Profiling biological applications for parallel implementation in multicore computers

Pedro Matos Geraldes Monteiro

Dissertation submitted to obtain the Master Degree in Information Systems and Computer Engineering

Jury
President: Prof. José Carlos Martins Delgado
Supervisor: Prof. Nuno Filipe Valentim Roma
Co-supervisor: Prof. Luís Manuel Silveira Russo
Member: Prof. José Carlos Campos Costa

November 2012
Acknowledgments

I would like to express my heart-felt gratitude to a group of people, without whom this thesis would not have been possible.

First of all, I would like to thank to my parents Domingas and Luís, to my sister Inês and to my grandparents Fernanda and Domingos for their unconditional love, support and encouragement to help me obtain a graduate degree in the field of my choice.

Secondly, I would like to thank to my friend Michel Cânovas, for all its support and friendship, but also for our endless and most of the time useless discussions on the results obtained in the scope of this work, which ended up by making me think and eventually move forward. I would also like to thank to my friend Nuno Marcos, for his support and friendship over the time we spent together in the course, at Instituto Superior Técnico.

Finally, I would like to thank to my advisers, Nuno Roma and Luís Russo, for their patience and endless faith in my work.

This work represents the end of an important stage in my life. Although I feel proud for the way how it ends, I feel particularly happy for the way how I lived it.

Thank you, one and all.

Pedro Matos Monteiro
Lisbon, October 27th 2012
Resumo

A informação biológica disponibilizada actualmente em diversas bases de dados continua a crescer todos os dias, o que representa um desafio constante para a área do desenvolvimento de novas arquiteturas de hardware e software dedicadas ao alinhamento de sequências. O principal objectivo deste trabalho é apresentar um conjunto de modelos de paralelização para um conjunto restrito de algoritmos de alinhamento de sequências baseados em programação dinâmica. É também proposta a arquitectura de um simulador que permite testar a solução apresentada no contexto de qualquer arquitectura baseada em GPPs. O algoritmo de Smith-Waterman foi considerado no contexto da implementação proposta. O modelo de paralelização, baseado em estratégias de paralelização de granularidade fina e grossa, obteve uma curva de desempenho bastante plana, escalando de forma praticamente linear com o número de processadores disponíveis na arquitectura. A implementação paralela do algoritmo conseguiu assim uma aceleração de mais de 480 vezes sobre a implementação sequencial, o que corresponde a um desempenho de mais de 77 GCUPS utilizando 32 worker threads, valores aproximadamente três vezes superiores aos obtidos com a implementação SWIPE, proposta por Rognes, para as mesmas condições. O estudo apresentado suporta o desenho conceptual de uma nova arquitectura heterogénea, em desenvolvimento no âmbito do projecto HELIX.

**Palavras-chave:** Algoritmos de bioinformática; Arquitecturas heterogéneas; Computação paralela; SIMD.
Abstract

Biological sequence databases keep growing every day, which represents a constant challenge in the context of the hardware and software architectures specially developed for high performance sequence alignment procedures. The main objective of this work is to propose a set of parallelization models for a restricted set of dynamic programming sequence alignment algorithms. A simulation environment is also proposed, making it possible to study the solution's behavior on a common GPP multi-core architecture. The Smith-Waterman algorithm was implemented in the context of the presented framework. The proposed parallelization model, based both on fine and coarse-grained parallelization approaches, obtained a rather flat performance curve in several different platforms, scaling almost linearly with the number of cores in the architecture. The algorithm's parallel implementation reached speedup values of over 480 times faster than the vanilla version of the same algorithm, which corresponds to a performance of more than 77 GCUPS obtained by using 32 parallel worker threads, which is almost three times faster than Rognes' SWIPE implementation running in the exact same conditions. The presented study supports the conceptual design of a dedicated heterogeneous architecture, which will be developed in the context of the HELIX project.

Keywords: Bioinformatic algorithms; Heterogeneous architectures; Parallel computing; SIMD.
This thesis was performed in the scope of project "HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis", funded by the Portuguese Foundation for Science and Technology (FCT) with reference PTDC/EEA-ELC/113999/2009, and project "TAGS: The power of the short - Tools and Algorithms for next Generation Sequencing applications", funded by FCT with reference PTDC/EIA-EIA/112283/2009.
# Contents

Acknowledgments ........................................................................................................ iii
Resumo ........................................................................................................................ v
Abstract ......................................................................................................................... vii
List of Figures ................................................................................................................. xv
Nomenclature ................................................................................................................. 1

1 Introduction ................................................................................................................. 1
   1.1 Objectives ........................................................................................................... 2
   1.2 Document Structure ......................................................................................... 2

2 Parallel Processing Systems ....................................................................................... 5
   2.1 Flynn's Taxonomy ............................................................................................ 5
   2.2 Thread-level Parallelism ................................................................................... 6
   2.3 Memory Models ............................................................................................... 6
   2.4 Parallel Architectures ...................................................................................... 7
      2.4.1 Homogeneous Architectures .................................................................. 8
      2.4.2 Heterogeneous Architectures ................................................................ 8
   2.5 Summary ........................................................................................................... 9

3 Sequence Alignment Algorithms .............................................................................. 11
   3.1 Needleman-Wunsch Algorithm ....................................................................... 11
   3.2 Smith-Waterman Algorithm ........................................................................... 12
   3.3 Summary .......................................................................................................... 14

4 Parallel Sequence Alignment .................................................................................. 15
   4.1 Coarse-grained Parallelism ............................................................................ 15
   4.2 Fine-grained Parallelism ................................................................................ 17
   4.3 Intermediate-grained Parallelism ................................................................. 22
   4.4 Summary .......................................................................................................... 24

5 Proposed Parallel Implementation .......................................................................... 27
   5.1 Thread-level Solution ..................................................................................... 27
      5.1.1 Wavefront Approach ............................................................................. 28
## 5.1.2 Master/Worker Model ............................................ 30
## 5.1.3 Helper Threads ................................................. 32
## 5.2 Instruction-level Solution ....................................... 33
## 5.2.1 SIMD Model Adaptation ....................................... 34
## 5.3 Summary .......................................................... 37

### 6 Implementation .................................................. 39

#### 6.1 Data Structures ............................................... 40
#### 6.2 Memory Access Model ......................................... 43
#### 6.3 Simulator Architecture ....................................... 45
#### 6.4 Summary ........................................................ 50

### 7 Results ............................................................. 51

#### 7.1 Thread-level Solution Analysis .............................. 52
##### 7.1.1 Evaluation of the optimal block size .................... 52
##### 7.1.2 Performance gain (Speedup) ............................ 54
##### 7.1.3 Helper Threads ............................................. 58
#### 7.2 SIMD Solution Analysis ....................................... 60
#### 7.3 Multi-level Solution Analysis ................................. 63
##### 7.3.1 Evaluation of the optimal block size .................... 63
##### 7.3.2 Performance gain (Speedup) ............................ 66
#### 7.4 Model Comparison .............................................. 68
#### 7.5 Summary ........................................................ 70

### 8 Conclusions ....................................................... 73

#### 8.1 Achievements .................................................... 73
#### 8.2 Future Work ..................................................... 74

## Bibliography ........................................................ 77
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Centralized shared-memory architecture.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Distributed-memory architecture.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Cell broadband engine architecture.</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Alignment matrix cell computation.</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Local alignment matrix for the above example sequences.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Database distribution across multiple computing nodes when following a <em>master/worker</em> processing approach.</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>SIMD vectorization approaches for the Smith-Waterman algorithm. For simplicity, vectors of only four elements are shown, while sixteen elements would normally be used. (A) Vectors along the anti-diagonal, described by A. Wozniak [16]. (B) Vectors along the query sequence, described by Rognes and Seeberg [17]. (C) Striped approach, described by Michael Farrar [18].</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>Multi-sequence vectors, described by Alpern et al. [22] and T. Rognes [19].</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>SWIPE’s core instructions.</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Wozniak’s anti-diagonal processing approach.</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Wavefront processing approach.</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Frontier dependencies between blocks.</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Simple wavefront processing approach issue.</td>
<td>31</td>
</tr>
<tr>
<td>5.5</td>
<td>Modified wavefront processing approach.</td>
<td>31</td>
</tr>
<tr>
<td>5.6</td>
<td><em>Master/Worker</em> communication model.</td>
<td>32</td>
</tr>
<tr>
<td>5.7</td>
<td><em>Master/Worker</em> communication model.</td>
<td>32</td>
</tr>
<tr>
<td>5.8</td>
<td><em>Master/Worker</em> communication model.</td>
<td>33</td>
</tr>
<tr>
<td>5.9</td>
<td>SWIPE’s database distribution strategy.</td>
<td>34</td>
</tr>
<tr>
<td>5.10</td>
<td>Multi-sequence SIMD vectorization approaches for the Smith-Waterman algorithm. For simplicity, vectors of only four elements are shown, while sixteen elements would normally be used. (A) SWIPE, which follows a parallel to the query sequence processing approach, described by T. Rognes [19]. (B) Proposed parallel processing approach, which adopts SWIPE’s SIMD model on multiple thread-level processing blocks.</td>
<td>35</td>
</tr>
</tbody>
</table>
6.1 Message data structure. .................................................. 41
6.2 Border table data structure. ............................................. 42
6.3 Global table data structure. ............................................. 43
6.4 Block processing operation. ............................................. 44
6.5 Memory system implemented in the simulator architecture. .... 47
6.6 Master thread execution cycle. ........................................ 48
6.7 Worker thread execution cycle. ...................................... 49

7.1 Performance dependency on the query block size using 32 worker threads and a profile block size of 1 024 amino acid residues. .................................................. 53
7.2 Performance dependency on the profile block size using 32 worker threads and a query block size of 2 048 amino acid residues. ............................................. 53
7.3 Performance dependency on the query block size using 4 worker threads and a profile block size of 2 048 amino acid residues. ............................................. 54
7.4 Performance dependency on the profile block size using 4 worker threads and a query block size of 2 048 amino acid residues. ............................................. 54
7.5 Variation of the obtained speedup with the number of workers (32-Core Architecture). ... 55
7.6 Variation of the obtained speedup with the number of workers (4-Core Architecture). ... 55
7.7 Performance dependency on the query length using 32 worker threads. ....................... 57
7.8 Performance dependency on the query length using 4 worker threads. ....................... 57
7.9 Number of anticipations in a set of 65 536 processing blocks, by adopting a line based prefetching approach. .................................................. 59
7.10 Number of anticipations in a set of 65 536 processing blocks, by adopting a column based prefetching approach. .................................................. 59
7.11 Comparing the parallel processing models with and without helper threads, by using 32 worker threads. .................................................. 60
7.12 Performance dependency on the query length using SIMD vectorial parallelization on a single worker thread. .................................................. 61
7.13 Variation of the total number of detected overflows using a 7-bit score range and different query lengths. .................................................. 62
7.14 Performance dependency on the query length using SWIPE SIMD vectorial parallelization model on a single thread. .................................................. 62
7.15 Performance dependency on the query block size using 32 worker threads and a profile block size of 16 384 amino acid residues. .................................................. 64
7.16 Performance dependency on the profile block size using 32 worker threads and a query block size of 4 096 amino acid residues. .................................................. 64
7.17 Performance dependency on the query block size using 4 worker threads and a profile block size of 32 768 amino acid residues. .................................................. 65
7.18 Performance dependency on the profile block size using 4 worker threads and a query block size of 4 096 amino acid residues. ............................. 65
7.19 Variation of the obtained speedup with the number of workers (32-Core Architecture). . . 67
7.20 Variation of the obtained speedup with the number of workers (4-Core Architecture). . . 67
7.21 Performance dependency on the query length using 32 worker threads. ................. 67
7.22 Performance dependency on the query length using 4 worker threads. ................. 68
7.23 Performance dependency on the query length for different parallel applications. ........ 68
7.24 Multi-sequence SIMD vectorization approaches for the Smith-Waterman algorithm. (A) SWIPE, which follows a parallel to the query sequence processing approach, described by T. Rognes [19]. (B) Proposed parallel processing approach, which adopts SWIPE’s SIMD model on multiple thread-level processing blocks. .................. 69
Chapter 1

Introduction

For many years, since the computer industry took its first steps, major manufacturers kept trying to make processors run faster, by gradually incrementing the number of transistors in the chip. However, in the late 1990s, a physical limit was reached: the processor’s global clock speed could no longer be increased without overheating [1]. As a consequence, architectures containing multiple independent processors became extremely popular.

Since then, single chips have increased their capacity, allowing the designers to be able to place several processors on a single die, which allowed several tasks to be simultaneously performed in the context of a single application. Thus, parallel programming techniques also became very popular, which highly leveraged the processing of large amounts of data.

These type of parallel architectures made it possible to more easily reach the computational power required to solve huge mathematical or scientific problems. As a consequence, many scientific communities started adopting them in order to improve the performance in the context of their applications.

One of such application areas is the bioinformatics research. Nowadays, biological data is becoming both more plentiful and more accessible to all the researchers around the world. The information in the GenBank keeps doubling every 18 months [2], which constantly increases the computational cost of biological sequence alignment performance between new genetic information and the existing information kept on multiple online databases.

Due to the demand for both efficient and sensitive searches, much effort has been made in order to produce faster implementations of biological sequence alignment algorithms. Thus, in the past few years, several parallel implementations have been presented, often adopting several different parallel approaches.

Some of these approaches exploit high performance heterogeneous architectures, which can even be specialized and fine-tuned in order to achieve performance levels as high as possible, to execute a very specific set of applications. One example of such specialized and dedicated multicore architectures is the one that is being developed in the scope of the HELIX project, at INESC-ID. Such processor is composed by a single general purpose processor (GPP), and multiple dedicated cores, specifically designed for biological sequence alignment.
This document, developed in the scope of a MSc thesis, proposes a set of parallelization models for a small set of sequence alignment algorithms. These models have been formalized and extensively evaluated, in order to conceptually design a dedicated heterogeneous architecture for these particular bioinformatic applications.

1.1 Objectives

The aim of the present study is to profile a set of new parallelization techniques, in order to support the design of a new heterogeneous multi-core architecture specially built for biological data processing.

The rapid growth of available genetic-sequence information represents a constant challenge in the development of new hardware and software platforms. The alignment of two different biological sequences is an important operation required in the implementation of multiple bioinformatics applications, such as sequence database searching, multiple sequence alignment, genome assembly or short-read mapping.

When this operation needs to be computed many times (for example when searching a sequence database), the computation time becomes substantial. Thus, several approaches have been presented in order to reduce the total amount of time needed to accomplish that task.

Algorithms can be implemented by making use of several different forms of parallelization. The approach where parallelization is carried out across multiple database sequences is known as inter-task parallelization, in contrast to intra-task parallelization, where the parallelization occurs within a single alignment procedure.

Instruction-level parallel solutions are mostly concentrated on parallelization within a single alignment. On the other hand, database distribution techniques are usually adopted at a very high computing level. In both cases, some of the original algorithm dependencies must be kept to avoid compromising the solution’s global performance or making it suitable for a set of specific computing architectures.

In order to achieve the proposed objective, both parallelization models were considered in the context of a single solution. These models are formally described, evaluated and discussed in the following chapters of this document.

1.2 Document Structure

First, a set of theoretical definitions are presented, which will support the remaining text and documented work (chapter 2). Then, the set of bioinformatic algorithms that will be considered in this study are extensively analyzed and described (chapter 3).

A description of the state of the art technology for parallel sequence alignment is presented in chapter 4. In particular, multiple implementations using several different levels of parallelism are described, in order to correctly contextualize the solution that is proposed in this work.

Chapter 5 presents the description of the proposed parallel solution. The adopted thread-level model is firstly described. However, multiple parallel models were considered in a single solution. Thus, an
SIMD solution is also formally described in this chapter. The integration and implementation of both models are described in chapter 6.

Finally, the results obtained by testing the previously described models in several computing platforms are extensively evaluated and discussed in chapters 7 and 8.
Parallel Processing Systems

Parallel processing consists in the ability to simultaneously carry out multiple operations or tasks. Although the term "parallel" has become extremely popular in computer science, it still remains ambiguous and ubiquitous if a program is regarded as a simple sequence of instructions on a considered set of data. In order to correctly define and understand the concepts behind parallel computing, this section presents a set of important theoretical definitions that will support the remaining text and documented work.

2.1 Flynn's Taxonomy

About 40 years ago, Michael J. Flynn proposed a simple model which still allows the categorization of any computer architecture. He began by adopting two important definitions [3]: the Instruction Stream, as the sequence of instructions performed by a machine, and the Data Stream, as the set of data manipulated by the so called instruction stream. By applying the concept of parallelism on both levels, he was able to place every single computer into one of four categories [3]:

- Single Instruction Stream, Single Data Stream (SISD) - corresponds to the common uniprocessor approach.

- Single Instruction Stream, Multiple Data Stream (SIMD) - the same instruction is executed by multiple processors using different data streams; this category exploits data-level parallelism, which consists in applying the same operation to multiple data items in parallel.

- Multiple Instruction Stream, Single Data Stream (MISD) - represents a vector pipeline of multiple independently executing functional units operating on a single stream of data, while the results are kept being pushed forward [4]. There are very few commercial applications of this type of architecture.

- Multiple Instruction Stream, Multiple Data Stream (MIMD) - each processor fetches its own instructions and operates on its own independent data set. This architecture exploits the so called thread-level parallelism, since multiple threads or processes are able to operate in parallel.
2.2 Thread-level Parallelism

MIMD architectures offer two major advantages over any other processor architecture [5]: flexibility, since they can still function as single-program processors, focusing on high performance for one or more applications; and cost-performance gain, since common processors can be used to build workstations or large single-processor servers. Furthermore, since single chips started increasing their capacity, designers were able to place several processors on a single die, making it possible to run multiple parallel processes on a single machine.

In this scope, a process corresponds to a segment of code that may be independently run, since it contains all the information needed to execute a given program on a single processor. Hence, in a multiprogrammed environment, each processor may be running an independent program, or more specifically, its corresponding process.

Although this represents an interesting and, in some cases useful approach, it is also useful to be able to have multiple processors executing a single program, by sharing its code and most of its address space. The term thread is used to describe processes which perform this type of sharing among each other. To take advantage of an MIMD multiprocessor composed by \( n \) processors or cores, there should be at least \( n \) threads to execute in parallel [1]. These independent threads are usually specified by the programmer, in the application code, and then recognized by the compiler. Thread-level parallelism is identified at a high level by software systems, where each thread consists of hundreds to millions of instructions that may be executed in parallel. Thus, in the scope of this thesis, it should be defined a more precise MIMD category, which is Multiple Thread Stream, Multiple Data Stream (MTMD).

It is also important to mention that threads can also be used to exploit data-level parallelism, although the data set must be sufficiently large in order to efficiently exploit data parallelism [5].

2.3 Memory Models

Since multiple parallel threads are able to access and manipulate shared data memory in the context of a given application, the adopted memory architecture model becomes extremely important.

The number of processors involved on a MIMD or MTMD multiprocessor architecture basically dictates its global memory organization and interconnect strategy. Multiprocessors composed by small processor counts (typically less than 100 cores) usually adopt a centralized shared-memory architecture, where all the processors share a single and centralized memory device (Figure 2.1).

These systems typically adopt an Uniform Memory Access model (UMA), where all the available cores share a single physical memory uniformly. Hence, all the processors end up by sharing a single memory bus.

Although scaling is technically conceivable in this type of architecture, sharing a centralized memory becomes less attractive as the number of processors increases. By having a single shared memory, access time typically becomes uniform for all the processors, which means that memory latency will be also uniform, eventually imposing the bus access physical limitation in the system [5].
In order to overcome this issue, some centralized shared-memory architectures adopt a Non-Uniform Memory Access model (NUMA). In this case, the available cores are grouped in small clusters, which have their own memory access buses. Hence, although the total number of available processors remains the same, each physical bus is shared only by a small group of cores, which improves the system's overall performance.

Although this has become an usual approach in centralized shared-memory architectures, due to the increase on the total number of cores built on a single die, the NUMA model is more often adopted in the context of distributed-memory architectures.

In contrast, the basic distributed-memory architecture adopts a quite different approach, based on an explicit exchange of data between a set of individual nodes containing a processor, some memory and an interface to an interconnection network that connects all the nodes (Figure 2.2). Distributing memory among several nodes has two major advantages [5]: first, it is a cost-effective way to scale memory bandwidth, if most of the accesses are to the local memory of the nodes; second, it reduces the latency of accesses to the local memory.

However, transferring data between processors becomes somewhat more complex, which demands a stronger effort in software development, in order to be really able to take advantage of the increased memory bandwidth afforded by the distributed memory architecture.

**2.4 Parallel Architectures**

As chip multiprocessors are gradually becoming the new direction for future systems in the industry, they are also spurring new bursts of computer architecture innovations. In fact, the need for high performance computing capacity in order to allow some particular tasks to be efficiently fulfilled requires the application of new techniques, in order to overcome the existing technology's limitations in ways that previous generation's techniques could not.
2.4.1 Homogeneous Architectures

Homogeneous computing architectures are characterized by the use of multiple and equal computational units or processors. This type of architecture is frequently used to aggregate several GPPs, in order to easily develop and improve the computational power required to solve huge mathematical or scientific problems. In fact, homogeneous architectures have recently become specially common, with the arise of multicore computers. Intel or AMD multicore GPP computers are well known commercial examples of this type of systems.

However, although such homogeneous systems based on the use of multiple GPPs are mainly characterized by their ease of use, they end up being too general in terms of their functionality and architecture, which makes them considerably inefficient while performing very specific high performance tasks.

2.4.2 Heterogeneous Architectures

Contrasting to homogeneous architectures, heterogeneous computing architectures refer to systems that use a variety of different types of computational units or processors.

The demand for increased heterogeneity in computing systems is mainly due to the need for higher levels of performance. These performance requirements usually arise from particular mathematical or scientific tasks, which require very specific computational environments in order to be successfully and efficiently completed. Currently, the most widely recognized method to gain extra performance out of the computing systems is based on the introduction of additional specialized resources, thus making the system heterogeneous. This technique allows the designers to use multiple types of processing units, each of them able to perform the tasks that they are best suited for [6].

There are currently several commercial systems based on heterogeneous architectures. One of the most well known is the Cell Broadband Engine. Cell is a microprocessor architecture jointly developed
by Sony, Toshiba, and IBM (an alliance known as “STI”), and had its first major commercial application in Sony’s Playstation 3 game console.

The processor’s typical configuration is based on a multi-core chip composed by a single GPP and multiple synergistic processing elements (SPEs) [6]. The GPP and the SPEs are linked together by an internal high speed communication bus (Figure 2.3). The GPP generally plays a “Master” role in the context of an application, by managing the data exchange operations between the SPEs, which are capable of performing double precision calculations at 3.2 GHz [6]. Each SPE manages its own independent cache, while the GPP is able to access and manage the global memory system. Thus, data transfers between processors must be explicitly performed at software level, which significantly hinders the programmer’s work in order to avoid a bottleneck imposed by memory access delays.

![Cell broadband engine architecture.](image)

Figure 2.3: Cell broadband engine architecture.

The potential of the Cell processor for scientific computing [7] ended up by inspiring several non-commercial processor architectures. One of these projects, which is still in a development phase, is the so called HELIX project, developed at INESC-ID.

The HELIX project is based on an integrated parallel hardware and software platform, which targets the acceleration of a wide range of bioinformatic algorithms. The remaining text and documented work was developed in the context of the HELIX project, whose design has several aspects in common with the previously described Cell architecture.

### 2.5 Summary

Although the term “parallel” has become extremely popular in computer science, it still remains ambiguous and ubiquitous if a program is regarded as a simple sequence of instructions on a considered set of data.

About 40 years ago, Michael J. Flynn proposed a simple model which still allows the categorization of any computer architecture. He began by adopting two important definitions [3]: the Instruction Stream

---

1HELIx project webpage, [http://sips.inesc-id.pt/~helix](http://sips.inesc-id.pt/~helix)
as the sequence of instructions performed by a machine and the *Data Stream* as the set of data manipulated by the so called instruction stream. By applying the concept of parallelism on both levels, he was able to place every single computer into one of four categories [3]:

- Single Instruction Stream, Single Data Stream (SISD);
- Single Instruction Stream, Multiple Data Stream (SIMD);
- Multiple Instruction Stream, Single Data Stream (MISD);
- Multiple Instruction Stream, Multiple Data Stream (MIMD).

MIMDs offer two major advantages over any other processor architecture [5]: *flexibility*, since they can still function as single-user processors focusing on high performance for one or more applications; and *cost-performance gain*, since common processors can be used to build workstations or large single-processor servers.

Since single chips started increasing their capacity, designers were able to place several processors on a single die, making it possible to run multiple parallel processes on a single machine. Hence, it became useful to be able to have multiple processors executing a single program by sharing its code and most of its address space. The term *thread* is used to describe processes which perform this type of sharing among each other. To take full advantage of an MIMD multiprocessor composed by \( n \) processors or cores, there should be at least \( n \) threads to execute in parallel [1].

The number of processors involved on a MIMD multiprocessor architecture basically dictates its global memory organization and interconnect strategy. Multiprocessors composed by small processor counts (typically less than 100 cores) usually adopt a centralized *shared-memory* architecture, where all the processors share a single and centralized memory device. On the contrary, the basic *distributed-memory* architecture adopts a quite different approach, based on the exchange of data between a set of individual nodes containing a single processor and their own memory units.

Both memory architectures can be implemented in the scope of multiple processor architectures. Homogeneous computing architectures are characterized by the use of multiple *equal* computational units or processors. This type of architecture is typically based on the use of several GPPs, in order to easily develop and improve the computational power required to solve huge mathematical or scientific problems.

On the other hand, heterogeneous computing architectures refer to systems that use a variety of different types of computational units or processors. One of the most well known heterogeneous systems is the Cell broadband engine, which consists of a multi-core chip composed by a single GPP and multiple synergistic processing elements (SPEs) [6].

The potential of the Cell processor for scientific computing [7] ended up by inspiring several non-commercial processor architectures. One of these projects, which is still in a development phase, is the so called HELIX project, developed at INESC-ID.
Chapter 3

Sequence Alignment Algorithms

Bioinformatics is a branch of biological science which deals with the application of computer science and information technologies to the fields of biology and medicine. Its primary goal is to increase the understanding of biological processes, which often involves the development and application of computational intensive techniques.

One of the greatest research efforts in these fields include DNA sequence analysis. The most basic sequence analysis task consists in verifying if two different sequences are related by comparing each of their residues, which define single units within a nucleotide or amino-acid structure. This is usually done by first aligning the sequences (or parts of them) and then decide if that alignment is more likely to have occurred because the sequences are actually related, or just by chance, which highly depends on the used scoring system to rank the alignments [8].

When an alignment is considered, there is only one possible global alignment for the complete sequences. However, things become more complicated once gaps are allowed, or once local alignments between subsequences of two main sequences are considered.

Algorithms that are able to find optimal alignments given an additive alignment score model are usually based on dynamic programming methods. The key idea behind dynamic programming is quite simple: in general, to solve a given problem it is necessary to solve many different parts of that problem (subproblems) and then combine the obtained solutions in order to reach an overall solution. However, many of these subproblems are often the same. The dynamic programming approach seeks to solve this global subproblem only once [9], which is especially useful when the number of repeating subproblems is very large, which ends up happening in general DNA/protein pairwise alignment problems. Dynamic programming sequence alignment algorithms are also guaranteed to find the optimal scoring alignment or set of alignments [8].

3.1 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm (1970) was one of the first techniques to obtain the global alignment using an iterative calculation method [10], which can easily be implemented by a top-down dynamic
programming based algorithm, where multiple results are stored in a matrix and later used in a larger and global calculation.

It is defined by considering a scoring matrix $F$, indexed by $i$ and $j$. The algorithm starts by initializing $F(0, 0) = 0$ and it then fills the matrix from the top left to the bottom right corner, by computing the following equation, where $s(a_i, b_j)$ corresponds to a comparison score between two sequence elements $a_i$ and $b_j$, and $d$ to a generic gap insertion cost:

$$F(i, j) = \max \begin{cases} F(i - 1, j - 1) + s(a_i, b_j), \\ F(i - 1, j) - d, \\ F(i, j - 1) - d. \end{cases} \tag{3.1}$$

The equation is repeatedly applied in order to fill in the matrix with the $F(i, j)$ values, by calculating the value in the bottom right-hand corner of each square of four cells from one of the remaining three values [8] (see Figure 3.1). By definition, the value in the bottom-right cell of the entire matrix, $F(n, m)$, corresponds to the best score for an alignment between $a_1...a_n$ and $b_1...b_m$.

![Alignment matrix cell computation.](image)

However, the Needleman-Wunsch algorithm is somewhat limited in the extend that it exclusively performs *global alignments* or, in other words, it assumes the search for the best end-to-end match between two different sequences. A much more common situation occurs when the best alignment between subsequences within different sequences is required. This arises, for example, when it is postulated that two different protein sequences may share a common domain. It is also useful to detect similarities when comparing two highly diverged sequences, even when they may have a shared evolutionary origin along their entire length [8].

### 3.2 Smith-Waterman Algorithm

In 1980, Smith *et al.* [10] extended the Needleman-Wunsch algorithm by enabling it to find pairs of segments, one from each of the two different sequences, such that there is no other pair of segments with greater similarity in the context of both sequences.

Formally defining the algorithm, let the two sequences be defined as:
A similarity score \( s(a, b) \) is given between each pair of sequence elements in eq. 3.2 and eq. 3.3. Gaps of variable lengths are also given a generic weight \( d \). In order to find segment pairs with a high degree of similarity, a dynamic programming matrix \( F \) is defined, such as follows:

\[
F_{k0} = F_{0l} = 0, \text{ for } 0 \leq k \leq n \text{ and } 0 \leq l \leq m
\]  

(3.4)

The remaining values of the matrix are calculated based on the following equation [10]:

\[
F(i, j) = \max \begin{cases} 
0, \\
F(i-1, j-1) + s(a_i, b_j), \\
F(i-1, j) - d, \\
F(i, j-1) - d.
\end{cases}
\]  

(3.5)

The pair of segments with maximum similarity is found by locating the maximum element on \( F \). The remaining matrix elements that conduct to this maximum value can be sequentially determined with a traceback procedure ending with an element that equals zero. The pair of segments with the next best similarity level is found by applying the same traceback procedure to the second largest element on \( F \) that was not associated with the first traceback [10].
procedure by following the arrows in bold, it was possible to correctly determine the sequences’ most similar pair of segments, which appears right under the table in Figure 3.2.

As it can be seen, the Smith-Waterman algorithm is closely related to the previously described Needleman-Wunsch algorithm for global alignments. There are actually just two differences between them [8]. First, in each matrix cell there is an extra possibility in eq. 3.1, allowing \( F(i, j) \) to take the value of 0 if all other options have a value less than 0, which corresponds to start a new local alignment from that point. Secondly, an alignment can now finish anywhere in the matrix. As a consequence, instead of taking the value in the bottom-right corner for the best score, the highest value of \( F(i, j) \) over the whole matrix has to be looked for, and the traceback procedure should start from there.

### 3.3 Summary

Bioinformatics corresponds to the application of computer science and information technologies to the fields of biology and medicine.

One of the greatest research efforts in these fields include DNA sequence analysis. The most basic sequence analysis task consists in asking if two different sequences are related. This is usually done by first aligning the sequences (or parts of them) and then decide if that alignment is more likely to have occurred because the sequences are actually related, or just by chance, which highly depends on the used scoring system to rank the alignments [8].

The Needleman-Wunsch algorithm (1970) was the first technique using an iterative matrix method of calculation [10], which can easily be implemented by a top-down dynamic programming based algorithm, where multiple results are kept being stored and later used in the context of a larger and global calculation. However, this algorithm is somewhat limited in the extend that it exclusively performs global alignments or, in other words, it assumes the search for the best end-to-end match between two different sequences. A much more common situation occurs when the best alignment between subsequences within different sequences is required.

In 1980, Smith et al. [10] extended the Needleman-Wunsch algorithm by enabling it to find pairs of segments, one from each of the two different sequences, such that there is no other pair of segments with greater similarity in the context of both sequences.

The Smith-Waterman algorithm is closely related to the previously described Needleman-Wunsch algorithm for global alignments. There are actually just two differences between them [8]. First, in each matrix cell there is an extra possibility in equation (3.1), allowing each cell to take the value of 0 if all other options have a value less than 0, which corresponds to start a new local alignment from that point. Secondly, an alignment can now finish anywhere in the matrix. As a consequence, instead of taking the value in the bottom right corner for the best score, the highest score value over the whole matrix has to be looked for, and the traceback procedure should start from there.
Chapter 4

Parallel Sequence Alignment

Since it was created to the present day, the number of bases in GenBank has approximately doubled every 18 months [2]. The rapidly increasing amounts of available genetic-sequence information represents a constant challenge in the development of hardware and software database searching and handling architectures. Furthermore, the rapid expansion of this information is actually exceeding the growth in computing power available at a constant cost. If this trend continues, increasingly longer time or increasingly expensive computers will be needed in order to search through an entire biological database.

The demand of fast and sensitive searches implies a continuous effort to produce faster implementations of several of the previously described dynamic programming algorithms.

These implementations use several different levels of parallelism. Most solutions typically adopt a set of techniques corresponding to one or more of Flynn’s categories. By doing so, applications are able to explore both instruction and data level parallelism.

This chapter presents a description of state of the art technology for parallel sequence alignment architectures.

4.1 Coarse-grained Parallelism

Coarse-grain refers to parallelism that is achieved at a very high computing level. It is typically related to data-level parallelism, which basically focus on distributing data across different parallel computing nodes. In a multiprocessor system, data parallelism is achieved when each processor simultaneously performs the same task on a different piece of distinct data.

In the context of parallel sequence alignment, this suggests the possibility of simultaneously aligning multiple sequences with a single query sequence. Thus, many hardware and software architectures exploit data-level parallelism in the context of sequence alignment algorithms by distributing data for several available resources, which independently perform a set of DNA alignments.

From an architecture point of view, data parallel implementations typically adopt a master/worker approach, where a single processor acts as master and the remaining processors as workers (see
Figure 4.1). Basically, the master process reads the query and splits the sequence database into $n - 1$ parts, which are distributed among the $n - 1$ workers so they can search the database and perform the alignments in parallel. After this task is completed, workers send the computed scores back to the master, which sorts and displays the alignments [11].

Figure 4.1: Database distribution across multiple computing nodes when following a master/worker processing approach.

This type of solution is generally adopted in the scope of large clusters. Several workstations are typically connected by an Ethernet network, which usually supports an MPI library based implementation for data exchange between processors and for processing coordination [11][12].

Solutions based on this implementation end up having good results [11]. However, they often present a difficult tradeoff, due to the significant time penalty imposed by the communication stages required to transfer data among several distributed nodes [12].

Based on this programming model, Sebastião et al. [13] presented a data parallel solution based on a field programmable gate array (FPGA). The idea behind this hardware solution is similar to what was previously described. Basically, a set of multiple short-read sequences are aligned over a multiple-stream array and aligned in parallel over a single larger and common sequence. The exact number of parallel arrays is configurable according to both the size of the short-read sequences to be aligned and the amount of available hardware resources, which imposes a physical limitation to the system.

Other proposal exist [14][15]. However, these solutions end up by leading to speedup values at a very high cost, since they are often extremely limited in terms of flexibility [12].

In conclusion, although data-level parallelism based solutions are able to obtain interesting results, they usually consist in changing the way the data is managed, not the way the algorithm is able to perform its central actions. As stated before, these implementations are typically limited in terms of scalability and flexibility, focusing the results in the short-read DNA sequence alignment area. In large alignments, data parallel implementations are not so efficient. Thus, data-level parallelism is typically used on top of other finer parallel based solutions [11][12].
4.2 Fine-grained Parallelism

Fine-grained parallelism refers to a set of solutions that exploit parallelism at a very low computational level: the processor’s instruction level. With the inclusion of SIMD vector instructions in the instruction set of most processor architectures, several parallel sequence alignment implementations were proposed. These solutions were typically based on current GPPs [16][17][18][19]. However, by following this same SIMD fine-grained based approach, other most recent proposals were also presented, by making use of several heterogeneous architectures, such as the Cell Broadband Engine [20] or general Graphics Processing Units (GPUs) [11][21].

A. Wozniak presented one of the first parallel implementations of the Smith-Waterman algorithm, based on a GPP architecture [16]. He based his solution on a video-oriented instruction set extension named VIS (Visual Instruction Set), which could be found in the SUN ULTRA SPARC processor.

The proposed approach was to parallelize the algorithm at the single sequence comparison level. As seen before, each Smith-Waterman’s matrix cell value depends on its \((i - 1, j), (i - 1, j - 1)\) and \((i, j - 1)\) neighbors, which can be visualized as its vertical, diagonal and horizontal dependencies. Thus, if the computation of the \(n^{th}\) row is shifted one place to the right, rows \(n - 1\) and \(n\) can be executed in parallel with only a small overhead of two steps: one on the beginning of the algorithm’s inner loop and the second on its end.

VIS instructions used special 64-bit registers, making it possible to add, in a single instruction, two sets of four 16-bit integers and get four 16-bit results. Thus, VIS instructions made it possible to process up to four matrix rows in parallel. To simultaneously process several rows, it uses a parallel minor diagonal approach (see Figure 4.2 - A). The resulting performance was greater than 18 million cell updates per second (MCUPS) on a single processor machine, which corresponds to a speedup factor of two compared to the same algorithm implemented with integer instructions on the same machine [16].

This model was then further improved in many different ways, to obtain higher speedup values. T. Rognes et al. proposed a solution based on the MultiMedia eXtensions (MMX) and on the Streaming SIMD Extensions (SSE) technologies, which were available in Intel’s most recent microprocessor architectures at the time [17]. MMX used 64-bit registers. However, in this implementation they are divided into as many units as possible, which means that eight 8-bit units could be independently processed. This option doubled the number of parallel operations, but limited the precision of the calculations to the range 0 - 255.

Furthermore, despite the loss of independence between the computation of each of the vector elements, a processing scheme considering vectors of cells parallel to the query sequence was used (see Figure 4.2 - B), instead of vectors of cells parallel to the minor diagonal in the matrix, as adopted in the previously described solution.

The main disadvantage that was introduced by this new processing method was that, due to the dependence between cells in different vector alignments, conditional branches were placed inside the algorithm’s inner loop. With conditional code inside the loop, the execution time became dependent not only on the length of the query string and on the global scoring matrix, but also on the number of gap
penalties that were introduced [18]. Still, this implementation lead to a performance of more than 150 MCUPS on a single Intel Pentium III microprocessor at 500 MHz, which is almost six times faster than Wozniak’s approach, running in the exact same conditions [17].

To circumvent this disadvantage, Michael Farrar proposed a new version of this previous implementation, where SIMD registers were also parallel to the query sequence but accessed by following a striped pattern (see Figure 4.2 - C) [18]. With such approach, conditional calculations were moved outside the algorithm’s inner loop. Basically, when calculating $F(i, j)$ in the Smith-Waterman algorithm (see equation 3.5), the value from the scoring matrix $d$ is added to $F(i - 1, j - 1)$. The previously described solutions had data dependencies between the previous and current vectors. However, by adopting a striped query access, these data dependencies could be moved outside the inner cycle and performed just once, while accessing the database for next sequence reading. This solution achieved performances of over 3 billion cell updates per second (GCUPS), reaching a speedup of approximately 8 times over the previously described SIMD implementations.

One common aspect of all these solutions is that they exploit intra-task parallelization techniques, where parallelism occurs within a single pair of sequences. Fine-grained parallelism is typically concentrated on this approach. However, other approaches where parallelization is carried out across multiple database sequences are also an option. The main advantage of such inter-task parallelization approaches, where multiple database sequences are processed in parallel, is that it significantly simplifies or even avoids all data dependences within the alignment matrix [22]. This approach does not seem to have been exploited much until T. Rognes presented his latest work [19].
The aim of his study was to exploit the use of an *inter-task* approach, by still using SIMD on ordinary CPUs. His model was implemented on common Intel processors with the SSSE3 instruction set extension [23], by using parallelization over multiple database sequences (see Figure 4.3).

Each SIMD vector has a fixed length of 128 bits. Thus, residues from up to sixteen different database sequences can be retrieved and processed in parallel. This approach was implemented in a software tool named SWIPE [19], which was mainly written in C++ and contains some parts hand coded in inline assembler.

Residues from several database sequences are processed in parallel by following a simple *inter-task* parallelization technique. Each of the up to sixteen residues are all simultaneously compared to the same query residue, and all operations are carried out using vectors consisting of sixteen independent bytes. As such, all the residues are fed into sixteen independent channels. When the first of these database sequences ends, the first residue of the next database sequence is loaded into that channel. In contrast with other solutions, the database sequences are read in the same order as they appear in the original database file.

Paying attention to the alignment procedure itself, the basis for the computations of the values in each cell are given by the algorithm’s general recurrence relations [10]. All the computations can be written in as little as ten assembly instructions, which constitute the core of the algorithm’s inner loop (see Figure 4.4). These ten instructions compute each vector of sixteen cells containing values from up to sixteen independent alignment matrices. The exact selection of instructions as well as their order are highly important. This part of the code was therefore hand coded by Rognes in order to maximize the performance.

In the code, $H$ represents a line of the algorithm’s global score matrix. Then, the $H$ vector is saved in the $N$ vector, in order to save the next cell on the diagonal. $E$ and $F$ represent the score vectors.
for alignments ending in a gap in the query and database sequences, respectively. $P$ is the vector of the substitution scores for the database sequences versus the query residue $q$. $Q$ represents the vector corresponding to the gap open plus gap extension penalty. $R$ represents the gap extension penalty vector. Finally, $S$ represents the current best score vector. It is important to note that all these vectors (except $N$) must be properly initialized prior to this code segment.

SWIPE’s global performance strongly depends on this part of the code. As a consequence, it has been studied and optimized in many ways. The values in $H$ and $E$ usually have to be read and written at least once in the context of each matrix cell. These arrays are usually small enough to be cached at a close cache level, so that the memory access time is not a major concern. However, they still need to be written and read back for each cell.

Since there are sixteen 128-bit registers available in the processor, there is enough space for keeping the $H$, $E$ and $F$ values of a few cells in the registers. This further allows to reduce the running time, by computing several consecutive cells along the database sequences before moving on to the next query residue. An amount of four consecutive cells was found to perform well so this was the adopted value.

Unrolling the inner loop once along the query sequence was also found to work well. Then, the basic computing blocks consist of $2 \times 4$ cells, which are simultaneously processed in the context of each inner loop iteration.

As it was referred before, each time a new database sequence enters into the processing loop in use of the available channels, the score of the previous database sequence must be recorded. In addition to that, the values in vectors $H$, $E$ and $S$ that are related to the previous sequence must be reset. Then, when a new column starts its processing, it is checked whether any database sequence ended in the previous column. If that is the case, some additional processing is carried out: the final scores are recorded and a mask is created in order to reset the values of vectors $H$, $E$ and $S$ in the appropriate channels. Then, the channels are correctly filled, and one or more new database sequences are started. No processing steps are carried out for a new column if no sequences ended in the previous column.

Most of the columns follow this simple type of processing. Thus, the overall performance will mostly depend on the speed of processing a simple step. In order to simplify the computations, each channel is padded with one to three null symbols after the end of a sequence whenever its length is not a multiple

```
paddsb P[q], H    // H = H + P[q]
pmaxub F, H       // H = max(H, F)
pmaxub E, H       // H = max(H, E)
pmaxub H, S       // S = max(S, H)
psubsb R, F       // F = F - R
psubsb R, E       // E = E - R
movdqa H, N       // N = H
psubsb Q, H       // H = H - Q
pmaxub H, E       // E = max(H, E)
pmaxub H, F       // F = max(H, F)
```
of four. This procedure ensures that a new database sequence will only begin at the beginning of a new block of cells. This padding increases the total number of cells a little bit. However, it allows these checks to be only carried out on every fourth residue, which increases the global solution's performance.

Despite all these optimizations, in order to make the computations fast it is essential that the vectors with the substitution score values can be loaded as quick as possible. As it was previously stated, each score vector corresponds to the score of a single query residue against sixteen residues from different database sequences. Thus, a kind of temporary score profile was also considered.

Temporary score profiles are created from any of the available substitution score matrices along with all database sequence residues by using a series of packed shuffle instructions. These instructions are only available on Intel processors with Supplemental Streaming SIMD Extensions 3 (SSSE3) [23]. On processors without SSSE3, these operations are replaced by a kind of matrix transpose procedure by using a set of unpack instructions, which only introduce a modest speed penalty.

Despite the fact that each SIMD vector has a fixed length of 128 bits (which means that up to sixteen 8-bit units can be considered), the performed computations are initially implemented using only a 7-bit score range. Although an 8-bit range would allow the same number of parallel computations as a 7-bit range, as well as it would also allow a wider score range, it was not used because it is slower.

Either the range from -128 to 127 or the range from 0 to 255 could have been used. There are both signed and unsigned versions of the instructions for parallel computation of the maximum of bytes \( \text{pmaxub} \), \( \text{pmaxsb} \) and for parallel addition and subtraction of bytes \( \text{paddusb} \), \( \text{psubusb} \), \( \text{paddsb} \), \( \text{psubsb} \). However, the \( \text{pmaxsb} \) instruction for the maximum of signed bytes is slower than \( \text{pmaxub} \).

Thus, a 7-bit score range is initially considered for the sixteen alignment score matrices to be computed in parallel using Intel's SIMD instructions. Additions and subtractions are performed using signed and saturated arithmetics, while the maximum operations are carried out on unsigned numbers. All scores are biased by an offset of 128. This will ensure that signed saturated addition and subtraction work well on lower boundaries, as well as the adopted unsigned maximum operation.

In conclusion, the 7-bit score range optimizes the system's global performance. However, it will limit the computed scores to a maximum value of 127 as well. Thus, alternative versions using 16-bit and 63-bit score ranges are also implemented and used every time an overflow situation is detected in the context of a computation running on a lower score range. In particular, the 16-bit version allows up to eight independent parallel computations, while the 63-bit score range considers a single alignment procedure at a time.

Hence, as soon as a potential overflow is detected in the computations being implemented using a narrow score range, the alignment score for that particular database sequence is recalculated using the next wider score range: first 7-bit, then 16-bit and finally 63-bit, if necessary. According to Rognes [19], rather few sequences will reach a score that cannot be represented by a 7-bit score range. Thus, the additional computation time is typically negligible.

The speed of SWIPE was heavily tested on a dual Intel Xeon X5650 six-core processor system, by using a wide range of query lengths. The tool proved to be about six times faster than Farrar's own solution [18], which has been implemented using an heterogeneous multi-core system based on Cell's
architecture [20]. SWIPE achieved performances of over 9 BCUPS for a single thread and up to 106 BCUPS for 24 parallel threads.

4.3 Intermediate-grained Parallelism

With the advent and widespread availability of multicore architectures based on GPPs or on other heterogeneous designs on today’s computers, a new intermediate-grained parallelism level became worth of exploiting. This intermediate level corresponds to thread-level parallelism, which allows an algorithm to run concurrently in the context of several threads or processes.

An implementation of the previously described striped version of the Smith-Waterman algorithm based on the Cell Broadband Engine heterogeneous architecture was proposed by Farrar himself [20]. The solution was based on a multi-threaded approach, where each thread independently handles the alignment of multiple chunks of the query sequence. A chunk corresponds to a small part of the global dynamic matrix, which is gradually filled with values by the algorithm to obtain the alignment maximum score result. Since his implementation was based on Cell's main architecture, only six of the eight SPEs were actually exploited. This means that up to six independent sequence chunks could be aligned in parallel.

This system has its limitations: despite the SPEs being extremely fast and efficient in terms of performance, they are strongly limited in terms of memory access. As it was previously stated, Cell's SPEs can manage only their own independent cache units (local stores). The small size of these local caches (256 KB) ends up imposing a strong limitation in terms of the length of the sub-sequences in each chunk. The adapted Smith-Waterman algorithm implementation running on a single 3.2 GHz Cell Broadband Engine achieved performance levels over 16 BCUPS [20].

By following a similar thread-level approach, other proposals were also presented by making use of stream processing flows implemented on Graphics Processing Units (GPUs) [11][21]. A GPU consists of a very specialized architecture which was specifically designed to rapidly manipulate and alter pixel data elements, in order to accelerate the processing and the rendering of graphical images in a frame buffer intended for output display. However, their highly parallel structure make them extremely effective in the context of other generic algorithms, where several data blocks may be processed in parallel.

Thus, the general idea behind a GPU implementation of a dynamic programming alignment algorithm is the same as the previously described for the Cell Broadband Engine architecture. GPUs have their own local memory and are able to run several parallel threads, which are grouped in blocks. Considering \( n \) independent processing blocks, up to \( n \) independent sequence chunks may be simultaneously aligned. However, the main difference between these architectures is that GPU’s multiple streaming processors can handle hundreds of independent parallel threads.

Unfortunately, the GPU’s global memory access is extremely limited when compared with its processing capacity. It has been observed that, for larger scale sequence databases, the limited bandwidth that is available to transfer data to the local memory units on the GPU tends to impose a significant bottleneck for solutions based on this architecture [12]. Furthermore, GPUs adopt a Single Program,
Multiple Data (SPMD) architecture model, which means that all the available processing threads must be constantly synchronized, in order to obtain consistent results. Still, processing speeds of more than 3.5 BCUPS are achieved on a workstation running two NVidia GeForce 8800 GPUs [21].

The described intermediate-grained solution can obviously be also applied to a GPP based multicore architecture, where each thread runs in the context of a single and independent core. Once more, each thread handles multiple and independent sequence chunks, which can be aligned in parallel. However, GPPs are not specially built for this kind of processing model. Their hardware specifications make them extremely flexible, but not especially capable in terms of very demanding processing tasks. Furthermore, memory access mechanisms are not optimized for mass multiple access conditions, when considering a large number of independent cores. Thus, despite being simpler and more accessible in terms of development effort, an 8-core architecture typically achieves no more than around 300 MCUPS [24].

From an analysis of these solutions we conclude that the key concept behind this level of parallelism is how data is distributed among the existing threads. As previously stated in section 4.1, coarse-grain parallel solutions typically suggest the possibility to simultaneously align multiple sequences with a single query sequence. Thus, this type of solution usually focus on the distribution of several database chunks across different computing nodes.

As stated in section 4.1, coarse-grained models are generally adopted in the scope of large clusters, where several workstations are connected by an Ethernet network, which supports MPI library based implementations. However, these implementations end up being limited in terms of scalability and flexibility, mainly due to the significant time penalty imposed by communication stages required to transfer data among several distributed nodes.

By integrating thread-level based solutions in such coarse-grained processing models, it is possible to solve part of these problems. From a thread-level point of view, each thread represents a different computing node. Then, by adopting a master/worker approach, the master is able to split the sequence database into different parts and distribute them among several local worker threads.

Local threads share their code and most of their addressing space. Hence, it is even possible to eliminate the master. In such case, each thread is allowed to access its database chunk directly in the main memory, thus avoiding any communication stages. However, this approach is only possible if a centralized shared-memory architecture is considered. T. Rognes followed this approach in his solution (SWIPE), which implements this exact thread-level model on a GPP based multi-core architecture [19].

In both cases, data communication becomes local. Thus, the time penalty imposed by the required communication stages in order to transfer the data among the several available nodes is significantly reduced. Furthermore, this model typically changes the way data is managed, not the way the algorithm is able to perform the alignment itself, which greatly simplifies its implementation.

Despite this set of improvements this approach keeps part of its original problems in terms of scalability and flexibility. The model focus its attention on the distribution of the several database chunks across different computing nodes, which means that each independent alignment will strongly depend on the size of the query sequence. Thus, the speed is gradually reduced for very long query sequences [19].
The way how database chunks are distributed across different nodes may also influence the results. SWIPE [19] estimates the size of each database chunk by following equation 4.1, where seqcount represents the total number of processing sequences, threads represents the total number of available workers and CHUNKDIVISION represents a constant parameter, used for chunk size adjustment. Thus, each thread will process $n$ consecutive sequences from the database. Since the input sequences have very different lengths, this type of distribution will typically raise severe load balancing issues.

$$\text{chunk} = \frac{(\text{seqcount} + \text{CHUNKDIVISION} \times \text{threads})}{\text{CHUNKDIVISION} \times \text{threads}}$$

Furthermore, the model is highly inappropriate for heterogeneous architectures, such as the Cell Broadband Engine, where multiple SPEs have exclusive access to their own and typically small memory units. Since the size of each chunk varies with the size of the sequences which are part of it, in most cases the chunk itself will have to be truncated for several times in order to be completely processed. This process is extremely difficult to implement, as well as highly inefficient in terms of processing.

However, one should recall that the aim of Rognes’ study was to exploit the use of an inter-task approach by using SIMD on ordinary CPUs. Thus, as it was previously stated in section 4.2, and despite the several aspects that may still be improved, SWIPE performs at speeds of up to 106 BCUPS by using 24 threads on a dual Intel Xeon X5650 six-core processor system.

### 4.4 Summary

The alignment of two different biological sequences is an important operation that forms part of multiple bioinformatics applications, such as sequence database searching, multiple sequence alignment, genome assembly or short-read mapping. When this operation needs to be computed repeatedly, for example when searching a sequence database, the computation time becomes substantial. Thus, several approaches have been presented in order to reduce the total amount of time needed to accomplish that task. These implementations use several different levels of parallelism.

Coarse-grain refers to parallelism achieved at a very high computing level. It is typically related to data-level parallelism, which basically focus on distributing data across different parallel computing nodes. This approach is generally adopted in the scope of large clusters.

Solutions based on this implementation end up having good results [11]. However, they often present a difficult tradeoff, due to the significant time penalty imposed by communication stages required to transfer data among several distributed nodes [12]. Thus, data-level parallelism is typically used on top of other parallel solutions [11][12].

Fine-grained parallelism refers to a set of solutions that explore parallelism at a very low computational level: the processor’s instruction level. With the inclusion of SIMD vector instructions in most processor architectures, several parallel sequence alignment implementations were proposed. These solutions are typically based on current GPP multicore architectures.

This type of parallelism mostly concentrates on parallelization within a single alignment, which is
known as *intra-task* parallelization. The opposite approach is know as *inter-task* parallelization, where multiple database sequences are processed simultaneously. The main advantage of an *inter-task* parallelization approach, is that it simply avoids all data dependencies within the alignment matrix [22]. This approach doesn’t seem to have been explored much until T. Rognes presented his latest work [19].

The described model has been implemented in a tool called SWIPE. The tool was heavily tested using a wide range of query lengths, achieving performances of over 9 BCUPS for a single thread and up to 106 BCUPS using 24 parallel threads.

Finally, with the widespread availability of multicore architectures based on GPPs or other heterogeneous designs on today’s computers, a new intermediate-grained parallelism level became worth of exploiting. This intermediate level corresponds to thread-level parallelism, which allows an algorithm to run concurrently in several threads or processes.

Several sequence alignment implementations were proposed based on the architecture of the Cell Broadband Engine [20]. Following a similar thread-level approach, other proposals were also presented by making use of stream processing flows available on a Graphics Processing Unit (GPU) [11] [21].

Both solutions end up having excellent performance results. However, by analyzing them it is possible to conclude that the key concept behind this level of parallelism is how data is distributed among the existing threads.
Chapter 5

Proposed Parallel Implementation

The aim of the present research is to propose an efficient solution for parallel implementation of several bioinformatic algorithms in the context of heterogeneous multi-core architectures. In order to take full advantage of a dedicated hardware platform, multiple parallelization techniques should be considered in a single software solution. Thus, several independent parallelization levels should be exploited. However, it is difficult to combine multiple parallel approaches in a single solution.

As previously stated in section 4.2, fine-grained parallelism mostly concentrates on parallelization within a single alignment, which is known as intra-task parallelization. The opposite approach is known as inter-task parallelization, where multiple database sequences are processed simultaneously. This technique is typically exploited at a very high computing level, being used on top of other finer parallel based solutions [11][12].

This Chapter presents a novel parallel model which exploits the exactly opposite processing approach. Firstly, a thread-level approach is presented, which will implement an intra-task processing model, based on Wozniak’s anti-diagonal approach [16].

Secondly, a fine-grained parallelization approach where multiple database sequences are processed simultaneously is proposed. This model will basically adapt SWIPE’s highly efficient SIMD core instructions [19] to the described thread-level parallelization model.

As a consequence, both intra-task and inter-task parallelization techniques will be considered in the context of a single parallel solution.

5.1 Thread-level Solution

State of the art commodity CPUs contain several processors on a single die, making it possible to run multiple parallel processes on a single machine. Each processor has the ability to run its own independent program, or more specifically, its corresponding process.

However, it is possible that all these processes belong to the execution of a single program. In this case, the term thread is used to describe processes which are able to operate in the context of a single program. Thus, in order to fully take advantage of an MIMD multiprocessor architecture composed by
processors or cores, at least \( n \) parallel threads should be considered [1].

This type of parallel approach offers two major advantages over any other [5]: flexibility and huge cost-performance gain, as described in section 2.2. However, independent threads are usually specified by the programmer, in the application code, and then recognized by the compiler, which means that those advantages are strongly dependent on the parallelization model.

Coarse-grained parallel solutions of sequence alignment algorithms typically suggest the possibility to simultaneously align multiple sequences with a single query sequence, by simply distributing several database chunks across different computing nodes. If a generic multicore architecture is considered, it is possible to adapt this model to a thread-level based solution, by considering each one of the available cores as an independent computing node. Thus, communication between different nodes becomes local, which improves scalability and flexibility. Furthermore, this only changes the way how data is managed, not the way the algorithm is able to perform the alignment itself, which greatly simplifies its implementation.

However, this approach ends up by keeping part of its original problems. Firstly, the model focus its attention on the distribution of several database chunks across different computing nodes, which means that each independent alignment will still depend on the size of the query sequence. Thus, the speed is gradually reduced for very long query sequences [19].

Secondly, the way the database chunks are distributed across different nodes may also influence the results. Solutions based on this model typically make each thread to process \( n \) consecutive sequences directly from the database. Although this represents a very simple solution in terms of implementation, sequences have very different sizes, which means that this type of distribution will typically raise several load balancing issues.

Furthermore, the way the data is distributed is highly critical for heterogeneous architectures, such as the Cell Broadband Engine or GPU based solutions, where multiple high-performance processors have exclusive access to their own small memory units. Since the size of each chunk varies with the size of the sequences which are part of it, in most cases the chunk itself would have to be truncated for several times in order to be completely processed. This process is extremely difficult to implement, as well as highly inefficient in terms of processing.

The primary objective of this work is to study the behavior of several parallelization techniques in the context of a new dedicated heterogeneous architecture. Hence, a different thread-level approach is proposed. This solution focus on distributing the sequences across several different nodes, in the context of a single alignment. By properly adapting a fine-grain based technique to a thread-level solution, some of the previously described issues are expected to be overcomed.

### Wavefront Approach

A. Wozniak presented one of the first parallel implementations for the Smith-Waterman algorithm [16]. As previously stated in section 4.2, his solution was based on a video-oriented instruction set extension named VIS (Visual Instruction Set), which made it possible to process up to four 16-bit integers in par-
allel. By shifting each matrix row one place to the right, a pipeline processing scheme is created, which means that several independent cells may be processed in parallel. Thus, in order to simultaneously process several matrix rows, a parallel minor diagonal approach was adopted (see Figure 5.1).

![Figure 5.1: Wozniak's anti-diagonal processing approach.](image)

The improvement that may be achieved by a thread-level parallel solution strongly depends on the way data is partitioned. Multi-core architectures, containing several dedicated or general purpose processors, typically run a single thread on each of its cores. Direct data sharing between threads running in the context of different physical cores is rarely considered simple or efficient. Thus, completely independent data chunks should be considered in the context of each thread.

By following the previous assumption, multiple independent matrix cells should be able to be simultaneously processed. For such purpose, we have adapted this model to a thread-level parallel approach, where multiple cell blocks are considered, instead of single unit cells.

In this context, each block corresponds to the alignment of two sub-sequences, corresponding to fractions of the original full sequences being processed. Thus, each thread processes an independent block, part of the algorithm's global score matrix.

Basically, the proposed model distributes the alignment itself across several nodes, not the database. This means that the alignment will not directly depend on the size of the query sequence anymore.

However, the sub-sequences must be aligned by a single specific order, due to the dependences beyond diagonal lines, which still remain. As a consequences, the number of available independent blocks will increase until the main anti-diagonal line is reached, and then decreases until the end of both sequences (see Figure 5.2).

![Figure 5.2: Wavefront processing approach.](image)
This describes the key concepts of the wavefront processing approach, which basically results from the adaptation of a typical fine-grained parallelization technique to a thread-level solution. Thus, both solutions end up being extremely close at this point. However, as it was previously stated in section 2.2, threads operate at software level, which represents a quite different environment. Thus, from a development point of view, it is necessary to implement a correct scheduling strategy, which guarantees that each block is processed as soon as it is possible, while simultaneously respecting the correct alignment order.

5.1.2 Master/Worker Model

As it was previously stated, data sharing between threads running in the context of different cores should not be considered simple or efficient to implement. As a consequence, completely independent matrix blocks should considered in the context of each independent thread. However, some dependences still remain, which means that a scheduling strategy is necessary in order to guarantee that every block is only processed after its top and left neighbors.

If each block is considered as a sub-part of the algorithm’s global score matrix, then the algorithm’s general equations can be independently applied to it in the context of each thread. Thus, since multiple diagonal lines are considered, only two frontier dependencies will remain for each block (see Figure 5.3). Instead of considering the entire score matrix structure, which would make the proposed solution extremely inefficient in terms of memory, only two block length arrays are kept in memory for each block.

![Figure 5.3: Frontier dependencies between blocks.](image)

This information has to be correctly managed in order to guarantee the correct alignment procedure. Evidently, this type of management implies handling several data structures. Although these structures are completely independent in the context of the alignment procedure itself, the information they keep should be accessible to every thread.

To keep data independency between each thread, a master/worker model was adopted. By following this approach, it is possible to completely separate the wavefront model logic from the alignment procedure itself.
The *master* process is responsible for the management and delivery of the arrays containing each block’s frontiers. The remaining threads act as *workers*: they receive the left and top frontiers and apply the alignment algorithm. When the alignment is concluded, the arrays corresponding to the right and bottom frontiers are sent back to the *master*. As soon as the *master* receives new information, it has two options: wait until all results are received in the context of a single diagonal line, or automatically send a new block for processing. This is obviously related to the considered scheduling policy.

By adopting the simplest and straightforward wavefront processing approach, threads remain blocked until all anti-diagonal line results are obtained. Thus, some of those threads will remain idle, while a set of new independent blocks can already be processed (see Figure 5.4).

![Figure 5.4: Simple wavefront processing approach issue.](image)

This approach is clearly inefficient. The *master* should be able to trigger any block processing as soon as both dependency frontiers have already been solved by previous *worker* processing. Thus, every single thread should be kept constantly working until the full alignment process is completed (see Figure 5.5).

![Figure 5.5: Modified wavefront processing approach.](image)

To get such a maximum efficiency, we adopted the second scheduling option. However, communication between a single *master* and several *worker* threads may represent a problem. If the information is delivered directly to a specific *worker*, it means that the remaining threads will have to wait for the *master* to finish its data dispatching and block delivery.

By considering an indirect block delivery system, it is possible to avoid such a bottleneck situation introduced by the *master* process. The proposed solution consists in implementing two *First In, First Out* (FIFO) lists through which the *master* is able to communicate asynchronously with all the *workers* (and vice versa). Thus, *messages* containing both frontiers are placed by the master in the first list. A free *worker* accesses to that *message*, processes the alignment, and places the results on the second list. The master accesses the results in the second list and repeats the whole process for each new block (see Figure 5.6).

However, the size of each block needs to be optimized in the context of a specific architecture, in
order to maximize the memory and cache usage efficiency. Furthermore, by optimizing each block’s size, the load is correctly distributed across the several existing workers, thus avoiding serious load balancing issues.

The described solution is extremely versatile in terms of architecture too. Since blocks are completely independent and always have the same size, it is possible to adapt the solution to highly restrictive architectures such as the Cell Broadband Engine, GPU based systems or other heterogeneous multi-core based architectures.

Finally, it is important to reinforce that the model’s logic is completely independent from the code of the alignment procedure itself. Thus, the described approach can be easily adaptable to any other alignment algorithm based on the processing of a dynamic programming matrix.

### 5.1.3 Helper Threads

The previously described master/worker model implements an indirect block delivery system, which depends on two FIFO lists through which the master is able to communicate asynchronously with all the workers.

As it was previously stated, the messages containing both frontiers are placed by the master in the first list. A free worker accesses to that message, processes the alignment, and places the results on the second list. Each of these list accesses involve accessing the global shared-memory, in order to read or write the necessary information. As a consequence, each worker performs two memory accesses per block processing, which ends up representing a tremendous overhead for the system (see Figure 5.7).

Figure 5.6: *Master/Worker* communication model.

Figure 5.7: *Master/Worker* communication model.
pletely disjoint blocks being sequentially processed by a single worker is quite high. In this case, the information kept in cache becomes useless as soon as a new block is fetched.

A possible solution to accelerate the program execution in such situations is provided by adopting helper threads [25]. In order to follow this approach, the programmer basically extracts small operations (often called slices) from the main thread, such that their execution in parallel with the main thread will lead to an improved execution efficiency of the latter. Furthermore, helper threads may also be used to resolve highly unpredictable branches and cache misses before these are required by the main thread.

In this particular case, two slices can be extracted from the original worker thread code segment. The first one is related with the moment in which the worker accesses the first list, in order to get a new processing block. The second refers to the moment in which the worker accesses the global memory, in order to place the obtained results in the second list. Both tasks are performed in the context of a single helper thread, which operates in the same core as its corresponding worker.

![Figure 5.8: Master/Worker communication model.](image)

In order to optimize the cache memory usage, each helper can even be modified in order to look at the master's data structures. Thus, each time a new block is fetched, the helper thread is able to check if the immediately following block is already available. If that is the case, the helper can automatically transfer the missing dependencies while the previous block is still being processed. Then, the processing procedure may continue along another contiguous block, taking full advantage of the values kept in cache from the previous computation (see Figure 5.8).

In conclusion, by following this approach, each worker thread is able to compute a block while the following block to be processed is already being fetched.

### 5.2 Instruction-level Solution

The thread-level model described in section 5.1 focus on the distribution of independent data blocks across multiple worker threads. In order to achieve that, a minor diagonal approach was followed. Thus, each block becomes an independent sub-part of the algorithm's global score matrix. This means that it adopts an intra-task parallelization method, where parallelism occurs within a single alignment procedure.

An approach where parallelization is carried out across multiple database sequences is possible too. This technique is known as inter-task parallelization.
In the past, Alpern et al. [22] described that the main advantage of such an inter-task approach is that it simply avoids all data dependences within the alignment matrix. Thus, it would be also possible to consider several database sequences in the context of a single thread-level block. As a consequence, multiple alignments can be simultaneously processed by each independent thread.

In conclusion, it is possible to exploit both parallelization techniques by considering an inter-task approach in the context of each worker thread. As a consequence, this section presents an inter-task SIMD solution, which extends the previously presented thread-level parallelization model.

The described solution follows T. Rognes’ approach [19], which uses Intel’s SSSE3 instructions in order to simultaneously process multiple database sequences.

5.2.1 SIMD Model Adaptation

As it was previously referred, SWIPE’s thread-level strategy focus on the distribution of several independent database chunks across multiple threads (see Figure 5.9). Thus, the described approach basically changes the way how data is managed, not the way how the algorithm is able to perform the alignment procedure itself, which greatly simplifies this model’s implementation.

![Figure 5.9: SWIPE’s database distribution strategy.](image)

However, by exclusively focusing on the distribution of the several database chunks across different nodes, the model becomes strongly dependent on the size of the query sequence. Each database sequence is partitioned and aligned along the entire query sequence, which means that each SIMD iteration will directly depend on the query length (see Figure 5.10 - A). Thus, the speed is gradually reduced for long query sequences when using multiple threads [19].

The way how database chunks are distributed across several different nodes may influence SWIPE’s overall performance as well. As it was previously described in section 4.3, chunks of database sequences are assigned to the available threads as soon as they are ready for more work. As a conse-
sequence, threads may not exactly process the same number of chunks each. Furthermore, each thread will process a fixed amount of consecutive sequences directly from the database in the context of each chunk. Since sequences may have very different sizes, this type of distribution will typically raise several load balancing issues.

Finally, this model also reveals to be highly inappropriate for heterogeneous systems, such as the Cell Broadband Engine or GPU based architectures, where multiple independent cores have exclusive access to their own and typically small memory units. Since the size of each chunk varies with the size of the sequences which are part of it, in most cases the chunk itself would have to be truncated for several times in order to be completely processed.

Figure 5.10: Multi-sequence SIMD vectorization approaches for the Smith-Waterman algorithm. For simplicity, vectors of only four elements are shown, while sixteen elements would normally be used. (A) SWIPE, which follows a parallel to the query sequence processing approach, described by T. Rognes [19]. (B) Proposed parallel processing approach, which adopts SWIPE’s SIMD model on multiple thread-level processing blocks.

However, despite all the described issues, SWIPE can still achieve extremely high performance levels, reaching up to 9 GCUPS, in the context of a single processor [19]. This is mainly possible due to the SIMD part of the solution, which implements the core procedures of the algorithm itself.

In fact, an in-depth look at these problems reveals that they mostly arise from the absence of an intra-task approach. In order to maximize the efficiency and improve the performance, an overall solution should be as complete as possible. Thus, it is desirable to exploit both intra-task and inter-task parallelism in the context of a single solution.

Hence, the aim of the present research is to propose a novel and even more efficient solution for parallel implementation of several bioinformatic algorithms in the context of heterogeneous multi-core architectures. Thus, the proposed approach is to adapt SWIPE’s highly efficient SIMD core to the
previously described thread-level distribution model. As a consequence, both intra-task and inter-task parallelization techniques will be considered in the context of a single parallel solution.

As it was described in section 5.1, the proposed thread-level approach focuses on the distribution of independent data blocks across multiple worker threads. In order to achieve a total independency between blocks, a minor diagonal approach was followed. Thus, each block becomes an independent sub-part of the algorithm’s global score matrix (see Figure 5.10 - B).

The main advantage of an inter-task approach is that it simply avoids all the data dependences within the alignment matrix. Thus, it is possible to consider several database sequences in the context of a single thread-level block. However, both solutions must be adapted to one another.

First of all, in the presented solution the SWIPE’s core instructions are included in the context of each thread, which means that each worker can handle up to sixteen database sequences at a time. As a consequence, each block now includes up to sixteen independent matrix sub-parts as well.

SWIPE follows a processing approach parallel to the database sequence, which means that the query sequence will remain intact during all the processing procedure. Thus, every column will be processed in the context of a single step. As a consequence, each new processing step will exclusively depend on the previous line. This means that by adopting a parallel minor diagonal approach, each block will only depend on its left and upper frontiers.

In the modified solution, the final scores stored in the $E$ vector are saved in memory, in order to be used in the context of the algorithm’s next processing step. In a same way, in order to adapt Rognes’s solution to this new approach, the final scores in $F$ should be also kept in memory.

All processing vectors, except $N$, are correctly initialized prior to the actual processing code segment. Thus, if the $F$ vector is initialized with the previously computed scores, the values will be then correctly computed by each worker, in the context of an independent block. The same applies to $E$, which already happens in the original SIMD implementation.

In order to achieve that, some parts of the original code segment hand coded by Rognes in inline assembler had to be modified. Some new lines also had to be added in order to store the correct values in memory. However, all these modifications took place outside of the algorithm’s critical code section. Hence, no major overhead is introduced by this new set of instructions.

In the original solution, the residues are directly fed into sixteen independent channels. Then, when the first of these database sequences ends, the first residue of the next database sequence is loaded into the empty channel. The concept of channel does not exist in this new solution. In fact, in the context of a wavefront processing approach, blocks must be aligned by a single specific order. This happens because each single block corresponds to the alignment of several database sub-sequences with a single query sub-sequence, both corresponding to parts of the original full sequences being processed.

As a consequence, each sequence is now considered in the context of a batch, which can support up to sixteen different database sequences. Thus, each batch will depend on the alignment of its longest database sequence. Since the database sequences will typically have very different sizes, the system may end up by spending a greater amount of time on a single larger alignment procedure. This means that up to fifteen processing slots may remain unused for several processing blocks.
This problem could be solved by sorting the whole database by length, which would homogenize the sequences’ size in each batch. However, the sorting procedure would represent an even greater overhead in the overall solution. Then, in order to keep reading database sequences in the same order as they are found in the original database file and still improve the system’s overall efficiency, several SIMD iterations were included in each thread-level processing block. By following this approach, it is possible to optimize the total amount of blocks per batch, thus minimizing the total number of blocks containing fewer database sequences than possible.

Just like the original SWIPE implementation, the presented solution also uses several bit score ranges. In Rogne’s solution, as soon as a potential overflow is detected, the alignment score for that particular database sequence is recalculated by using the next wider score range available. Thus, if a single potential overflow is detected, the corresponding database sequence is processed alone, which means that up to seven processing slots would remain empty during the necessary computations.

In order to avoid that, the new solution keeps a list, which contains the sequence ID of each detected potential overflow. Then, when the global database processing is over, the alignment for each sequence in that list is calculated again by using the next wider score range. Thus, several profile sequences can be simultaneously aligned in the context of a new batch, by considering a different and wider score range.

Hence, although there are some major differences, SWIPE’s core instructions remain basically the same. The developed work on this area focus on the adaptation of Rogne’s instruction-level solution to a completely new thread-level parallel approach.

In conclusion, it is possible to take full advantage of an highly optimized implementation in the context of a whole new thread-level approach. The result is a new and extremely flexible solution, which exploits completely different techniques and different parallelization levels, in order to achieve the system’s maximum performance.

5.3 Summary

The primary objective of this work is to study the behavior of several parallelization techniques in the context of a new dedicated heterogeneous architecture.

Intermediate-grained parallel approaches offer two major advantages over any other parallelization approach [5]: flexibility and huge cost-performance gain. Hence, a novel thread-level parallel approach is proposed, which focus on distributing the sequences across multiple processing nodes, in the context of a single alignment.

The described model follows the A. Wozniak’s minor diagonal processing approach [16]. However, it extends it to larger blocks, which can be simultaneously processed in the context of multiple and independent threads. Thus, since multiple diagonal lines of blocks are considered, only two frontier dependencies remain for each block.

However, the sub-sequences must be aligned by a single specific order, due to the dependences beyond diagonal lines, which still remain. As a consequences, it is necessary to implement a correct
scheduling strategy, which guarantees that each block is processed as soon as it is possible, while simultaneously respecting the correct alignment order. Hence, a master/worker approach is followed, which makes it possible to completely separate the model’s logic from the alignment procedure itself.

Furthermore, the model can be extended by implementing helper threads in the context of each independent processing node. By following this approach, each worker thread is able to compute a block while another one is already being fetched by its corresponding helper.

The described model basically focus on the distribution of independent data blocks across multiple worker threads. Each block corresponds to an independent sub-part of the algorithm’s global score matrix, which means that an intra-task parallelization method is adopted, where parallelism occurs within a single alignment procedure.

An approach where parallelization is carried out across multiple database sequences is possible too. This technique is known as inter-task parallelization.

In order to maximize the efficiency and improve the performance, an overall parallel solution should be as complete as possible. Thus, it is desirable to exploit both intra-task and inter-task parallelism in the context of a single solution. Hence, an inter-task SIMD solution is also proposed, which extends the previously described thread-level parallelization model.

The proposed approach adapts SWIPE’s highly efficient SIMD processing core [19] to the previously described thread-level distribution model. SWIPE’s core instructions are included in the context of each thread, which means that each worker can handle up to sixteen database sequences at a time. As a consequence, each block now includes up to sixteen independent matrix sub-parts as well.

In conclusion, by combining both parallel approaches, it is possible to exploit both intra-task and inter-task parallelization techniques in the context of a single parallel solution.
Chapter 6

Implementation

In the previous sections it was proposed a set of conceptual parallelization models, which can be applied to several sequence alignment algorithms.

As it was previously stated, the aim of this study is to describe a general solution for parallel implementation of several bioinformatic algorithms in the context of an heterogeneous multi-core architecture. This specialized architecture is being developed in the scope of the HELIX project, and it will basically depend on multiple dedicated cores. The architecture of each core will be specifically designed for parallel biological sequence alignment, based on the previously described parallel approaches. Thus, the proposed set of parallel models should be formalized and evaluated in order to conceptually design this particular dedicated architecture.

Hence, this chapter presents a set of important implementation details for the previously described models. The described models basically depend on the exchange of information between a master thread and several worker threads. This exchange of information has to be correctly managed, in order to guarantee the correct alignment process.

This type of management implies handling several data structures. These structures should be correctly formalized and described as part of the proposed solutions. Their correct management and implementation are fundamental in order to maximize the final system's efficiency.

Heterogeneous multi-core architectures are typically composed by a single GPP and multiple dedicated processors. These systems typically adopt a distributed-memory architecture. In this case, each dedicated processor manages its own independent cache, while the GPP is able to access and manage the global memory system.

As it was previously referred, the described solution depends on the exchange of information between a master thread and several worker threads. This information will basically correspond to a memory block, which is considered to be a sub-part of the algorithm's global score matrices. Since multiple anti-diagonal lines are considered, two frontier dependencies will have to be remained for each alignment, in the context of each block.

Thus, the left and top frontiers, as well as the corresponding sub-sequences must be transferred by each worker thread from the global memory to its local cache unit. This operation describes a well
defined memory access model, which is presented in the context of this chapter.

Finally, the described parallel processing models have been implemented using a preliminary version of the HELIX project architecture. This architecture conceptually follows a master/worker approach, where a single master processor manages a set of independent dedicated processors, which act as workers.

To study the several aspects of the proposed solutions, a set of GPP based multi-core computers had to be considered, in order to simulate the described preliminary model of the architecture. Thus, a software level platform was created in order to achieve the correct computational environment conditions on a GPP based computer.

The proposed fine-grained parallelization model uses SIMD vector instructions, which are available in most GPP architectures. Thus, the proposed simulating framework will depend on the use of several instruction set extensions. As a consequence, it is already expected to include SIMD processing units in the architecture of each worker in the HELIX platform, in order to implement the described parallel model using the correct hardware specifications.

Thus, the implementation of the overall solution in this simulating framework makes it possible to study and support the definition of a wide set of parameters in the scope of the HELIX processor architecture (e.g. performance according to the total number of available dedicated processors, considered memory block size, etc.).

6.1 Data Structures

As it was previously stated in section 5.1.2, the proposed thread-level model follows a master/worker processing approach. By following this approach, the master thread is able to trigger any block processing as soon as their dependences have already been solved by a previous worker.

Furthermore, it is also possible to avoid any bottleneck situation introduced by the master thread, by considering an indirect block delivery system (see Figure 5.6). In this case, the information is not delivered directly to a specific worker, which means that the master thread is able to communicate asynchronously with all the workers and vice versa.

This type of management implies handling several data structures. It is the master who is responsible for the management and delivery of the arrays containing each block’s frontiers. Thus, in order to correctly handle the available blocks, the master thread manages two different lists and a table, containing all the available frontier dependencies between the blocks.

For such purpose, the solution uses two FIFO lists through which the master is able to asynchronously communicate with all the workers and vice versa. The messages are placed by the master in the first list. A free worker accesses that message, processes the alignment, and places the results on the second list. The master accesses the results in the second list and repeats the whole process for each new block (see Figure 5.6).

Each message represents an independent block, which is considered to be as a sub-part of the algorithm’s global score matrices being processed. Since we are considering multiple anti-diagonal
lines, two frontier dependencies (left and top) remain for each alignment, in the context of each block. Thus, each message will contain pointers (addresses) for two large memory arrays, containing the correct values for \( E \) and \( F \) (see Figure 6.1). These fields are used by the master, in order to specify the address of the values to be processed by any worker thread, or used by any worker, to return the results back to the master.

![Figure 6.1: Message data structure.](image)

These arrays have a fixed size, which determines the dimension of each processing block. However, the type of data that they keep will vary. As it was explained in chapter 5, the computations are initially performed using a 7-bit score range. Hence, the arrays initially hold several byte units. Bytes are implemented using the \texttt{unsigned char} data type. In this case, up to sixteen database sequences are processed in the context of a single algorithm iteration.

When a potential overflow condition is detected in computations using a 7-bit score range, the alignment scores for those particular database sequences are recalculated using a 16-bit score range. In this case, the arrays hold several word units, which are implemented using the \texttt{unsigned short} data type. This means that up to eight different database sequences can be simultaneously processed in the context of a single processing iteration.

Finally, if no instruction-level parallelism is considered, which means that a single cell comparison is processed at a time, each array holds multiple \texttt{long} type elements. The type of data being used is indicated to the workers by a specific field in the message (see Figure 6.1).

In all cases, the exchanged messages include pointers for both the query sequence and the database sequences being aligned (see Figure 6.1). The message also includes their corresponding offsets. This information allows each worker thread to correctly access all the necessary sequences.

If several database sequences are considered in the context of a single block, a mask is used in order to keep track of which alignment slots are still active (see Figure 6.1). Hence, each message includes a small array where each position represents an independent processing slot. All its values are initially set to \texttt{TRUE}, which means that all the slots are available and should be processed in the context of the alignment code segment. Each time a database sequence ends, its corresponding slot value is set to \texttt{FALSE}. Thereafter, those slots should be ignored by the alignment code segment, in order to avoid
incorrect score updates.

Partial score values are computed in the context of each independent processing block. These values are sent by each worker to the master thread, which evaluates them in order to keep the alignment’s maximum scores. Thus, the exchanged messages must also include an array containing the partial scores computed in the context of their particular blocks (see Figure 6.1).

Finally, each message also includes a tableID, which basically defines the scope of each block (see Figure 6.1). As it was previously stated, the master thread is responsible for the management of two different lists and a table. This table consists of an array containing multiple structures referred to as border tables.

A border table consists of an array structure keeping a node for each matrix block. Each node contains two FIFO lists. These lists basically keep track of the available frontier dependencies during the entire alignment process (see Figure 6.2).

![Border Table Data Structure](image)

Figure 6.2: Border table data structure.

Each frontier dependency is calculated during the processing of the neighboring block belonging to the previous anti-diagonal line. Thus, each time a message reaches the master thread, a new set of results is pushed into two different nodes, corresponding to the block's bottom and right neighbors. Whenever this happens, both node’s lists are checked. If none of them is empty, it means that both frontier dependencies are available. Then, the master is able to create a new message and place it in the outgoing list, where it will wait for processing.

These structures basically implement the logic behind the solution described in sections 5.1.1 and 5.1.2. By following this approach, the model’s logic becomes completely independent from the code of the alignment itself, which makes this an extremely flexible and robust solution.

As it was previously stated, several border table structures are considered in a single execution. This ensures that all the available processing resources start to consume the available information evenly.

In a wavefront approach, the number of available independent blocks increases until the full diagonal
line is reached, and then decreases until the end of the alignment process (see Figure 5.2). Thus, the system’s maximum performance is not immediately reached if a single border table structure is considered.

In order to avoid such a bottleneck situation, the number of processing structures should be equal to the number of available worker threads in the system. This number usually matches the number of available cores, which means that we end up by having a single border table structure per available core. As a consequence, several score matrices are simultaneously processed (see Figure 6.3). Furthermore, it is important to reinforce that each one of these matrices can support up to sixteen parallel alignments.

![Figure 6.3: Global table data structure.](image)

Each time a complete alignment procedure reaches its end, the default values in its border table are totally restored. Then, a new set of database sequences is loaded and the whole process is repeated using the same structure. By following this approach, it is possible to avoid constant memory allocation of new border table structures, which would make the system run slower.

Finally, each time a potential overflow is detected, an ID tag is added to a list. This list contains the sequence ID of all the overflow cases detected during a database alignment process. When the global database processing is over, the alignment for each sequence in that list is recalculated by using the next wider score range. Thus, several profile sequences can be simultaneously aligned, by using a different and wider score range. This list is managed by the master thread, just like all the previously described data structures.

### 6.2 Memory Access Model

The proposed solution is based on a well defined memory access model, which was already studied and established. Figure 6.4 presents a typical block processing operation in the context of the thread-level proposed solution.

Each worker thread has to perform two independent global memory accesses per block processing operation. Let $T_r$ and $T_w$ represent the time it takes to perform a single direct read and write access operation to the global memory, respectively.

First, both frontier dependencies (left and top) have to be read from the global memory (see Figure
6.4, on the left). This operation will occur in the context of every block, so the following expression can be considered, which defines the total amount of time that is spent in memory accesses to perform this particular operation:

\[
\text{bound}_{\text{read}} = T_r \times 2 \times \text{blockSize}
\] (6.1)

Each block processing operation corresponds to the alignment of two independent sub-sequences extracted from the complete sequences being processed. Thus, both sub-sequences must also be read from memory (see Figure 6.4, on the left). This operation will also occur in the context of each processed block. Then, the following expression can be considered:

\[
\text{seq}_{\text{read}} = T_r \times 2 \times \text{blockSize}
\] (6.2)

After the block is processed, the resulting boundary scores (right and bottom) must be written back to global memory structures (see Figure 6.4, on the right). Once more, this access will occur in the context of each processed block. Thus, the following expression can be considered, which defines the total amount of time that is spent writing data to global memory:

\[
\text{bound}_{\text{write}} = T_w \times 2 \times \text{blockSize}
\] (6.3)

Hence, the model directly depends on the size of each processing block. Two different memory arrays with the same size are always accessed, which means that square blocks are being considered. However, both sequences can have very different sizes. Thus, it is correct to consider the following expression:

\[
\text{blockSize} = \text{queryBlockSize} + \text{profileBlockSize}
\] (6.4)

As a consequence, the previously described expressions 6.1, 6.2 and 6.3 respectively become:

\[
\text{bound}_{\text{read}} = T_r \times (\text{queryBlockSize} + \text{profileBlockSize})
\] (6.5)
seq\_read = T_r \times (\text{queryBlockSize} + \text{profileBlockSize}) \tag{6.6}

bound\_write = T_w \times (\text{queryBlockSize} + \text{profileBlockSize}) \tag{6.7}

If several database sequences are considered in a single processing block, then the situation becomes slightly different. In this case, the \textit{profileBlockSize} variable is related to a set of database sequences, not to a single sequence, as before. Thus, several processing slots are now available and multiple residues can be simultaneously processed in a single SIMD operation cycle. Furthermore, in order to optimize the global memory access, several processing iterations are considered in a single block. Then, the following expression should be considered, where \textit{processingSlots} represents the total number of processing sequences per block, \textit{SIMDBlockSize} represents the number of residues per database sequence being simultaneously processed by a single SIMD operation and \textit{workerIterations} represents the total number of worker SIMD iterations per processing block:

\textit{profileBlockSize} = \#\text{processingSlots} \times \text{SIMDBlockSize} \times \#\text{workerIterations} \tag{6.8}

Finally, the following expression defines the total amount of time that is spent in memory accesses, as a function of specific \(T_r\) and \(T_w\) values.

\textit{total} = \textit{seq\_read} + \textit{bound\_read} + \textit{bound\_write} \tag{6.9}

As it can be seen, the described model strongly depends on the size of the processing block. One of the objectives of this work is to study the previously described model in order to optimize this value. With such information, we are able to develop a computer architecture completely optimized for the context of this particular application.

## 6.3 Simulator Architecture

The proposed solution was mainly implemented using the C programming language. The POSIX threads library (\textit{pthreads}) was used in addition to the common language libraries, in order to handle multiple thread management operations. Some parts of the solution were coded using inline x86 assembly language and part of this code uses SSE2 and SSSE3 vector instructions.

Hence, the simulating framework that is now proposed makes it possible to study and optimize multiple aspects of this solution in the context of a particular heterogeneous multi-core architecture. However, in order to accomplish this purpose, an accurate simulation environment must be considered.

As it was previously stated, the described parallel processing model is being developed in the context of a preliminary version of the HELIX project architecture. This architecture conceptually implements a \textit{master/worker} model, where a single \textit{master} processor manages a set of independent and dedicated processors, which will act as \textit{workers}. Each \textit{worker} processor exclusively processes data in its own
local memory unit. Thus, a global memory system will be shared by a single bus being accessed by all the existing worker processors, which will be permanently required for multiple data exchanges between the global shared memory and the worker’s local independent cache units.

In order to study the several aspects of the proposed solution, a set of GPP based multi-core architectures is considered to simulate the described HELIX architecture model. Thus, if a \( n \)-core GPP architecture is considered, the \( n \) available cores will act as the previously described dedicated worker processors. To ensure that each worker thread remains attached to a single specific core, CPU pinning was enabled.

Processor affinity or CPU pinning techniques take advantage of the fact that some remains of a process or thread are fixed to the processor’s state (more particularly, to its cache) since the last time the process ran [26]. By scheduling each worker to always run in the same processor results in more efficiency, since it allows to minimize part of the performance-degrading situations, such as cache misses. Furthermore, this also better approximates to the original scenario, where each worker represents a real physical core, with exclusive access to its own cache and memory unit. Each independent processing block should be completely computed by a single specific core. In fact, only by following this approach it is possible to predict the real performance of each independent worker, when using a generic GPP multi-core architecture.

In contrast, the master thread does not need to be attached to any specific physical core. Each available core will actually interleave its operations with the master’s activity. Thus, this particular thread will be executed by all the available cores in the system. Since the load introduced by the master in the system is typically very small, this does not represent any problem.

Accordingly, only the several available worker threads are attached to multiple independent cores. However, exclusive global memory access also has to be considered by each of them. Thus, several mutual exclusion tools from the operating system were used. These tools were used in order to simulate certain hardware characteristics, which will be adopted by the original HELIX architecture.

On the other hand, GPPs implement a considerably different memory model, when compared with the previously described one, in the HELIX architecture. In a GPP based architecture, the global memory is shared by all the available cores, which means that each core is able to directly access it and manipulate it. Thus, in order to correctly simulate the previously described HELIX memory model, \texttt{memcpy} operations are performed by each thread every time some data is required by a worker core. Furthermore, a \textit{mutex} mechanism is used in order to ensure that a single thread is able to access the global memory data structures at a time, which simulates a common global memory access bus with arbitration (see Figure 6.5).

Hence, in order to simulate the HELIX memory environment in the context of a GPP architecture, each thread performs its own independent memory access operations at a time. Thus, each worker thread is responsible for reading its corresponding sub-sequences, as well as the two frontier dependencies, on which its values will depend. After the processing, the worker threads must write the resulting dependencies back to the main memory. To correctly implement these operations, some \textit{master/worker} synchronization mechanisms are also necessary.
As it was previously stated in section 5.1.2, the described solution considers two FIFO lists through which the master is able to asynchronously communicate with all the workers and vice versa. This scenario describes a typical producer/consumer problem, where one thread (the producer) generates data items and another thread (the consumer) receives those items and uses them [27]. In this particular case, both types of thread will act as producers and consumers. Thus, two different counting semaphores are used, in order to synchronize the access to both lists.

The first semaphore ensures that each worker waits until new messages are placed by the master in the first list. The second one ensures that the master thread waits until new results are placed in the second list by any of the available workers. As a consequence, each time a new message is placed in one of the lists, its corresponding semaphore should be posted. Furthermore, two different mutex variables are used, in order to ensure that each list is accessed by a single thread at a time.

Again, it is important to note that the original HELIX architecture will not have an operating system installed. The architecture will basically implement a shared-memory system with exclusive and atomic access policies defined by an arbiter [28]. Thus, each one of the previously described synchronization mechanisms will end up by being implemented on hardware.

Figures 6.6 and 6.7 present the previously described execution cycles for the master and worker threads, respectively.

It is worth noting that the simulating framework uses a special message containing the ID tag KILL_THREAD in order to stop each worker’s execution as soon as the global alignment procedure is over. These messages are sent by the master, which corresponds to the main process. In the HELIX architecture, each worker directly corresponds to a physical core, which means that its code will run indefinitely. However, in this case it is obviously useful to stop the execution at the end of each alignment, in order to evaluate the system’s overall performance in numerous situations.

The described simulation framework was also extended in order to support helper threads, which implied a small set of modifications. For such purpose, the code which is responsible for memory
access (memcpy) was implemented in the scope of a different thread. This thread is created by each worker. Thus, each worker will have its corresponding helper.

As it was previously stated, each worker corresponds to a real physical core in the HELIX architecture. Hence, each helper should be pinned to its corresponding worker core. As a consequence, both threads are scheduled on two hyper-threads running on the same core. The simultaneous execution of these two threads on the same core was achieved by means of the hyper-threading feature, offered by the considered GPP architecture.

Hyper-threading enables a single processor core to be used for two or more concurrent executions with just a little extra hardware. For each processor core that is physically present, the operating system is able to address two virtual or logical cores, and share the workload between them when possible. If two hyper-threads execute completely different code (i.e., the two threads are treated just like separate processors by the operating system, in order to execute separate processes) the cache size will be cut in half, which results in a significant increase in cache misses [29].

However, in this particular case, both threads will cooperate. Furthermore, the helper will remain idle part of the time, avoiding unnecessary memory accesses that would pollute the shared memory unit. As a consequence, the arithmetic operations that are performed by the worker thread and the memory load operations performed by the helper do complement each other. Hence, the resource collisions are minimal, which causes a synergistic effect [29]. In conclusion, hyper-threads, which are often not useful.
due to cache pollution, shine in these type of situations and should be taken advantage of.

Hence, as it was previously stated in section 5.1.3, those helpers will access the master's structures, in order to anticipate the processing of a particular block. As a consequence, a mutex had to be used, in order to avoid concurrent accesses to any of the available border table structures.

To conclude, the HELIX architecture can thus be simulated on any GPP based platform. By following the previously described approach, a wide set of parameters (e.g. the total number of available worker processors, the considered block size, etc.), can be extensively studied in order to optimize the achievable performance of this particular solution. The size of each worker’s cache unit can also be accurately optimized, since its dimension will be closely related to the amount of data that is transferred in each thread operation.

In conclusion, the implementation of the previously described model in the proposed simulating framework will make it possible to study and support the definition of three particular parameters, in the scope of the HELIX processor architecture:

- Total number of available dedicated processors (workers);
- Each data block’s size, which should be optimized in order to improve multiple processor performance;
- Required memory space, both in the global memory system and in each one of the multiple independent cache units.
6.4 Summary

This chapter presents a set of important implementation details for the previously described parallelization models.

The described models basically depend on the exchange of information between a master thread and several worker threads. This exchange of information has to be correctly managed, in order to guarantee the correct alignment process.

This type of management implies handling several data structures. It is the master who is responsible for the management and delivery of the arrays containing each block's frontiers. Thus, in order to correctly handle the available blocks, the master thread manages two different lists and a set of border tables, containing all the available frontier dependencies between the blocks.

Furthermore, the master keeps a list containing the sequence ID of all the overflow cases detected during a database alignment process. When the global database processing is over, the alignment for each sequence in that list is recalculated by using the next wider score range.

The proposed solution is based on a well defined memory access model. Each worker thread has to perform two independent global memory accesses per block processing operation, which correspond to read the top and left frontier dependencies and write the resulting bottom and right frontier arrays back in the global memory system. Thus, the described model will strongly depend on the size of each processing block.

Finally, a simulating framework is proposed, making it possible to study and optimize multiple aspects of this solution in the context of a particular heterogeneous multi-core architecture. However, in order to accomplish this purpose, an accurate simulation environment must be considered.

Hence, a set of GPP based multi-core architectures is considered to simulate the described HELIX architecture model. In this case, if a \( n \)-core GPP architecture is considered, the \( n \) available cores will act as the dedicated worker processors.

However, GPPs implement a considerably different memory model, when compared with the previously described one, in the HELIX architecture. As a consequence, exclusive global memory access also has to be considered by each independent core. Thus, several mutual exclusion tools from the operating system were used, in order to simulate certain hardware characteristics, which will be adopted by the original HELIX architecture.

The simulating framework can also be extended, in order to support helper threads. In this case, threads are scheduled on two hyper-threads running on the same core.

The HELIX architecture can thus be simulated on any GPP based platform. By following the described approach, a wide set of parameters (e.g. the total number of available worker processors, the considered block size, etc.), can be extensively studied, in order to optimize the achievable performance of the proposed parallel solution.
Chapter 7

Results

The performance of the conceived parallel implementation of the Smith-Waterman algorithm was extensively evaluated using different processing parameters. This chapter presents the evaluation methodology that was followed in order to correctly study the results obtained by the described solution.

As it was previously referred, the presented model simulates the operation of an heterogeneous computing environment in any generic GPP based computer. Thus, in order to correctly evaluate the performance of the proposed solution, two different multi-core platforms were considered:

- 4-Core Intel i7 950 processor running at 3.07 GHz with six 2 GB DDR3 RAM modules running at 2.0 GHz.
- Dell PowerEdge R810 platform, equipped with four 8-Core Intel Xeon E7-4830 processors running at 2.13 GHz and with thirty two 8 GB DDR3 RAM modules running at 1066 MHz. This system adopts the Intel's Smart Cache technology [30], which means that each core is equipped with an individual 24 MB L2 cache. Each of these cores is able to handle up to eight concurrent threads, making this processor capable of executing up to 32 SMP threads.

The code was compiled for 64-bit Linux operating system using the Intel C compiler version 12.1. The source code corresponding to Rogne’s SWIPE software was downloaded from the author’s website and compiled using Intel’s compiler as well, as specified in the supplied Makefile [19].

The Zv9 zebrafish genome sequence was used in the tests [2]. Sequences with different lengths were extracted from this sequence and used as queries. These sequences have lengths ranging from 16 to 524 288 amino acid residues, which made it possible to study the solution’s overall performance against multiple different query sizes.

For the database sequences, the *nr.07* database, released on 31 July 2012, was used in all the tests [2]. The *nr/int* database corresponds to the largest database available through NCBI BLAST. It generally includes all GenBank, PDB (Protein Data Bank) and SwissProt sequences. This specific version of the database consists of 455 432 sequences with a total of 159 236 774 amino acid residues. All these sequences are aligned against a single query sequence in the context of each test.

As it was referred before, due to its universal application in most toolboxes that guarantee an optimal
alignment, the Smith-Waterman algorithm was implemented in the context of the presented framework. The described solution supports multiple score matrices and gap penalties. However, all the tests were performed with the BLOSUM62 score matrix and gap open and extension penalties of 11 and 1, respectively, which correspond to the Basic Local Alignment Search Tool (BLAST) default values [31].

The results obtained with the proposed solution were compared with the values obtained using SWIPE [19]. Each alignment was performed using both tools and the obtained results were compared in order to verify that they were the same.

The speedup ranges provided by the proposed solution were extensively evaluated. Both sequential and parallel versions of the two algorithms under comparison were considered under the same computational circumstances, in order to collect the corresponding processing time values.

The obtained results were evaluated both in terms of the achieved performance, measured in Cell Updates Per Second (CUPS), and overall speedup. The following expressions were used, where \( a \) and \( b \) represent two generic sequences being aligned, and \( t \) represents the alignment time:

\[
CUPS = \frac{\text{len}_a \times \text{len}_b}{t}
\]

\[
\text{speedup} = \frac{t_{\text{sequential}}}{t_{\text{parallel}}}
\]

The next sections present a detailed discussion of the results obtained by the previously described parallel models in the context of several computational environments.

### 7.1 Thread-level Solution Analysis

This section presents the results that were obtained by adopting the proposed thread-level model, which follows an *intra-task* parallel approach, where each alignment is divided into multiple parts. Each part corresponds to an independent processing block, which is considered as a sub-part of the algorithm’s global score matrix.

#### 7.1.1 Evaluation of the optimal block size

Figures 7.1 and 7.2 present the variation of the achieved speedup to align a sequence composed by 4,096 amino acid residues against the *nr.07* database, when considering multiple processing block sizes. Dell PowerEdge R810 processing platform was used for this test, so a total number of 32 worker threads was used.

The obtained values clearly show that the achieved performance strongly depends on the considered block size. This applies both to the query sequence and to the profile sequence being aligned. In this particular architecture, 2048 x 1024 processing blocks should be considered, in order to optimize the system’s performance.

The same tests were conducted using the Intel 4-Core architecture, by considering the same query
Figure 7.1: Performance dependency on the query block size using 32 worker threads and a profile block size of 1,024 amino acid residues.

Figure 7.2: Performance dependency on the profile block size using 32 worker threads and a query block size of 2,048 amino acid residues.
sequence and database file. However, only 4 worker threads were used. Figures 7.3 and 7.4 present the obtained results.

![Figure 7.3: Performance dependency on the query block size using 4 worker threads and a profile block size of 2 048 amino acid residues.](image)

The obtained results show that the block's optimal size depends on the total number of worker threads being used. A closer look shows that the optimal query block size remained the same for this setup in both architectures. However, the number of database residues per block is doubled. This happens because the number of worker threads trying to simultaneously access the global memory is considerably smaller. Thus, the risk of collision is considerably lower, making it better to reduce the total number of memory accesses by loading greater amounts of data at a time.

### 7.1.2 Performance gain (Speedup)

By considering the obtained block's optimal size, the system was then tested by varying the number of worker threads. A query sequence composed by 8 192 amino acid residues was used in these tests. The obtained results show that the algorithm scales almost linearly with the number of workers in both architectures (see Figures 7.5 and 7.6).
Figure 7.5: Variation of the obtained speedup with the number of workers (32-Core Architecture).

Figure 7.6: Variation of the obtained speedup with the number of workers (4-Core Architecture).
Amdahl’s law was also adopted in order to theoretically evaluate the expected speedup results obtained in these tests. This law states that the performance improvement when using some faster mode of execution is limited by the fraction of time in which that faster mode can be used [32]. Hence, the following expression should be considered, where \( S \) represents the solution’s overall speedup, \( f \) represents a parallelizable part of the original algorithm and \( N \) represents the total number of available processing nodes:

\[
S = \frac{1}{(1-f) + \frac{f}{N}}
\]  
(7.3)

By analyzing the expression, it is possible to conclude that:

\[
(1-f) = \frac{N}{S} - 1
\]  
(7.4)

Hence, since a maximum speedup factor of approximately 23 was obtained when using 32 cores on a single processing platform, it is possible to conclude that approximately 98.7% of the original algorithm is parallelizable by the proposed thread-level parallelization model. This value represents a great achievement.

However, if this fraction is taken into account, the remaining observed speedup values become different from the expected. In fact, a rather flat performance curve was expected instead of a linear increase of the algorithm’s overall performance.

This happens because a set of parameters end up by influencing the obtained speedup results. In fact, by comparing the proposed parallel solution with its corresponding vanilla version, where each database sequence is processed at a time, we established a real and fair evaluation model, since the parallel solution is being directly compared with its sequential implementation.

However, both solutions naturally end up by being run on very different processing environments. The vanilla version of the algorithm runs on a single processing core, which means that a single cache memory unit is available. In the proposed parallel solution, each worker runs on an independent processing core, which means that if 32 workers are available, then 32 cache memory units are available, one per each available core. Hence, in addition to the fact that multiple processing blocks are simultaneously processed in the context of several available cores, a different factor should be considered, which is introduced by the increase in the total number of cache hits per processing operation on a single core.

Furthermore, recent computer architectures use multiple cache levels. Hence, each core has its own dedicated Level 1 (L1) and Level 2 (L2) cache memory units. However, the Level 3 (L3) cache is shared by a set of independent cores. The Dell PowerEdge R810 processing platform adopts an architecture composed by four 8-Core Intel Xeon processors [30], which means that each L3 cache will be shared by a set of 8 independent cores. On the other hand, the Intel 4-Core platform groups all its cores in a single set, which means that a single L3 cache unit is shared by all the available processors in the architecture.

When the L3 cache is shared by a set of processors, each independent unit is able to take advantage of memory loaded into cache by other processing units in the same set. Hence, several global memory accesses can be avoided since the necessary data is already in cache, in most cases. The described
procedure is obviously impossible when a single core is being used. In this case, all the information has to be loaded by a single worker core, from the global memory system.

As a consequence, it is possible to conclude that the proposed implementation scales almost linearly with the number of available worker threads. However, the obtained results should be carefully analyzed, in order to realize that a set of parameters also interfere with the established comparison model.

The variation of the achieved speedup was also tested by using different query lengths. The tests were conducted in both processing platforms. Figures 7.7 and 7.8 present the obtained results.

Previous thread-level parallel approaches, such as the one adopted by SWIPE [19], typically follow inter-task processing models. Those models typically focus on the distribution of the database across different parallel computing nodes. In such cases, it is possible to simultaneously align multiple database sequences with a single query sequence. However, the size of the query sequence remains the same in the context of each independent alignment procedure. As a consequence, the approach will strongly depend on the query length.

On the contrary, by analyzing the results obtained with the proposed solution in both architectures, it is possible to conclude that the achieved gain does not directly depend on the global size of the query sequence being aligned.

![Figure 7.7: Performance dependency on the query length using 32 worker threads.](image)

![Figure 7.8: Performance dependency on the query length using 4 worker threads.](image)
As it was predicted before, this problem was circumvented by adopting a thread-level *intra-task* approach and by splitting both the profile and the query sequences into multiple independent chunks.

Hence, the proposed solution ends up by having good results in both architectures. As a consequence, it is possible to conclude that the proposed thread-level parallel model greatly increases the processing speed of sequence database searches based on the Smith-Waterman algorithm, when compared with earlier thread-level parallel approaches.

### 7.1.3 Helper Threads

A small set of tests was also conducted in order to evaluate the solution's overall performance when *helper* threads are implemented in the scope of the previously described thread-level parallelization model.

As it was previously stated in section 6.3, each *worker* corresponds to a real physical core in the HELIX architecture. Hence, each *helper* was pinned to its corresponding *worker* core. As a consequence, both threads were scheduled on two hyper-threads running on the same core, which means that a total of 64 threads (32 *workers* and 32 *helpers*) should be considered in a 32-core processing platform. Hence, each memory access operation is now treated by the *helper* thread, while its corresponding *worker* is able to permanently keep the processing of the algorithm itself.

Furthermore, each *helper* is also able to directly access the *master*’s data structures, in order to anticipate the computation of a contiguous processing block. By following this approach, the *helper* thread is able to automatically load the missing frontier dependencies, while the immediately preceding block is still being processed in the scope of its corresponding *worker*.

As it was previously stated, the proposed model adopts an anti-diagonal processing data flow, which means that each block will have two contiguous dependent blocks: one on its right and another one on its bottom. As a consequence, the described anticipation procedure can take place in two different directions (line or column).

The solution was then tested in order to evaluate the number of possible block anticipations in both scenarios. The test consisted of a single alignment procedure between two different sequences, both of them composed by 131 072 amino acid residues. Square processing blocks of 512 x 512 residues were used, which means that a set of 65 536 processing blocks is considered. This configuration simplified the model's overall evaluation. This same test was repeated fifty times in the context of each scenario. The Dell PowerEdge R810 processing platform was used for this test, so a total number of 32 *worker* threads was used. Figures 7.9 and 7.10 present the obtained results for lines and columns, respectively.

As it can be observed, when the first approach is adopted, it is possible to anticipate an average of 42 699.12 blocks per alignment procedure, which means that the following processing block will be available in approximately 65% of the times. On the other hand, when a column approach is followed, it is only possible to anticipate an average of 17 261.14 blocks per alignment, which corresponds to approximately 26% of the times.

As it was previously stated in section 6.3, a *mutex* was used to avoid concurrent accesses to any
Figure 7.9: Number of anticipations in a set of 65,536 processing blocks, by adopting a line based prefetching approach.

Figure 7.10: Number of anticipations in a set of 65,536 processing blocks, by adopting a column based prefetching approach.
of the available border table structures. When both prefetching strategies are considered, each helper potentially accesses the master’s data structures for two times, per block processing procedure. Each of these operations will block the master’s, preventing it from accessing its own processing structures. This observation naturally leads to the need of achieving a consequent compromise, in order to obtain the greatest performance level as possible.

When looking at the results corresponding to the second case, it is possible to observe that only approximately 26% of the accesses would result in a hit, which means that the master would constantly be blocked in vain. Hence, in order to circumvent this problem, a single line approach was implemented.

The previously described test conditions were also used in order to evaluate the proposed solution’s overall performance when adopting multiple helper threads. The test was performed thirty times by using only 32 worker threads, and another thirty times by using an additional helper thread per core. Figure 7.11 presents the results for the achieved processing speedup, obtained by the proposed parallelization model, by using only workers (red) and by using both a worker and an helper in each core (blue). The achieved speedup results were obtained by comparing the proposed parallel model with its corresponding vanilla implementation.

![Figure 7.11: Comparing the parallel processing models with and without helper threads, by using 32 worker threads.](image)

As it can be observed, the solution’s overall performance is typically better when an helper thread is implemented in the context of each core. In fact, by comparing the obtained average speedup values, it is possible to observe that a speedup increment of approximately 2 units is achieved with the version implementing the helpers, over the version implementing a single thread per core.

Hence, despite the fact that this technique can still be further improved, it is possible to conclude that the described approach significantly increases the processing speed of the previously presented thread-level parallelization model.

### 7.2 SIMD Solution Analysis

This section presents the results that were obtained by adopting the proposed SIMD parallelization model, which implements an inter-task parallel approach. As it was referred before, the main advantage
of an *inter-task* approach is that it simply avoids all data dependences within the alignment matrix. Thus, by using a vectorial processing model implemented with SIMD instructions in the core of each *worker* thread, it is possible to process several database sequences in the context of a single thread-level processing block.

In order to focus the attention to the performance of the adopted SIMD parallel solution, a single *worker* thread was used in the following tests. Thus, all the blocks are processed by a single core, which uses the SSE vectorial extension to the instruction set in order to process up to sixteen database sequences in parallel. Dell PowerEdge R810 processing platform was used in this test.

The variation of the achieved speedup was tested using different query lengths. Figure 7.12 presents the obtained results.

By adopting a vectorial model implemented with SIMD instructions, each *worker* thread can handle up to sixteen database sequences at a time. As expected, the solution was approximately sixteen times faster than the *vanilla* version of the algorithm. These linear performance results were obtained when aligning sequences with more than 128 amino acid residues, which means that this parallel solution demonstrates to be adequate for short and for long alignment procedures, reaching the expected performance in both cases.

![Figure 7.12: Performance dependency on the query length using SIMD vectorial parallelization on a single worker thread.](image)

As it was previously stated in section 5.2.1, the implemented solution supports multiple score ranges. A 7-bit score range is initially considered, which allows sixteen alignment score matrices to be computed in parallel. However, this range limits the scores magnitude to a maximum value of 127. When a potential overflow condition is detected in the computations using a narrow score range, the alignment score for that particular database sequence is recalculated by using a wider score range: 16-bit, which only allows up to eight database sequences to be processed in parallel.

The variation of the total number of detected overflows for the entire *nr.07* database was studied using different query sizes. Figure 7.13 presents the obtained results. As expected, the number of overflows strongly depends on the size of the query sequence. As it was described in chapter 5, the adopted solution keeps a list containing the sequence *ID* of each detected overflow. As soon as the global database processing is over, the alignment of each sequence in that list
is recalculated, by using the next wider score range. This ensures that most of the profile sequences are kept under processing by using the minimum possible score range.

Furthermore, when aligning a query sequence with 131,072 amino acid residues, only approximately 3,700 overflow conditions are detected, which means that around 98% of the total processing time is spent by considering the 7-bit score range configuration. Thus, it is possible to conclude that the additional computation time to process the sequences identified with an overflow condition is typically negligible.

In conclusion, the proposed SIMD approach reaches excellent performance results, showing almost no loss of performance for long query alignments, which greatly increases the speed of the algorithm.

The firstly described tests were also performed using Rogne’s parallel implementation [19], in order to directly compare both parallelization models. SWIPE was tested using a single thread, in order to evaluate the performance of its SIMD core implementation. Figure 7.14 presents the obtained results using the Dell PowerEdge R810 processing platform.

As it can be observed, SWIPE obtains remarkable speedup values when compared with the *vanilla* version of the algorithm. In fact, each SIMD instruction can handle up to sixteen database sequences at
a time, which means that a maximum speedup factor of approximately 16 would be expected. However, instead of that, several supra-linear speedup values are obtained.

This happens because the algorithm itself is processed in a slightly different way when using a vectorial parallelization model. In this particular case, four consecutive database residues per sequence are simultaneously loaded and processed in each algorithm iteration. As a consequence, the total number of memory accesses is reduced to only a quarter of the originally performed by the vanilla version of the algorithm.

Furthermore, the use of SIMD instructions requires all the values to be correctly aligned in memory, which optimizes cache memory usage. Hence, by comparing the obtained results with the results obtained by using the proposed SIMD parallelization model, it is possible to conclude that SWIPE is able to achieve much better results, in the exact same processing conditions.

Accordingly, as it was previously stated in section 5.2.1, the proposed parallel solution adapts SWIPE's highly efficient SIMD implementation to a different thread-level approach, which adopts an anti-diagonal processing strategy. As a consequence, two frontier dependency arrays have to be loaded per algorithm iteration. On the contrary, the original SWIPE adopts a parallel to the query sequence processing approach, which means that a single memory array is loaded per algorithm iteration.

Furthermore, the proposed processing approach focus on a distributed-memory architecture model, while Rogne’s tool adopts a rather simplistic shared-memory architecture. Thus, several memory access operations are performed in the proposed solution, which do not exist in the scope of Rogne's implementation.

In conclusion, despite the fact that both models end up by being the same in terms of processing, they are significantly different in terms of memory access operations. As a consequence, the proposed SIMD model turns out to be significantly slower than SWIPE in these particular tests. However, the proposed SIMD solution should be considered as being part of a bigger solution, which includes several parallelization models working together in order to improve the system's overall performance.

7.3 Multi-level Solution Analysis

As it was previously stated in chapter 5, it is desirable to exploit both the intra-task and the inter-task parallelism in the context of a single solution, in order to make it as complete and efficient as possible. Thus, SWIPE’s SIMD inter-task approach and the previously proposed thread-level intra-task model were jointly combined in a single parallel solution, which is now evaluated.

7.3.1 Evaluation of the optimal block size

Several database sequences are now considered in a single processing block. Naturally, this influences each block’s optimal size. Thus, the variation of the algorithm’s performance to align a query sequence against the entire nr.07 database by considering several different processing block sizes had to be re-evaluated, in the scope of this new model. For such purpose, a query sequence composed by 16 384
amino acid residues was used in these tests. Figures 7.15 and 7.16 present the obtained results.

Figure 7.15: Performance dependency on the query block size using 32 worker threads and a profile block size of 16,384 amino acid residues.

Figure 7.16: Performance dependency on the profile block size using 32 worker threads and a query block size of 4,096 amino acid residues.

The Dell PowerEdge R810 processing platform was used for this test, providing a total number of 32 worker threads. In this case, it was observed that the block size corresponding to 4,096 x 16,384 amino acid residues should be considered, in order to optimize the system’s performance.

The same tests were also performed using only 4 worker threads, in the Intel 4-Core platform. Figures 7.17 and 7.18 show the obtained values.

As expected, the results were slightly different from the obtained in the previous architecture. In this case, the block size corresponding to 4,096 x 32,768 amino acid residues should be considered in order to optimize the system’s performance.

Hence, the obtained results demonstrate that the block’s optimal size depends on the total number of worker threads being used. A closer look shows that the query block size remains the same in both architectures. However, the number of database residues per block is doubled. As a consequence, the number of worker iterations per block is also doubled. This happens because the number of worker threads trying to simultaneously access the global memory is considerably smaller. Thus, the risk of collision is considerably lower, making it better to reduce the total number of memory accesses by
Figure 7.17: Performance dependency on the query block size using 4 worker threads and a profile block size of 32,768 amino acid residues.

Figure 7.18: Performance dependency on the profile block size using 4 worker threads and a query block size of 4,096 amino acid residues.
loading greater amounts of data at a time.

7.3.2 Performance gain (Speedup)

By considering the previously obtained values for the optimal block size in each configuration, the solution was re-evaluated by varying the number of workers. A query sequence composed by 16 384 amino acid residues was used for these tests. Figures 7.19 and 7.20 show the obtained results in both architectures.

As it was previously stated in section 7.1.2, it is in fact possible to conclude that the proposed implementation scales almost linearly with the number of available worker threads. However, the obtained results should be now carefully analyzed.

In this particular case, both the thread-level and the SIMD parallelization models are considered in a single solution, while the vanilla version of the algorithm remains the same. Thus, the obtained results are affected by a set of parameters, which end up by interfering with the previous results that were separately obtained using the described comparison model.

Firstly, the vanilla version of the algorithm runs on a single processing core, which means that a single cache memory unit is available. On the contrary, in the proposed parallel solution, each worker runs on an independent processing core, which means that if 32 workers are available, then 32 cache memory units are available, one per each available core. This difference ends up by increasing the solution's overall performance due to the increase in the total number of cache hits per processing operation on a single core. Furthermore, recent computer architectures use multiple cache levels. Hence, each core has its own dedicated L1 and L2 cache units. However, the L3 cache is usually shared by a set of independent cores. In this case, each processor is able to take advantage of memory loaded into cache by other processing units in the same set, which is impossible when a single worker core is being used.

Secondly, as it was previously stated in section 7.2, the algorithm is processed in a slightly different way when the SWIPE vectorial parallelization model is adopted, which is the case. As a consequence, the total number of memory accesses is reduced and the cache memory usage is optimized, which increases the algorithm's overall performance, making it possible to achieve supra-linear speedup results.

In this case, the obtained speedup should approximately correspond to the value obtained by multiplying the total number of database sequences being simultaneously processed in each SIMD register for the total number of available workers. The obtained values by separately following this procedure are close to the obtained in the previous tests. However, there is still a significant difference, due to the described aspects.

Hence, despite some observed variances in the expected values, it is possible to conclude that the described solution scales almost linearly with the number of available workers.

The variation of the achieved speedup was also measured by using different query lengths. The tests were conducted in both processing platforms. Figures 7.21 and 7.22 present the obtained results.

As expected, the proposed solution presents highly satisfactory results in both architectures. In fact, it is possible to observe that significantly smaller losses of performance occur for long query sequences,
Figure 7.19: Variation of the obtained speedup with the number of workers (32-Core Architecture).

Figure 7.20: Variation of the obtained speedup with the number of workers (4-Core Architecture).

Figure 7.21: Performance dependency on the query length using 32 worker threads.
which makes this an highly scalable solution.

### 7.4 Model Comparison

The previously described tests were also performed using Rogne’s parallel implementation [19], in order to directly compare both parallelization models.

In fact, besides the *inter-task* SIMD model, SWIPE also implements a thread based *inter-task* approach, where multiple threads can simultaneously work on different parts of the sequence database. However, Rogne’s tool adopts a rather simplistic shared-memory architecture. To obtain a comparison as fair as possible, a shared-memory version of the proposed solution was also considered, in order to compare both parallelization models. The tests were conducted by using the Dell PowerEdge R810 processing platform. Figure 7.23 presents the results for the achieved processing speed (in GCUPS), obtained with the proposed parallelization model, using both shared and distributed-memory architectures (blue and red, respectively), and SWIPE (black).

As it can be observed, SWIPE presents better results for query sequences shorter than approximately 32 768 amino acid residues. However, the attained speed suffers a significant degradation for
larger query sequences in a massive multi-threaded environment.

As it was previously stated, SWIPE adopts an *inter-task* processing approach focused on the distribution of the database across different parallel computing nodes [19]. In this case, it is possible to simultaneously align multiple database sequences with a single query sequence. However, the size of the query sequence remains the same in the context of each independent alignment procedure. As a consequence, the approach will strongly depend on the query length.

On the contrary, the results obtained by the proposed solution remain constant for very long query sequences. This happens in both shared and distributed-memory architectures because a thread-level *intra-task* approach is adopted, which makes it possible to circumvent the described problem by splitting both the profile and the query sequences into multiple independent chunks (see Figure 7.24).

![Figure 7.24: Multi-sequence SIMD vectorization approaches for the Smith-Waterman algorithm. (A) SWIPE, which follows a parallel to the query sequence processing approach, described by T. Rognes [19]. (B) Proposed parallel processing approach, which adopts SWIPE’s SIMD model on multiple thread-level processing blocks.](image_url)

It is also important to note that slightly different results are obtained by implementing the proposed parallel approach on different memory architectures. By observing the obtained results, it is possible to conclude that the shared-memory version is specially faster, when small query sequences are considered. In fact, as it was previously stated in section 6.3, when a distributed-memory architecture is adopted, each *worker* thread has to explicitly copy the data from the global memory system. Thus, the total amount of data loaded in a single memory access has to be optimized in order to avoid collisions. In this particular case, this value corresponds to the optimal size of each independent processing block, which has been studied in the context of several processing platforms, in sections 7.1 and 7.3.

As a consequence, when a distributed-memory architecture model is adopted, the solution is only expected to reach its maximum performance when the query sequence size is equal or greater than
the block’s optimal size. Only then it is possible to guarantee that each worker’s processing time and memory access time are even, making it possible to avoid as much collision situations as possible. In the considered case, the obtained values become stable for query sequences composed of more than 4 096 amino acid residues, which exactly corresponds to the block’s optimal size, according to the tests described in section 7.3.

On the other hand, when a shared-memory architecture model is considered, it is no longer needed to explicitly copy the sequence data from the global memory to each worker’s local memory, thus assuming a traditional transparent data transfer mechanism provided by the memory and cache controller systems. Hence, the obtained results will be more stable for any query sequence length. As a consequence, the remaining values still present a significant difference due to the overhead imposed by the several memcpy operations, which are not implemented in the context of a shared-memory architecture. Hence, the performed memory operations will be considerably lighter in a shared-memory architecture, which results in better overall performance results.

To conclude, the shared-memory version of the proposed application attained performance levels of more than 77 GCUPS by using 32 parallel worker threads, which is almost three times faster than the software based on Rognes’ approach for long query sequences. The presented implementation also obtained a performance of more than 71 GCUPS using 32 parallel worker threads on a distributed-memory architecture, which is nearly 2.5 times faster than SWIPE, running on a different memory architecture.

7.5 Summary

The performance of the conceived parallel implementation of the Smith-Waterman algorithm was extensively evaluated using different processing parameters and techniques.

In order to correctly evaluate the performance of the proposed solution, two different multi-core platforms were considered:

- 4-Core Intel i7 950 processor running at 3.07 GHz with six 2 GB DDR3 RAM modules running at 2.0 GHz.
- Dell PowerEdge R810 platform, equipped with four 8-Core Intel Xeon E7-4830 processors running at 2.13 GHz and with thirty two 8 GB DDR3 RAM modules running at 1066 MHz. The system adopts the Intel’s Smart Cache technology [30], which means that each core is equipped with an individual 24 MB L2 cache. Each of these cores is able to handle up to eight concurrent threads, making this processor capable of executing up to 32 SMP threads.

The Zv9 zebrafish genome sequence was used in the tests [2]. Sequences with different lengths were extracted from this sequence and used as queries. For the database sequences, the nr.07 database was used in all the tests [2].

Due to its universal application in most toolboxes that guarantee an optimal alignment, the Smith-Waterman algorithm was implemented in the context of the presented framework. The described solution supports multiple score matrices and gap penalties. However, all the tests were performed with
the BLOSUM62 score matrix and gap open and extension penalties of 11 and 1, respectively, which correspond to the Basic Local Alignment Search Tool (BLAST) default values [31].

The proposed thread-level model was firstly tested. As it was previously stated, this model follows an *intra-task* parallel approach, where each alignment is divided into multiple chunks. Each chunk corresponds to an independent processing block, which is considered as a sub-part of the algorithm’s global score matrix.

In contrast, previous thread-level parallel approaches, such as the one adopted by SWIPE [19], typically follow *inter-task* processing models. These models typically focus on the distribution of the database across different parallel computing nodes. In this case, it is possible to simultaneously align multiple database sequences with a single query sequence. However, the size of the query sequence remains the same in the context of each independent alignment procedure. As a consequence, the approach will strongly depend on the query length.

As it was predicted before, by adopting a thread-level *intra-task* approach it is possible to circumvent this problem by splitting both the profile and the query sequences into multiple independent chunks. Hence, the proposed solution ends up by having good results in both architectures.

The proposed SIMD parallel approach was also tested, which turned out to be significantly slower than the original SWIPE implementation. However, the proposed SIMD parallelization model should be considered as being part of a bigger solution, which includes several parallelization levels working together in order to improve the system’s overall performance.

Thus, the proposed multi-level implementation of the algorithm was tested, achieving a good set of results in both architectures. In fact, it was possible to observe that significant smaller losses of performance occur for long query sequences, which makes this an highly scalable solution, just as expected.

Finally, the conducted tests were also performed using Rogne’s parallel implementation [19], in order to directly compare both parallelization models. Contrasting with the proposed approach, Rogne’s tool adopts a rather simplistic shared-memory architecture. Hence, in order to obtain a comparison as fair as possible, a shared-memory version of the proposed solution was also considered.

The shared-memory version of the proposed application obtained performance levels of more than 77 GCUPS by using 32 parallel *worker* threads, which is almost three times faster than the software tool based on Rognes’ approach for long query sequences.

The presented implementation also attained a performance of more than 71 GCUPS by using 32 parallel *worker* threads on a distributed-memory architecture, which is nearly 2.5 times faster than SWIPE, running on a different memory architecture.
Chapter 8

Conclusions

Nowadays biological sequence alignment procedures represent an enormous demanding task. Thus, multiple hardware and software solutions have been proposed in the past few years. To account for this problem, this document started with a description of parallel computing architectures in the context of biological sequence alignment algorithms.

One example of such parallel architecture is the heterogeneous processing structure that was proposed in the context of the HELIX project. However, there are lots of restrictions and limitations when designing dedicated hardware without simulating it in first place.

In order to build a dedicated processor architecture in the context of a very specific set of algorithms, their behavior must be carefully studied and correctly understood. This study defines the scope of this thesis, where several parallelization approaches and processing models have been formally studied and assessed in what concerns the attainable processing performance in an heterogeneous and dedicated processing architecture.

8.1 Achievements

The main outcome of the presented work is the proposal of the most suited set of parallel models to be applied in the implementation of a set of bioinformatic algorithms in the context of the HELIX architecture.

In the previous sections, a completely new thread-level parallel approach implementation was proposed. An instruction-level model was also proposed, based on Rognes’ SIMD processing approach. The proposed implementation considers both parallelization models, exploring both intra-task and inter-task parallelism in the context of a single solution.

A simulation environment was also presented, which allowed the solution to be tested on several common GPP based multicore architectures. The proposed model was tested using the Smith-Waterman algorithm, in order to align a single query sequence against an entire database.

The proposed parallelization model presented a rather flat performance curve in several different platforms, scaling almost linearly with the number of available cores in the architecture. In a 32-core

\(^1\)HELIX project webpage, http://sips.inesc-id.pt/~helix
GPP architecture, the algorithm's parallel SIMD based implementation reached speedup values of over 480 times faster than the *vanilla* version of the same algorithm.

The achieved performance corresponds to more than 77 GCUPS using 32 parallel *worker* threads, which is almost three times faster than SWIPE implementation running in the exact same conditions.

The described model is also extremely flexible in terms of architecture, which means that it can easily be adapted to any platform, pushing its overall performance to the limit.

### 8.2 Future Work

The presented solution already considers both *intra-task* and *inter-task* processing approaches. However, the described models could be extended to GPUs and large clusters.

In both cases, the objective would be to explore as much available computing resources as possible. The GPU unit can be considered as an additional independent processing node. Hence, it could work just like all the remaining *worker* nodes in the system, which receive and compute multiple processing blocks.

However, this would require a different processing routine, which should exclusively be run by the GPU *worker*. By following this approach, the system's processing performance should be significantly improved. However, the GPU's global memory access is extremely limited when compared with its processing capacity. As a consequence, the optimal size of each processing block would be different, which means that the whole approach would have to be reassessed, in order to optimize the system's overall processing performance.

On the other hand, if the system is part of a large cluster, it should be easy to extend the proposed model in order to support an additional data-level parallel approach, in which the data is distributed across all the available computing nodes in the cluster. This suggests the possibility of simultaneously aligning multiple database chunks with a single query sequence in the context of different nodes and architectures.

This would require some modifications in order to implement an high-level master/slave model, making it possible to distribute several database chunks across the cluster's processing nodes and then collect the obtained results, which would increase the system's overall performance. However, a significant time penalty would also be introduced in the system, which would be imposed by the communication stages required to transfer data among several distributed nodes.

Another possible use for this algorithm, in addition to database searches, would be optimizing it for a single alignment, concentrating both SIMD and thread-level parallelism in a single procedure. In this case, each *worker* would implement an *intra-task* processing approach.
Bibliography


[29] Ulrich Drepper. What every programmer should know about memory. 2007.


