem, making it easy to adopt. When choosing the best implementation of square root, we found that the reciprocal square root was the most convenient option for GK104 and GK106 GPUs such as the GTX 680 and GTX 765M, respectively. When running the LTM map on Nvidia's Tesla K40 GPU, which is a GK110 architecture, we found that the computation of the square root can be performed just using the default function $sqrtf(x)$ without any performance penalty. This makes the implementation of LTM even simpler.

The performance of mapping strategies for *td-problems* still varies depending on the GPU used and on the way memory is accessed, making it hard to choose the best map. Nevertheless, we have found some important performance patterns among our results; the RB and LTM maps are good for global memory problems while the LTM map is the only one good for a shared memory based problem. This scenario puts LTM in advantage over the other maps, since it is the only strategy that can work for both global and shared memory based problems.