

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Mention : Informatique
École doctorale MATISSE

présentée par

Guillaume Chapuis

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique et systèmes aléatoires / Inria Rennes

Exploiting parallel features of modern computer architectures in bioinformatics

Applications to genetics, structure comparison and large graph analysis.

Thèse soutenue le 18 décembre 2013
devant le jury composé de :

Nouredine Melab
Professeur / *rapporteur*
Gunnar Klau
Professeur / *rapporteur*

Frédéric Guinand
Professeur / *examinateur*
Patrice Quinton
Professeur / *examinateur*

Rumen Andonov
Professeur / *co-directeur de thèse*
Dominique Lavenier
DR CNRS / *directeur de thèse*

Contents

1 Introduction **7**

2 Background **11**
2.1 Introduction . . . 11
2.2 Overview of CPU and GPU architectures . . . 12
2.3 Hierarchy of computational problems . . . 13
2.4 Coarse-grain parallelism . . . 14
2.4.1 Multicore CPU programming . . . 15
2.4.2 GPU programming . . . 17
2.5 Fine-grain parallelism . . . 21
2.5.1 Vector instructions . . . 22
2.5.2 Bit-level parallelism . . . 23
2.5.3 Instruction-level parallelism . . . 23
2.6 Parallelization in bioinformatics . . . 23
2.6.1 Sequence comparison . . . 24
2.6.2 Structure comparison . . . 26
2.7 Conclusion . . . 28

3 GPU accelerated QTL mapping **29**
3.1 Introduction to QTL mapping . . . 29
3.2 Methods and algorithms . . . 32
3.2.1 Linkage Analysis . . . 32
3.2.2 Linkage Disequilibrium and LDL Analyses . . . 33
3.2.3 Thresholds detection . . . 33
3.2.4 Algorithms for QTL detection . . . 34
3.3 GPU implementation . . . 36
3.3.1 Mapping computations on the GPU . . . 37
3.3.2 Optimizing GPU memory usage . . . 38
3.3.3 Reducing CPU/GPU transfers . . . 39
3.3.4 Optimizing homoskedastic analyses . . . 39
3.4 Experiments and results . . . 40
3.4.1 Execution times . . . 40
3.4.2 Speedups . . . 41
3.5 Conclusion . . . 43

4 Efficient Multi-GPU Computation of All-Pairs Shortest Paths **47**

4.1 Introduction . . . 47
4.2 Related Work . . . 50
4.3 Algorithm details . . . 51
4.3.1 Overview . . . 52
4.3.2 Step 1: Graph decomposition . . . 52
4.3.3 Step 2: Computing distances within each graph component . . 52
4.3.4 Step 3: Computing distances in the boundary graph . . . 54
4.3.5 Step 4: Distances between non-boundary vertices . . . 55
4.4 Implementation . . . 57
4.4.1 Data organization . . . 57
4.4.2 Work analysis . . . 59
4.4.3 Parallel implementation . . . 61
4.4.4 Memory limitations . . . 62
4.5 Results and perspectives . . . 62

5 Parallel seed-based approach to protein structure similarity detection 67
5.1 Introduction . . . 67
5.1.1 Alignment graphs . . . 68
5.1.2 Relation to protein structure comparison . . . 69
5.1.3 Measures for protein alignments . . . 69
5.2 Methods . . . 70
5.2.1 Our approach . . . 70
5.2.2 Overview of the algorithm . . . 70
5.2.3 Seed enumeration . . . 71
5.2.4 Seed extension . . . 72
5.2.5 Extension filtering . . . 73
5.2.6 Guarantees on resulting alignments' $RMSD$ scores . . . 74
5.2.7 Result ranking . . . 75
5.2.8 k-to-k alignments . . . 76
5.2.9 Graph splitting . . . 77
5.3 Parallelism . . . 78
5.3.1 Overview of the implemented parallelism . . . 78
5.3.2 Coarse-grain parallelism . . . 78
5.3.3 Fine-grain parallelism . . . 80
5.4 Results and perspectives . . . 81

6 Conclusions and perspectives 85
6.1 Conclusions . . . 85
6.2 Perspectives . . . 86
6.2.1 QTL detection . . . 86
6.2.2 Large graph analysis . . . 86
6.2.3 Protein structure comparison . . . 87
6.2.4 General remarks . . . 87
6.3 Acknowledgments . . . 88

List of Figures

2.3.1 Rough classification of computational problems from a parallelism point of view. 15
2.5.1 Differences between scalar and vector instructions. 22
2.5.2 Bit parallel set intersection. 23
2.5.3 Example of data dependencies. 24

3.1.1 Repartition of markers M1/M2 and N1/N2 on alleles Q1 and Q2. . . 30
3.2.1 Description of a contingency matrix used for computations at each genome position and for each simulation. 36
3.3.1 Example of gridification on the GPU. 37
3.4.1 Evolution of the execution time with respect to the number of simulations. 41
3.4.2 Evolution of the execution time with respect to the number of half-sib families. 42
3.4.3 Evolution of the execution time with respect to the number of genome positions. 42
3.4.4 Speedup with respect to the number of simulations. 43
3.4.5 Speedup with respect to the number of half-sib families. 44
3.4.6 Speedup with respect to the number of genome positions. 44

4.3.1 Illustration to the proof of Lemma Theorem 1. The shaded region illustrates a component C with the subpath $q = (x_{b_i}, x_{b_i+1}, \ldots, x_{b_{i+1}})$ of p inside it. 55
4.3.2 Illustration to the proof of Lemma Theorem 2. Note that while in the figure both v_i and v_j are non-boundary, the proof does not make such an assumption. 57
4.4.1 Adjacency matrix after reordering of the vertices. Vertices from the same component are stored contiguously starting with boundary vertices (in red). 58
4.4.2 The boundary matrix, here in red, is scattered over the adjacency matrix. Step 3 consits in reconstituting the boundary matrix and computing shortest distances. 59
4.4.3 Computations associated to each non-diagonal sub-matrix uses data from 2 diagonal sub-matrices and part of the non-diagonal sub-matrix itself. Computations are similar to matrix multiplications. 60

4.5.1 Evolution of run times with respect to the number of vertices. Two implementations are compared: our implementation using external memory and the GPU Dijkstra implementation from [OATLGE13]. Computations were run using two GPUs on a single cluster node. . . 63
4.5.2 Evolution of speedups with respect to the number of GPUs. The ideal scaling line is given as a reference. 64
4.5.3 Evolution of run times with respect to the number of vertices. Three implementations are compared: our two implementations - with and without using external memory - and a distributed Dijkstra implementation referred to as CPU Dijkstra. All computations were run on 64 cluster nodes. 65

5.1.1 Example of an alignment graph used here to compare the structures of two proteins. The presence of an edge between vertex $(1, 1)$ and vertex $(3, 2)$ means that the distance between atoms 1 and 2 of protein 1 is similar to the distance between atoms 1 and 3 of protein 2. The clique $(2, 1)$ $(3, 2)$ $(4, 3)$ indicates that RMSD of structures $(2, 3, 4)$ and $(1, 2, 3)$ is less than 2τ. 68
5.2.1 Example of symmetry issues. Even though, vertex $v_l = (L, L')$ belongs to the extension of $seed(v_i, v_j, v_k)$, points L and L' lie on different sides of the plane defined by optimally superimposed triangles IJK and $I'J'K'$. 73
5.2.2 Illustration of the guarantee on the similarity of internal distances between two pairs of atoms $v_l = (L, L')$ and $v_m = (M, M')$, here represented in yellow, added to a seed (v_i, v_j, v_k) represented in blue. Dashed lines represent internal distances, the similarity of which is tested in the alignment graph. 75
5.2.3 Example of 1-to-1 alignments retrieved from a k-to-k alignments. In red, a 1-to-1 alignment of optimal length but sub-optimal RMSDc and in green a 1-to-1 alignment of optimal length and optimal RMSDc. Solving the assignment problem on this graph yields the green alignment. 77
5.3.1 Overview of the implemented parallelism. 79
5.3.2 Bit vector representation of the neighbors of vertex v_i in an alignment graph $G(V, E)$. In this example, v_j unlike v_k is a neighbor of v_i. . . . 80
5.3.3 Intersection of neighbors of vertex v_i and vertex v_j. 81
5.4.1 These two proteins are both composed of two similar domains - named A and B for 4clna (left), and C and D for 2bbma (right). These domains are separated by a a flexible bridge. 82
5.4.2 Visualizations of the results for the comparison of proteins 4clna and 2bbma returned by CMO, PAUL and the four top alignments of our approach. 82
5.4.3 Evolution of run times with respect to # of edges in the alignment graph. 84

List of Algorithms

3.1 Algorithm for heteroskedastic analysis 35
3.2 Algorithm for homoskedastic analysis 35

4.1 Floyd-Warshall algorithm. 49
4.2 Dijkstra's Single Source Shortest Path algorithm. 49
4.3 Partitioned All-Pairs Shortest Path algorithm 53

5.1 Overview of the algorithm 71
5.2 Seed enumeration 72
5.3 Seed extension 73
5.4 Extension filtering algorithm 74
5.5 Graph splitting algorithm 78

List of Tables

3.2 Values and ranges of values for fixed and variable parameters used in Fig. 3.4.4, Fig. 3.4.5, and Fig. 3.4.6. 45
3.1 Values and ranges of values for fixed and variable parameters used in Fig. 3.4.1, Fig. 3.4.2, and Fig. 3.4.3. 45

5.1 Details of the alignments returned by other tools - columns 2 through 4 - and our method - columns 5 through 8. Best scores are in italics. . 83
5.2 Run times and speedups for varying # of cores. 83

1 Introduction

The recently renewed interest in parallel computing stems from a conjunction of two very different factors. On the one hand, data generation in many fields is becoming exponentially cheaper yielding a data tsunami and greatly increasing the demand for time-consuming computations. On the other hand, computer architectures are undergoing a drastic shift from exponentially increasing clock frequencies to exponentially increasing parallel capabilities.

The exponential growth of available data to process lead to the emergence of the concept of Big Data. Although the term was first used in 1941 according to the Oxford English Dictionary, it really became popular around the year 2007; a special issue of *Nature* in 2008 was even entirely dedicated to the Big Data concept [HCF+08, Lyn08, W+08]. Coping with the Big Data phenomenon poses several challenges in terms of storage, search, sharing, analysis and transfers of the data. In this thesis, we will concentrate on the computational aspect of the challenges, *ie.* how to analyze large datasets using the limited capabilities of modern computers. From a computational point of view, large datasets require efficient implementation in order to reduce runtimes to a reasonable level as well as a careful usage of the limited main memory available on a given computer.

The simultaneity of the regain in popularity of the Big Data concept and the recent change in computer architectures may not be solely coincidental. The sudden halt in the evolution of processor clock frequencies drastically accentuated the challenges of Big Data, from a computational point of view at least. Before this shift in computer architectures, any program could process a larger amount of data by simply having it run on a newer computer with a higher clock frequency. Increasing the parallel capabilities of a computer will however have no immediate impact on a sequential program's runtime. Parallel capabilities of modern computers require efforts from programmers to be fully exploited. Some computational problems will not even benefit from parallelism; others may be well suited for certain types of parallelism but will not be accelerated by other types. Implementing a parallel version of an algorithm must therefore be preceded by a careful analysis to exhibit parallelism and find suitable parallel hardware to target.

This conjunction of factors is particularly noticeable in bioinformatics. Recent advances in data generation such as High-Throughput Sequencing technologies have stressed the need for efficient implementations, fully exploiting parallel capabilities of modern computers, to handle the massive amount of data to process. The same phenomenon can be observed in other domains such as protein comparison and protein interaction analyses, where protein databases have known the same exponential growth. In the past few years, parallel implementations have flourished targeting

various parallel architectures such as multicore Central Processing Units (CPUs) and manycore Graphics Processing Units (GPUs).

This thesis focuses on exploiting the parallel capabilities of modern computers for computationally intensive bioinformatics problems. Various problems are studied, from which different types of parallelism can be exhibited and exploited.

Chapter 3 describes the various parallelism techniques that can be employed on a modern computer and the types of computational problems to which they can benefit. We then detail some recurring problems in bioinformatics that have previously been ported onto parallel hardware. Two types of parallelism are discussed here:

- Fine-grain parallelism, in which we include the use of CPU vector instructions and bit-level parallelism techniques;
- Coarser-grain parallelism such as multicore CPU implementations and manycore GPU implementations.

This distinction between fine- and coarse-grain parallelisms is of course arbitrary and depends on the application; GPU parallelism can be considered fine-grain parallelism in the context of a multinode cluster parallelization.

Chapter 4 proposes a GPU implementation of a tool for Quantitative Trait Locus (QTL) detection called QTLMap. The embarrassingly parallel structure of the statistical approach developed in QTLMap, makes it an ideal candidate for a porting to the GPU architecture. This new implementation is up to 75 times faster than the previous multicore CPU version. This speedup can however only in part be imputed to the use of a GPU. Some optimizations have been specifically made to accelerate the GPU implementation; these optimizations could also be implemented on the CPU version of the tool. Faster QTL analyses allow geneticists to consider more precise computations and the processing of larger datasets.

In chapter 5, we discuss a new algorithm for the All-Pairs Shortest Paths (APSP) problems. APSP consists in finding the minimum distance between any two vertices of a weighted graph. This new algorithm, derived from the Floyd-Warshall algorithm, targets graphs with good community properties and develops a partitioned approach to the problem. The two-level parallelism exhibited by this algorithm allow for a multi-node GPU implementation. Computations are intended on large clusters of multi-GPU nodes and thus for very large instances of the problem - graphs with up to 10^9 vertices. Computing the shortest distances between all pairs of vertices in a graph is the first step to obtaining many graph measures that are useful in various domains. Large graph analysis becomes crucial in bioinformatics when studying large protein protein interaction networks for instance.

In chapter 6, we propose a new approach to protein structure comparison and its parallel implementation. This new algorithm was early on developed with parallelism in mind. The implementation exploits multiple levels of parallelism such as vector instructions, bit-level parallelism with a bit-parallel set representation and computations across multiple cores of a CPU. These multiple levels of parallelism allow a more in depth analysis of the similarities between two proteins by providing

more than a single pair of similar regions to be returned as well as more complicated configurations to be taken into account, such as sequence inversions and local flexibility of proteins.

2 Background

2.1 Introduction

The aim of bioinformatics is to use state of the art techniques stemming from the computer science field to tackle computationally intensive biological problems. Bioinformatics embraces fields such as DNA or protein sequence alignments, analysis and comparison of protein or RNA structures. All these fields have known an exponential increase in the amount of available data in the past decades.

DNA sequencing is perhaps the most striking example of this rapid increase in data availability. Sequencing techniques have dramatically improved since the first sequencing of an organism in 1977, a 5386 base-pair long bacteriophage [SNC77]. Sequencing of the human genome, more than 3 billion base-pairs, was achieved in the year 2000 at the steep price of several billion dollars. Nowadays, next generation sequencing technologies have drastically reduced the price down to about $7k$ dollars.

The consequence of the decrease in price of data generation is an explosion in size of databases in many fields. This phenomenon, referred to as Big Data, has rendered traditional tools incapable of outputting results in a reasonable time frame. The Big Data phenomenon also poses serious issues in terms of storage. Simultaneously, microprocessor architecture are rapidly evolving in a completely new direction.

Up until around 2007, processor clock frequencies increased exponentially, doubling approximately every 2 years. This exponential growth then came to a halt due to the power wall. Sustaining increasing clock frequencies would come at the price of a prohibitively high power consumption. Microprocessor manufacturers however still manage to follow Moore's Law, which states that the number of transistors on a single chip grows exponentially [M$^+$65, Moo] - doubling approximately every 2 years in practice. This increasing number of transistors does not however go towards increasing clock frequencies anymore but instead mostly towards more computational units on a single chip.

From a programmer's point of view, an increase in the number of processing units is very different from an increase in clock frequency. Doubling the clock frequency means that the same sequential program will run effortlessly up to twice as fast. On the other hand, doubling the number of computational units will have absolutely no impact on the run time of the sequential program. In order to benefit from the additional computational units, one has to undergo the tedious process of parallelizing the sequential program. Not all programs, however, can benefit from this parallelizing process. With twice as many computational units available, run times of a parallel program will at most be reduced by a factor 2 but this ideal case is far

from being the norm.

Central Processing Units (CPU) have only recently adopted parallel architectures as a standard for even general public computers. Graphics Processing Units (GPUs) have had that approach for a little longer, which is why the computational capabilities of these components has increasingly drawn the attention of the high performance computing community. GPUs present a massively parallel architecture with an impressive theoretical computational throughput and are an integral part of most modern computers.

Taking advantage of all the computational power offered by a modern computer means using its CPU cores simultaneously as well as its GPUs. In terms of parallelism, modern CPUs also offer vector instructions allowing executions of the same instruction over multiple data simultaneously. These parallelizing techniques are however not suited for every problem. Deciding which approach to consider and implementing it for a particular problem requires a careful analysis of the problem to be solved.

In this chapter, we first describe the differences between the CPU and the GPU architectures. We then give an overview of the different types of existing computational problems as well as some hints about which types of parallelism that can be considered. The following sections give an overview of the parallel capabilities offered by modern computers. We finally present recent examples of parallel applications in bioinformatics.

2.2 Overview of CPU and GPU architectures

Central processing units (CPUs) and graphics processing units (GPUs) have very different architectures. These differences stem from the fact that these two components had very distinct original purposes. On the first hand, CPUs are designed to be all purpose processing units. CPUs must be able to run a variety of heterogeneous programs and in particular an operating system. GPUs on the other hand, are components initially dedicated to image rendering. They were originally designed to compute the values of each pixel to display on the screen.

A typical computer screen displays millions of pixels that need to be refreshed dozens of times every second. The tasks associated to updating each of these pixels are almost always identical and independent - or solely depend on the values of other pixels in the near vicinity. GPU architecture was therefore designed to exploit the massive parallelism inherent to image rendering. In that regard, GPUs can nowadays be successfully used in general computations to solve problems that present the same properties as image rendering - *i.e.* problems that can be decomposed in a large number of independent tasks.

Modern CPUs are composed of several cores offering interesting parallel capabilities. These cores are fast, all-purpose processing units that benefit from a large cache hierarchy and a dedicated control unit. New technologies such as Intel's Hyper-threading even allow each of these cores to run two threads at full speed

simultaneously. Each thread also has access to larger registers and an associated set of vector instructions. These instructions allow each core to perform multiple identical instructions simultaneously over different data items. Due to their dedicated control units, two CPU cores of a single CPU can execute different instructions at the same time; this property allows multicore CPUs to run independent processes simultaneously.

Modern GPUs, on the other hand, are composed of a large number of multiprocessors. Each multiprocessor has its own control unit and a small manual cache memory; both of them are shared among a large number of GPU cores. Recent GPUs also offer a small automatic cache shared among GPU cores of each multiprocessor. The fact that the control unit is shared among GPU cores from the same multiprocessors forces these GPU cores to always execute the same instruction over different data items at all times. In this sense, GPU multiprocessors are very similar to a single CPU core only executing vector instructions.

With the increasing use of GPUs in general purpose computations, the need for more CPU-like features becomes greater. GPU vendors are adapting to this recent demand. This trend can be observed in the integration of more precise floating point units, conforming with IEEE standards; in the increasing set of available instructions or in the recent adding of l1 and l2 caches.

In the meantime, CPU vendors, limited by the power wall, cannot keep increasing processor frequencies as they used to until around the year 2007. CPU performances are however still improving, though not by increasing clock frequencies. Instead, better performances are obtained, for example, by issuing more instructions per clock cycle, a process referred to as instruction level parallelism (ILP), by increasing the size of SSE vectors and their related set of instructions or by increasing the number of CPU cores on a single chip. Thus, CPU performances nowadays mostly improve by the addition of parallel features at different levels.

Both the CPU and GPU architectures are slowly merging to a hybrid architecture exhibiting traits inherited from traditional CPUs, i.e. all-purposeness, and from GPUs, i.e. massive parallelism. However, as of today, clear distinctions between the two architectures remain and it is up to developers to determine which type of parallelism to exploit depending on the studied problem, and what architecture best suits their needs.

2.3 Hierarchy of computational problems

Having access to highly parallel hardware with modern CPUs and GPUs does not mean that any implementation of a computational problem can be significantly accelerated. Some computational problems are inherently sequential and will not benefit from an attempt at parallelizing them. A common analogy for such problems comes from the software development field and claimed that adding manpower to a late project makes it later. The analogy states that “nine women can’t make a baby in a month”.

This analogy is also valid in our case. Some problems simply do not benefit from additional computational units. Thus no attempt should be made at parallelizing such computational problems. An example of such an inherently sequential problem is optimally playing a card game; computing which card is the best to play at a given round always depends on the previously played cards, thus one cannot compute rounds n and $n + 1$ simultaneously. However computing a single round may be breakable into independent tasks.

In order to be consider parallel, a computational problem should be breakable into independent tasks. Two tasks are considered independent if one does not need the result of the other to be computed. These independent tasks can then be computed independently and simultaneously on different processing units. Being able to break down a problem into independent task does not however ensure that a parallelization attempt will yield a decent speedup. If said tasks are greatly unbalanced for instance, balancing the total workload between the available processing may become tricky or even impossible. If, for example, one task represents 90 percent of the total execution time, any effort at balancing the workload is vain.

A computational problem is considered embarrassingly parallel if it is parallel and if all the sub-tasks composing the problem are perfectly balanced and identical. This strong property makes load balancing between the computational units trivial. Problems exhibiting this property are also candidates for executions on hardware that require that identical operations are computed between different computing units at all times. Such hardware include vector instructions on a CPU and GPU multiprocessors.

Figure Fig. 2.3.1 shows a rough classification of computational problems. Belonging to a given class of problem gives hints into which type of hardware is best for a given problem but is not nearly enough to make a decision. Many other aspects of the problem must also be taken into account to decide whether it fully complies with specific hardware restrictions. Such aspects include memory access patterns, number of independent tasks or required memory.

2.4 Coarse-grain parallelism

Parallelism can be exploited at many different hardware levels in today's computers. Parallelism can range from distributing computations over a grid to executing two instructions simultaneously on a single computation unit. The former type of parallelism is considered coarse grain parallelism, while the latter is considered fine grain parallelism. The grain traditionally refers to the size of the tasks that are run in parallel. When distributing computations over a grid, the overhead induced by the communication between nodes is far from negligible and, in order to better amortize it, the initial problem must be split into large tasks; therefore, parallelism over a grid of computers is considered coarse grain.

In this chapter, we focus on parallelism on a single machine. What we will consider coarse grain parallelism here is using multiple cores of a modern CPU and offsetting

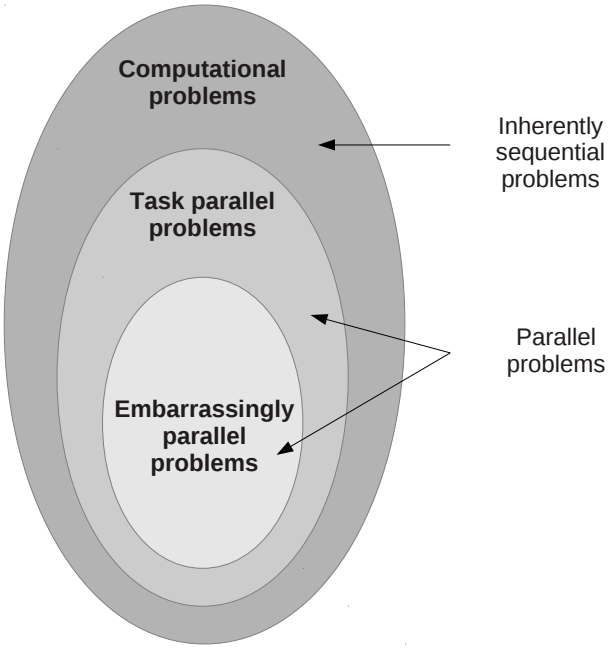

Figure 2.3.1: Rough classification of computational problems from a parallelism point of view.

computations to a GPU. In these two cases also, substantial speedups can only be obtained when tasks are large enough. When programming on a multicore CPU, the size of the tasks needs to amortize the overhead induced by spawning multiple threads and in the case of GPU programming it needs to amortize the overhead induced by the communications between the CPU and the GPU.

2.4.1 Multicore CPU programming

Most modern general public CPUs are multicore. In order to make full use of a recent CPU, a program must exploit the computational power of all CPU cores simultaneously. Each CPU core can be seen as an independent CPU in the sense that very different tasks can be assigned simultaneously to two different cores. This form of coarse-grain parallelism is referred to as MIMD (for Multiple Instructions Multiple Data); at any moment, two cores can be executing different instructions over different pieces of data.

CPU cores share the same memory. Shared memory has the advantage of allowing easy communication between cores but leaves to the programmer the tedious responsibility of ensuring data consistency. In order to prevent two or more cores from modifying the same memory address concurrently, complex mechanisms must be implemented. Various strategies exist to do so and depend on the framework chosen by the programmer. Two multicore frameworks are exposed in this section:

- the POSIX threads library (or Pthreads), a low-level approach to multicore

programming;

- The OpenMP interface, a higher-level approach.

2.4.1.1 The POSIX threads library

The POSIX threads library is a standardized C language threads programming interface for UNIX systems that was specified by the IEEE POSIX 1003.1c standard in 1995. This standard programming interface emerged to provide a cross-platform alternative to already existing proprietary APIs proposed by hardware vendors. Nowadays, most hardware vendors provide an implementation of the Pthreads standard along with their own proprietary APIs.

Task definition

In the POSIX threads paradigm, parallelism is exploited by splitting the initial problem into tasks that are attributed to different Pthreads. A Pthread can be seen as a lightweight process, in the sense that Pthread creation generally induces much less system overhead than process creation. Pthreads also share the same address space; thus, inter-thread communication is usually more efficient than inter-process communication.

Tasks are defined by specific C functions. When no data dependency exist between two functions - *i.e.* the result of the second function does not depend on the result of the first function and vice versa - they can be run in parallel. Parallelism is achieved by attributing both functions to different threads. The developer can decide to create as many threads as necessary but optimal performances are usually obtained by a number of threads that is less or equal to the number of available CPU cores on the machine. The developer has very little control over the way threads are scheduled and should not make any assumption. Great care must be exercised when two or more threads access the same data. In general, the sharing of data must remain as low as possible. When data sharing is inevitable, synchronization techniques can be applied to preserve data coherency.

Synchronization

Three types of thread synchronization are available in the POSIX threads library:

- Joins, which allow a thread to wait for another thread to finish its execution;
- Condition variables, which are used to prevent a thread from starting its execution until the condition is met;
- Mutexes, which can be used to prevent concurrent accesses to a memory address.

Joins are a task level type of synchronization. They can be used to manage the logical order in which tasks are to be executed. Often, a master thread handles the creation of threads, assigns the various tasks of the problem and takes care of the logical execution order of these tasks using joins. Condition variables provide a finer way of synchronizing threads, by not requiring a thread to wait for another thread to terminate but instead wait for it to reach a specific point in the resolution of its task. Finally, mutexes are a variable level type of synchronization. They are generally used to ensure sequential accesses to shared variables. In all forms of parallelism, synchronization between units is best kept to a minimum, because it serializes operations and thus reduces parallel computations.

2.4.1.2 The OpenMP interface

OpenMP is also a programming interface for shared memory parallel computing. It offers a higher level of abstraction compared to the PThreads library. It is available on various environments (including Unix and Windows) and programming languages (C/C++, FORTRAN).

The main advantage of the OpenMP interface is to let users design parallel programs from existing sources with very little modifications to the code. Since OpenMP is directive based, in many cases the modified program can still be compiled in a sequential way by simply ignoring the directives.

Task definition is also not as cumbersome as it is with the Pthreads library. In the OpenMP paradigm, a task can simply be a portion of code or even iterations of a loop. OpenMP also offers synchronization mechanisms similar to the ones provided by the Pthreads library.

2.4.2 GPU programming

GPUs have recently attracted the interest of the scientific computing community with their tremendous computational capacity. The use of GPUs for computations other than graphic computations is referred to as General-purpose Processing on Graphics Processing Units (GPGPU). Programming for a GPU requires developers to provide work for a massive number of GPU cores. In order to be efficient, a GPU implementation needs to use all these available GPU cores simultaneously. In this section, we first describe how parallelism is exploited in the GPU paradigm. We then present limitations inherent to the GPU architecture. These limitations take form in the limited synchronization available between GPU cores, in the particular memory hierarchy offered by current GPUs and the memory transfers between the CPU and GPU. We finally introduce three GPU programming environments: a proprietary environment proposed by NVidia, namely the CUDA environment, a low-level, cross-platform, open standard called OpenCL, and an emerging cross-platform standard - the OpenHMPP environment.

2.4.2.1 Grid-based computations

A typical GPU is composed of several multiprocessors. Each of these multiprocessors is itself composed of several GPU cores. Mapping computations to these two levels of parallelism is done by creating a computation grid. A computation grid is a matrix with up to three dimensions composed of blocks. A block is itself a sub-matrix with up to three dimensions composed of GPU threads. Grids and blocks map to the two-level parallelism of GPUs. Each block is to be executed on a single multiprocessor and each GPU thread is to be executed on a GPU core.

In order to successfully implement a problem on the GPU, one needs to decompose the given problem into tasks, which will constitute the blocks of the computation grid. Each task must be in turn be decomposed into sub-tasks, which will constitute the GPU threads of these blocks. The number of cores available per multiprocessor depends on the GPU and influences the minimal efficient size for a block - typically a multiple of the number of cores per multiprocessor. In NVidia's terminology, a block should have at least the size of a half-warp, with a warp being a set of 32 threads.

Not all problems however can be translated into these block/thread decompositions. The limited synchronization available between two GPU cores further reduces the range of problems that can be efficiently addressed on a GPU.

2.4.2.2 Limited synchronization

Three types of synchronizations are discussed here:

- Within warp synchronization; the lowest level of synchronization between subsets of threads of a same block;
- Intra-block synchronization;
- Inter-block synchronization.

Due to the SIMD nature of the GPU architecture, synchronization within a half-warp is automatic. Threads that belong to the same half-warp always execute the same instructions simultaneously. Therefore one can make assumptions about the order in which instructions will be executed within a single half-warp. The size of a warp - and thus that of a half-warp - however depends on the number of CPU cores in a multiprocessor; this number may change with future generations of GPUs. It may thus be unsafe to rely on this automatic synchronization.

The downside of this automatic synchronization is that whenever a branching occurs - *i.e.* conditional or loop statements - if threads within a half-warp do not all take the same branch, a process referred to as warp divergence, both parts of the code will be executed sequentially. In other words, the first branch will be executed by a first group of threads while the second group remains idle; the second branch will then be executed by the second group of threads while the first group is idle.

Intra-block synchronization is possible using language dependent, device-side primitives. This type of synchronization allows developers to manage, for example, accesses to the small manual shared cache available for each block.

Inter-block synchronization is theoretically not possible. In most cases, this type of synchronization can be emulated by separating the kernel in which inter-block synchronization is wanted into two separate kernels. The sequential computations of these two kernels can then be enforced using host-side synchronization primitives. Recent work by [XcF10]showed that such synchronization is possible with decent performances but many assumptions are made to ensure a one-to-one mapping of blocks to SIMD processors. These assumptions are hardware dependent, hindering code portability and evolutivity. New developments in the CUDA language, such as the ability to call new kernels directly from the device (a technology called dynamic parallelism), could however open the door for new research in this direction.

The limited synchronization available on a GPU means that the tasks associated to the various blocks composing the grid must have a high degree of independence.

2.4.2.3 Host/ Device memory transfers

In the GPGPU paradigm, the CPU is referred to as the host whereas the GPU is referred to as the device. General computations on a GPU are always performed at the request of the host. Since the host and the device do not share the same memory space, when the CPU offsets computations to the GPU, it is necessary to first copy the input data from the host's main memory to the device's main memory, then request computations on the device and finally copy the results from the device's memory to the host's memory.

Transfers between the host and the device can be considered as additional work to the initial problem to solve. These transfers can be costly and must therefore be minimized. They can however be hidden by computations on both the host and the device and, if the device allows it, by another transfer in the opposite direction.

2.4.2.4 Memory accesses

GPUs typically offer two types of memory:

- slow, global memory; a large memory space - typically several GB - used for input and output data;
- fast, on-chip memory; a small memory space - typically several kB - that can be used as a manual cache for input data reuse, intra-block communication and intermediate data.

NVidia GPUs also offer a second type of on-chip memory referred to as constant memory. This memory space can only be filled by the CPU before GPU computations. Only read accesses are allowed from the GPU. This memory space can for instance be used to send additional parameters to GPU threads.

A single memory transaction from or to global memory may take up to 800 clock cycles [NVI12]. More than a single data item can however be modified in a single global memory transaction. Namely, if threads belonging to a same warp access consecutive memory addresses, these accesses can be merged into a single memory transaction using a larger part of the bus width. This process, referred to as coalescing, stresses the need for programmers to carefully manage data access patterns in their GPU code. As in traditional CPU code, data locality is of the utmost importance when it comes to GPU programming. If non-consecutive memory accesses are to be performed, these accesses may in some cases be translated to coalesced accesses and stored in on-chip memory for later use.

Coalescing memory accesses drastically improves data throughput but does not address the above mentioned,800 cycle, memory latency. This latency can be addressed in several ways. One way is to provide independent instructions in the same instruction block as the memory access. These instruction will be computed while waiting for the memory item to be retrieved. However, providing enough instructions to keep GPU cores busy for 800 cycles may prove challenging.

A second way is to provide more threads in the computation block than there are GPU cores available in the multiprocessor (usually a multiple of the warp size); therefore, when GPU cores stall on a memory access, a context switch can happen and provide independent instructions to compute in the meantime.

Finally, the latency of memory accesses can be hidden by caching data that has been retrieved from global memory into local, on-chip memory and reusing the data. Best performances can be achieved by using a combination of these three techniques. Fast, on-chip memory is however shared among threads of a GPU block; the more GPU threads, the less shared memory is available per GPU thread. A trade-off must thus be found between potential context switches and available on-chip memory to cache global data.

2.4.2.5 The CUDA environment

NVidia provides a framework to program their line of GPUs. This framework, named CUDA for Compute Unified Device Architecture, takes the form of extensions to several popular programming languages such as C/C++ and FORTRAN. This environment lets programmers define specific functions - kernels - to be executed on the device. Grid and block dimensions are dynamically specified at each kernel launch. Specific modifiers are also provided to indicate storage location of variables on the device. Transfers from and to the device are explicitly requested by a set of routines.

Device code is a subset of the original programming language with specific intrinsic functions to manage either host- or device side synchronization and other GPU specific actions. Special variables are available to identify the position of the current thread in the grid/ block environment. NVidia also offers a set of tools to debug and optimize GPU code. Although efficient and easy to use, the CUDA framework produces code that lacks portability and evolutivity. CUDA code only

targets NVidia GPUs and optimizations made for a specific generation of GPUs will most likely have to be - at least partially - rewritten in order to be efficient on later generations of GPUS.

2.4.2.6 The OpenCL environment

OpenCL is a framework for computations over heterogeneous systems such as a computer equipped with a modern graphics card. Designed to be platform-independent, openCL code can therefore be compiled and executed on both AMD and NVidia graphics cards for instance. OpenCL does not provide a higher level of abstraction than CUDA but has the advantage of targeting multiple brands of accelerators in a single code. When executing code on NVidia GPUs, OpenCL is however limited by the back-end provided by NVidia, which uses the CUDA API. Therefore, when run on an NVidia GPU, OpenCL code will never outperform its CUDA equivalent. Nevertheless, [FVS11, KDH10] showed that, when correctly tweaked, OpenCL code could reach similar performances than CUDA code.

2.4.2.7 The OpenHMPP environment

OpenHMPP is an emerging standard for GPGPU. It provides a cross-platform environment with a high level of abstraction. Much like OpenMP for multicore programming, OpenHMPP provides a set of directives to annotate existing CPU code. Specific functions can be annotated to be offset to the GPU. A default behavior is provided and trigger implicit data transfers between the host and the device. Useless transfers can be avoided by specifying which pieces of data should remain on the GPU between two computations. Explicit transfers are also available for finer tuning.

OpenHMPP provides safe directives that will not change the semantic of the original code. Annotated code can therefore still be compiled with a regular CPU compiler to provide a GPU free alternative in case no GPU is available. Code generated by the proprietary compiler can be executed on various brands of device accelerators including NVidia and AMD GPUs. Evolutivity of the annotated code is guaranteed by compiler updates.

2.5 Fine-grain parallelism

In this section, we present three forms of parallelism that take place at the instruction level:

- Vector instructions (also known as SSE instructions for Streaming SIMD Extensions, which replaced a former set of vector instructions known as MMX for MultiMedia eXtensions). These instructions allow developers to work with registers that are larger than normal (32 or 64 bit) registers.

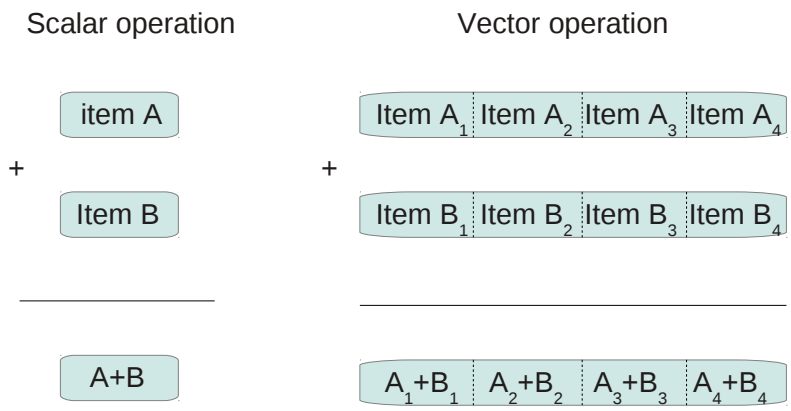

Figure 2.5.1: Differences between a scalar addition and a vector addition. In this example, the length of the vector registers is 4 machine words.

- Bit-level parallelism. In this type of parallelism, more than one piece of data are represented in a single register, allowing developers to compute operations on multiple data items at once.

- Instruction level parallelism (ILP). This technique refers to the way several consecutive and independent instructions can be computed simultaneously on modern architectures. This type of parallelism is not truly controlled by programmers and is made possible by recent improvements in hardware designs and control- and data-flow analysis. Being aware of ILP may however help programmers write code that will better benefit from this type of parallelism.

2.5.1 Vector instructions

Vector instructions are a set of instructions that work on multiple word length registers. As opposed to scalar instructions, vector instructions allow simultaneous computations over different data items. Several data items, or vectors, can be loaded into large registers and multiple operations can be performed simultaneously - see Fig. 2.5.1. More vector instructions are added regularly increasing the number of available operations.

Not all programs however can benefit from vector instructions. Vector instructions will really be beneficial if several identical scalar instructions need to be computed at one given time to be able to merge them into a vector instruction. Moreover, the data items over which the vector instructions is to be computed should ideally be contiguous in memory as packing scattered data items into a single register may take a while and reduce the performance gain. Current $x86$ processors offer registers up to 256 bit long, allowing up to 8 32-bit words to be computed in one single instruction.

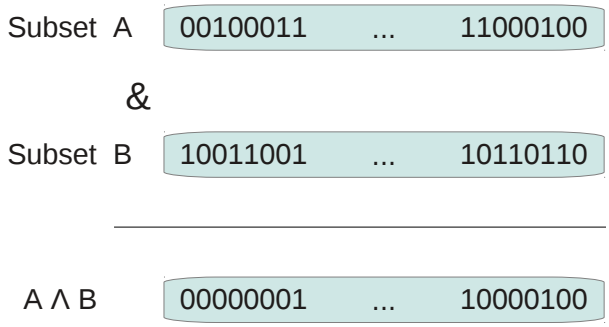

Figure 2.5.2: Bit parallel intersection of subsets. Using a logic *and* between these two bit vectors yields the expected result.

2.5.2 Bit-level parallelism

Bit-level parallelism relies on the same principle as vector instructions except that instead of increasing the size of the registers, we reduce the size of the data items. Several smaller data items can be packed into a single register and computed simultaneously using traditional word size instruction or even vector instructions.

A typical bit-level parallelism example consists in representing subsets of a larger set of size N using a bit vector the length of the super-set. The nth bit in the bit vector represents the presence or absence of the nth item in the subset. In this case, every bit of the subset is a data item. The intersection of two such subsets can then be obtained in a reduce number of instructions using a simple logic *and* operation - see Fig. 2.5.2. The number of instructions required for such an operation thus is N/RS, where RS is the size of the register. In the same way, a logic *or* yields the union of the two subsets. Inclusion of a subset A in a subset B can be tested by an equality test between the result of $A \wedge B$ and subset A.

2.5.3 Instruction-level parallelism

Instruction-level parallelism (ILP) refers to the simultaneous execution by a single processor of independent instructions belonging to a similar code region. Two instructions are considered independent if one does not read nor write a value that is written by the other - see Fig. 2.5.3. ILP is not directly the responsibility of the programmer and is exploited by the compiler and the hardware but knowing how ILP works may let programmers write code that will allow the compiler to increase the amount of ILP. Several techniques, such as loop unrolling, may favor ILP.

2.6 Parallelization in bioinformatics

Bioinformatics as a field of research encompasses a wide variety of data and computation intensive problems. Recent technologies, such as Next Generations Sequencers (NGS), have greatly increased the amount of data to process for a single analysis,

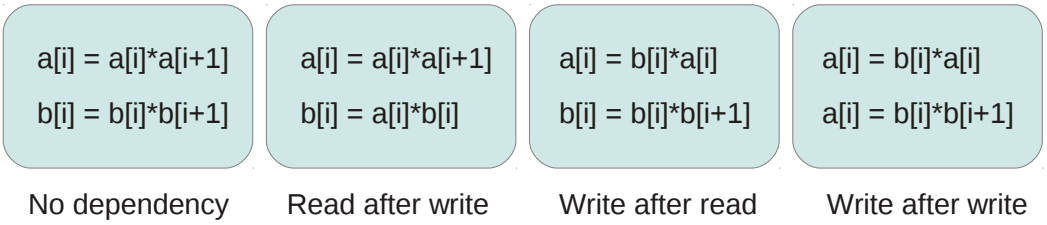

Figure 2.5.3: Examples of possible dependencies between two instructions.

thus accentuating the need for efficient implementations in a large array of known problems. The aim of this section is to give a non exhaustive list of problems important to bioinformaticians that have benefited from one or more forms of parallelism. When available, speedups claimed by the original authors are also reported in this section. However, these speedups should not be compared directly, since reference times were not taken on similar architectures.

2.6.1 Sequence comparison

Sequence comparison is a central topic in bioinformatics. Comparing DNA, RNA or protein sequences allows bioinformaticians to study the functional, structural or evolutionary relationships between two sequences that may come from a single individual, two individuals from a same species or even individuals from different species. The recent exponential increase in the amount of data available has stressed the need for efficient implementations, taking advantage of all the available computational power. In this context, many parallel implementation have been proposed.

2.6.1.1 Sequence alignment

Sequence alignment in bioinformatics is, from a computer science point of view, string alignment over a very succinct alphabet. Therefore, bioinformatics relies heavily on advances made in computer science. Aligning two sequences consists in finding the superpositions of the two sequences that minimize the number of mismatched characters. Gaps may be introduced in either sequence to allow a better superposition of the two sequences. A gap penalty must be applied to minimize the number of gaps.

[Mye99] proposed an elegant bit-parallel algorithm for approximate string matching based on dynamic programming. This algorithm was recently adapted to DNA sequence alignment by [KKN12]. In this implementation, the authors did not compare execution times with a previous non bit-parallel implementation, thus no speedup is reported here. They however estimated the comparison time to 28 ns per base pair including file I/O on an Intel Xeon E5540 (2.53 GHz) PC. More recently, [Edg04] proposed a bit-parallel improvement of their own software named MUSCLE, a tool for creating multiple protein sequence alignments. In this improved implementation, a bit vector is used to represent the presence or absence of

k-mers in a given sequence; a k-mer being a subsequence of length k. In this representation, the nth bit is set if the nth k-mer is present in the sequence and unset otherwise. This representation allows MUSCLE to quickly compute set operations such as intersection of two sets - see sec. 2.5.2.

Vector instructions have also been used to speed up sequence alignment programs. [Far07] proposed a very efficient implementation of the Smith-Waterman algorithm that takes advantage of Intel SSE2 intrinsic functions. The performance of their implementation greatly surpassed that of other SSE implementations of the same algorithm, namely [Woz97, RS00].

Sequence alignment has also been extensively parallelized over multicore CPUs. [KT10] proposed a threaded implementation of MAFFT, a program for multiple sequence alignment. Their parallel implementation scales decently up to about 10 CPU threads on a machine with a 16 core CPU. In terms of results, [KT10] obtained a slightly lower accuracy than [Edg04] with much longer runtimes. [LSM10b] also proposed a multicore implementation of a program for multiple sequence alignment called MSAProbs. No information is given about how well their implementation scales with the number of applied threads; they however show that their results obtain better accuracy scores than [Edg04, KT10] but for runtimes even longer than [KT10]. [L$^+$09] also proposed a local alignment tool named Plast that takes advantage of both SSE instructions and multicore parallelization.

Programs taking advantage of GPUs for sequence alignment started to appear from the beginning of the GPGPU era with [LHJV06] for local sequence alignment and [LSVMW06, WSVMW07] for multiple alignments. All of these programs used graphics libraries to offset computations to the GPU as neither CUDA nor OpenCL existed then. When CUDA was released by NVidia in 2007, new programs emerged benefiting from the enhanced programmability of GPUS. [MV08, LR09] improved previous performances on GPUs for local sequence alignment and [LSM09] for multiple sequence alignment. More recent efforts by [LSM10a, VS11, ZC13] focused on improving with GPUs the performances of BLAST [AGM$^+$90], the reference in local alignment search.

2.6.1.2 The longest common subsequence problem

Given two strings, the longest common subsequence problem (LCS) consists in finding the longest sequence common to both strings. A subsequence differs from a substring in the sense that some characters from a given substring can be omitted to form a substring. For example, CAT is a subsequence of the string $CAGT$, where the character G is omitted. The generalization of this problem to an arbitrary number of strings is called the multiple longest common subsequence problem (MLCS). The MLCS problem is known to be NP-hard, whereas the LCS problem can be solved in polynomial time. Both problems are generally tackled using dynamic programming (DP) techniques; parallel implementations for both problems thus mostly rely on parallel DP techniques.

In bioinformatics, the length of the longest common subsequence of two sequences

is used as a similarity measure between the two sequences. As mentioned previously, sequence similarity may indicate an evolutionary relation between two sequences. Such similarity measures can be used in phylogeny for classification purposes or to determine whether two proteins can have similar functions.

However, the LCS problem does not take into account any prior biological knowledge for the found longest common subsequence. To address this issue, [Tsa03, TLC+03] formalized the Constrained Longest Common Subsequence problem (CLCS). The CLCS problem restricts the output of the LCS problem to subsequences containing a given subsequence P; for two input sequences M and N, only common subsequences of M and N containing P are considered. Biological knowledge can be incorporated in this additional constraint.

[Hyy04] proposed a bit-parallel solution to the LCS problem and improved two existing bit-parallel implementations by [AD86, CIPR01]. [Hyy04] compared his implementation to the fastest non-parallel implementation by [KC89] for increasing alphabet sizes. [Hyy04]'s implementation is rather constant with respect to alphabet size and at least twice as fast as that of [KC89], even though the latter is specifically optimized for very large alphabets. [Deo10] also proposed a bit-parallel algorithm but for the recently formulated CLCS problem; the performance of his implementation surpassed all other existing implementations for this problem and achieved a speedup between 2 and 4 when compared to the fastest known implementation.

Multithreaded implementations exist as well for these problems. In 2011, [WKS11] proposed a new algorithm and it Pthread implementation for the NP-hard MLCS problem. [WKS11] compared their results to that of traditional sequence aligners, namely MUSCLE [Edg04] and ClustalW [THG94]. Since sequence aligners differ from MLCS solvers, [WKS11]'s run-times are much longer but yielded sensibly longer results.

[YXS10] designed a new algorithm for the LCS problem that exhibits more parallelism than previous approaches. They proposed two parallel implementations of this new approach: a multicore CPU implementation and a GPU implementation. They compared the run-times achieved by both implementations to that of the sequential implementation from [UI93]. The multicore CPU version performed about 3 times as fast as the sequential version and the GPU version showed a 6 fold speedup over the multicore CPU version.

2.6.2 Structure comparison

Protein three-dimensional structures are more closely related to their functions than protein amino-acid sequences; therefore, structures tend to be more evolutionary preserved than sequences. Finding protein structure similarities thus yields more information about functional similarities than sequence similarities. However, comparing three-dimensional structures proves more challenging than comparing one-dimensional sequences.

A common way to address the problem of aligning three-dimensional structures is to create an alignment graph or product graph. For two structures S_1 and S_2 or

lengths m and n respectively, an alignment graph of these two structures is an $m * n$ grid shaped graph, where vertex (i, j), located on the ith row and the jth column corresponds to the matching of the ith element of S_2 with the jth element of S_1.

In this alignment graph, an edge between vertices (i_1, j_1) and (i_2, j_2) indicates that matching elements i_1 and i_2 with elements j_1 and j_2 in the same alignment is relevant - see sec. 5.1.1 for a more detailed description of alignment graphs. A common way to add edges top the graph is if, for a given distance function d - often a euclidean distance - $d(i_1, i_2) \approx d(j_1, j_2)$; in other words, if the two distances being matched are similar in both proteins. In such a graph, a subset of vertices with high edge density denotes a similarity between the two structures.

2.6.2.1 The maximum clique problem

The highest possible edge density in a subset of vertices is achieved when any any two vertices are connected to each other. Such a subset of vertices is called a clique. Finding the largest clique - or one of the largest cliques, as two distinct cliques can be of equal and maximum size - in a given graph is an NP-hard problem referred to as the maximum clique problem. The computational challenge it represents and the large amount of parallelism it induces - testing the presence of all edges in different subsets - seem to make the maximum clique problem a perfect candidate for various parallelization techniques.

A common way to approach the maximum clique problem is through branch and bound techniques. Branch and bound consists in presenting possible solutions in the form of tree. In the case of the maximum clique problem, where a brute force approach would consider every subset of vertices in the input graph, each branch between two levels of the exploration tree denotes the addition of a vertex to the current set of selected vertices. Each node of the tree is therefore a potential solution to the problem. With this representation, a branch and bound approach estimates the potential maximal clique that a given subtree can offer using an approximation. If this potential exceeds the current known best clique, the subtree is explored and pruned otherwise.

Parallelizing the exploration of a branch and bound tree is however challenging and can even lead to a slowdown [LS84]. If multiple threads explore the branch and bound tree in parallel, only communicating to improve the best known solution, the whole tree is explored in a different order than it would in a sequential approach. Some branches that would have been pruned with a sequential traversal of the tree may end up being explored when processed in parallel, thus increasing the total amount of work.

Despite this issue, [TKM07] proposed a multicore solution to the maximum clique problem and achieved substantial speedups when compared to their own sequential implementation. They however compared executions over only 12 randomly generated graphs. We can expect that some other instances may show much poorer results if the lower bound of a clique close to the maximum size is found late in the execution due to a different exploration order of the search tree induced by

parallelism.

A finer type of parallelism is nevertheless achievable without potentially increasing the overall workload. Bit-level parallelism is particularly well suited to handle set operations. [SSRLJ11] proposed a bit-parallel implementation for the maximum clique problem and later published an improved version in [SSMRLH13]. This improved version obtained better performances on average than the leading program from [TSH+10].

To the best of our knowledge, no GPU implementation exist for the maximum clique problem. The properties of the problem make it difficult to implement efficiently on a GPU; the number of subtasks is unpredictable and these tasks unbalanced. A GPU implementation would greatly suffer from warp divergence and non-coalesced memory accesses. Algorithms that can be used to speedup the branch-and-bound search tree exploration can however be successfully implemented on a GPU. [GZL+11] proposed a GPU implementation of a graph coloring heuristic. graph coloring heuristics are often used for the maximum clique problem as an upper bound for the maximum clique that can be found in a branch of the exploration search tree. Graph coloring consists in finding the smallest number of colors required to color a graph so that no two adjacent vertices have the same color. The implementation from [GZL+11] proved that runtimes on the GPU could be equivalent to that of traditional CPU implementations and yield better approximations of the actual number of required colors.

2.7 Conclusion

Many applications in bioinformatics can benefit from parallelizing techniques. Modern computers nowadays offer a large array of parallel capabilities that range from the fine-grain vector instructions to coarser-grain approaches such as multicore CPU and manycore GPU programming. Each parallelizing technique provides a potential speedups; reaching this theoretical speedup is however almost impossible to achieve depending on the problem to solve.

Each parallelizing technique has its specificities that make it suitable only for some computational problems. Identifying which techniques are susceptible of giving the best speedups for a given problem can be done through a thorough analysis of the problem to solve and a deep comprehension of the hardware. Some problem, known to be inherently sequential, will never benefit from parallelization. Once a potentially successful parallelizing technique has been identified, developing a parallel implementation is a tedious and time-consuming process.

Since parallel architectures seem to be becoming the norm in computer architecture, new methods and algorithms should be developed early on with parallelism in mind to reduce the cost of a subsequent parallelizing process.

3 GPU accelerated QTL mapping

This chapter discusses a GPU implementation of a QTL mapping tool named QTLMap. QTLMap is developped by the animal genetics division at INRA. The original release dates back to 2008 [GLRM+08]. The tool was later reimplemented to take advantage of multicore CPUs in 2009. sec. 3.1 first gives an overview of the underlying principles of QTL Mapping. sec. 3.2 then describes the specific methods implemented in QTLMap. sec. 3.3 details the GPU implementation of QTLMap. Finally, sec. 3.4 gives some experimental results. This chapter is mostly based on an article published in the Journal of Computational Biology ([CFE+13]).

3.1 Introduction to QTL mapping

Most of the traits characterizing individuals (their "phenotypes": performance level, susceptibility to disease etc..) are influenced by heredity. Geneticists are interested in detecting, localizing and identifying genes, the polymorphism of which explains a part of observed trait variability. Such genes are often called QTLs (for "Quantitative Trait Locus"), the term locus pointing to a physical position on the genome.

QTL detection procedures consist in a series of statistical hypotheses tests at successive putative locations on the genome. Many experimental designs, sampling protocols and test statistics were proposed and used. QTLMap's algorithm focuses on regression approaches performed on sets of large families. These approaches were developed for exploiting the linkage disequilibrium - the discrepancy between a random distribution of haplotypes and the observed distribution of haplotypes in the studied population - observed on a per family basis and / or at the population level. Amongst the available software dealing with QTL regression techniques, QTLMap was developed ([EMG+99]).

The general principle of Linkage Analysis (LA) for detecting QTLs within a family is to correlate for each tested genome position, the performance trait measured in the progenies and the grand parental origins of the piece of chromosome they received from a common parent. These origins are inferred from "genomic marker information" which describe the parental chromosomes (in diploid species, chromosomes are in pairs and each individual carries two copies, or "alleles", of QTLs, say Q1 and Q2, and markers, say M1/M2, N1/N2) and the way they are transmitted to their progenies (see Fig. 3.1.1). Locations of QTLs are pointed on chromosomal segments, which display high correlations.

Linkage Disequilibrium Analysis (LDA) does not exploit family structure but considers the whole population as a large sample of independent individuals. Due

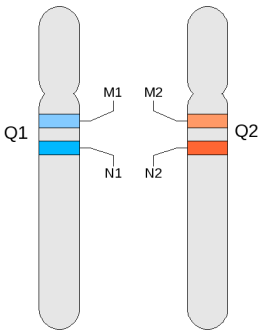

Figure 3.1.1: Repartition of markers M1/M2 and N1/N2 on alleles Q1 and Q2.

to various demographical events along the population histories (selection, breeds mixtures, bottlenecks etc), allelic forms found at two close chromosomal positions are generally not independent. A few measurements of this dependence were proposed in the past ([Lew64, HR68]). This phenomenon is fully proven for markers, which are easy to visualize (e.g. [FCA+00] for the Bovine species) and certainly true between markers and QTLs.

In LDA, the direct effect of genetic information (to make presentation simple, say the marker genotype effect) on the quantitative trait variability is tested at successive positions. The basic idea is that groups of individuals defined by their genetic class (e.g. their genotype for the marker located at the tested position) will display significant differences in their quantitative performance if a QTL is located close to this position. The signal will be stronger if:

1. the linkage disequilibrium between markers and QTL is higher and

2. the QTL explains a larger part of the variability.

A third category of techniques combines LA and LDA. In Linkage Disequilibrium Linkage Analysis (LDLA), both the family structure and the population history are exploited. The population is described as a set of "founders", supposed unrelated, but subject to linkage disequilibrium, and "non-founders" which inherited from the founders, intact or recombinant chromosomes, after one or more generations of transmission. In LDLA, the performance trait measured in the non-founders are correlated, as in the LA, with the founder origins of the transmitted piece of chromosomes, and, as in the LDA, information about the population history is extracted from the degree of similarity between founder pieces of chromosomes. Thus, detecting QTLs is basically a three steps procedure:

1. inferring from the marker information the "phase" of the parents ([EMG+99, FEDGL10]), *i.e.* the way the two alleles of each marker are positioned on the chromosomes. The output is the haplotype pairs of each parent, *i.e.* the list of successive marker alleles carried by each chromosome (e.g. M1N1 and M2N2, or M1N2 and M2N1);

2. estimating parents to progenies transmission probabilities of chromosomal segments;

3. evaluating the likelihood of performance traits observation under alternative hypotheses (a QTL is present or not at the chromosomal segment location) and modeling (LA, LDA, LDLA).

A statistical test is operated at each genome location tested and the best location for the QTL is given by the most significant test. In terms both of theoretical developments and /or computation burden, a major difficulty is to obtain correct statistical test rejection thresholds. Different strategies exist to estimate these thresholds. Efforts were made to find the distribution of the statistical process under the null hypothesis (no QTL on the chromosome), but they did not fully consider the real life situations where unbalanced designs are the rules (e.g. [RAED10]). Alternatively, and this is the usual procedure, thresholds can be estimated empirically after many permutations of the data breaking the marker-phenotypes correlations, or after many simulations under H0 ([CD94]).

QTL mapping analyses, such as LDLA, are computationnally intensive. For QTLMap, run times increase linearly with the size of the studied population, the exploration step of the studied genome region, and the number of simulations required to determine the thresholds. As opposed to sizes of studied populations, which should remain rather stable, the density of marker data increases exponentially and allow a finer exploration of the genome regions, with a higher number of tested positions. Current QTL mapping analyses may take weeks to run in the most difficult cases (e.g. when looking for QTL interactions) on modern computers and run times will increase linearly with the density of available genetic markers. Therefore, dividing run times by an order of magnitude would allow geneticists to run multiple analyses or consider even more time consuming analyses, such as multiQTL ones.

Despite the computational burden that QTL mapping represents, very few parallel tools exist. The first attempt was made by [SHG+06] with gridQTL. This tool is derived from QTLexpress ([SHK+02]), a popular web based tool for QTL analyses, and harnesses the power of computational grids to try and reduce run times. Another approach was developped by [FMG+10] with a tool called QTLMap. This tool takes advantage of modern CPUs by using all their cores simultaneously. Finally, epiGPU ([HTWH11]) uses any commercially available GPUs for QTL analyses but focuses on the detection of epistasis - a complex interaction between genes, where the effect on a given phenotype of a single gene is altered by one or more other genes. Other QTL software, such as eQTL, specialized in detecting expression QTLs, still run on a single CPU core.

The empirical approach used in QTLMap makes it an ideal candidate for GPU computations. In order to determine efficient relevance thresholds, the analysis not only needs to be performed at each genome position but must also be repeated for each simulated dataset - typically 10^3 to 10^4 times. The increasing density of genetic marker data allows for ever more precise analyses, meaning that more and more genome locations need to be considered. Computations at neighboring

genome locations are correlated; it is however more efficient in practice to consider them independant. Computations for each simulation are independant. These computations can therefore be run in parallel. In this chapter, we propose a new version of QTLMap, which performs about 70 times faster than the previous multicore implementation, while maintaining the same level of precision.

We first describe the empirical methods for QTL Mapping ported to GPU in QTLMap and give details about the algorithms of the methods. We then describe the implementation of these algorithms on the GPU. We finally show the details of the experiments we ran to test our new implementation and the results we obtained.

3.2 Methods and algorithms

QTLMap relies on three methods to determine possible QTL locations on linkage groups. The first method, called Linkage Analysis - LA -, aims at determining the transmission probability of each chromosomal segment, based on available marker information in the studied population.

The second method, referred to as Linkage Disequilibrium Analysis - LDA -, relies on studying the discrepancy between an expected random distribution of haplotypes in the studied population and the observed distribution. The third method, referred to as Linkage Disequilibrium Linkage Analysis - LDLA - combines the first two approaches. Contrary to a Linkage Analysis, a Linkage Disequilibrium Linkage Analysis does not solely take into account the length of chromosomal segments to determine transmission probabilities, it also takes into account the more complex Linkage Disequilibrium effect.

Once transmission probabilies have been determined, QTLMap statistically computes the likelihood of the observations under the hypothesis that a QTL is present at successive genome locations in the studied linkage groups. QTLMap then uses an empirical approach to determine thresholds, above which a QTL effect can be considered significant. The following two sections give a brief overview of the QTL mapping methods implemented in QTLMap. A more detailed description of these methods can be found in [EMG+99].

3.2.1 Linkage Analysis

In QTLMap, the hypothesis is tested that one QTL affecting a single trait is located at a position x in a linkage group (e.g. a chromosome). Successive positions on this linkage group are scanned. The test is performed with the interval mapping technique applied to an approximation of the likelihood of having a QTL at a given location ([KEH96, EMG+99, LREB+98]).

Let ns and nd be the number of sires and dams respectively in the studied population. All parents are supposed heterozygous at the QTL, with specific alleles, giving a total of $2(ns + nd)$ QTL genotypes. Performance expectation of progeny k of parent i and j is described as the sum of parental mean values $\mu_i + \mu_{ij}$ and

of the deviations $\pm\alpha l$ to this mean due to the QTL. In this model, it is assumed that the parents are unrelated, the markers in linkage equilibrium - *i.e.* a random distribution of haplotypes is assumed - and the trait normally distributed.

As proposed by [GD99] and by [LREB$^+$98], the residual variance of the quantitative trait is estimated within sire. Considering that sub-populations - in our case, descendants of a given sire - can have different variances is called a *heteroskedastic* hypothesis. This heteroskedastic parametrization better fits different patterns (between sires) of segregation of other QTLs, unlinked to the tested position. The homoskedastic hypothesis, which considers that the variance is equal for any two sub-populations, is also implemented.

Parameters maximizing the likelihood can be obtained in an iterative two step procedure:

1. Solving a linear system (see [EMG$^+$99] for details);

2. Estimating the within sire family variances.

The steps are repeated until convergence, detected when the distance between the likelihood ratio test (LRT) at iteration t and the likelihood ratio test at iteration $t+1$ is arbitrarily small enough.

3.2.2 Linkage Disequilibrium and LDL Analyses

QTLMap implements the "LDA Decay" regression approach described by [LF09]. This Linkage Disequilibrium Analysis is particularly adapted to experimental populations, characterized by a family structure, the target of this software. In this approach, parental haplotypes are pooled in classes, the classification being open to the user decision. In QTLMap, only the most probable sire and dam phases are considered, and the classes (following the example given by [LF09]) are simply defined by the haplotype (to a class corresponds a single haplotype).

To a given class corresponds a specific effect on the quantitative trait. The quantitative performance of a progeny depends on the haplotypes as found in the parental chromosomes from which the putative QTL alleles are originating and not to the (possibly recombinated) haplotypes the progeny itself is carrying. The "LDLA" approach described by [LF09] was also implemented. This approach combines the previous LD Decay and LA models, the QTL effect being defined within the parental haplotype effect.

3.2.3 Thresholds detection

The QTL mapping procedures described in sec. 3.2.1 and sec. 3.2.2 determine, for each position on the studied chromosomal region(s), the likelihood of having a QTL related to a given trait. This score can only be considered relevant if it is above a certain threshold that has yet to be determined.

We define $H(n)$ as the hypothesis of having a n-QTL at n given positions. QTLMap uses an empirical approach to determine relevance thresholds. In order to estimate the probability of having a QTL at a given location ($H(1)$), it is compared to the null hypothesis, $H(0)$. Distribution under $H(0)$ is calculated by running the previous algorithm on random sets of data. Randomly generated datasets share the same architecture as the actual dataset. They contain the same population but for each invidual, performance vectors are randomly generated.

In order to compute the distribution under $H(0)$, a user set number of simulations are randomly generated and run. Efficient empirical thresholds can be obtained by computing a large number of simulations. A single analysis with QTLMap explores *npos* genome positions; rejection thresholds are obtained by running *nsim* analysis on simulated data, leading to a total of *nsim.npos* likelihood computations. Computations at each genome position are correlated but it is better in practice to consider them independent. Computations for each simulation are independent. These computations can therefore be run in parallel.

3.2.4 Algorithms for QTL detection

QTLMap provides three types of analyses, presented in sec. 3.2.1 and sec. 3.2.2, and allows for two types of parametrizations : hetero- and homoskedastic parametrizations. In a heteroskedastic analysis, the variance of subpopulations can differ, whereas in a homoskedastic analysis, the variance is considered stable within the studied population. This section gives information about the structure of the algorithm depending on the analysis and on the parametrization. This section also describes the nature of the data used for computations.

Algorithm 3.1 describes the algorithm implemented in QTLMap for a heteroskedastic analysis. The listing does not describe in details how to solve the linear system - line 7 of Algorithm 3.1 - nor how to estimate the variance - line 8 of Algorithm 3.1 - (see [EMG+99] for details). Instead, attention is brought to the structure of the computations and more precisely to the three loops - lines 2, 3, and 5 of Algorithm 3.1 - that are offset to the GPU. The type of analysis - LA, LDA, LDLA - does not change the structure of the algorithm and will only affect the way the linear system is solved.

Iterations of the first two loops - lines 2 and 3 of Algorithm 3.1 - are completely independent and can be run in parallel. However, iteration n of the third loop - line 5 of Algorithm 3.1 - depends on the result of iteration $n-1$, therefore the third loop cannot be parallelized.

In the more specific case of a homoskedastic analysis, results can be obtained in one pass without waiting for convergence. Details of the algorithm are given in Algorithm 3.2. As in Algorithm 3.1, iterations of the two loops - lines 2 and 3 of Algorithm 3.2- are independent and can be run in parallel.

At each iteration of the two independent loops, a contingency matrix, described in Fig. 3.2.1, is used for computations. For each individual in the studied population - referred to as descendants - these matrices contain values in various effects, some

Algorithm 3.1 Algorithm for heteroskedastic analysis

```
Begin
  for each genome_position
    for each simulation
      LRT = 0
      do
        LRT_old = LRT
        solve_linear_system()
        LRT = estimate_variance()
      while |LRT − LRT_old| > ε
End
```

Algorithm 3.2 Algorithm for homoskedastic analysis

```
Begin
  for each genome_position
    for each simulation
      solve_linear_system()
End
```

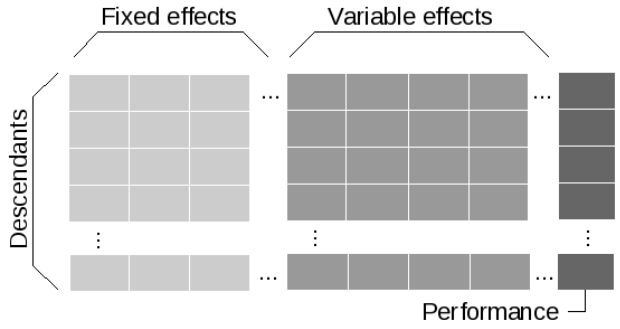

Figure 3.2.1: Description of a contingency matrix used for computations at each genome position and for each simulation.

of which are independent of the current genome position - *i.e.* fixed - and others are dependent of the position - *i.e.* variable -, and a performance value, which describes their performance with respect to the studied trait. Performance values are also independent of the current genome position but change for each simulation. All matrices have strictly the same dimensions for every iteration. Typical sizes for these matrices are about 10^2 for the number of descendants and 10^2 for the total number of effects (including performance).

The properties exhibited by Algorithm 3.1 and Algorithm 3.2 make them ideal candidates for computations on a GPU. First, both algorithms mainly consist of two independent loops, meaning that all iterations can be processed in any order or in our case simultaneously. Second, dimensions of input data - see Fig. 3.2.1 - are identical for every iteration. This consistency of the dimensions of input data allows for regular data access patterns as well as a stable number of instructions to process each iteration. Finally, input data for every iteration is partly redundant, thus leaving room for optimization.

3.3 GPU implementation

Computations offset to the GPU consist mainly in operations on matrices. Operations such as Cholesky decompositions and matrix multiplications are ideally suited for execution on a GPU and can be achieved at near peak performance on such devices ([VD08]). Several highly optimized libraries exist providing linear algebra routines benefitting from modern hybrid architecture ([HPS+10, TDVD09]). However, these libraries specifically target operations on large matrices. In our case, computations are done on a large number of rather small matrices - typically $10^2 * 10^2$ -, therefore, no performance would be gained from using these libraries for such small matrix sizes. Single Instruction Multiple Data (SIMD) parallelism can nevertheless be drawn from the large number of matrix operations performed on different small matrices.

This section describes how the algorithms presented in sec. 3.2.4 are mapped onto

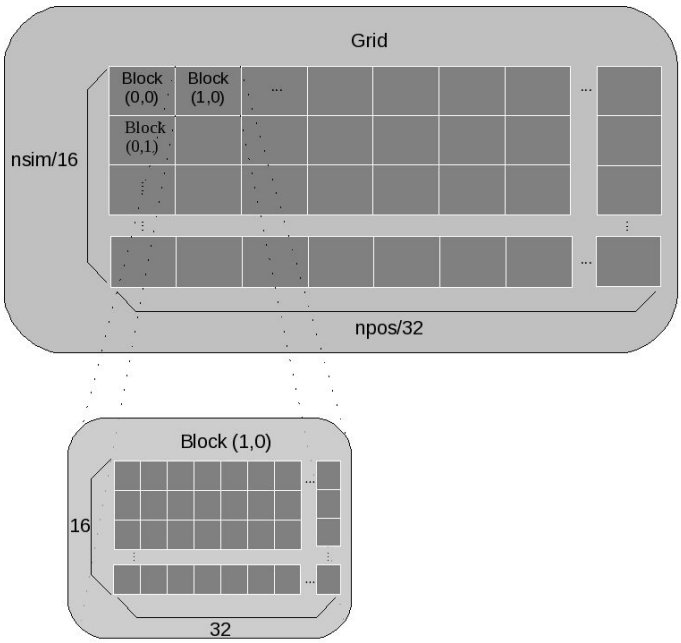

Figure 3.3.1: Example of gridification on the GPU.

the GPU's architecture and what otpimizations were applied to accelerate computations.

3.3.1 Mapping computations on the GPU

Parallelism is present at two levels on a GPU:

- SIMD cores running simultaneously on the GPU;
- hardware threads running simultaneously within a single core.

Mapping computations on a GPU, a process also called gridification, consists therefore in separating the given problem on these two levels of parallelism. The problem must first be broken down into blocks; each block is executed on a single SIMD core. Each block must then be divided into threads, which will be mapped to the hardware threads of the SIMD core.

Several gridifications are implemented in QTLMap depending on the nature of the computations and data access paterns. Fig. 3.3.1 shows an example of such a gridification. In this example, each block handles computations on $32 * 16$ matrices - described in Fig. 3.2.1 - corresponding to different genome positions and simulations. A single thread handles the computations for a single genome position and a single simulation.

3.3.2 Optimizing GPU memory usage

As mentioned in sec. 3.2.3, the algorithm for QTL detection needs to be run on the actual input dataset and a large number of randomly generated datasets in order to test the results against the null hypothesis. In our case, these computations are independent and an obvious data parallelism pattern can be exploited. The amount of available memory on a GPU is nevertheless limited when compared to a CPU's memory. Therefore great care must be exercised when offsetting data to a GPU. The amount of memory required for a single analysis can be divided into three categories:

- memory for input data;
- memory for intermediate results;
- memory for end results.

Input data consist in contingency matrices at each position and simulation - see Fig. 3.2.1. The amount of memory M required for contingency matrices for a linkage analysis is given by the following formula:

$$M = nsim * npos * (1 + (1 + nqtl) * ns) * sizeof(DOUBLE)$$

Where $nsim$ is a user set number of simulations to run, $npos$ is the number of positions to test on the linkage group, $nqtl$ is the number of QTL to look for, ns is the number of sires and $sizeof(DOUBLE)$ is the size in bytes of a double precision float on the given architecture. The previous formula is valid if one decides to store integrally every incidence matrix. Memory optimizations can however be performed. Each matrix contains a first set of population averages, which are independent of both the position in the genome and the family, and a second set of polygenic effects, which are independent of the genome position. All these effects can be factored out and stored only once. The resulting amount of memory required M_{input} is now:

$$M_{input} = nsim * npos * nqtl * ns * sizeof(DOUBLE)$$

The amount of memory required for intermediate results depends on the type of analysis and represents a significant part of the total memory requirements. For large analyses, the input data can be sent to the GPU but computations are limited by the low amount of memory available for intermediate results. In these cases, computations are grouped into workloads, which are then handled sequentially on the GPU. Each workload consists in a number of genome positions that can be computed together within the GPU's global memory. The optimal size Max_pos for a workload is calculated using the following formula:

$$Max_pos = \lfloor (Free_mem - M_{input})/IRsize \rfloor$$

where $Free_mem$ is the amount of memory available on the GPU, M_{input} is the total amount of memory required for input data, and $IRsize$ is the amount of memory

required for intermediate results for a single genome position. Recent CUDA versions allow data transfers between the CPU and the GPU to overlap with computations on the GPU. A possible optimization would be to reduce the size of a workload to half the available memory on the GPU and transfer a workload while the previous one is being computed. Another optimization would be to partition the input data into workloads as is done with intermediate results.

3.3.3 Reducing CPU/GPU transfers

Data transfers between the host (CPU) and the accelerator (GPU) are rather time consuming and need to be optimized. Part of solving the linear system, as mentioned in Algorithm 3.1, consists in determining confounding effects, *i.e.* effects correlated with other effects. These effects are identified by a Choleski decomposition and need to be removed from the dataset for further computations. Subsequent computations are performed on a subset of each matrix - the structure of these matrices is described in Fig. 3.2.1 - excluding confounding effects.

To avoid recopying the matrices, confounding effects are excluded using conditional statements. Nevertheless, branching statements (such as 'if', 'while' etc) can significantly reduce performances on a GPU. This is due to the fact that consecutive Cuda threads are grouped together in warps of 16 threads. Whenever a branching statement occurs, if threads within a single warp take different paths, both paths are executed sequentially, thus breaking parallelism at this level. However, due to the biological nature of the problem, diverging branches never occur since confounding effects are identical for each matrix. Therefore, the overhead induced by adding these branching statements is negligible compared to the overhead induced by transferring the matrices back and forth between the CPU and the GPU.

3.3.4 Optimizing homoskedastic analyses

Each step of the analysis, either using real data or a simulated set, shares a small amount of computations with other steps. This is due to the fact that only performance vectors are randomly generated for simulations. In order to avoid redundancy, matrix multiplications involved in solving the linear system, described at line 4 in Algorithm 3.2 are split into three phases:

- multiplications solely involving fixed effects, shared by all matrices (*i.e.* without performance effects);
- multiplications involving performance effects, which differ from one dataset to another, as well as fixed effects;
- multiplications solely involving performance effects.

The first phase is computed only once on the CPU, while the second and third phases are computed for each dataset in parallel on the GPU. Computations that are

common to each matrix multiplication are thus factored out. This represents a very slight improvement over the previous CPU implementation and was only relevant in the GPU implementation, where these computations are done simultaneously. Dividing these computations also allows us to only keep one copy of the part common to all matrices, while the rest is stored on the GPU in a compact form ; only the relevant halves of the triangular matrices are kept contiguously in memory.

Phases two and three are computed in two distinct Cuda kernels on the GPU in order to optimize memory accesses. Requirements imposed by the Cuda model on memory access patterns to the GPU's global memory are very strict and have a tremendous impact on performances. Memory accesses in these two kernels are optimized for coalescing either by reorganizing data on the GPU or by preloading subsets of the data into shared memory. When breaking coalescing is unavoidable, keeping data locality allows us to benefit from the small cache available on recent graphics cards.

3.4 Experiments and results

Tests were run on machines with two quadcore Intel® Xeon® E5420 (12M Cache, 2.50 GHz, 1333 MHz FSB) processors. Multicore cpu tests were run on the Genotoul platform (`http://www.genotoul.fr`). GPU tests were run on a machine equipped with an Nvidia® C2050 card, with 448 cuda cores at 1150 MHz and 3 GB of main memory. Each test consists of an LDLA analysis over simulated datasets from the 2011 QTL-MAS workshop (`https://colloque.inra.fr/qtlmas`). Two versions of QTLMap are compared here:

- the previous multicore CPU version running with 8 threads [FMG+10];
- the new GPU version in double precision.

For the CPU executions, each of the 8 threads had a dedicated CPU core. Input parameters ranged from 500 to 10000 for the number of simulations, from 9 to 998 for the number of genome positions, and from 5 to 20 for the number of sires.

3.4.1 Execution times

Fig. 3.4.1, Fig. 3.4.2, and Fig. 3.4.3 show the evolution of execution times for both the CPU and the GPU versions over the number of simulations, the number of half-sib families and the number of genome positions respectively. Times for the CPU version are given on the left Y-axis, while times for the GPU version are given on the right Y-axis.

The amount of computations required for the analysis grows linearly with the number of simulations (see Fig. 3.4.1) and the number of considered genome positions (see Fig. 3.4.3). These linear growths were expected, given the structure of the algorithm - lines 2 and 3 of Algorithm 3.1. On the other hand, run times grow

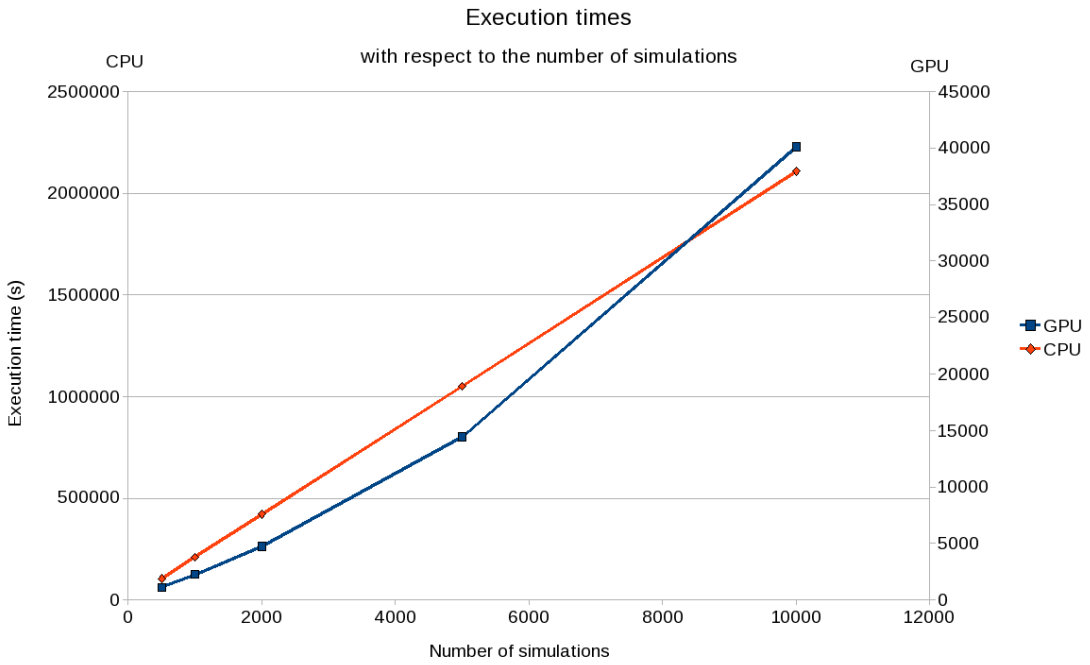

Figure 3.4.1: Evolution of the execution time with respect to the number of simulations.

polynomially with the number of sires - the polynome depends on the type of analysis performed - (see Fig. 3.4.2). Tab. 3.1 shows the values and ranges of values for fixed and variable parameters used in Fig. 3.4.1, Fig. 3.4.2, and Fig. 3.4.3. The most time consuming analysis, using 10000 simulations, 20 sires, and covering 998 genome positions, took more than 3 weeks to compute on the CPU, and slightly more than 11 hours on the GPU.

3.4.2 Speedups

Fig. 3.4.4 (respectively Fig. 3.4.5 and Fig. 3.4.6) shows the evolution of the speedups between the two versions of QTLMap with respect to the number of simulations (respectively to the number of sires and to the number of genome positions).

Tab. 3.2 shows values and ranges of values for fixed and variable parameters used in Fig. 3.4.4, Fig. 3.4.5, and Fig. 3.4.6. Fig. 3.4.4, Fig. 3.4.5, and Fig. 3.4.6 show that speedups remain stable with increasing values in all three dimensions - number of genome positions, number of simulations and size of the population. Overall, the GPU version performs about 70 times faster than the multicore CPU version. This speedup, however, cannot entirely be attributed to the use of a graphics card. Indeed, the CPU version does not benefit from certain optimizations applied specifically to the GPU version - one of which is described in sec. 3.3.4 -, nor does it take advantage of SSE instructions. Optimizing the CPU version would probably reduce

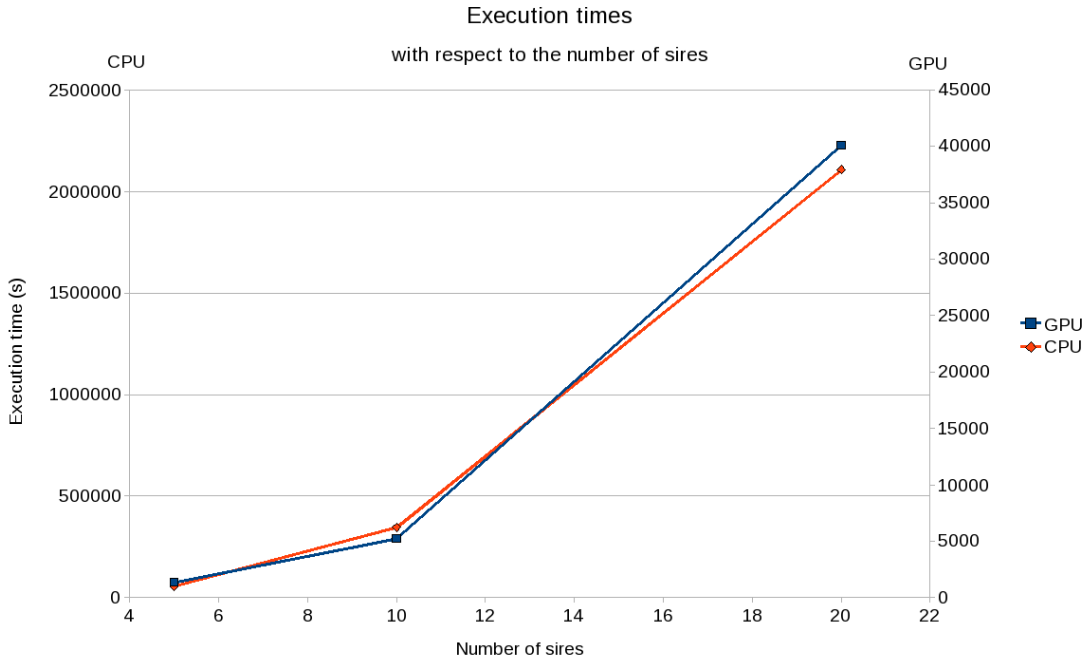

Figure 3.4.2: Evolution of the execution time with respect to the number of half-sib families.

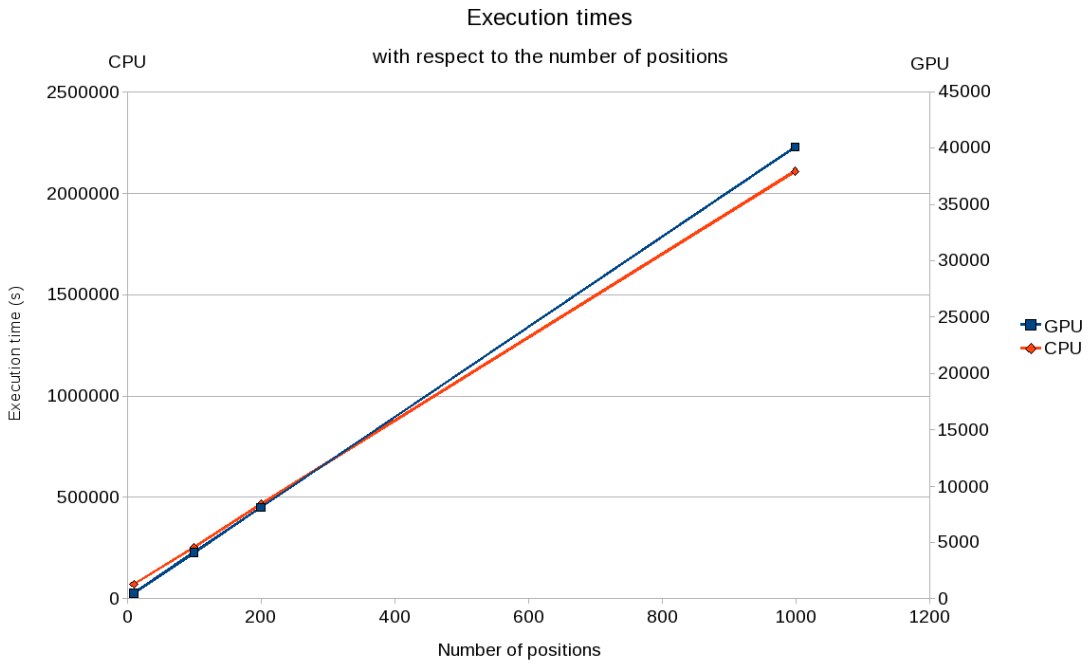

Figure 3.4.3: Evolution of the execution time with respect to the number of genome positions.

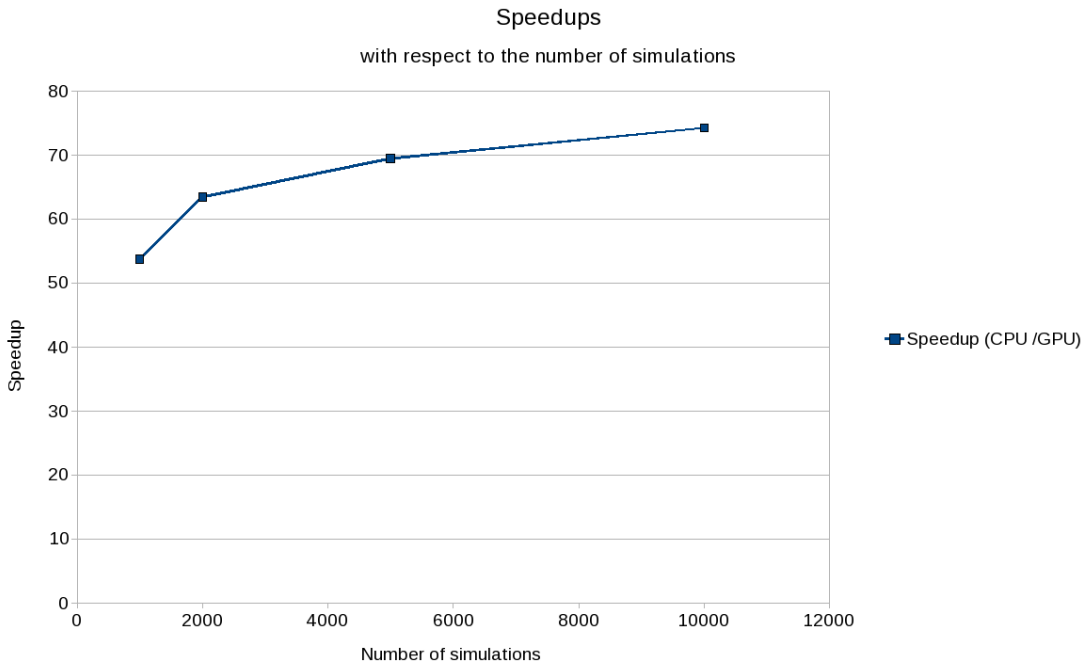

Figure 3.4.4: Speedup with respect to the number of simulations.

its run times by a factor of 3 or 4.

The multicore CPU version of QTLMap is not designed to run optimally for low numbers of genome positions. In the multicore CPU version of QTLMap, data structures are allocated and initialized for each simulation and then amortized over computations for each genome position. On the contrary, in the GPU version of QTLMap, a single set of data structures is allocated and initialized for a large set of simulations and then amortized over both genome positions and simulations. Consequently, large speedups are observed on Fig. 3.4.6 between the GPU and the CPU versions for low numbers of genome positions. These speedups are not representative of the true acceleration obtained by porting QTLMap on the GPU; they simply illustrate the fact that the CPU version does not perform optimally for low numbers of genome positions.

3.5 Conclusion

We propose a new version of existing software QTLMap. QTLMap is a tool for QTL detection, a computationally heavy procedure. This new version takes advantage of GPUs to speed up computations. Computations using this new version are between 50 and 75 times faster than computations using the previous multicore implementation, while maintaining the same results and precision. Reduced runtimes allow geneticists to consider more precise and time consuming analyses by

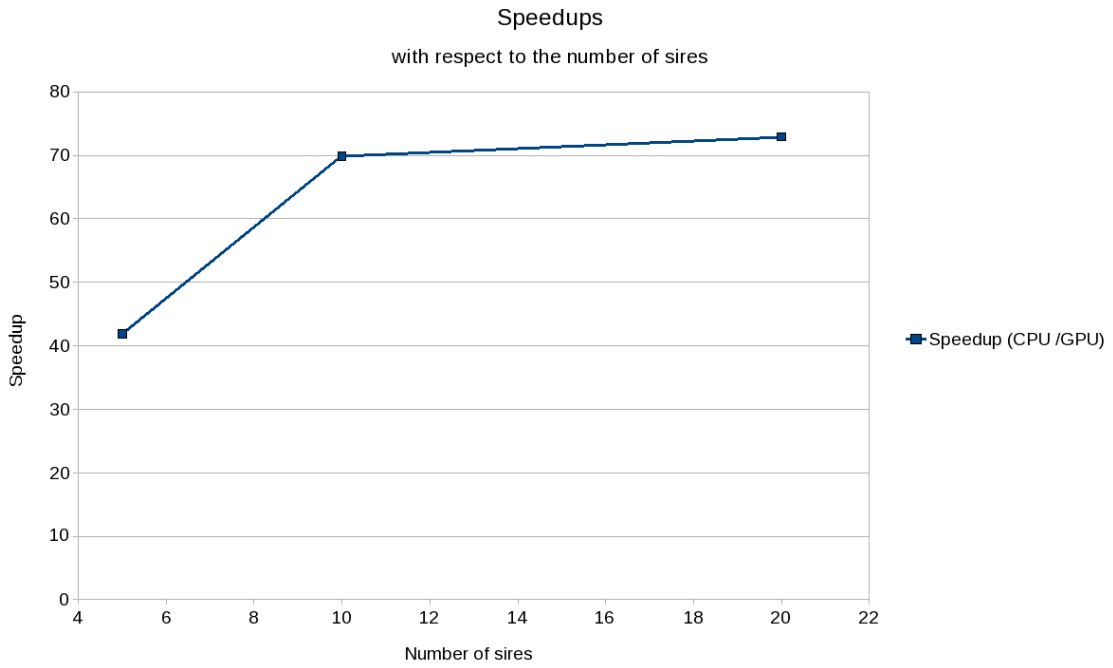

Figure 3.4.5: Speedup with respect to the number of half-sib families.

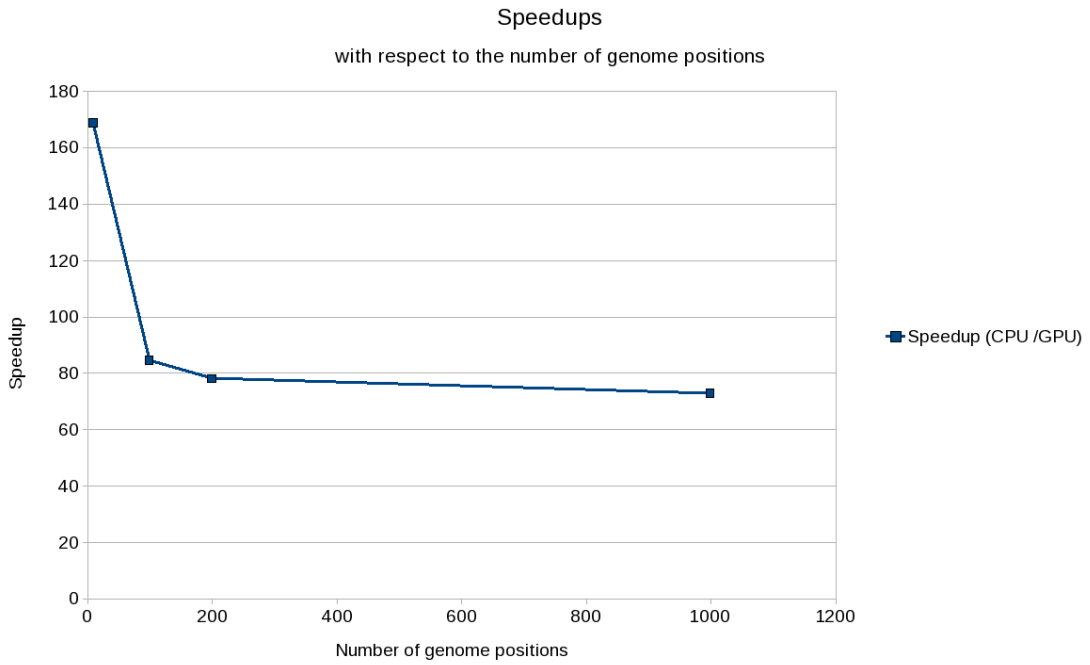

Figure 3.4.6: Speedup with respect to the number of genome positions.

	# of simulations	# of sires	# of genome positions
Fig. 3.4.4	$500 - 10000$	20	998
Fig. 3.4.5	5000	$5 - 20$	998
Fig. 3.4.6	5000	20	$9 - 998$

Table 3.2: Values and ranges of values for fixed and variable parameters used in Fig. 3.4.4, Fig. 3.4.5, and Fig. 3.4.6.

increasing the number of simulations or the number of studied genome positions. Reduced runtimes also allow geneticists to consider new analyses, such as multi-QTL analyses. All versions of QTLMap are available under CeCILL licences at `http://www.inra.fr/qtlmap/`.

Future work include the promotion and use of parallel computing in statistical genetics, focusing on two applications of the Single Nucleotide Polymorphism (SNP) chip technology:

- Dissection of the genetic architecture of characters through Genome Wide Association Studies (GWAS);
- Genomic Selection (GS).

SNP chip technology now makes possible the genotyping on millions of SNPs of tens or hundreds of thousands of individuals, thus increasing the demand for much faster computations. Faster computations are needed both for implementing more precise genetic models in research of trait genetic determinants, and for the industrial exploitation of genomic data, with production of statistical information at regular time intervals.

The algorithms used in QTLMap were not originally thought in parallel. They however exhibit a high degree of parallelism and proved to be perfect candidates for a GPU implementation. The large number of identical and independent operations to execute are ideal to map onto the SIMD architecture of the GPU. This ideal case is far from being the norm and in many cases, one has to either discard the GPU as a potential accelerator for a specific problem or where possible, find a new algorithm exhibiting more parallelism.

	# of simulations	# of sires	# of genome positions
Fig. 3.4.1	$500 - 10000$	20	998
Fig. 3.4.2	10000	$5 - 20$	998
Fig. 3.4.3	10000	20	$9 - 998$

Table 3.1: Values and ranges of values for fixed and variable parameters used in Fig. 3.4.1, Fig. 3.4.2, and Fig. 3.4.3.

4 Efficient Multi-GPU Computation of All-Pairs Shortest Paths

We describe a new algorithm for solving the all-pairs shortest-path (APSP) problem for planar graphs and graphs with small separators that exploits the massive on-chip parallelism available in today's Graphics Processing Units (GPUs). Our algorithm, based on the Floyd-Warshall algorithm, has near optimal complexity in terms of the total number of operations, while its matrix-based structure is regular enough to allow for efficient parallel implementation on the GPUs.

By applying a divide-and-conquer approach, we are able to make use of multi-node GPU clusters, resulting in more than an order of magnitude speedup over fastest known Dijkstra-based GPU implementation and a two-fold speedup over a parallel Dijkstra-based CPU implementation.

4.1 Introduction

Shortest-path computation is a fundamental problem in computer science with applications in diverse areas such as transportation, robotics, network routing, and VLSI design. The problem is to find paths of minimum weight between pairs of nodes in edge-weighted graphs, where the weight $|p|$ of a path p is defined as the sum of the weights of all edges of p. The distance between two nodes v and w is defined as the minimum weight of a path between v and w.

There are two basic versions of the shortest-path problem: in the single-source shortest-path (SSSP) version, given a source node s, the goal is to find all distances between s and the other nodes of the graph; in the all-pairs shortest-path (APSP) version, the goal is to compute the distances between all pairs of nodes in the graph. While the SSSP problem can be solved very efficiently in nearly linear time by using Dijkstra's algorithm [Dij59], the APSP problem is much harder computationally.

Two main families of algorithms exist to solve the APSP problem exactly: the first family is based on the Floyd-Warshall algorithm [CSRL01], while the second derives from Dijkstra's algorithm. The Floyd-Warshall's approach consists in iterating through every vertex v_k of the graph to improve the best known distance between every pair of vertices (v_i, v_j) - see Algorithm 4.1. The complexity of this approach is $O(|V|^3)$, regardless of the density of the input graph. While the algorithm works for arbitrary graphs (including those with negative edge weights), its cubic complexity makes it inapplicable to very large graphs.

Given that the Dijkstra's algorithm solves the SSSP problem, it is possible to

solve the APSP problem by simply running the Dijsktra's algorithm over all source vertices in the graph (see Algorithm 4.2). When using min-priority queues, the complexity of this approach is $O(|E| + |V| \log |V|)$ for the SSSP problem, where V and E are the sets of the vertices and edges, respectively. For the APSP problem, the total complexity is thus $O(|V| * |E| + |V|^2 \log |V|)$, which becomes $O(|V|^3)$ when the graph is complete, but only $O(|V|^2 \log |V|)$ when $|E| = O(|V|)$, making this approach faster than Floyd-Warshall for sparse graphs.

Solving the All-Pairs Shortest Path problem is important not only in transportation-related problems, but in many other domains. It is the first step to obtaining several network measures that are of importance in domains such as social network analysis or in bioninformatics. One such measure is the *betweenness centrality*, which is defined, for any vertex v, as the number of shortest paths between all pairs of vertices that pass through v, and is a measure of a v's centrality (importance) in the network.

Some algorithms use the centrality of the nodes in a network in order to compute its community structure. Furthermore, in several applications, the networks that need to be analyzed may have negative weights, and hence one needs an algorithm that solves the APSP problem for graphs with real (positive as well as negative) weights. In online social networks, for instance, negative weights may be used to indicate antagonism between two individuals [LHK10] or even conflicts and alliances between two groups [TB09]. Causal networks in bioinformatics also use negative edges to represent inhibitory effects [IDN11].

In this chapter, we present an algorithm for solving the APSP problem for graphs with real weights that exploits the great degree of parallelism available in today's Graphics Processing Units (GPU). GPUs and other stream processors were originally developed for intensive media applications and thus advances in the performance and general purpose programmability of these processors have hitherto benefited applications that exhibit computational similarities to graphics applications, namely high data parallelism, high computational intensity, and data locality.

However, many theoretically optimal graph algorithms exhibit few of these properties. Such algorithms often use efficient data structures storing as little redundant information as possible, resulting in highly unstructured data and un-coalesced memory access making them less-than-ideal candidates for streaming processor manipulations. Nevertheless, given the wide applicability of graph-based approaches, the massive parallelism afforded by today's graphics processors is too compelling to ignore; current GPUs support hundreds of cores per chip and even future CPUs will be many core.

Our approach aims to exploit the structure of the input graphs and specifically their partitioning properties. Our algorithm will be especially efficient if the input graph has a good separator, which means (informally) that it can be divided into two or more equal parts removing $o(n)$ vertices or edges, where n is the number of the vertices of the graph. Such graphs are frequently seen in road networks, geometric networks and social networks; all planar graphs also satisfy this property. To harness the parallel computing power for solving the path problem on such

Algorithm 4.1 Floyd-Warshall algorithm.

```
INPUT: A graph G(V,E), where V is a set of vertices and E a set of
    weighted edges between these vertices.
OUTPUT: The distance of the shortest path between any two pairs of
    vertices in G.
for each vertex v in V
  dist[v][v] = 0 end for for each edge (u,v) in E
  dist[u][v] = w(u,v)  // the weight of the edge (u,v)
end for
for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
      dist[i][j] =
        min(dist[i][j], dist[i][k] + dist[k][j])
    end for
  end for
end for
return dist
```

Algorithm 4.2 Dijkstra's Single Source Shortest Path algorithm.

```
INPUT: A graph G(V,E), where V is a set of vertices and E a set of
    weighted edges between these vertices. A source vertex from V.
OUTPUT: The distance of the shortest paths between the source vertex
    and every vertex in V.
for each vertex v in V
  dist[v] = infinity
  previous[v] = undefined
end for
dist[source] = 0
Q = V
while Q is not empty
  u = vertex in Q with smallest distance in dist[]
  Q = Q\{u}
  if dist[u] = infinity
    break
  for each neighbor v of u in Q
    alt = dist[u] + dist_between(u, v)
    if alt < dist[v]
      dist[v] = alt
      previous[v] = u
      decrease-key v in Q
    end if
  end for
end while
return dist
```

graphs, we partition the input graphs into an appropriate number of parts and solve the APSP on each part and then use the partial solutions to compute the distances between all pairs of vertices in the graph.

Our algorithm, based on the Floyd-Warshall algorithm, has near quadratic (i.e. near optimal) complexity with respect to the number of nodes, while its matrix-based structure is regular enough to allow for efficient parallel implementation on the GPUs. By applying a divide-and-conquer approach, we are able to make use of multi-node GPU clusters, resulting in more than an order of magnitude speedup over fastest known (Dijkstra-based) GPU implementation and a two-fold speedup over a parallel Dijkstra-based CPU implementation.

In what follows, sec. 4.2 presents recent parallel implementations for solving the APSP problem; in sec. 4.3, we detail the principles of our partitioned algorithm; sec. 4.4 focuses on the structure of the data and the computations and how the algorithm is implemented on large multi GPU clusters. Finally, sec. 4.5 shows the results of two experiments and possible ways to improve our implementation.

4.2 Related Work

When considering a distributed GPU implementation, both the Floyd-Warshall and Dijkstra's approaches have advantages and drawbacks. Though slower for sparse graph, a Floyd-Warshall approach has the advantage of having regular data access patterns that are identical to those of a matrix multiplication. The amount of computations required for a given graph, using a Floyd-Warshall approach, solely depends on the number of vertices in the graph; therefore, balancing workloads between different processing units can be achieved easily. Dijkstra's approach is much faster for sparse graphs but, to achieve best performance, requires complex data structures which are difficult to implement efficiently on a GPU.

Implementing parallel solvers for the APSP problem is an active field of research. [HN07] proposed GPU implementations of both the Dijkstra and Floyd-Warshall algorithms to solve the APSP problem and compared them to parallel CPU implementations. Both approaches however require that the whole graph fit in the GPUs memory. They report solving APSP for a $100k$ vertex graph in around 22 minutes on a single GPU. A cache-efficient parallel, blocked version of the Floyd-Warshall algorithm for solving the APSP problem in GPUs is described in [KK08]. While the graphs mentioned in [KK08] are larger than what would fit onto GPU on-board memory, the largest graph instances described in the paper are still only around $10k$ vertices.

[BGB10] proposed a blocked-recursive Floyd-Warshall approach. Their implementation, running on a single GPU, shows a speedup of 17-45 when compared to a parallel CPU implementation and outperforms both GPU implementations from [HN07]. Their blocked-recursive implementation also requires that the entire graph fit in the GPU's global memory; therefore, they only report timings for graphs with up to $8k$ vertices. [OIH12] proposed an improvement over the GPU implementation

of Dijkstra for APSP from [HN07] by caching data in on-chip memory and exhibiting a higher level of parallelism. Their approach showed a speedup of $2.8 - 13$ over Dijkstra's SSSP-based method of [HN07]. [MNS12] also proposed a blocked Floyd-Warshall algorithm that they implemented for computations on a single GPU and a multicore CPU simultaneously. Their implementation handles graphs with up to $32k$ and achieves near peak performance. Only [OATLGE13] report solving APSP on large graphs - up to $1024k$ vertices. Using an SSSP-based Dijkstra approach, their implementation runs on a multicore CPU and up to 2 GPUs simultaneously. Recent experimental work on parallel algorithms for solving just the SSSP problem for large graph instances using a Δ-stepping approach [MS03] is described in [MBBC07].

Our Contribution: We propose a novel APSP algorithm and its parallel implementation to compute all shortest distances between all pairs of vertices of a graph with good partitioning properties. To make the algorithm scalable to large graphs, our implementation uses a combination of shared and distributed-memory GPU computing; the current implementation targets executions on large clusters of GPUs in order to handle graphs with up to a million vertices. Experimentations showed that the trillion shortest distances of a million vertex graph can be found in less than 25 minutes using 64 cluster nodes with 2 GPUs each.

We view our contributions in the following:

1. We develop a new Floyd-Warshall-based algorithm that is simultaneously work-efficient, has a high-degree of parallelism, and is build upon matrix operations; we are aware of no previous APSP algorithm with such properties.

2. Our implementation is using massive parallelism; both fine-grained at GPU level as well as coarse-grained employing up to 300 GPUs.

3. Our algorithm beats the previous algorithms by orders of magnitude with respect to running times using the same or similar computational resources.

4. In addition to the fact that our algorithm is faster than Dijkstra-based algorithms, it also has the advantage that it works with arbitrary-negative as well as positive-weights.

5. The matrix structure of our algorithm will allow it to get additional efficiency boost from any pipelined vector features not available in current GPUs.

4.3 Algorithm details

In this section we give the overall structure and the idea of the algorithm and describe its individual steps, but without discussing details of the GPU implementation. We start with an overview of the algorithm and then give details on each of its steps.

4.3.1 Overview

Our algorithm takes as input a weighted directed or undirected graph G with n vertices and computes the distances between all pairs of vertices of G. We currently do not output routing information, which can be used to reconstruct the shortest paths, but computing such an information requires a minor modification in the algorithm and would increase the run times and memory requirements by at most a constant factor of two.

Our algorithm is based on a divide-and-conquer approach and consists of four steps (see Algorithm 4.3). In the first step, the original graph G is partitioned into k components of roughly equal sizes using a min-cut like heuristic - our implementation uses a k-way partitioning method from the METIS library [KK98b]. In the second step, the APSP problem is solved on each component independently; in the third step the distance information computed for the components is used to compute distances between all pairs of boundary vertices of G (a *boundary* vertex is one that is adjacent to a vertex from another component); and in the final step the information obtained in steps two and three is combined to compute shortest paths between non-boundary pairs of vertices of G.

We will use the following notation: $\text{dist}_i(v, w)$ will denote the (approximate) value of the distance between v and w computed in Step i, for $i = 2, 3, 4$, and $\text{dist}_G(v, w)$ will denote the (exact) distance in G. Next we will describe the steps in more detail.

4.3.2 Step 1: Graph decomposition

In Step 1 the input graph G is divided into k components of roughly equal sizes. The decomposition is done be identifying a set of edges (a *cut* set) whose removal from G results into a disconnected graph of k parts we call *components*. The set of all components is called a *partition*. Note that while by the standard definition in graph theory a component is connected, this is not a requirement in our case (although in the typical case our components will be connected). A requirement is that every vertex in G belongs to exactly one component of the partition. Moreover, in order for the resulting APSP algorithm to be efficient, the cut set of edges should be small. Not all classes of graphs have such partitions, but some mportant classes do. These include the class of planar graphs, the class of graphs of low genus, some geometric graphs, and graphs corresponding to networks with good community structure.

4.3.3 Step 2: Computing distances within each graph component

Step 2 involves computing the distances in each component of the partition $\mathcal{P}$ of G using a conventional algorithm, e.g., the Floyd-Warshall's or Dijkstra's algorithm. For each component $C \in \mathcal{P}$ and any two vertices s and t of C, the output of this step is the minimum length of a path between s and t that is restricted to lie entirely in C. Hence, the distance computed between s and t may be larger that the distances

Algorithm 4.3 Partitioned All-Pairs Shortest Path algorithm

```
INPUT: A graph G(V,E), where V is a set of vertices and E a set of
    weighted edges between these vertices.
OUTPUT: The distance of the shortest path between any two pairs of
    vertices in G.
function partitioned_APSP(G)
  // Step 1
  for each Component C in G
      Floyd-Warshall(C) %compute_APSP(C)
  end for
  // Step 2
  Graph BG = extract_boundary_graph(G)
  compute_apsp(BG)
  // Step 3
  for each Component C in G
    Floyd-Warshall(C) %compute_APSP(C)
  end for
  // Step 4
  for each Component C1 in G
    for each Component C2 in G
      compute_apsp_between_components(C1, C2)
    end for
  end for
end function
```

between s and t in G, if there is a shorter path between them that goes out of C. Nevertheless, as we will show in the next subsections, the computed approximate distances can be used to efficiently compute the accurate distances in G.

In order to implement this step, for each component $C \in \mathcal{P}$, a subgraph is extracted containing vertices from the current component and existing edges between these vertices. Any APSP algorithm can then be applied in order to compute distances in each of these sub-graphs. This step thus has k independent tasks - one for each sub-graph - that can be computed in parallel. Since each component contains roughly n/k vertices, using an algorithm whose complexity solely depends on the number of vertices allows these tasks to be computed in roughly the same number of operations. This property can be advantageous depending on the type of parallelism that we want to exploit.

4.3.4 Step 3: Computing distances in the boundary graph

In step 3, we first extract the *boundary graph* BG of G with respect to the partition $\mathcal{P}$. The vertices of BG are defined to be all boundary vertices of G. There are two types of edges of BG. The first type of edges are edges in G between boundary vertices from different components. The weights on these edges are the same as their weights in G. The second type of edges, which we call *virtual* edges, are between boundary vertices in the same components - for any two boundary vertices v and w belonging to the same component C there is an edge (v, w) in BG with weight equal to the distance between v and w computed in Step 2. Hence, BG is a compressed version of the original graph, where all non-boundary vertices have been removed, and instead of them shortest path information encoded in the weights of the new edges of BG. Having constructed BG, we then solve for it the APSP problem using a conventional APSP algorithm.

Despite the fact that the distances encoded in the weights of the new edges of BG are only approximate, the distances between the boundary nodes of BG computed at the end of Step 3 are exact. The next lemma formally establishes this fact.

Lemma 1. *For any two boundary vertices v and w, the distance between v and w in BG is equal to their distance in G.*

Proof. Let $p = (v = x_1, x_2, \ldots, x_l = w)$ be a shortest path in G and let $(x_{b_1}, x_{b_2}, \ldots, x_{b_j})$ be the subsequence of all boundary vertices in p, i.e., $1 = b_1 < \cdots < b_j = l$ and there are no boundary vertices on p between x_{b_i} and $x_{b_{i+1}}$. Hence $p' = (x_{b_1}, x_{b_2}, \ldots, x_{b_j})$ is a path in BG. We are going to estimate the length of p'.

Let $h = (x_{b_i}, x_{b_{i+1}})$ be an edge of p'. If x_{b_i} and $x_{b_{i+1}}$ are from different components, then, by the definition of BG, h is also an edge of G with the same weight as in BG. If x_{b_i} and $x_{b_{i+1}}$ are from the same component C (Figure Fig. 4.3.1), then h corresponds to a subpath $q = (x_{b_i}, x_{b_i+1}, \ldots, x_{b_{i+1}})$ of p consisting of vertices from only C, by the assumption that p' contains all the boundary vertices of p. Hence, the weight of h and the length of q are the same. By induction on the number of the edges of p', p and p' have the same length, which implies that the distance between

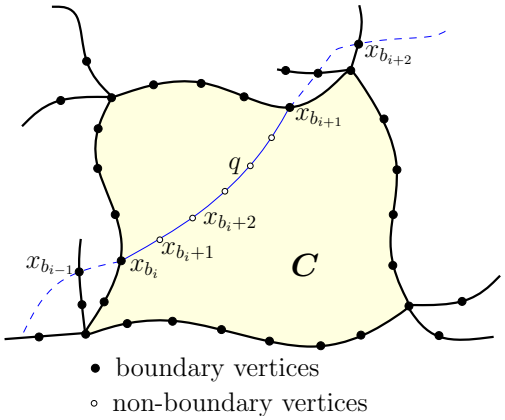

Figure 4.3.1: Illustration to the proof of Lemma Theorem 1. The shaded region illustrates a component C with the subpath $q = (x_{b_i}, x_{b_i+1}, \ldots, x_{b_{i+1}})$ of p inside it.

v and w in BG is no greater than the distance between them in G. The reverse inequality is obtained in the same way, namely, by showing that any path in BG can be transformed into a path of the same length in G by replacing each virtual edge of the former with the corresponding shortest path computed in Step 2. The claim follows. □

This step presents no apparent parallelism, since only one task needs to be computed. This absence of parallelism at this step may be a major bottleneck for a coarse-grain parallel implementation as boundary graphs can be very large. This issue can however be mitigated by applying our current algorithm recursively on the boundary graph. Boundary graphs are nevertheless denser than the original graph with the addition of virtual edges at Step 2. Boundary graphs are therefore less easily partitioned than input graphs - the number of edges cut per node for a given number of components will be higher.

4.3.5 Step 4: Distances between non-boundary vertices

In Step 4 we compute distances where at least one vertex is non-boundary using the information computed in Steps 2 and 3. In order to compute the distance between two non-boundary vertices v_i and v_j from (not necessarily different) components C_i and C_j respectively, we need to find boundary vertices b_i and b_j from components C_i and C_j, respectively, that minimize the sum $\text{dist}_2(v_i, b_i) + \text{dist}_3(b_i, b_j) + \text{dist}_2(b_j, v_j)$, where dist_2 and dist_3 are the distances computed in Step 2 and Step 3, respectively. By our analysis above, dist_3 is the same as the distance in G, but dist_2 is not. We need therefore to prove that such a method produces accurate distances in G.

Lemma 2. *Let v_i and v_j be two vertices from different components C_i and C_j,*

respectively. Define $B_i = C_i \cap BG$, $B_j = C_j \cap BG$, *and*

$$dist_4(v_i, v_j) = \min\{dist_2(v_i, b_i) + dist_3(b_i, b_j) + dist_2(b_j, v_j) \mid b_i \in B_i, b_j \in B_j\}. \quad (4.3.1)$$

Then $dist_4(v_i, v_j)$ *is equal to the distance in* G *between* v_i *and* v_j.

Proof. Let p be a shortest path in G between v_i and v_j. Since v_i and v_j belong to different components, then p will contain at least one vertex from B_i and at least one vertex from B_j. Let b_i be the first vertex on p from B_i and b_j be the last vertex on B_j (Figure Theorem 2). Let p_1 be the portion of p between v_i and b_i, p_2 be the portion between b_i and b_j, and p_3 – the portion between b_j and v_j. Since any subpath of a shortest path is also a shortest path between the corresponding endpoints, p_1 is a shortest path in G between v_i and b_i, i.e., $|p_1| = \text{dist}_G(v_i, b_i)$. Moreover, by the definition of b_i as the first boundary point of C_i on p, p_1 is entirely in C_i and hence $|p_1| = \text{dist}_2(v_i, b_i)$. In the same way one can prove that $|p_2| = \text{dist}_2(b_j, v_j)$. Finally, $|p_3| = \text{dist}_G(b_i, b_j) = \text{dist}_3(b_i, b_j)$ by Lemma Theorem 1. Hence

$$|p| = |p_1| + |p_2| + |p_3| = \text{dist}_2(v_i, b_i) + \text{dist}_3(b_i, b_j) + \text{dist}_2(b_j, v_j).$$

By the definition of $\text{dist}_4(v_i, v_j)$ as a minimum over all $b_i \in B_i, b_j \in B_j$, the last equality implies $\text{dist}_4(v_i, v_j) \leq \text{dist}_G(v_i, v_j)$. But since $\text{dist}_4(v_i, v_j)$ is a length of a path between v_i and v_j, while $\text{dist}_G(v_i, v_j)$ is the length of a shortest path, then $\text{dist}_4(v_i, v_j) \geq \text{dist}_G(v_i, v_j)$. Combining the last two inequalities we infer that none of them can be a strict inequality, i.e., $\text{dist}_4(v_i, v_j) = \text{dist}_G(v_i, v_j)$. □

Lemma 3. *Let* v_i *and* v_j *be two vertices from component* C_i. *Then* $dist_G(v_i, v_j) = \min\{dist_2(v_i, v_j), dist_4(v_i, v_j)\}$, *where* $dist_4$ *is as defined in Lemma Theorem 2.*

Proof. Consider the following two cases. If p leaves C_i, then p should cross the boundary B_i at least twice. Define b_i and b_j as the first and last vertex from B_i on p. Then exactly the same arguments as in Lemma Theorem 2 apply to the three paths into which b_i and b_j divide p. In this case $\text{dist}_G(v_i, v_j) = \text{dist}_4(v_i, v_j)$. If p does not leave p, then Step 2 will compute the accurate distance in G between v_i and v_j, and therefore $\text{dist}_G(v_i, v_j) = \text{dist}_2(v_i, v_j)$. □

The lemmas imply that the distances in G between all pairs of vertices where at least one of the vertices is non-boundary can be computed by using (eq:step4). Since we don't know which pair (b_i, b_j) of boundary nodes corresponds to the minimum in (eq:step4), we have to try all such pairs, resulting in total of $|B_i||B_j|$ operations needed for computing $\text{dist}_G(v_i, v_j)$. For a graph with k components, we need to compute the distances between pairs in any pair of components; we therefore have k^2 independent tasks. Components being of roughly equal sizes, these tasks also represent the same amount of computations. This step is the most computationally intensive, but presents massive, already balanced, coarse-grain parallelism.

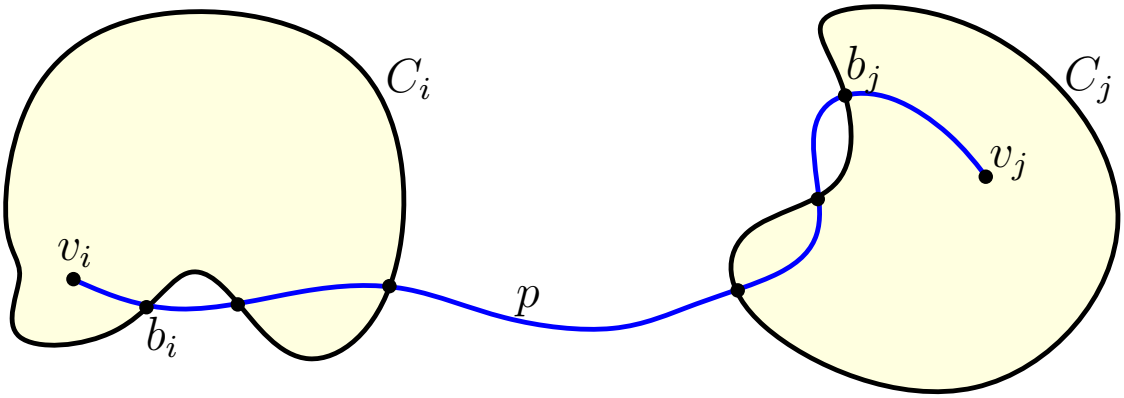

Figure 4.3.2: Illustration to the proof of Lemma Theorem 2. Note that while in the figure both v_i and v_j are non-boundary, the proof does not make such an assumption.

4.4 Implementation

In this section, we first focus on how operations described in the previous section translate in terms of data structures. We then detail the two-level parallel aspect of our implementation. We finally describe the current main memory bottleneck of our approach.

4.4.1 Data organization

A simple way to represent a weighted graph is to use an adjacency matrix. For very large graphs however, such a memory intensive representation is often avoided. Instead, large sparse graphs are stored using lists; sub-matrices, corresponding to sub-graphs, are extracted from these lists. For simplicity reasons, we can however assume that a large adjacency matrix representation is available and keep in mind that sub-matrix extraction operations are slightly more costly than they appear. We are also taking into account the fact that, even when the input graph (matrix) is sparse, the output is always a dense matrix as it encodes the distances between all pairs of vertices.

Partitioning the graph is performed using a k-way partitioning routine from the METIS library [KK98b]. The result is a partitioning of the graph into k parts such that the number of edges with endpoints in different parts is minimized. Since that partitioning problem is NP-hard, METIS computes an approximation based on heuristics. Vertices are then reordered so that vertices belonging to the same component are numbered consecutively starting with the boundary vertices - see Figure Fig. 4.4.1.

Diagonal sub-matrices contain information about sub-graphs for each component; non-diagonal sub-matrices contain known shortest distances between components. Within each diagonal sub-matrix, the top left sub-matrix contains information about

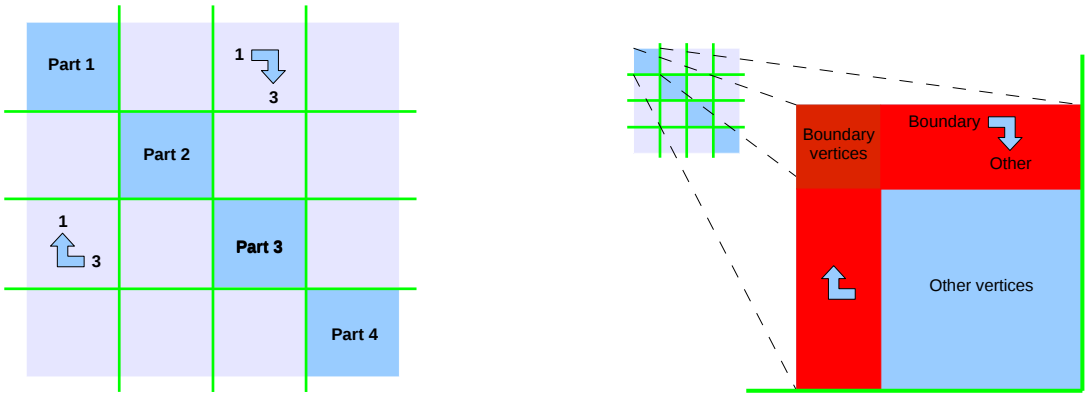

Figure 4.4.1: Adjacency matrix after reordering of the vertices. Vertices from the same component are stored contiguously starting with boundary vertices (in red).

the sub-graph induces by boundary vertices of the component; the bottom right sub_matrix contains information about the sub-graph induced by non-boundary vertices of the component and the rest of the diagonal sub-matrix contains known shortest distances between boundary and non-boundary vertices.

For Step 2, diagonal sub-matrices are extracted; a Floyd-Warshall approach is then used to compute shortest distances. The Floyd-Warshall algorithm guarantees that the total number of operations for a single matrix solely depends on the size of the matrix. Since all components of the graph have roughly the same number of vertices, all diagonal sub-matrices represent roughly the same amount of operations.

For Step 3, the boundary matrix is extracted – see Figure Fig. 4.4.2. We then apply the same algorithm recursively reducing the number k of component at each iteration. Recursion stops when $k = 1$ or when the boundary graph becomes so dense that it does not have good partitioning (in terms of number of boundary vertices). At that point the APSP subproblem is solved using Floyd-Warshall.

For Step 4, we compute shortest distances between every pair of distinct components. This process corresponds to filling non-diagonal sub-matrices. For two components I and J, filling the associated, I to J, non-diagonal sub-matrix requires information from three sub-matrices:

- the non-diagonal sub-matrix being filled. We are particularly interested in the part of the sub-matrix containing shortest distances between boundary vertices from component I to boundary vertices from component J.

- the diagonal sub-matrix corresponding to component I - located in the same row as the non-diagonal sub-matrix being filled. We are particularly interested in the part of this diagonal sub-matrix that contains shortest distances from any vertex of component I to boundary vertices.

- the diagonal sub-matrix corresponding to component J - located in the same column as the non-diagonal sub-matrix being filled. We are particularly in-

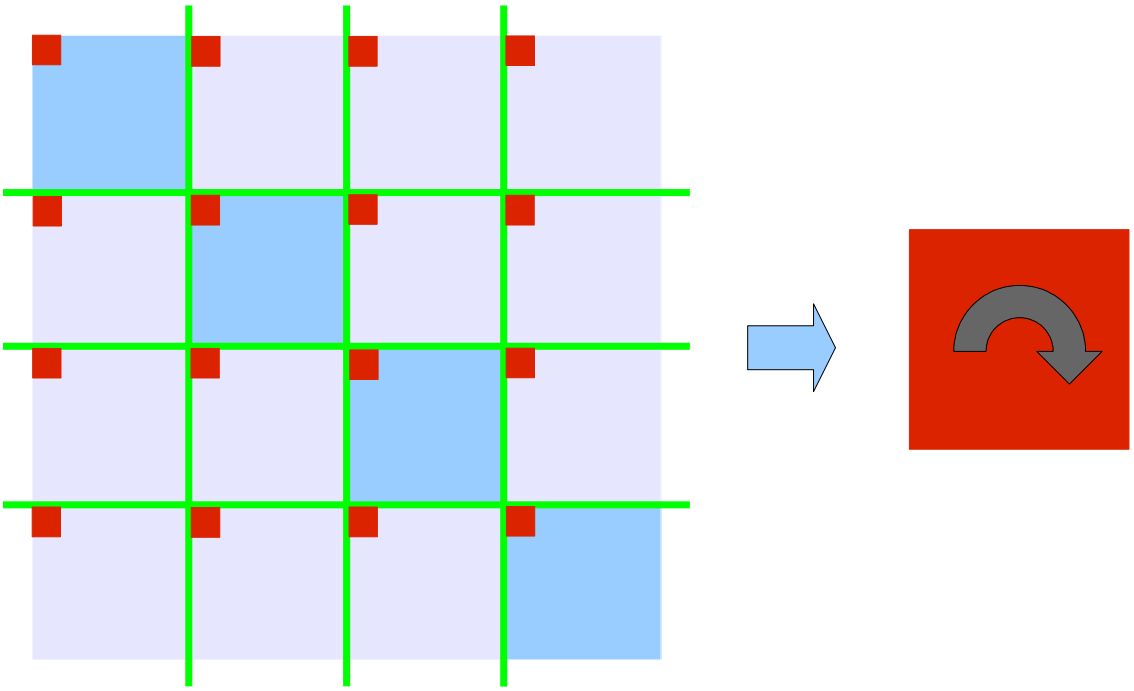

Figure 4.4.2: The boundary matrix, here in red, is scattered over the adjacency matrix. Step 3 consits in reconstituting the boundary matrix and computing shortest distances.

terested in the part of this diagonal sub-matrix that contains shortest distances from boundary vertex of component J to any vertex - see left of Figure Fig. 4.4.3.

Shortest distances from vertices from component I to vertices from component J are obtained by multiplying the three parts of sub-matrices - as shown on the right of Figure Fig. 4.4.3 - where $(+, *)$ operations are replaced with $(min, +)$ operations.

4.4.2 Work analysis

Next we will try to estimate the work (number of operations) of the algorithm. Since the work depends on the partitioning properties of the input graph, we will do the analysis for the case of planar bounded-degree graphs. For that class of graphs, there exists a partitioning of any n-vertex graph into k parts such that the number of boundary vertices in each part is $O(\sqrt{n/k})$ [Fre87]. We make the assumption that METIS produces a partition with such properties. Although the partition METIS produces does not come with theoretically guaranteed bounds, it works in practice better than alternative algorithms that have such guarantees, which is the reason we chose it. The time needed for Step 1 is $O(n \log n)$.

In Step 2, we have k subtasks of computing APSP on graphs of size $O(n/k)$ using an algorithm of cubic complexity, so the number of operations for that step is

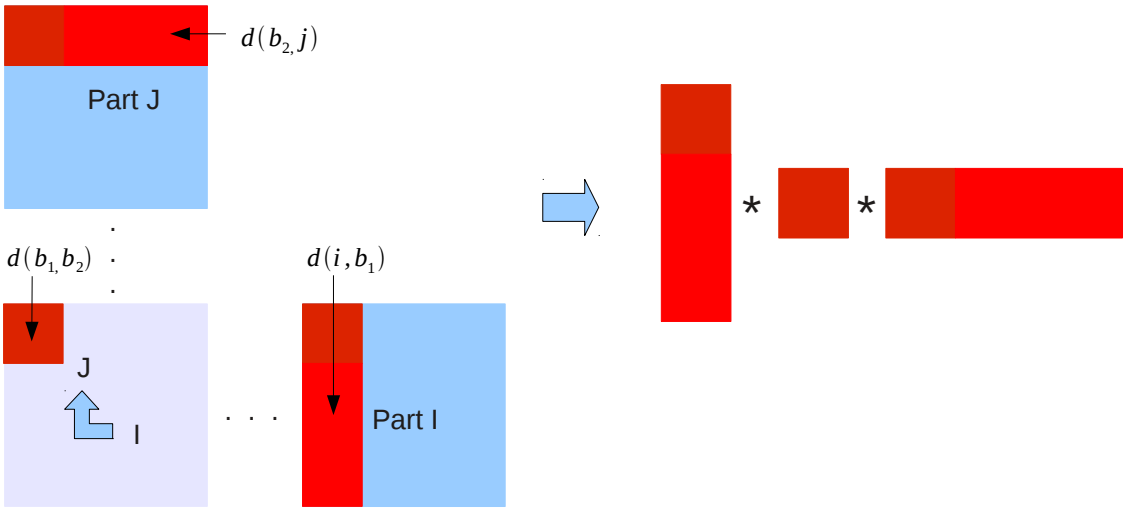

Figure 4.4.3: Computations associated to each non-diagonal sub-matrix uses data from 2 diagonal sub-matrices and part of the non-diagonal sub-matrix itself. Computations are similar to matrix multiplications.

$k(n/k)^3 = n^3/k^2$.

In Step 3, we have to solve the APSP on a graph of size $O(k\sqrt{n/k}) = O(\sqrt{kn})$. Using an algorithm with complexity $O(N^\alpha)$, where N is the number of the vertices of the subgraph, the number of operations for this step is $O((kn)^{\alpha/2})$. For Step 4, we have k^2 tasks and each tasks involves the multiplication of three matrices with dimensions $n/k \times \sqrt{n/k}$, $\sqrt{n/k} \times \sqrt{n/k}$, and $\sqrt{n/k} \times n/k$, respectively. Computing the product of the first and the second matrix takes

$$O((n/k)\sqrt{n/k}\sqrt{n/k}) = O((n/k)^2)$$

operations and finding the product of the resulting $n/k \times \sqrt{n/k}$ matrix and the third matrix takes

$$O((n/k)\sqrt{n/k}(n/k)) = O((n/k)^{5/2})$$

operations, which is the dominating term. Hence, the total number of operations for Step 4 is

$$O(k^2(n/k)^{5/2}) = O(n^{5/2}/k^{1/2}).$$

The total number of operations is the sum of the numbers computed for Steps 1, 2, 3, and 4 and is minimized when $(kn)^{\alpha/2} = n^{5/2}/k^{1/2}$ or $k^{\alpha+1} = n^{5-\alpha}$. If in Step 3 Floyd-Warshall is used, then $\alpha = 3$ and $k = n^{1/2}$ is optimal, resulting in a bound of $O(n^{9/4})$ for the total number of operations, slightly worse than the theoretical lower bound of $O(n^2)$. Our implementation in fact uses recursion in Step 3 so the total complexity is even closer to quadratic, but we will skip the details of the exact evaluation since the analysis gets much more complex.

4.4.3 Parallel implementation

Our implementation specifically targets large clusters of hybrid systems - possessing both a multicore CPU and manycore GPUs. This implementation exploits parallelism at two levels. At a coarse-grain level, large independent tasks - corresponding to computations of diagonal and non-daigonal sub-matrices - can be performed simultaneously on different nodes of a cluster. At a fine-grain level, each task is computed on a massively parallel GPU. Remaining CPU cores handle tasks that are not suited for GPUs: input/output file operations and communication with other nodes.

Coarse-grain parallelism

Steps 2 and 4 of our algorithm exhibit interesting parallel properties: a large number of balanced, independent tasks; k tasks for Step 2 and $k^2 - k$ for Step 4. Using the MPI standard [SOW+95], these tasks are distributed accross nodes of the cluster for simultaneous computations. One master node is in charge of reading the input graph file, calling the partitioning routine and sending tasks to a number of slave nodes equal to the number of available GPUs on the cluster. Depending on the cluster's topology, the number of master and slave nodes will not match the number of physical nodes used on the cluster if each cluster node contains more than one GPU.

For Step 3, the large initial boundary matrix is computed recursively using the same algorithm with decreasing values for the number k of components. The amount of independent tasks therefore decreases with k, until a single, smaller boundary matrix is obtained and computed by a single slave node.

Fine grain parallelism

Upon receiving a task from the master node, each slave node then sends the corresponding data to its GPU for computations, retrieves results and send them back to the master node. Tasks are of two different kinds: diagonal workloads, which consist in computing shortest distances over a small subgraph, and non-diagonal workloads, which consist in multiplying three matrices.

Computations of diagonal workloads are implemented on the GPU using a blocked-recursive Floyd-Warshall approach developped by [BGB10] and adapted for non-power of 2 matrices. Non-diagonal workloads require less synchronization and can be implemented using a fast matrix-multiplication approach derived from [Vol10] and adapted for $(min, +)$ operations.

In this configuration, each physical node on the cluster makes use of as many CPU cores as there are available GPUs. If more CPU cores are available than GPUs, computational power is still available. On slave nodes, remaining CPU cores are used for outputting final results to disk. On large clusters, communication between the master node and slave nodes can become a bottleneck, leaving slave nodes idle while waiting for the master node to be available. In order to increase the availability of

the master node, a single CPU thread is used to initiate communications with slave nodes while remaining CPU cores handle the rest of the communications, updating data structures with temporary results and outputting final results to disk.

4.4.4 Memory limitations

For very large input graphs, memory usage becomes an issue. As stated previously, an entire adjacency matrix for the graph cannot be allocated; the graph is instead kept in memory as a list of edges, a much more memory-efficient representation. Even with this efficient representation, temporary sub-matrices need to be kept in memory: diagonal sub-matrices and boundary matrices. When recursively computing Step 3, boundary matrices are output to files so as to only keep a single boundary matrix in memory.

Final results for diagonal sub-matrices are only obtained at the end of Step 3. As soon as final values for these diagonal sub-matrices are obtained, they are output to files; only relevant parts are kept in memory for Step 4; namely, parts of these sub-matrices containing shortest distances from and to boundary vertices. Shortest distances between non-boundary vertices are thus discarded from main memory at the end of Step 3. The current limiting factor in terms of memory usage is the initial boundary matrix. The first boundary matrix has to fit in the main CPU memory. Section sec. 4.5 discusses ways to overcome this limitation. It is however probable that prohibitive run-times or an amount of results too large to process may become the limiting factor before main memory usage does.

4.5 Results and perspectives

In this section, we compare our implementation to two parallel Dijkstra implementations. It is important to note that our implementation allows graphs with negative edges - but no negative cycles - unlike Dijkstra-based approaches.

In order to test our implementation, we generated random graphs with increasing numbers of vertices, ranging from 1024 to 1024k. These graphs, generated using the LEDA library [MNU99], were made planar to ensure good partitioning properties.

Computations were run on a cluster of more than 300 computer nodes; each node is equipped with two NVIDIA C2090 GPUs, a 16 core Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz and 32 GB of RAM.

Our implementation handles instances up to $512k$ vertices without using external memory. For the very last instance, the use of external memory was required to fit in the 32 GB of main memory. We later refer to our implementation without using external memory as “Part. APSP no EM” and our implementation using external memory as “Part. APSP EM”.

The GPU Dijkstra implementation from [OATLGE13] is, to the best of our knowledge, the only implementation that was reported to solve APSP for graphs with up

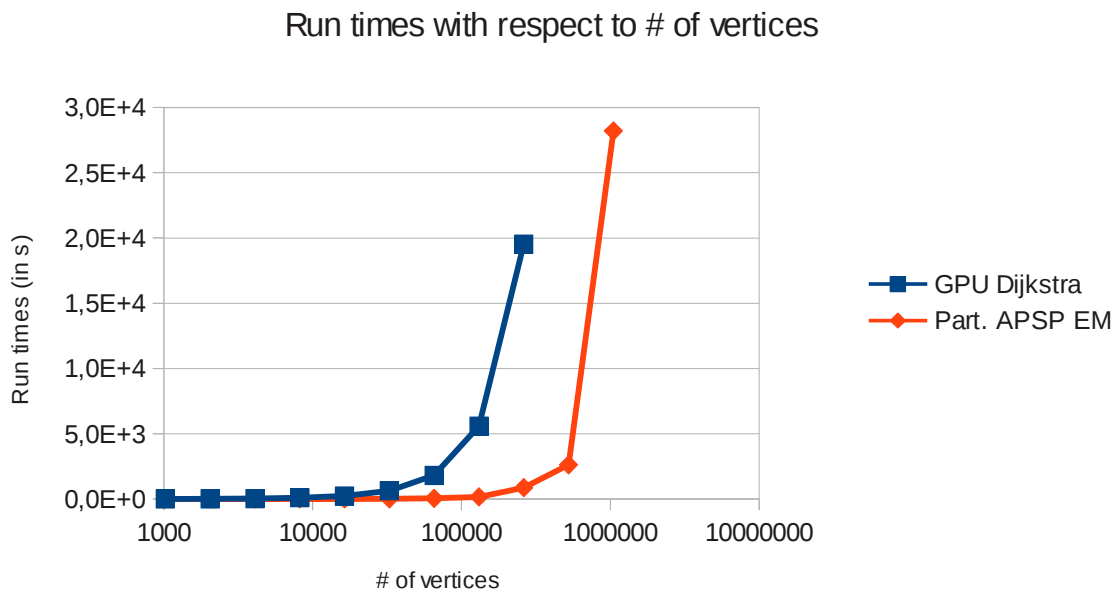

Figure 4.5.1: Evolution of run times with respect to the number of vertices. Two implementations are compared: our implementation using external memory and the GPU Dijkstra implementation from [OATLGE13]. Computations were run using two GPUs on a single cluster node.

to $1024k$ vertices; we later refer to this implementation as "GPU Dijkstra". This implementation parallelizes SSSP computations on a single computer using two GPUs and a multicore CPU. In order to compare this implementation to ours, we restricted computations of both implementations to using only two GPUs. Both implementations could therefore run on a single cluster node; no communication between nodes were therefore required. Figure Fig. 4.5.1 shows the runtimes for GPU Dijkstra and Part. APSP EM for graphs with numbers of vertices ranging from 1024 to $1024k$ using only two GPUs. GPU Dikstra could not compute the last two instances - $512k$ and $1024k$ vertices - within the 10 hour limit enforced on the cluster. We can see that our implementation is significantly faster than GPU Dijkstra.

Figure Fig. 4.5.2 shows the evolution of the speedup of our method without using external memory with respect to the number of GPUs used for the computations. Speedups are calculated using the run time obtained using only one GPU as a reference. Computations were done for the $512k$ vertex instance using the Part. APSP no I/O implementation. We can see that coarse-grain parallelism is close to optimal up to around 31 GPUs; almost no benefit can however be gained from using more than about 63 GPUs. The reason for this stagnation of the speedup above 63 GPUs is the saturation of communication with the master node.

The scalability can be improved using a coarse-grain parallelism approach that would relieve the master node of some of its communication. A work-stealing approach, for instance, would reduce the amount of communication required for the master node by decentralizing some of the memory transfers. A work-stealing ap-

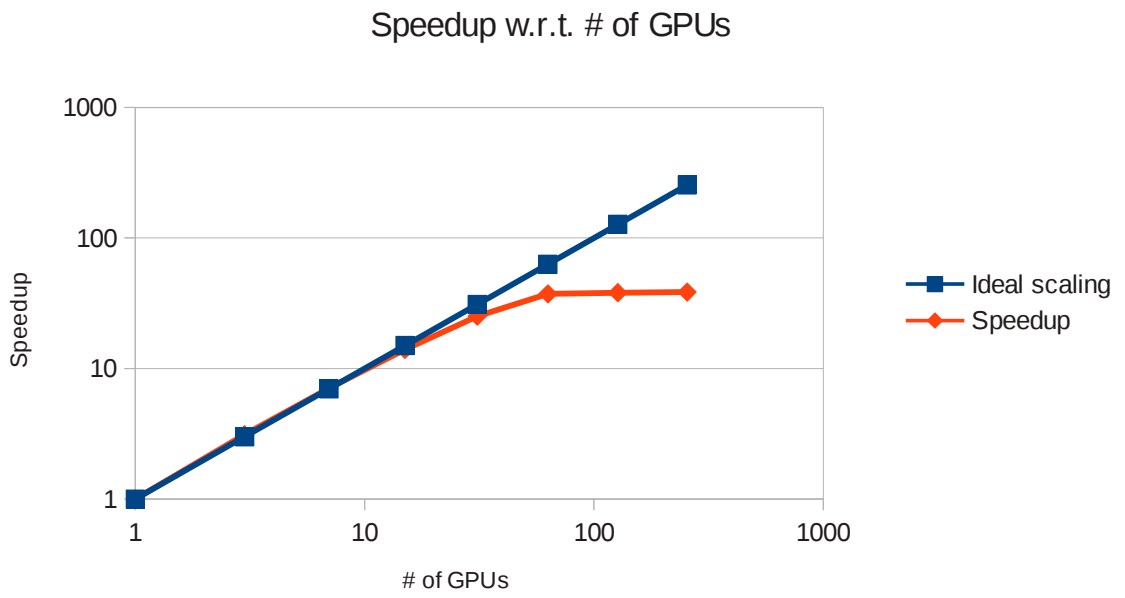

Figure 4.5.2: Evolution of speedups with respect to the number of GPUs. The ideal scaling line is given as a reference.

proach is however difficult to implement, due to the two-sided communication scheme enforced by the MPI standard. [PCMM07] showed that such an efficient approach was nevertheless feasible. This issue could also be addressed by creating a hierarchy of master nodes; some computations would be redundant between the different master nodes - handling the main data structure - but this would only represent a negligible fraction of the overall workload.

Figure Fig. 4.5.3 shows a comparison between our two implementations and a distributed Dijkstra approach - later referred to as CPU Dijkstra - for graphs ranging from 1024 to $1024k$ vertices. The distributed Dijkstra approach was implemented by dynamically distributing SSSP computations for each vertex of the graph over every core of every available cluster node. The Dijkstra-based implementation used is that of the Boost C++ library [DAR09]. This experiment is not intended to compare directly the performances of 2 GPUs versus a multicore CPU. Instead, we intend to show that our approach is competitive with a distributed Dijkstra approach given a fixed number of heterogeneous cluster nodes. The run times presented in Figure Fig. 4.5.3 were obtained using 64 cluster nodes. We can see that our version using external memory obtains very similar run times to that of the distributed Dijkstra version, while allowing graphs with negative edges to be computed. Our version without external memory is however significantly faster.

In order to test our implementation on a real dataset, we retrieved the Californian road network dataset from [LLDM09]. This dataset consists in the entire road network of the state of California; it contains $1,957,027$ vertices corresponding to road intersections and more than 5 million edges corresponding to roads. Computing the 10^12 shortest distances in this network took 31 minutes, using 64 cluster nodes.

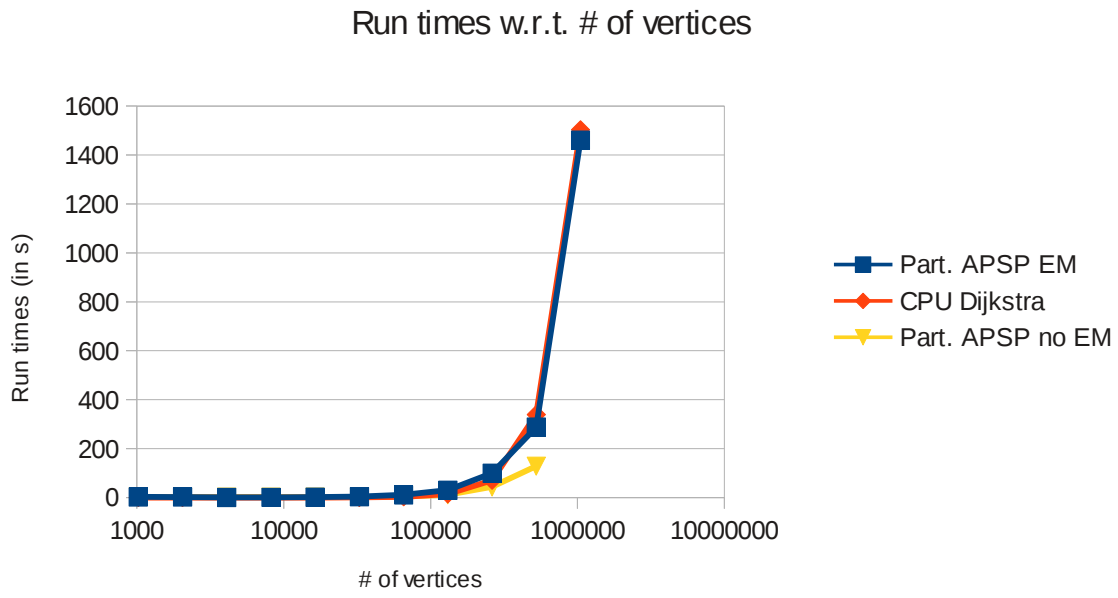

Figure 4.5.3: Evolution of run times with respect to the number of vertices. Three implementations are compared: our two implementations - with and without using external memory - and a distributed Dijkstra implementation referred to as CPU Dijkstra. All computations were run on 64 cluster nodes.

5 Parallel seed-based approach to protein structure similarity detection

5.1 Introduction

A protein's three dimensional structure tends to be better evolutionarily preserved than its sequence. Therefore, finding structural similarities between two proteins can give insights into whether these proteins share a common function or whether they are evolutionarily related. Structural similarities between two proteins are expressed by a one-to-one mapping (also called alignment) of their three dimensional representations. The quality of these alignments is crucial to correctly estimate protein functions and protein relations. Detecting the longest alignment, when comparing protein structures, is frequently modeled as finding the maximum clique [MDAY10, KJ10, SBS05], or enumerating all maximal cliques [GMB96, SKK+02]. Both problems are NP-hard. In these approaches, cliques are looked for in so-called product (or alignment) graphs, where each edge corresponds to matching of similar internal distances (up to a user-defined threshold τ). All edges in the target cliques satisfy this condition, but exactly this requirement leads to solving NP-hard problems.

Here, we relax this condition and accept cliques such that edges correspond to matching of similar internal distances up to 2τ. For this relaxed problem we propose a polynomial algorithm and its efficient parallel implementation comparing two protein structures that guarantees to return alignments with both $RMSDc$ and $RMSDd$ less than a given threshold value, if such alignments exist. This methodology also offers the possibility to return more than one alignment for a single pair of proteins to address cases where two proteins share more than a single similar region. Our approach takes advantage of internal distance similarities among both proteins to search for an optimal transformation to superimpose their structures. To the best of our knowledge, our tool is unique in the capacity to generate multiple alignments with "good" $RMSDc$ and $RMSDd$ values. Thanks to this property, the tool is able to detect structural repetitions within a single protein and between related proteins. We do not require vertices in the alignment graph to be ordered which makes our algorithm suitable for detecting similar domains when comparing multiple domained proteins. The computational burden is addressed by extensive use of parallel computing techniques.

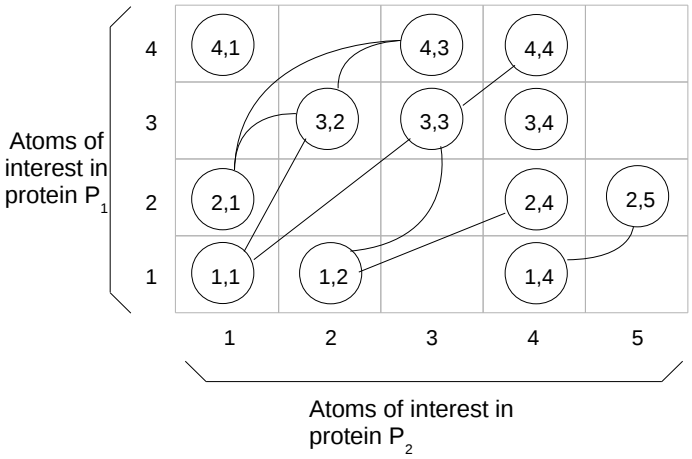

Figure 5.1.1: Example of an alignment graph used here to compare the structures of two proteins. The presence of an edge between vertex $(1, 1)$ and vertex $(3, 2)$ means that the distance between atoms 1 and 2 of protein 1 is similar to the distance between atoms 1 and 3 of protein 2. The clique $(2, 1)$ $(3, 2)$ $(4, 3)$ indicates that RMSD of structures $(2, 3, 4)$ and $(1, 2, 3)$ is less than 2τ.

5.1.1 Alignment graphs

Undirected graphs $G = (V, E)$ are represented by a set V of vertices and a set E of edges between these vertices. In this chapter, we focus on a subset consisting of grid-like graphs, referred to as alignment graphs.

An $m \times n$ *alignment graph* $G = (V, E)$ is a graph in which the vertex set V is depicted by an $m \times n$ array T , where each cell $T[i][k]$ contains at most one vertex (i, k) from V. An example of such an alignment graph for protein comparison is given in Fig. 5.1.1.

The matching of two proteins P_1 and P_2 can be solved by analyzing an alignment graph $G = (V, E)$, where $V = \{(v_1, v_2)|v_1 \in V_1, v_2 \in V_2\}$ and V_1 (resp. V_2) is the set of atoms of interest in protein P_1 (resp. protein P_2). A vertex (i, k) is present in V only if atoms $i \in V_1$ and $k \in V_2$ are compatible. An example of incompatibility could be different electrostatic properties of the two atoms. An edge $((i, k), (j, l))$ is in E if and only if the distance between atoms i and j in protein P_1, $d(i, j)$, is similar to the distance between atoms k and l in protein P_2, $d(k, l)$. In our case, these distances are considered similar if $|d(i, j) - d(k, l)| < \tau$, where τ is a given threshold.

Vertices in an alignment graph are arbitrarily ordered and given a corresponding index. In subsequent sections, the arbitrary notion of successors of a vertex v refers to all the vertices that are adjacent to v in the alignment graph and have a higher index than v. The notion of neighbors of a vertex v refers to all the vertices that are adjacent to v in the alignment graph regardless of their respective indices. Let $G(V, E)$ be the input alignment graph, where V is the set of vertices and E the set of edges. We define :

$$successors(v_i \in V) = \{v_j \in V | (v_i, v_j) \in E \,\&\, i < j\}$$
$$neighbors(v_i \in V) = \{v_i\} \cup \{v_j \in V | (v_i, v_j) \in E\}$$

5.1.2 Relation to protein structure comparison

In an alignment graph between two proteins P_1 and P_2, a subgraph with high density of edges denotes similar regions in both proteins. Finding similarities between two proteins can therefore be performed by searching the corresponding alignment graph for subgraphs with high edge density. The highest possible edge density is found in a clique, a subset of vertices that are all connected to each other.

DAST [MDAY10], for Distance-based Alignment Search Tool, aims at finding the maximum clique in an alignment graph. DAST uses alignment graphs where rows (resp. columns) represent an ordered set of atoms V_1 (resp. V_2) from protein P_1 (resp. protein P_2). A vertex (i, j) is present in the graph if and only if residues i and j belong to similar secondary structures in both proteins. An edge is present between vertex (i, j) and vertex (k, l) if and only if $|d(i, j) - d(k, l)| < \tau$, where τ is a given threshold. By construction, alignments returned by DAST are guaranteed to have associated RMSDd strictly less than τ.

5.1.3 Measures for protein alignments

Many measures have been proposed to assess the quality of a protein alignment. These measures include additive scores based on the distance between aligned residues such as the TM-score [ZS04], the DALI score [WAK13], the PAUL score [WPDK09] and the STRUCTAL score [SLL93] and Root Mean Square Deviation (RMSD) based scores, such as RMSD100, SAS and GSAS [KKL05]. Given a set of n deviations $S = s_1, s_2, ..., s_n$, its Root Mean Square Deviation is: $RMSD(S) = \sqrt{\frac{1}{n} * \sum_{i=1}^{n} s_i^2}$. Two different RMSD measures are used for protein structure comparison: $RMSDc$, which takes into account deviations consisting of the euclidean distances between matched residues after optimal superposition of the two structures; $RMSDd$, which takes into account deviations consisting of absolute differences of internal distances within the matched structures. The measured deviations are $|d(i, j) - d(k, l)|$, for all couples of matching pairs "$i \leftrightarrow k, j \leftrightarrow l$". Let P be the latter set and N_m, its cardinality. We have that $RMSDd = \sqrt{\frac{1}{N_m} * \sum_{(ij,kl) \in P} (|d(i, j) - d(k, l)|^2)}$.

5.2 Methods

5.2.1 Our approach

Looking for the maximal clique in a graph is a NP-complete problem [Kar72]. Being an exact solver, DAST faces prohibitively long run times for some instances. We propose a polynomial approach to protein structure comparison that guarantees to return alignments with the following properties $RMSDd < 2\tau$ and $RMSDc < \tau$, if such exist. Our approach offers the possibility to return an arbitrary number of distinct alignments. Returning multiple similar regions can prove useful, for instance, when looking for a structural pattern that may be present more than once in a protein or when comparing highly flexible proteins. However, enumerating multiple similar regions requires a more systematic approach than those developed in other existing heuristic-based tools. The computational burden associated with such a systematic approach can nevertheless be addressed by making use of multiple levels of parallelism.

Our method is inspired by the maximal clique search implemented in DAST. Instead of testing for the presence of all edges among a subset of vertices as done in DAST, we only test for the presence of edges between every vertex of the subset and an initial 3-clique, referred to as seed. The correctness of the resulting algorithm follows from geometric arguments, namely that the position of any 3-dimensional solid object is determined by the positions of three of its points that are not collinear.

5.2.2 Overview of the algorithm

Algorithm 5.1 gives an overview of our approach. The algorithm consists of the following three steps:

- Seeds in the alignment graph are enumerated. In our case, a seed is a set of three points in the alignment graph that correspond to two triangles (one in each protein) with similar internal distances. This step is detailed in sec. 5.2.3.
- Each seed is then extended. Extending a seed consists in adding all pairs of atoms, for which distances to the seed are similar in both proteins, to the set of three pairs of atoms that make up the seed. Seed extension is detailed in sec. 5.2.4.
- Each seed extension is filtered - cf. lines 5 through 11 of Algorithm 5.1. Extension filtering is detailed in sec. 5.2.5 and consists in removing pairs of atoms that do not match correctly.

Filtered extensions are then ranked according to their size - number of aligned pairs of atoms - and a user-defined number of best matches are returned. This process is explained in sec. 5.2.7. For very large alignment graphs, the graph can be partitioned into a user-defined number of parts to speed up computations. The graph is partitioned using a min-cut alike heuristic to preserve the quality of the

Algorithm 5.1 Overview of the algorithm

```
1 function find_alignments(graph)
  INPUT: graph, an alignment graph between atoms from two proteins
3 OUTPUT: resList, a list of the largest distinct alignments found

5 ResultList resList = empty_result_list()
  SeedList seeds = enumerate_seeds(graph)
7 For each seed in seeds
      VertexSet set = extend_seed(seed)
9     VertexSet result = empty_set()
      For each vertex in set
11        If(is_valid(vertex))
              result.add(vertex)
13        End If
          resList.insert_if_better(result)
15    End For
  End For
```

results. Each subgraph is then processed independently. This process is explained in Algorithm 5.5. The overall worst-case complexity of this algorithm without partitioning is $O(|V| * |E|^{3/2})$.

5.2.3 Seed enumeration

A seed consists of three pairs of atoms that form similar triangles in both proteins. A triangle IJK in protein P_1 is considered similar to a triangle $I'J'K'$ in protein P_2 if the following conditions are met: $|d(I,J) - d(I',J')| < \tau$, $|d(I,K) - d(I',K')| < \tau$ and $|d(J,K) - d(J',K')| < \tau$. Here, d denotes the euclidean distance and τ is a user-defined threshold parameter. The default value for τ is 2.0 Ångströms.

In the alignment graph terminology, these conditions for a seed $(v_i = (I, I'), v_j = (J, J'), v_k = (K, K'))$ in graph $G(V, E)$ translate to the following: $(v_i, v_j) \in E$, $(v_i, v_k) \in E$ and $(v_j, v_k) \in E$.

A seed thus corresponds to a 3-clique in the alignment graph; i.e., three vertices that are connected to each other. Enumerating all the seeds is therefore equivalent to enumerating every 3-clique in the input alignment graph.

Not all 3-cliques, however, are relevant. Suitable 3-cliques are composed of triangles for which a unique transformation can be found to optimally superimpose them. Namely, 3-cliques composed of triangles that appear to be too "flat" will not yield a useful transformation. We thus ensure that the triangles in both proteins defined by a potential seed are not composed of aligned points (or points which are close to being aligned). The method is detailed in Algorithm 5.3. The worst-case complexity of this step is $O(|E|^{3/2})$ using, e.g., the algorithms from [SW05].

Algorithm 5.2 Seed enumeration

```
function enumerate_seeds(graph)
INPUT: graph, an alignment graph between atoms from two
    proteins
OUTPUT: seedList, a list of suitable 3-cliques (i.e. triplets
    of vertices that are connected to each other and correspond
     to non-degenerated triangles in both proteins)

SeedList seedList = empty_seed_list()
For each vertex1 in graph
    For each vertex2 in get_successors(vertex1)
        For each vertex3 in get_successors(vertex2)
            If is_edge(vertex1, vertex3)
                If collinearity_check(vertex1, vertex2, vertex3)
                    seedList.add(vertex1, vertex2, vertex3)
                End If
            End If
        End For
    End For
End For
```

5.2.4 Seed extension

Extending a seed consists in finding the set of vertices that correspond to pairs of atoms that potentially match well (see sec. 5.2.5 for details) when the two triangles defined by the seed are optimally superimposed. Finding a superset of pairs of atoms that match well is performed by triangulation with the three pairs of atoms composing the seed. Let $(v_i = (I, I'), v_j = (J, J'), v_k = (K, K'))$ be a seed of an alignment graph $G(V, E)$ as defined in sec. 5.2.3.

$$\begin{aligned} extension(v_i, v_j, v_k) = \{&v_l = (L, L') | \\ &|d(L, I) - d(L', I')| < \tau \wedge \\ &|d(L, J) - d(L', J')| < \tau \wedge \\ &|d(L, K) - d(L', K')| < \tau\} \end{aligned}$$

Where I (resp. I') is the atom of the first (resp. second) protein associated to vertex v_i. In the alignment graph terminology, the previous definition translates to:

$$extension(v_i, v_j, v_k) = neighbors(v_i) \cap neighbors(v_j) \cap neighbors(v_k)$$

The detail of seed extension is given in Algorithm 5.3. The computational complexity associated to this step is $O(|V|)$.

Algorithm 5.3 Seed extension

```
function extend_seed(vertexA, vertexB, vertexC)
INPUT: a seed represented by three vertices (or pairs of atoms) from
    the alignment graph
OUTPUT: res, a set of pairs of atoms that potentially match well
    when atoms from the seed are optimally superimposed;
      size, the size of the returned set

BinaryVertexSet setA = get_neighbors(vertexA)
BinaryVertexSet setB = get_neighbors(vertexB)
BinaryVertexSet setC = get_neighbors(vertexC)
BinaryVertexSet tmp = intersection(setA, setB)
BinaryVertexSet res = intersection(tmp, vertexC)
int size = pop_count(res)
```

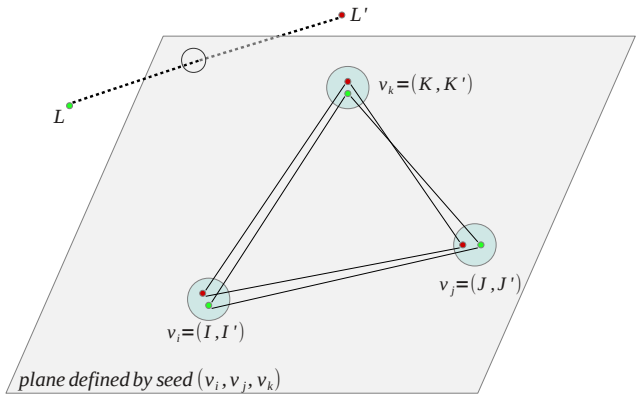

Figure 5.2.1: Example of symmetry issues. Even though, vertex $v_l = (L, L')$ belongs to the extension of $seed(v_i, v_j, v_k)$, points L and L' lie on different sides of the plane defined by optimally superimposed triangles IJK and $I'J'K'$.

5.2.5 Extension filtering

The triangulation performed when extending a seed is not sufficient to find alignments with good $RMSD$ measures. Indeed, in most cases, knowing the distance of a point in a three dimensional space to three other non-aligned points yields two possible locations. These locations are symmetrical with respect to the plane defined by the three reference points. A vertex in a seed extension represents a pair of atoms, one in each studied proteins. By construction, these atoms have similar distances to the three points of their respective triangles. It may happen that one of the two points, say L, is located, in protein P_1, on one side of the plane defined by its reference triangle, while the second point, says L', in protein P_2, lies on the other side of the plane defined by the two optimally superimposed reference triangles - see Fig. 5.2.1.

Algorithm 5.4 Extension filtering algorithm

```
function filter_extension(extension)
INPUT: extension, a set of pairs of atoms
OUTPUT: result, a subset of the extension containing only pairs of
    atoms that match well

VertexSet result = empty_set()
Matrix transformation = get_optimal_transformation(seed)
For each vertex in extension
    Point L = get_coordinates_in_first_protein(vertex)
    Point L_prime = get_coordinates_in_second_protein(vertex)
    Point L_transformed = apply_transformation(L, transformation)
    Float distance = dist(L_transformed,L_prime)
    If(distance < threshold)
        result.insert(vertex)
    End If
End For
```

Using quadruplets of vertices as seeds does improve the quality of seed extensions but greatly increases the computational cost of seed enumeration and degeneration check on the corresponding tetrahedra. Moreover, larger seeds do not completely ensure the quality of extensions. Namely, in cases where, for a vertex $v_l = (L, L')$, atom L (resp. L') is very distant from atoms I, J and K (resp. atoms I', J' and K') of a seed $(v_i = (I, I'), v_j = (J, J'), v_k = (K, K'))$, distance similarities to the atoms of the seed do not ensure similar positions of atoms L and L' in the two proteins.

In order to remove issues with symmetry (where the atoms in the extending pair are roughly symmetrical with respect to the plane determined by the seed atoms) and distance from the seed, we implemented a method to filter seed extensions. This method consists in computing the optimal transformation T to superimpose the triangle from the seed corresponding to the first protein onto the triangle corresponding to the second. The optimal transformation is obtained in constant time with respect to the size of the alignment by using the fast, quaternion-based method of [LAT10]. For each pair of atoms (L,L') composing the extension of a seed $(v_i = (I, I'), v_j = (J, J'), v_k = (K, K'))$, we compute the euclidean distance between $T(L)$ and L'. If the distance is greater than a given threshold τ, the pair is removed from the extension. The filtering method is detailed in Algorithm 5.4. The complexity of this step is $O(|V|)$ per seed.

5.2.6 Guarantees on resulting alignments' $RMSD$ scores

By construction, the filtering method ensures that the RMSD for a resulting alignment is less than τ: the distance between two aligned residues after superimposition of the two structures is guaranteed to be less than τ.

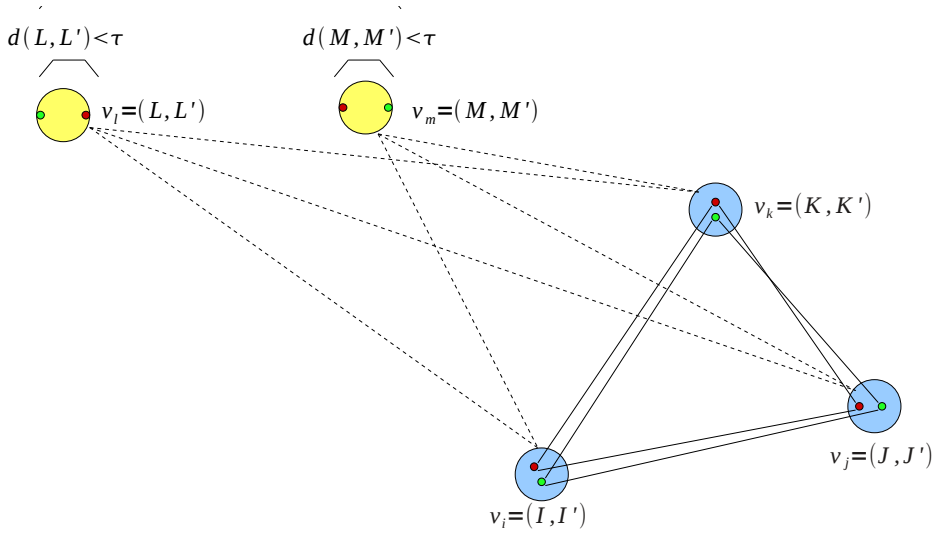

Figure 5.2.2: Illustration of the guarantee on the similarity of internal distances between two pairs of atoms $v_l = (L, L')$ and $v_m = (M, M')$, here represented in yellow, added to a seed (v_i, v_j, v_k) represented in blue. Dashed lines represent internal distances, the similarity of which is tested in the alignment graph.

Internal distances between any additional pair of atoms and the seed is also guaranteed, by construction to be less than τ. Concerning internal distances between two additional pairs of atoms, we ensure that in the worst possible case, the difference is $2 * \tau$, see Fig. 5.2.2. The worst possible case happens when two additional pairs of atoms $v_l = (L, L')$ and $v_m = (M, M')$, added to the extension of a seed (v_i, v_j, v_k), have atoms L, L', M and M' aligned, after superimposition, and atoms from one protein lie within the segment defined by the two other atoms. In such a case, the filtering step ensures that $d(L, L') < \tau$ and $d(M, M') < \tau$; it follows that $|d(L, M) - d(L', M')| < 2 * \tau$.

5.2.7 Result ranking

When comparing two proteins, we face a double objective: finding alignments that are both long and have low $RMSD$ scores. The methodology described in sec. 5.2.5 ensures that any returned alignment will have a $RMSDd$ lower or equal to twice a user-defined parameter τ. We can therefore leave the responsibility to the user to define a threshold for $RMSD$ scores of interest. However, ranking alignments that conform to this $RMSD$ threshold simply based on their lengths is not an acceptable solution. In a given alignment graph, several seeds may lead to very similar transformations and thus very similar alignments. The purpose of returning multiple alignments for a single comparison is to find distinct similar regions in both proteins. Therefore, when two alignments are considered similar, we discard the shorter of the two.

Two alignments are considered similar, when they share a defined number of pairs

of atoms. This number can be adjusted depending on the expected length of the alignments or even set to a percentage of the smaller of the two compared alignments. This methodology of ranking results ensures that no two returned alignments match the same region in the first protein to the same region in the second protein.

5.2.8 *k*-to-*k* alignments

With this approach, results may not be alignments in the traditional sense, *i.e.* a 1-to-1 mapping of amino-acids. It may happen that two residues from one protein are matched with the same residue from the second protein. Such alignments, referred to as k-to-k alignments, may be interesting when studying the possible docking of two proteins; in such a case, a k-to-k alignment may be used to identify two regions with a large number of contacts between two protein surfaces. Our efforts in this direction were however unfruitful.

When 1-to-1 alignments are necessary, we propose two different approaches to retrieve a 1-to-1 alignment from a k-to-k alignment. The first approach is independent of the sequences of the two proteins, while the second approach offers sequence dependent alignments for cases where following the original protein sequences is required.

In order to retrieve a traditional alignment from a k-to-k alignment, we consider the subgraph composed of all the vertices in the k-to-k alignment. In this subgraph, which we assume to be complete, we remove edges between two vertices (M, M') and (N, N') if $M = N$ or $M' = N'$. This subgraph is also an alignment graph with columns that represent the atoms of the first protein that are present in the k-to-k alignment and rows that represent the atoms of the second protein that are present in the k-to-k alignment.

Finding the largest 1-to-1 alignment in a k-to-k alignment thus consists in finding the largest set with at most one vertex per row and column in the subgraph previously described. This largest set is found by solving an assignment problem. Multiple 1-to-1 alignments of optimal length may however be present in the k-to-k alignment graph. In order to retrieve the 1-to-1 alignment of optimal length and optimal RMSDc, each tile of the k-to-k alignment graph is filled with the distance between the two paired atoms after superimposition of the seed; solving the assignment problem thus yields the longest 1-to-1 alignment that minimizes the sum of the distances, hence minimizing the RMSDc - see Fig. 5.2.3.

Resulting 1-to1 alignments are independent of the original sequences of the two proteins. If sequence dependence is required, a 1-to-1 alignment can also be retrieved from the k-to-k alignment by looking for the longest increasing path in the previously described subgraph. Resulting 1-to-1 alignments therefore follow the amino-acid sequences of both proteins and do not take into account possible sequence inversions between the two proteins.

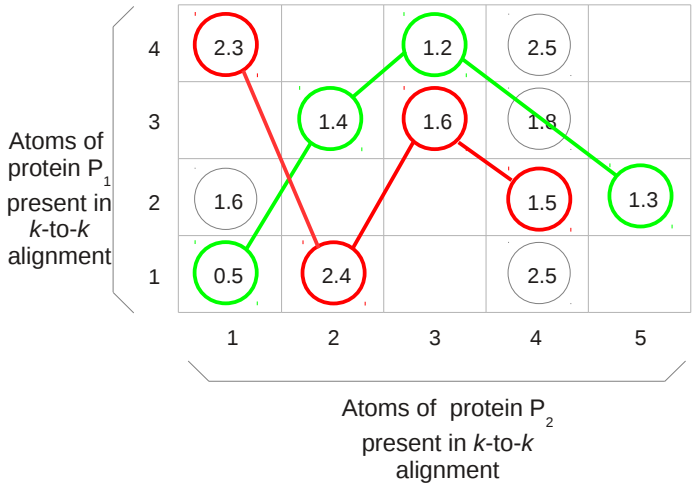

Figure 5.2.3: Example of 1-to-1 alignments retrieved from a k-to-k alignments. In red, a 1-to-1 alignment of optimal length but sub-optimal RMSDc and in green a 1-to-1 alignment of optimal length and optimal RMSDc. Solving the assignment problem on this graph yields the green alignment.

5.2.9 Graph splitting

Large protein alignment graphs can contain millions of edges. In order to reduce the computations induced by such large graphs, a graph splitting scheme is implemented.

Graph splitting is performed using a min-cut like heuristic, also known as multi-level graph partitioning, provided by the METIS library [KK98a]. This heuristic partitions the graph in k components of similar number of vertices and aims at minimizing the number of inter-component edges - edges between vertices that belong to distinct components. In order to further minimize the number of inter-component edges, we allow the sizes in terms of numbers of vertices of the components to vary up to an order of magnitude. The assumption is that such partitions will keep each area of interest in the graph within a single component.

Once a partition is obtained, subgraphs corresponding to the k components are sorted according to their respective numbers of vertices. Each subgraph is then solved starting with the largest subgraph. The list of best results is transmitted from one subgraph to another, in order to be able to discard seeds whose extensions are smaller than the best results found so far.

In practice, partitioning the graph tends to group vertices of each of the best results within a single component. However, several of these vertices may be placed in different components. To address this issue, seeds yielding to the best results in a subgraph are extended and filtered once more using atoms from the initial global graph.

This second extension and filtering phase significantly improves the length of resulting alignments but does not guarantee to provide the same results as without partitioning. However, experimental results show that a given graph could be par-

Algorithm 5.5 Graph splitting algorithm

```
1 function split_and_solve(globalGraph)
  INPUT: globalGraph, an alignment graph between atoms from two
      proteins
3 OUTPUT: globalRes, a list of the longest distinct alignments found
      in the graph

5 ResultList globalRes = empty_result_list()
  Graph[] subGraphs = split(globalGraph)
7 sort(subgraphs)
  For each subGraph in subGraphs
9     SeedList best_seeds = empty_list()
      SeedList seeds = enumerate_seeds(subGraph)
11    For each seed in seeds
          VertexSet current_res = extend_and_filter(subGraph, seed)
13        best_seeds.insert_if_better(seed)
      End For
15    For each seed in best_seeds
          VertexSet current_res = extend_and_filter(globalGraph, seed)
17        globalRes.insert_if_better(current_res)
      End For
19 End For
```

titioned in up to 10 components with only a 2% loss in terms of alignment length and a four fold overall speedup.

The graph splitting scheme is described in Algorithm 5.5.

5.3 Parallelism

5.3.1 Overview of the implemented parallelism

The overall complexity of our algorithm being $O(|V| * |E|^{3/2})$, handling large protein comparison with a decent level of precision - i.e., using alignment graphs with a large number of edges - can prove time-consuming. Our approach is however parallelizable at multiple levels.

Fig. 5.3.1 shows an overview of our parallel implementation. Multiple seeds are treated simultaneously to form a coarse-grain level of parallelism, while a finer grain parallelism is used when extending a single seed.

5.3.2 Coarse-grain parallelism

Computations for enumerating seeds - see sec. 5.2.3, extending seeds - see sec. 5.2.4, and filtering the resulting extensions - see sec. 5.2.5, are independent processes, which can be performed in parallel. A user-defined number of threads can be spawned to

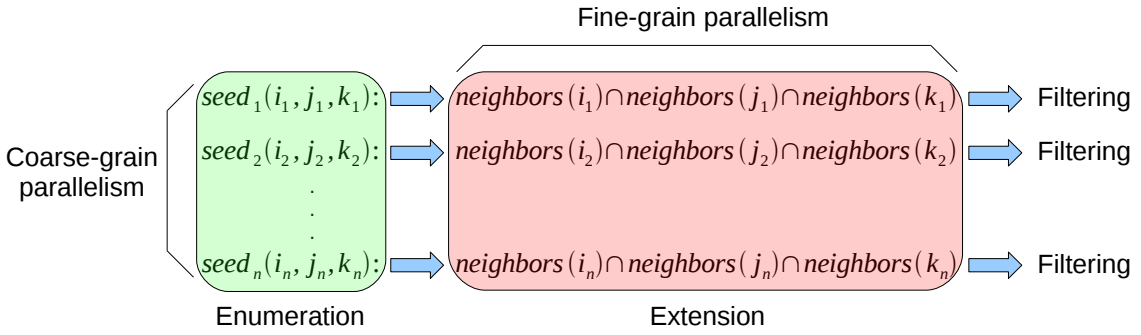

Figure 5.3.1: Overview of the implemented parallelism.

handle, in parallel, computations for the various seeds present in the graph. This parallelism is implemented using the openMP standard [DM98].

Threads, however need to share their results to populate a global list of results. Inserting new entries in this global-result list would prove rather inefficient, because thread safety would need to be ensured by using locks around accesses to this result list. With such locks, threads would often stall whenever inserting a new alignment and the time lost on these accesses would only increase with the number of threads in use. In order to avoid any bottleneck when inserting a new alignment in the result list, each thread has its own private list. These lists are merged at the end of the computations to form a global result list. This method prevents the need for a synchronization mechanism and allows threads to be completely independent.

However, using this method can, in some cases, increase the total amount of computations. Whenever a seed extension is smaller than the smallest alignment present in the result list, it is discarded, thus avoiding the cost of a filtering step. Since each thread has its own result list, the minimal size required for the thread to consider filtering an extension is only a lower bound of the global minimal size found so far by all threads. Sharing only this global minimal size among threads is not a suitable solution, because no guarantee could be made on the distinctness of two alignments from different threads. Therefore, smaller similar regions would be wrongly discarded.

Even if threads were to share a global minimal size, parallelism at this level could still induce more computations. The order in which seeds are treated can, in some cases, be important. When n results are required, if all n seeds yielding the n best results are treated first, more seed extensions will be discarded and the total amount of computations will be reduced. In this regard, our approach is similar to a branch and bound algorithm. Parallelizing our approach at this level therefore induces the same challenges that parallel branch and bound implementations face[LS84].

Though not implemented, an even higher level of parallelism could be considered when graph splitting is performed. Computations for each subgraph are also independent and could therefore be run in parallel. Since a multicore parallelism implementation is already provided, a cluster level parallelism could be implemented. Each subgraph would be sent to a single cluster node using for example using an

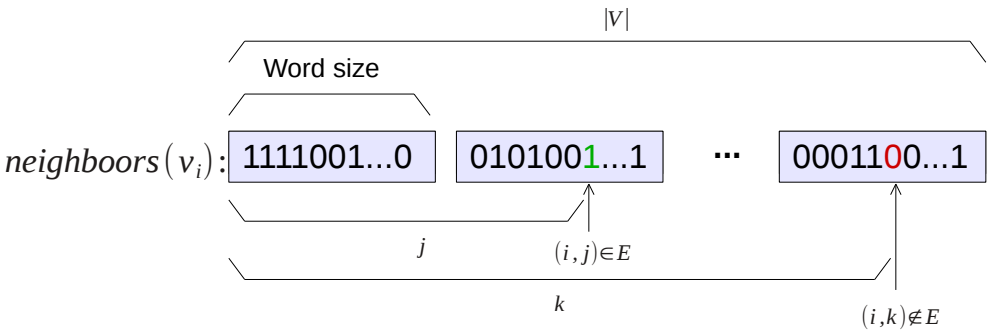

Figure 5.3.2: Bit vector representation of the neighbors of vertex v_i in an alignment graph $G(V, E)$. In this example, v_j unlike v_k is a neighbor of v_i.

MPI approach(for Message Passing Interface [GLS99]). However, load balancing would be a challenging task due to the limited number of subgraphs that can be generated without a prohibitive loss of accuracy and the difference in terms of numbers of vertices of these subgraphs. Moreover, the total amount of computations would increase if subgraphs were treated in parallel, since the optimal lower bound found in one subgraph could not be used to solve other subgraphs. This issue would also be similar to that observed in parallel branch and bound algorithms and first described in [LS84].

5.3.3 Fine-grain parallelism

Seed extension makes extensive use of set intersection operations. In order to speed up these particular operations, we implemented a bit vector representation of the neighbors set of each vertex of the alignment graph. These bit vectors represent the neighbors in the alignment graph of each vertex (cf. Fig. 5.3.2). For a vertex v_i, a bit is set at position j if and only if vertices v_i and v_j are connected in the alignment graph.

This bit vector representation of the neighbors sets allows bit parallel computations of set intersection. A simple logic *and* operation over every word element of the two sets yields the intersection.For faster traversal of the neighbors set a traditional list representation is also kept. This list representation allows easy access to the first and last elements of the neighbors set. Knowing the first and last elements of the sets allows us to restrict the area of interest for intersection operations (see Fig. 5.3.3).

Intersection operations also benefit from SSE[1] instructions. A number of atomic operations equal to the size of the SSE registers available on the machine (typically 128 or 256) can be computed simultaneously. However, this sparse approach to computing set intersections increases the number of atomic operations to perform. Namely, vertices, which are not neighbors of any of the two vertices for which the

[1]Streaming SIMD Extensions

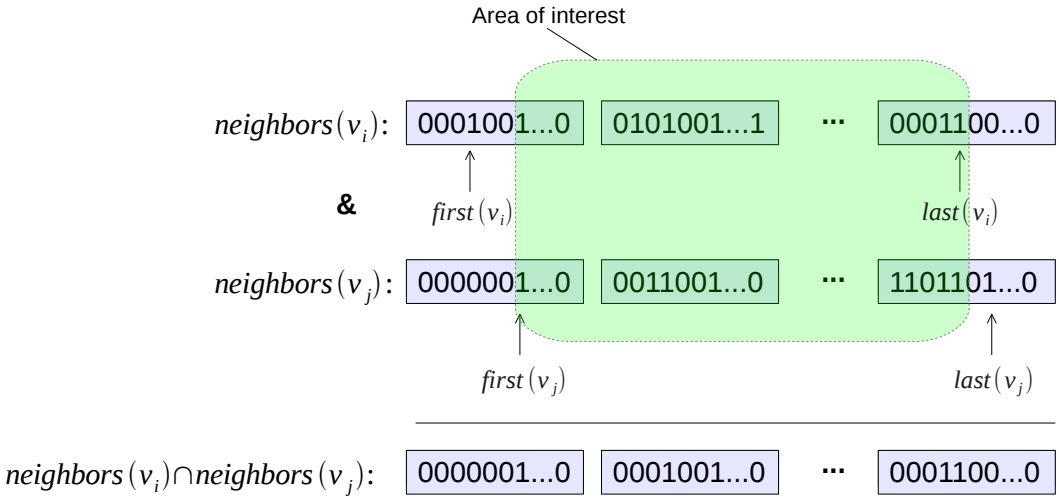

Figure 5.3.3: Intersection of neighbors of vertex v_i and vertex v_j.

intersection is computed, will induce atomic operations; provided such vertices reside in the area of interest. Such vertices would not be considered in a traditional approach to set intersection. This sparse approach is still faster in our case because alignment graphs tend to be dense enough. The size of the resulting intersections is required for the rest of our algorithm. Knowing the size of an intersection allows us to discard seeds, when larger results have already been found. Computing the size of a sparse set is not as trivial as it is with a dense set. In order to compute the size of a sparse set, we use a built-in population count instruction (POPCNT) available in SSE4. This operation returns, in constant time, the number of bits set in a single machine word. For architectures without a built-in population count instruction, a slower alternative is provided.

5.4 Results and perspectives

In order to test the capacity of our approach to detect multiple regions of interest, we considered two proteins (PDB IDs 4clna and 2bbma). These proteins are each composed of two similar domains - named A and B (resp. C and D) for the first protein (resp. second protein), separated by a flexible bridge (see Fig. 5.4.1).

Existing approaches, such as PAUL [WPDK09] and ones based on contact map overlap (CMO) [AMDY11], tend to match both proteins integrally, yielding larger alignments but poorer RMSD scores. TM_align [ZS05], the reference tool for protein comparison, only matches domain A onto domain C. The four top results of our tool correspond to all four possible combinations of domain matching, cf. Fig. 5.4.2. Our tool was run using 12 cores of an Intel(R) Xeon(R) CPU E5645 @ 2.40GHz and the distance threshold was set to 7 Ångströms and to 2 Ångströms in the alignment graph. Scores corresponding to these alignments are displayed in Table Tab. 5.1.

In this chapter, we introduce a novel approach to find similarities between protein

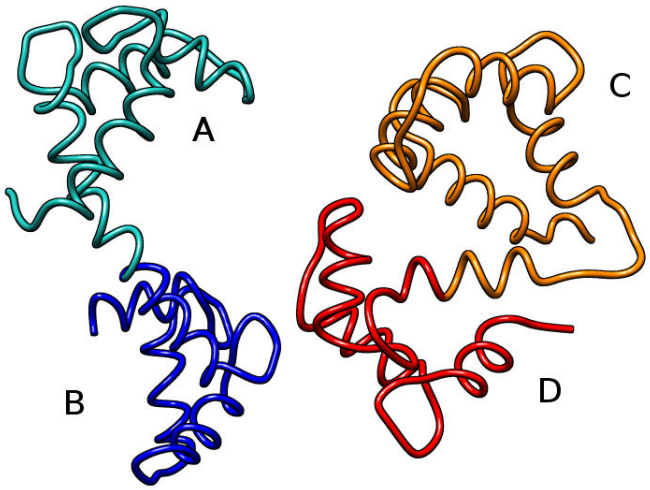

Figure 5.4.1: These two proteins are both composed of two similar domains - named A and B for 4clna (left), and C and D for 2bbma (right). These domains are separated by a a flexible bridge.

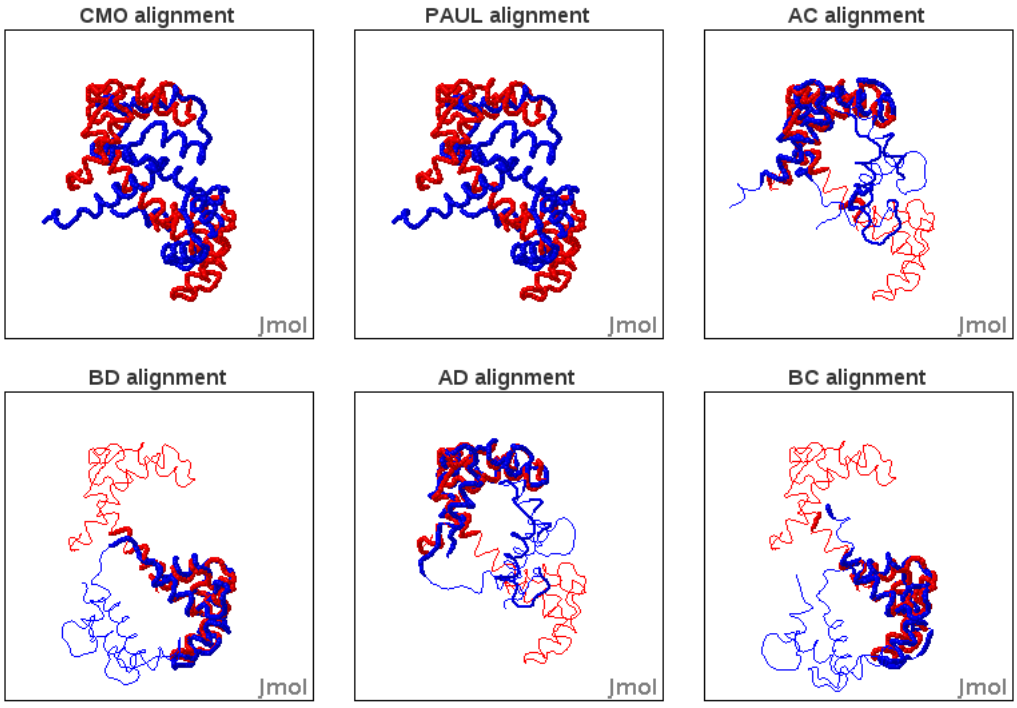

Figure 5.4.2: Visualizations of the results for the comparison of proteins 4clna and 2bbma returned by CMO, PAUL and the four top alignments of our approach.

	CMO	**PAUL**	**TMAlign**	**AC**	**BD**	**AD**	**BC**
# of aligned residues	148	148	79	72	70	66	64
% of aligned residues	100	100	53.4	48.7	47.3	44.6	43.2
RMSDc	14.781	14.781	2.935	2.048	1.731	*1.592*	2.210
RMSDd	10.838	10.838	2.627	1.797	1.475	*1.414*	1.770
TM_score	0.161	0.161	*0.422*	0.411	*0.422*	0.405	0.358

Table 5.1: Details of the alignments returned by other tools - columns 2 through 4 - and our method - columns 5 through 8. Best scores are in italics.

# of cores	1	2	3	4	6	8	12	16	20	24
Run time (s)	6479	3696	2494	1932	1374	1072	781	723	676	643
Speedup	1	1.8	2.6	3.4	4.7	6.0	8.3	9.0	9.6	10.1

Table 5.2: Run times and speedups for varying # of cores.

structures. Resulting alignments are guaranteed to score well for both $RMSD_d$ and $RMSD_c$, while remaining polynomial. This approach takes advantage of internal distance similarities, described in an alignment graph, to narrow down the search for an optimal transformation to superimpose two substructures of the proteins.

In order to test our coarse-grain parallel implementation, we compare run times obtained with various numbers of threads on a single artificially large instance. Any instance can be made artificially large by allowing a large number of vertices and edges when creating the alignment graph. The input alignment graph for this instance contains 15024 vertices for 9565358 edges. Computations were run using a varying number of cores of an Intel(R) Xeon(R) CPU E5645 @ 2.40GHz. Tab. 5.2 shows run times and speedups with respect to the number of CPU cores. The gain in terms of speedup becomes less significant beyond 12 cores. Note that similar results - both in terms of length and $RMSD$ scores - can be obtained in less than 30 seconds with a sparser alignment graph.

Fig. 5.4.3 shows run times for graphs with a varying number of edges and the same number of vertices - 21904. Computations were run using 12 cores of an Intel(R) Xeon(R) CPU E5645 @ 2.40GHz. Input alignment graphs were all generated from the same two proteins and different parameters to allow a varying number of edges.

This approach could be used to find similarities between RNA structures. However, such structures can be much larger than proteins. Therefore, future work includes further optimizations to allow larger alignment graphs to be computed.

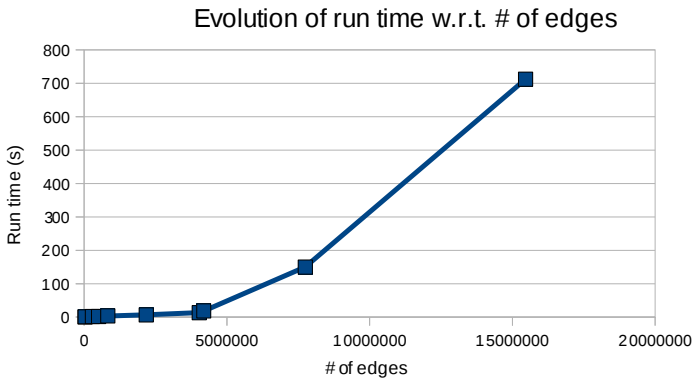

Figure 5.4.3: Evolution of run times with respect to # of edges in the alignment graph.

6 Conclusions and perspectives

In many domains, the increasing amount of data to process along with an architectural shift from increasing processor clock frequencies to increasing parallel capabilities in modern computers stress the need for parallel algorithms and efficient implementations. Our work focused on designing parallel algorithms for relevant bioinformatics problems and implementing them for suitable parallel architectures.

6.1 Conclusions

This thesis described contributions to three different domains: genetics, with a GPU implementation of a tool for QTL Mapping; large graph analysis, with a multi-GPU implementation for a new algorithm for the ALL-Pairs Shortest Path problem; and protein structure comparison, with a new algorithm for protein similarity detection and its multicore implementation.

We described the background of this present work in chapter 2. We focused on detailing the parallel capabilities offered by modern computers and the programming methods to exploit these parallel capabilities. We also described recent parallelization efforts conducted in two areas in bioinformatics: sequence and structure comparison. These two particular areas have seen many recent publications for parallel implementations using a wide range of parallelization techniques and are representative of the global trend in bioinformatics.

In chapter 3, we presented a new version of existing software QTLMap. QTLMap is a tool for QTL detection, a computationally heavy procedure. This new version takes advantage of GPUs to speed up computations. Computations using this new version are between 50 and 75 times faster than computations using the previous multicore implementation, while maintaining the same results and precision. Reduced runtimes allow geneticists to consider more precise and time consuming analyses by increasing the number of simulations or the number of studied genome positions. Reduced runtimes also allow geneticists to consider new analyses, such as multiQTL analyses.

In chapter 4, we described a new algorithm for solving the all-pairs shortest-path (APSP) problem for planar graphs and graphs with small separators that exploits the massive on-chip parallelism available in today's Graphics Processing Units (GPUs). Our algorithm, based on the Floyd-Warshall algorithm, has near optimal complexity in terms of the total number of operations, while its matrix-based structure is regular enough to allow for efficient parallel implementation on the GPUs. By applying a divide-and-conquer approach, we are able to make use of multi-node GPU clusters,

resulting in more than an order of magnitude speedup over fastest known Dijkstra-based GPU implementation and a two-fold speedup over a parallel Dijkstra-based CPU implementation. The applications of this new algorithm lie beyond the scope of bioinformatics.

In chapter 5, we presented a novel approach to protein structure comparison. A traditional approach to finding structural similarities between proteins is to search for the maximum clique in an alignment graph. However, searching for the maximum clique in a graph is an NP-complete problem and can therefore lead to unreasonable runtimes. Our approach relaxes some of the constraints imposed when looking for the maximum clique and returns regions of high density in the alignment graph. The relaxation of the constraints allows our approach to have a polynomial complexity, while maintaining guarantees on the quality of resulting alignments. The computational burden of our approach is addressed by extensive use of parallel computing techniques.

6.2 Perspectives

6.2.1 QTL detection

QTL detection is a time consuming procedure. The accuracy of an analysis is determined by the precision of the discretization of the studied chromosomal region and the number of simulations to test the results under the null hypothesis. With reduced runtimes, geneticists can now consider analyses with smaller discretization steps and more simulations.

Future QTL detection analyses will however require to consider the entire DNA sequence of the studied individuals in order to be more precise. Such analyses, also referred to as genome-wide association studies, require many more computations than traditional QTL detection, where entire DNA sequences of studied individuals are discretized. In order to cope with the computational burden of genome-wide studies, QTLMap could be adapted to take advantage of more than one GPU on a single computer or even adapted to run on large GPU clusters.

6.2.2 Large graph analysis

Our method allowed the resolution of the All-Pairs Shortest Path problem on graphs with up to several million vertices in reasonable times using up to hundreds of GPUs simultaneously. Increasing the size of target graphs will require efforts in two directions.

On the one hand, the representation of the data will need to be improved in order to reduce the memory footprint of the program. The current limiting factor is the size of the initial boundary matrix, which is currently represented in memory by a large distance matrix. Switching to a denser representation of this initial boundary

matrix will drastically reduce its size in memory and allow larger graphs to be computed.

On the other hand, efforts will need to improve the coarse-grain parallelism of our approach. Larger graph will drastically increase the demand for computations. This increasing demand for computations can be addressed by increasing the number of GPUs in order to guarantee reasonable runtimes. Our current implementation however does not benefit from using more than around 100 GPUs. This is due to the fact that the master node, which coordinates all computations, becomes overwhelmed with communications with slave nodes. In order to improve the scaling of our implementation, we could implement a work-stealing approach, which should relieve the master node of some of the communications or increase the number of master nodes.

6.2.3 Protein structure comparison

In the future, methods that have been successfully applied to finding local similarities between proteins will need to be adapted to look for similarities between RNA sequence. RNA sequences can be much larger than protein sequences. Though a heuristic, our approach to protein structure comparison is computationally intensive. In order to cope with the computational burden of finding similarities between large RNA strands, our approach will need to be improved.

Parts of our approach could benefit from using available GPUs for computations. Extending a set of seeds, as defined in our algorithm, is especially suited for computations on a GPU. Seed extension consists in computing the intersection of three binary sets. This process does not present any branching nor irregular data access patterns; implementing it on a GPU is rather straight-forward as all the required instructions are available on a GPU. A potential improvement of our implementation would be to use one CPU thread to enumerate a set of seeds. Once a suitable number of seeds have been identified, these seeds can be sent to a GPU for extension. Remaining CPU threads could then filter the seed extensions computed on the GPU.

6.2.4 General remarks

The recent data tsunami in bioinformatics yields many challenges in terms of storage and transfer of information, computing of analyses... This worked showed that parallelism can be used to alleviate the computational burden that represents this increasing mass of data. We also showed that parallelism allows more precise analyses to be run when the mass of data is not an issue. However, parallelizing an existing approach can be a tedious task. Therefore, developing new approaches with parallelism in mind from the beginning is crucial with the recent ubiquity of parallel architectures.

6.3 Acknowledgments

This work was supported by the region of Brittany, France.

Bibliography

[AD86] Lloyd Allison and Trevor I Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23(5):305–310, 1986.

[AGM+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[AMDY11] Rumen Andonov, Noël Malod-Dognin, and Nicola Yanev. Maximum contact map overlap revisited. *Journal of Computational Biology*, 18(1):27–41, 2011.

[BGB10] Aydın Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the GPU. *Parallel Computing*, 36(5):241–253, 2010.

[CD94] G.A. Churchill and R.W. Doerge. Empirical threshold values for quantitative trait mapping. *Genetics*, 138(3):963, 1994.

[CFE+13] Guillaume Chapuis, Olivier Filangi, Jean-Michel Elsen, Dominique Lavenier, and Pascale Le Roy. Graphics Processing Unit-accelerated Quantitative Trait Loci detection. *Journal of Computational Biology*, 20(9):672–686, 2013.

[CIPR01] Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DAR09] Beman Dawes, David Abrahams, and Rene Rivera. Boost C++ libraries. *URL http://www. boost. org*, 35:36, 2009.

[Deo10] Sebastian Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fundamenta Informaticae*, 99(4):409–433, 2010.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[Edg04] Robert C Edgar. MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC bioinformatics*, 5(1):113, 2004.

[EMG+99] Jean-Michel Elsen, Brigitte Mangin, Bruno Goffinet, Didier Boichard, and Pascale Le Roy. Alternative models for QTL detection in livestock. i. general introduction. *Genet. Sel. Evol.*, 31:1–12, 1999. 10.1186/1297-9686-31-3-213.

[Far07] Michael Farrar. Striped smith–waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

[FCA+00] F. Farnir, W. Coppieters, J. J. Arranz, P. Berzi, N. Cambisano, B. Grisart, L. Karim, F. Marcq, L. Moreau, M. Mni, C. Nezer, P. Simon, P. Vanmanshoven, D. Wagenaar, and M. Georges. Extensive genome-wide linkage disequilibrium in cattle. *Genome Res.*, 10:220–227, Feb 2000.

[FEDGL10] A. Favier, J.M. Elsen, S. De Givry, and A. Legarra. Optimal haplotype reconstruction in half-sib families. In *ICLP-10 workshop on Constraint Based Methods for Bioinformatics, Edinburgh, UK*, 2010.

[FMG+10] O. Filangi, C. Moreno, H. Gilbert, A. Legarra, P. Le Roy, and JM Elsen. QTLMap, a software for QTL detection in outbred populations. In *Proceedings of the 9th World Congress on Genetics Applied to Livestock Production: 1-6 August; Leipzig*, number 787, 2010.

[Fre87] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[GD99] B. Goffinet and R. Didier. Alternative models for qtl detection in livestock.iii. heteroskedastic model and models corresponding to several distributions of the qtl effect. *Genet. Sel. Evol.*, 31:341–350, 1999.

[GLRM+08] H. Gilbert, P. Le Roy, C. Moreno, D. Robelin, and J.M. Elsen. QTLMap, a software for QTL detection in outbred populations. In *Annals of Human Genetics*, volume 72, pages 694–694. WILEY-BLACKWELL COMMERCE PLACE, 350 MAIN ST, MALDEN 02148, MA USA, 2008.

[GLS99] William Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI-: Portable Parallel Programming with the Message Passing Interface*, volume 1. MIT press, 1999.

[GMB96] Jean-Francois Gibrat, Thomas Madej, and Stephen H Bryant. Surprising similarities in structure comparison. *Current opinion in structural biology*, 6(3):377–385, 1996.

[GZL+11] Andre Vincent Pascal Grosset, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary Hall. Evaluating graph coloring on GPUs. *SIGPLAN Notices*, 46(8):297, 2011.

[HCF+08] Doug Howe, Maria Costanzo, Petra Fey, Takashi Gojobori, Linda Hannick, Winston Hide, David P Hill, Renate Kania, Mary Schaeffer, Susan St Pierre, et al. Big data: The future of biocuration. *Nature*, 455(7209):47–50, 2008.

[HN07] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[HPS+10] J.R. Humphrey, D.K. Price, K.E. Spagnoli, A.L. Paolini, and E.J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7705, page 1, 2010.

[HR68] W. G. Hill and Alan Robertson. Linkage disequilibrium in finite populations. *TAG Theoretical and Applied Genetics*, 38:226–231, 1968. 10.1007/BF01245622.

[HTWH11] G. Hemani, A. Theocharidis, W. Wei, and C. Haley. EpiGPU: Exhaustive pairwise epistasis scans parallelised on consumer level graphics cards. *Bioinformatics*, 27:1462–1465, April 2011.

[Hyy04] Heikki Hyyrö. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, pages 16–27, 2004.

[IDN11] Katsumi Inoue, Andrei Doncescu, and Hidetomo Nabeshima. Hypothesizing about causal networks with positive and negative effects by meta-level abduction. In *Inductive Logic Programming*, pages 114–129. Springer, 2011.

[Kar72] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[KC89] Shufen Kuo and George R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. In *ACM SIGIR Forum*, volume 23, pages 89–99. ACM, 1989.

[KDH10] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

[KEH96] S. Knott, J. Elsen, and C. Haley. Methods for multiple-marker mapping of quantitative trait loci in half-sib populations. *TAG Theoretical and Applied Genetics*, 93:71–80, 1996. 10.1007/BF00225729.

[KJ10] Janez Konc and Dušanka Janežič. ProBiS algorithm for detection of structurally similar protein binding sites by local structural alignment. *Bioinformatics*, 26(9):1160–1168, 2010.

[KK98a] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KK98b] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[KK08] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[KKL05] Rachel Kolodny, Patrice Koehl, and Michael Levitt. Comprehensive evaluation of protein structure alignment methods: scoring by geometric measures. *Journal of molecular biology*, 346(4):1173–1188, 2005.

[KKN12] Kouichi Kimura, Asako Koike, and Kenta Nakai. A bit-parallel dynamic programming algorithm suitable for DNA sequence alignment. *Journal of Bioinformatics and Computational Biology*, 10(04), 2012.

[KT10] Kazutaka Katoh and Hiroyuki Toh. Parallelization of the MAFFT multiple sequence alignment program. *Bioinformatics*, 26(15):1899–1900, 2010.

[L+09] Dominique Lavenier et al. PLAST: parallel local alignment search tool for database comparison. *BMC bioinformatics*, 10(1):329, 2009.

[LAT10] Pu Liu, Dimitris K Agrafiotis, and Douglas L Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. *Journal of computational chemistry*, 31(7):1561–1563, 2010.

[Lew64] R. C. Lewontin. The interaction of selection and linkage. II. Optimum models. *Genetics*, 50:757–782, Oct 1964.

[LF09] A. Legarra and R. L. Fernando. Linear models for joint association and linkage QTL mapping. *Genet. Sel. Evol.*, 41:43, 2009.

[LHJV06] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. GPU accelerated smith-waterman. In *Computational Science–ICCS 2006*, pages 188–195. Springer, 2006.

[LHK10] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.

[LLDM09] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[LR09] Lukasz Ligowski and Witold Rudnicki. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[LREB+98] P. Le Roy, JM Elsen, D. Boichard, B. Mangin, JP Bidanel, and B. Goffinet. An algorithm for QTL detection in mixture of full and half-sib families. In *Proceedings of the 6th World Congress on Genetics Applied to Livestock Production*, volume 26, pages 257–260, 1998.

[LS84] Ten-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[LSM09] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 121–128. IEEE, 2009.

[LSM10a] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. CUDASW++ 2.0: enhanced smith-waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC research notes*, 3(1):93, 2010.

[LSM10b] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. MSAProbs: multiple sequence alignment based on pair hidden markov models and partition function posterior probabilities. *Bioinformatics*, 26(16):1958–1964, 2010.

[LSVMW06] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In *High Performance Computing-HiPC 2006*, pages 363–374. Springer, 2006.

[Lyn08] Clifford Lynch. Big data: How do your data grow? *Nature*, 455(7209):28–29, 2008.

[M+65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[MBBC07] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*. SIAM, 2007.

[MDAY10] Noël Malod-Dognin, Rumen Andonov, and Nicola Yanev. Maximum cliques in protein structure comparison. In *Experimental Algorithms*, pages 106–117. Springer, 2010.

[MNS12] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G. Sedukhin. Blocked united algorithm for the all-pairs shortest paths problem on hybrid CPU-GPU systems. *IEICE TRANSACTIONS on Information and Systems*, 95(12):2759–2768, 2012.

[MNU99] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. Leda: A platform for combinatorial and geometric computing. 38, 1999.

[Moo] Gordon E Moore. Excerpts from a conversation with gordon moore: Moore's law, 2005.

[MS03] Ulrich Meyer and Peter Sanders. [delta]-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[MV08] Svetlin A. Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[Mye99] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

[NVI12] NVIDIA. *NVIDIA CUDA Programming Guide 4.2*. 2012.

[OATLGE13] Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo Gonzalez-Escribano. A tuned, concurrent-kernel approach to speed up the APSP problem. 2013.

[OIH12] Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara. A task parallel algorithm for finding all–pairs shortest paths using the GPU. *International Journal of High Performance Computing and Networking*, 7(2):87–98, 2012.

[PCMM07] Guilherme P Pezzi, Márcia C Cera, Elton Mathias, and Nicolas Maillard. On-line scheduling of MPI-2 programs with hierarchical work stealing. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 247–254. IEEE, 2007.

[RAED10] C.E. Rabier, J.M. Azais, J.M. Elsen, and C. Delmas. Threshold and power for Quantitative Trait Locus detection. hal:http://hal.archives-ouvertes.fr/hal-00482142/en/, 2010.

[RS00] Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[SBS05] D.M. Strickland, E. Barnes, and J.S. Sokol. Optimal protein structure alignment using maximum cliques. *Oper. Res.*, 53(3):389–402, 2005.

[SHG+06] G. Seaton, J. Hernandez, J.A. Grunchec, I. White, J. Allen, DJ De Koning, W. Wei, D. Berry, C. Haley, and S. Knott. GridQTL: a grid portal for QTL mapping of compute intensive datasets. In *Proceedings of the 8th World Congress on Genetics Applied to Livestock Production*, pages 13–18, 2006.

[SHK+02] George Seaton, Chris S. Haley, Sara A. Knott, Mike Kearsey, and Peter M. Visscher. QTL Express: mapping quantitative trait loci in simple and complex pedigrees. *Bioinformatics applications note*, 18(2):339–340, 2002.

[SKK+02] Stefan Schmitt, Daniel Kuhn, Gerhard Klebe, et al. A new method to detect related function among proteins independent of sequence and fold homology. *Journal of molecular biology*, 323(2):387–406, 2002.

[SLL93] S. Subbiah, D.V. Laurents, and M. Levitt. Structural similarity of dna-binding domains of bacteriophage repressors and the globin core. *Current Biology*, 3(3):141–148, 1993.

[SNC77] Frederick Sanger, Steven Nicklen, and Alan R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[SOW+95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[SSMRLH13] Pablo San Segundo, Fernando Matia, Diego Rodríguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, pages 1–13, 2013.

[SSRLJ11] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[SW05] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[TB09] VA Traag and Jeroen Bruggeman. Community detection in networks with positive and negative links. *Physical Review E*, 80(3):036115, 2009.

[TDVD09] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. Magma library, version 0.1, 2009.

[THG94] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson. ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994.

[TKM07] LR Thimm, DL Kreher, and P Merkey. A parallel implementation for the maximum clique problem. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 63:183, 2007.

[TLC+03] Chuan Yi Tang, Chin Lung Lu, Margaret Dah-Tsyr Chang, Yin-Te Tsai, Yuh-Ju Sun, Kun-Mao Chao, Jia-Ming Chang, Yu-Han Chiou, Chia-Mao Wu, Hao-Teng Chang, et al. Constrained multiple sequence alignment tool development and its application to rnase family alignment. *Journal of Bioinformatics and Computational Biology*, 1(02):267–287, 2003.

[Tsa03] Yin-Te Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.

[TSH+10] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and computation*, pages 191–203. Springer, 2010.

[UI93] Naoto Ukiyama and Hiroshi Imai. Parallel multiple alignments and their implementation on CM5. *Genome Informatics*, 4:103–108, 1993.

[VD08] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[Vol10] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[VS11] Panagiotis D Vouzis and Nikolaos V Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[W+08] Mitch Waldrop et al. Big data: wikiomics. *Nature*, 455(7209):22, 2008.

[WAK13] Inken Wohlers, Rumen Andonov, and Gunnar W Klau. DALIX: Optimal DALI protein structure alignment. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 10(1):26–36, 2013.

[WKS11] Qingguo Wang, Dmitry Korkin, and Yi Shang. A fast multiple longest common subsequence (MLCS) algorithm. *Knowledge and Data Engineering, IEEE Transactions on*, 23(3):321–334, 2011.

[Woz97] Andrzej Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer applications in the biosciences: CABIOS*, 13(2):145–150, 1997.

[WPDK09] Inken Wohlers, Lars Petzold, Francisco Domingues, and Gunnar Klau. PAUL: Protein structural alignment using integer linear programming and lagrangian relaxation. *BMC Bioinformatics*, 10(Suppl 13):P2, 2009.

[WSVMW07] Liu Weiguo, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1270–1281, 2007.

[XcF10] Shucai Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[YXS10] Jiaoyun Yang, Yun Xu, and Yi Shang. An efficient parallel algorithm for longest common subsequence problem on GPUs. In *Proceedings of the world congress on engineering, WCE*, volume 1, 2010.

[ZC13] Kaiyong Zhao and Xiaowen Chu. GPU-BLASTN: Accelerating nucleotide sequence alignment by GPUs. *Poster at RECOMB*, 2013.

[ZS04] Yang Zhang and Jeffrey Skolnick. Scoring function for automated assessment of protein structure template quality. *Proteins: Structure, Function, and Bioinformatics*, 57(4):702–710, 2004.

[ZS05] Yang Zhang and Jeffrey Skolnick. TM-align: a protein structure alignment algorithm based on the TM-score. *Nucleic acids research*, 33(7):2302–2309, 2005.