
# The pairwise problem with High Performance Computing Systems, contextualized as a key part to solve the Multiple Sequence Alignment problem

Alberto Montañola Lacort

http://hdl.handle.net/10803/385282

**Escola Politècnica Superior**
**Departament d'Informàtica i Enginyeria Industrial**
**Universitat de Lleida**

# The pairwise problem with High Performance Computing Systems, contextualized as a key part to solve the Multiple Sequence Alignment problem.

Dissertation presented for obtaining the degree of
Doctor by the University of Lleida
by

**Alberto Montañola Lacort**
alberto.montanola@udl.cat

Supervised by

**Concepció Roig Mateu**
Computer Science and Industrial Engineering Department
Universitat de Lleida
roig@diei.udl.cat

**Porfidio Hernández Budé**
Computer Architecture and Operating Systems Department
Universitat Autónoma de Barcelona
Porfidio.Hernandez@uab.es

Programa de doctorat en Enginyeria i Tecnologies de la Informació.
December 9, 2015

*A mi familia.*

# Resum

El problema d'alineació de seqüències, ha estat durant anys un repte important per a la computació en la bioinformàtica. La capacitat d'alinear i comparar les seqüències biològiques entre elles, s'ha convertit en una eina molt poderosa per als biòlegs per tal d'ajudar en la seva recerca. L'alineació de seqüències no és només, la clau per esgarrapar la superfície del que sabem sobre el genoma humà, també és la clau per ajudar a la comprensió del mateix.

Per arribar a ser una eina d'èxit, l'alineació de seqüències ha de ser un procés senzill i ràpid, sent la seva complexitat de càlcul i els requisits de memòria els principals factors que limiten la quantitat de seqüències que poden ser processades. Avui en dia, els biòlegs estan alineant milers de seqüències. Un dels reptes de càlcul en l'alineació de seqüències és alinear tantes seqüències com sigui possible amb el mínim de temps. A més, ha d'estar garantida la millor qualitat d'alineació.

A partir dels antics sistemes de còmput en sèrie, ja passats de moda, avui dia els sistemes multi-core són part del nou entorn utilitzat per augmentar el rendiment dels sistemes informàtics moderns. És un repte utilitzar de manera eficient aquests recursos, de manera que aquestos puguin ser aprofitats i gestionats correctament. És important determinar la quantitat eficaç de recursos de càlcul i de memòria necessària per resoldre el problema, per tal d'evitar el malbaratament innecessari d'aquests recursos, reduint així els costos innecessaris.

En aquesta tesi, ens centrem en l'explotació eficient dels sistemes multi-

core per a l'execució de les múltiples aplicacions de problemes d'alineament múltiple de seqüències (MSA). En el camp de la biologia, el MSA es un procés de alta demanda de recursos de còmput, el qual, consisteix en dur a terme l'alineació d'un conjunt de seqüències d'entrada mitjançant la recerca de diferents similituds entre aquestes. L'objectiu és alinear tantes seqüències com sigui possible en una quantitat acceptable de temps i amb un nivell de qualitat que fa que l'alineació sigui biològicament significativa. S'han estudiat diferents implementacions d'alineament de seqüències, aquestes es comparen i es presenten en aquesta investigació. Hem contribuït en la millora dels primers passos del problema alineació de seqüències múltiples de diverses maneres. Amb l'objectiu de reduir el temps de càlcul i l'ús de memòria, adaptem l'alineador T-Coffee per treballar en paral·lel amb ús de fils lleugers en lloc de processos fork. Més endavant en aquest treball, hem desenvolupat un mètode de alineació de parells paral·lel, amb una assignació eficient de seqüències a nodes. Finalment es presenta un mètode per determinar la quantitat mínima de recursos del sistema, necessaris per resoldre un problema d'una mida determinada, per tal de configurar el sistema perquè pugui ser utilitzat de manera eficient.

# Resumen

El problema de la alineación de secuencias, ha sido durante años un reto importante de cómputo en el campo de la bioinformática. La capacidad de alinear y comparar las secuencias biológicas entre ellas, se ha convertido en una herramienta muy poderosa para los biólogos con el fin de ayudar en su investigación. La alineación de secuencias, no es sólo la clave para arañar en la superficie de lo que sabemos sobre el genoma humano, también es la clave para ayudar en la comprensión del mismo.

Para llegar a ser una herramienta de éxito, la alineación de secuencias tiene que ser un proceso sencillo y rápido, siendo su complejidad de cálculo y los requisitos de memoria los principales factores que limitan la cantidad de secuencias que pueden ser procesadas. Hoy en día, los biólogos están alineando miles de secuencias. Uno de los retos de cálculo en la alineación de secuencias es alinear tantas secuencias como sea posible con la menor cantidad de tiempo. Además, debe estar garantizada la mejor calidad de alineación.

A partir de los antiguos sistemas de cómputo en série, ya pasados de moda, hoy en día los sistemas multi-core son parte del nuevo entorno utilizado para aumentar el rendimiento de los sistemas informáticos modernos. Es un reto utilizar de manera eficiente estos recursos, de manera que estos puedan ser aprovechados y gestionados correctamente. Es importante determinar la cantidad adecuada de recursos de cálculo y de memoria necesarios para resolver el problema, con el fin de evitar el desperdicio innecesario de estos recursos, reduciendo así costes innecesarios.

En esta tesis, nos centramos en la explotación eficiente de los sistemas multi-core para la ejecución de las múltiples aplicaciones de problemas de alineamiento de secuencias (MSA). En el campo de la biología, el MSA como un proceso de computación exigente, consiste en llevar a cabo la alineación de un conjunto de secuencias de entrada mediante la búsqueda de diferentes coincidiencias entre estas. El objetivo es alinear tantas secuencias como sea posible en una cantidad aceptable de tiempo y con un nivel de calidad que hace que la alineación sea biológicamente significativa. Se estudiaron diferentes implementaciones de alineamiento de secuencias que se comparan y se presentan en esta investigación. Hemos contribuido en la mejora de los primeros pasos del problema de alineación de secuencias múltiples de varias maneras. Con el objetivo de reducir el tiempo de cálculo y el uso de memoria, adaptamos el alineador T-Coffee para trabajar en paralelo con el uso de hilos ligeros en lugar de procesos fork. Más adelante en este trabajo, hemos desarrollado un método de alineamiento de pares en paralelo, con una asignación eficiente de secuencias a nodos. Finalmente se presenta un método para determinar la cantidad mínima de recursos del sistema necesarios para resolver un problema de un tamaño determinado, con el fin de configurar el sistema para que pueda ser utilizado de manera eficiente.

## Abstract

The sequence alignment problem, has been during years an important computing challenge for the bioinformatics field. The capability to align and compare biological sequences between them, has become an extremely powerful tool for biologists in order to aid in their research. Sequence alignment is not only, the key to scratch the surface of what we know about the human genoma, it is also the key to aid in the comprehension of it.

In order to become a successful tool, alignment of sequences needs to be a simple and fast process, being its computation complexity and memory requirements the main factors that limit the amount of sequences that can be processed. Nowadays, biologists are aligning thousands of sequences. One of the computation challenges in sequence alignment is to align as many sequences as possible with the smallest amount of time. In addition, it should be warranted the best alignment quality.

From the past old-fashioned serial systems, nowadays, multi-core systems are part of the new environment used to increase the performance of modern computer systems. It is a challenge to use them efficiently, in a way that no more resources that the needed are being used. It is important to determine the efficient amount of computation and memory resources required to solve the problem, in order to avoid the unnecessary waste of them, thus reducing the required costs.

On this thesis, we focus on the efficient exploitation of multi-core systems for the execution of the Multiple Sequence Alignment (MSA) problem applications. On the biology field, the MSA as a high computing demanding process, consists on performing the alignment of a set of input sequences by searching

the different matches between them. The goal is to align as many sequences as possible in an acceptable amount of time and with a level of quality that makes the alignment biologically meaningful. Different sequence alignment implementations were studied, compared and are presented in this research. We contributed in the improvement of the firsts steps of the multiple sequence alignment problem in several ways. With the goal of reducing computation time and memory usage, we adapted the T-Coffee aligner to work in parallel with threads rather than processes. Later in this work, we develop a parallel pair-wise method, with an efficient mapping of sequences to nodes. Finally we present a method to determine the minimal amount of system resources required to solve a problem of a determined size, in order to configure the system so it can be efficiently used.

# Acknowledgments

Writing a PhD thesis is one of the most important and complex challenges that a PhD student may achieve in his career. The path to write a thesis is not always straight and easy, as a big mountain is ahead on the path, with several challenges and sometimes you may fell of the path, and you will need to keep going, and most of the time you will have to run, like if you were running away for your life from a volcanic eruption.

I would like to start this section by expressing my sincere gratitude to my supervisors Concepció Roig and Porfidio Hernández, who have patiently supported me during the course of this thesis, have given invaluable advice, suggestions and encouraged me to keep going on the right path. Without their help, I wouldn't have successfully finished this thesis.

This thesis wouldn't be possible without mentioning all the people at the Distributed Computing Group (GCD) at the University of Lleida, who have helped in some way in the development of it. It has been an honour and a complete pleasure to work with the professors: Fernando Cores, Francesc Giné, Fernando Guirado, Josep Lluís Lérida and Francesc Solsona. I also give special thanks to my colleagues Josep Rius, Ignasi Barri, Miquel Orobitg, Anabel Usié and Ivan Teixidó for all the good and bad moments we shared together in the group. Thank for their knowledge and experience on their fields, and for their support at several levels given during the development of this thesis.

Futhermore, I would like to thank professor Toni Espinosa from Universitat Autònoma de Barcelona, and Yandi Naranjo for its support and initial advice while I was starting my PhD.

I also would like to extend my gratitude to Cedric Notredame and all his

colleagues, for letting me work with him at the Center for Genomic Regulation at Barcelona, where I have learnt a lot about bioinformatics. And I would also like to thank to professor Rita Casadio from the Biocomputing Group at the University of Bologna, and all the members of CINECA supercomputing center in Bologna, specially Giovanni Erbacci, Silvia Monfardini and Mirko Cestari for their support and assistance at my research in my stay with them at Bologna.

Finally, I would like to dedicate this thesis to my friends, and specially to my family. My parents Juan and Begoña, my brother Javier, my grandparents Raquel and Agustí, and my uncle Alfredo. And a very special thanks, for Edith, for her patience and support in the final path of my PhD, and also for all my family in-law in México.

I don't want to terminate this section, without a special mention to the reader, for following my research, and sharing a bit of what was the development of this thesis, finally this gratitude is extended to the scientific community in general, the university and the humanity in general.

Contents

# List of Figures

# List of Tables

CHAPTER 1

# Introduction

This introductory chapter starts describing the thesis overall research objectives for the work done during these years. Furthermore, it briefly describes preliminary concepts about bioinformatics and genetics, and finally it fully explains to the reader concepts about the sequence alignment and the multiple sequence alignment process. The thesis outline based on the work done is presented at the end of the chapter.

## 1.1 Overview

Traditionally, the increase of performance in computer systems has been based on the increment of the processor clock frequency. This has a technological limit, being that increasing the frequency causes an increment of the energy cost and the dissipated heat by the system [MM08]. Nowadays, multi-core systems are being proposed as the new environment in order to increase the performance of modern computer systems [D05]. Traditional systems, were composed by only one central processing unit. In multi-core environments

there are at least two or more processing units, or cores, in the central processing unit. On such systems, the number of integrated cores on the same chip is increasing for newer chips. Figure 1.1 displays the layout of a modern multi-core processor.

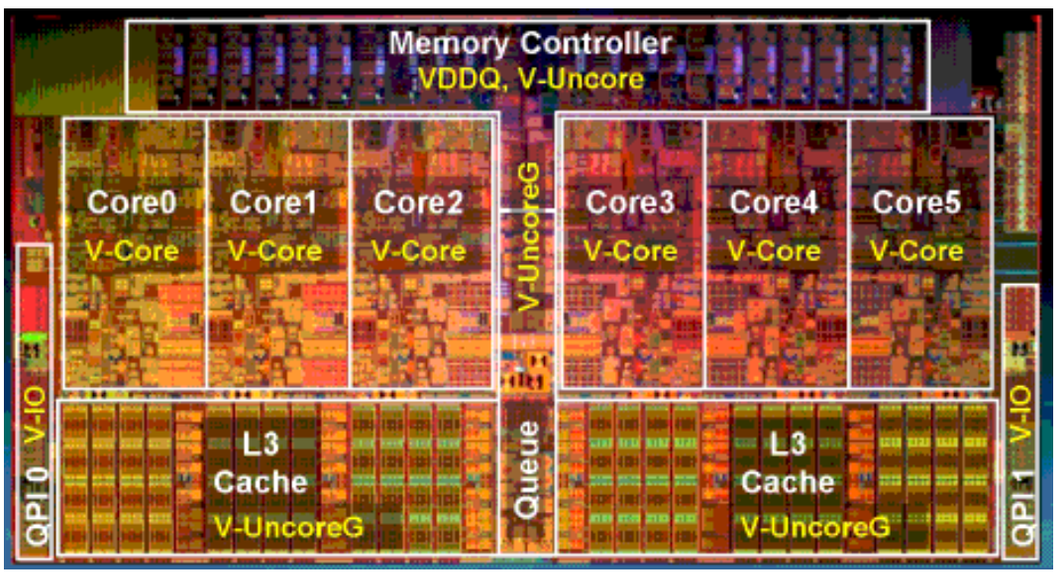


Figure 1.1: Intel Xeon six-core processor. Adapted from [PM10].

Now, we have availability of systems with a significant number of cores per node. Not only the CPU is capable of processing data, as most systems have availability of a GPU graphical processor unit, which may be used in combination with the CPU. Consequently this exposes an interesting challenge as the architecture of the GPU is different to the CPU, and this requires also two different algorithm implementations, one for the CPU and the other for the GPU. The current challenge is to use multi-core systems and GPU in an efficient way during the execution of applications. One should optimistically determine the amount of work to send to each processor unit, in order to achieve the least computing time possible to solve a problem.

There is a wide range of scientific problems that can benefit from an efficient exploitation of parallel systems, and specifically of multi-core systems. One of the scientific fields with large scale problems to be solved, is computational biology. In the biotechnology field, the deployment of the Multiple Sequence

Alignment (MSA) problem [EB06], which is a high performance computing demanding process, is one of the new challenges to address on the new parallel systems. In this work, we specifically focus on the exploitation of multi-core systems for solving the biological problem of Multiple Sequence Alignment.

MSA is one of the most used techniques in the study of biological sequences of genomes. The main goal is to find coincidences between different sequences. This allows biologists to study the differences between genomes and study the evolution of species. The current MSA implementations are a few, and not all of them are completely written to fully use efficiently the current multi-core architectures, as a consequence some of them don't scale so well. The most common MSA implementations found in literature are T-Coffee [NHH00], MUSCLE [Edg04], MAFFT [KT08] and ClustalW [LBB+07].

One MSA implementation currently used by the bioinformatics community, and one which will be deeply used in our research, is T-Coffee [NHH00]. T-Coffee, that is a sequential sequence aligner, has been designed to work on standalone systems with one processor. Although the latest implementation creates as many sequential processes as processor cores are found in the system, it is not the most efficient. Parallel-T-Coffee [ZYRA07] and Clus-T-Coffee [Y09] are two parallel implementations based on T-Coffee, that can be deployed over a cluster, thus improving its overall efficiency. The goal of this work is to analyse the behaviour of these different implementations on different systems, in such a way that, the memory and processing time parameters are being monitored with a set of instrumentation tools. In addition, these implementations will be compared with other ones found in literature.

The obtained results of this study, are determinant in the process of researching ways to improve current T-Coffee implementations in order to achieve better computing times, while maintaining the quality of the generated alignments. In order to achieve our goal, we have performed an study of some parallel implementations of T-Coffee in a set of different system configurations with different input data. Then, the results are studied in order to determine the benefits and drawbacks of each implementation. Based on these conclusions we propose different solutions. We are proposing two solutions based on T-Coffee that would increase the average efficiency of the MSA problem.

These solutions will be presented on the following chapters, accompanied with their experimental tests and compared with the other implemented solutions.

Finally, from the obtained experimental data, we are capable to study and extract an empirical prediction model, this model implemented on a sequence aligner, allows the possibility to dynamically configure the systems resources according to the specified problem size, in order to avoid the waste of unnecessary resources. This model aids in the predictions of the required memory and the best amount of processing resources to be implemented to achieve a results in the most efficient computing time.

## 1.2 High performance computing

High performance computing (HPC) is a terminology used to describe a set of complex programming techniques and hardware solutions used and involved in the resolution of complex problems, such as, for example, the MSA problem.

The HPC term includes any kind of supercomputers or parallel systems that can be used to implement and solve such kind of problems. This kind of systems have an important amount of computational and memory resources, compared to any personal general-purpose computer. While in personal computers the computing capacity is measured in (MIPS) million instructions per second, in powerful systems it is measured with (MFLOPS) million floating point operations per second.

First supercomputers were introduced in the 1960s, at Control Data Corporation (CDC), by Seymour Cray [CLHH09]. While these first systems had only a few processors, nowadays systems are composed of thousands of them, being nowadays the China's Tianhe-2 supercomputer the most fastest in the world at 33.86 petaFLOPS [top15].

Traditional systems were based on a single CPU, which frequency had been increased over the years until it reached the maximal technological level. The paradigm behind HPC systems is the use of parallel computing, bypassing the frequency limitation and dividing the processing power between different processor units. This solution has now been introduced into traditional systems creating nowadays more powerful parallel machines with multi-core processors.

Consequently, this forced programmers to change the way to design and implement software for this kind of machines, although this was already being done by the HPC community since the first parallel machines.

Bioinformatics researchers are nowadays more careful considering these kind of systems as a base for their implementations, in order to improve the current limits of traditional and simpler computational systems.

In the following section, different categories of HPC will be described according to the different features that can be considered.

### 1.2.1 Flynn's taxonomy classification

Flynn [Fly72] classified applications depending of the usage of single or multiple sets of parallel instructions executed concurrently and whether these instructions were working over a single or multiple set of data.

- **SISD: Single Instruction Single Data.** A single processor (one execution thread) works over a single set of data. Also known as a sequential processing.

- **SIMD: Single Instruction Multiple Data.** A set of processing units will execute the same instructions over different sets of data. This exploits the data parallelism paradigm. As an example, most processors have a set of SIMD multimedia instructions, thus signal processing benefits most of this kind of parallelism. A graphical processor unit (GPU), is another example of an architecture that uses this paradigm.

- **MISD: Multiple Instruction Single Data.** Different set of instructions will operate in parallel over the same set of data. This architecture is rarely used, normally this could be used for fault tolerance were all threads must agree on the results. As an example, this is used by flight control computers like the one in a Space Shuttle.

- **MIMD: Multiple Instruction Multiple Data.** Different sets of instructions will operate in parallel over different sets of data. This is the most common parallelism paradigm used, and is implemented by nowadays multi-core processors and co-processors. Other systems under this

category are the clusters, where we run multiple tasks (processes) on different nodes processing different data at the same time.

A complex HPC system will be formed by series of multi-core processors and co-processors (MIMD) and may integrate one or more GPUs (SIMD). Thus, both technologies will be available exposing a challenge in order to implement software that use both of them efficiently.

### 1.2.2 Application Granularity

Parallel applications can be modelled as dependency graphs where nodes indicate the tasks with their computation time and edges indicate dependencies and communication costs [YA99]. Depending on the amount of dependencies between each task, we can measure the type of granularity as fine-grained or coarse-grained.

- **Fine-grained:** Applications have many smaller tasks with many dependencies between them. Thus, these applications will suffer of bigger communication overheads.

- **Coarse-grained:** Tasks are bigger, in means of computation time and code involved, thus they may have less dependencies and less communication overhead.

Applications with fine-grained granularity are more capable of obtaining higher speed-up that coarse-grained ones. However, communications overhead can affect negatively this performance. Thus, the suitable granularity has to be found.

### 1.2.3 Memory model

Systems may be classified between two paradigms: Shared memory and distributed memory.

- **Shared Memory**: Shared memory systems will expose the whole of the system memory to all processor units, and it will be available immediately. This memory sits close to the processor. However, it presents scalability issues, as the number of processors increases.

- **Distributed Memory**: The whole memory is distributed along the processors of the system. Each processor unit sits close its own private memory, but has no access to memory of other processors. This architecture is stronger against race conditions and scales well. As a disadvantage whom may have to replicate the same data if applicable on all processors.

- **Distributed Shared Memory**: Based on the distributed memory model architecture, it tries to emulate with hardware or software mechanisms to present a fully shared memory model. These mechanisms consists on abstracting and hiding the communications layer from the programmer, thus it will exactly be formed by different processors units with its own isolated memory next to them, but with transparent mechanisms to offer a global view of all the system memory. While it may work as a Shared Memory systems, accessing memory from other units will incur in huge penalizations in terms of computation time.

### 1.2.4 Distributed computing systems

Depending of the previously seen classifications, and depending of the kind of implemented hardware, the network topology and the kind of tasks to run on the system, we have the following distributed systems: Cluster, Multi-Cluster, Grid, P2P and Cloud. These systems can also be classified as homogeneous and heterogeneous, being the first one formed by nodes, all of them of the same type, with the same resources (memory, processor, disk...), and the second one with nodes with different hardware specifications.

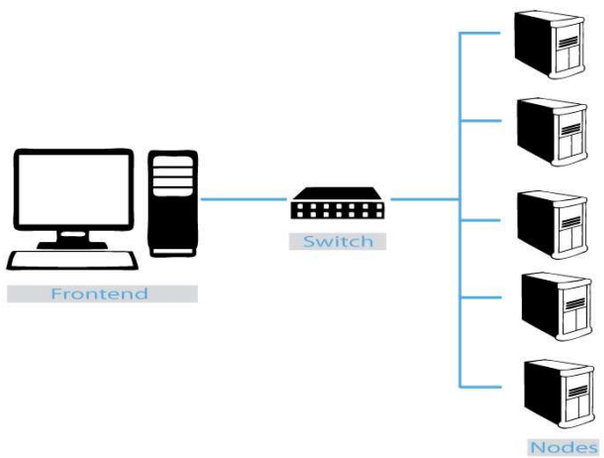


Figure 1.2: Cluster architecture.

- **Cluster**: The cluster model, displayed on Figure 1.2, is the traditional distributed computing model based on a determined amount of compute nodes. Depending of the hardware of the nodes, we can have an homogeneous or heterogeneous cluster. All nodes are connected between them, through a private switch, and a front-end node. The front-end node is responsible of running all required services for the cluster, as for example the queue services. Bandwidth between nodes tends to be high and there can be dedicated network interfaces for data exchange.

- **Multi-Cluster**: A multi-cluster, seen on Figure 1.3 is formed by two or more clusters that are linked together by a dedicated known communications link. When a cluster most of times tends to be in general homogeneous, the multi-cluster model breaks this rule as it may expose computers from different clusters with different computing resources.

- **Grid**: As displayed on Figure 1.4, the grid explores the Multi-Cluster solution by interconnecting different clusters over the Internet. In this case, connection between clusters is more prone to failures, and is slower than in a multi-cluster.

- **P2P Computing**: P2P systems, as seen on Figure 1.5, are composed by

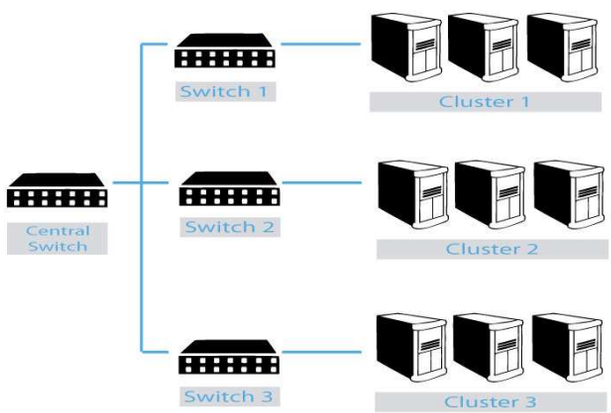


Figure 1.3: Multi-Cluster architecture.

different kind of nodes, interconnected between them over internet, there is no central node on this kind of networks. This exposes also a different paradigm to work, as nodes may process a unit of work, and send back its results to a master node (the one which initiated the request).

- **Cloud Computing**: Cloud computing, see Figure 1.6, is a new paradigm, where usually providers are providing the solicited platform as a service (PAAS). This model can be viewed as an extension to Grid, with higher level abstraction of the system.

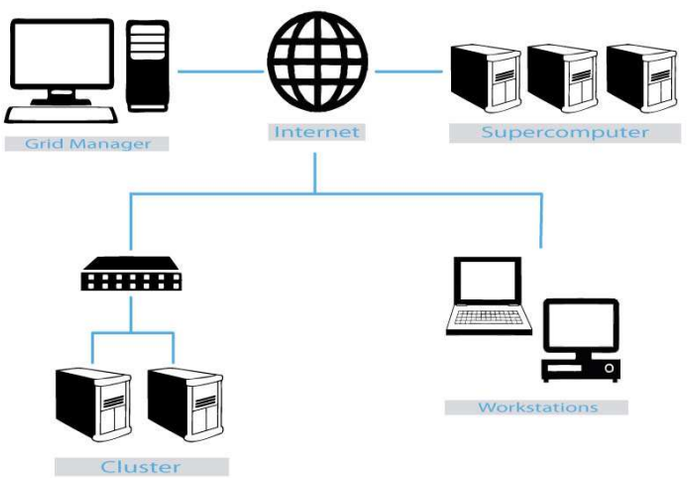


Figure 1.4: Grid architecture

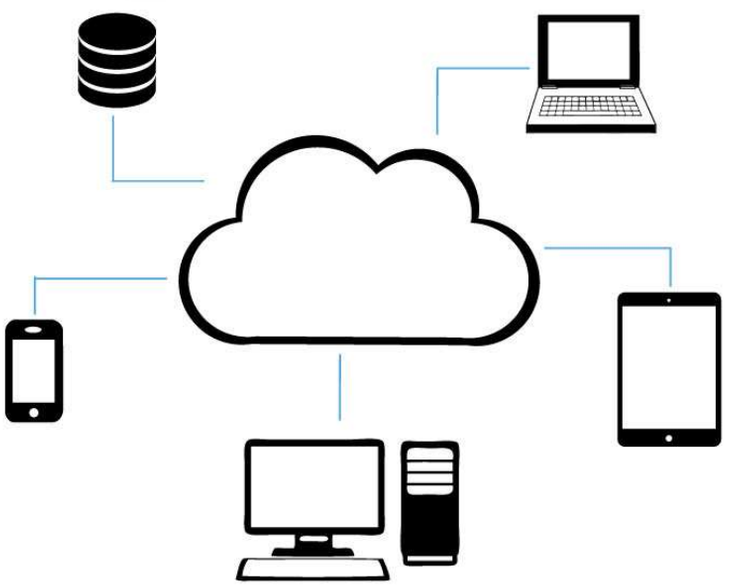

Figure 1.6: Cloud architecture.

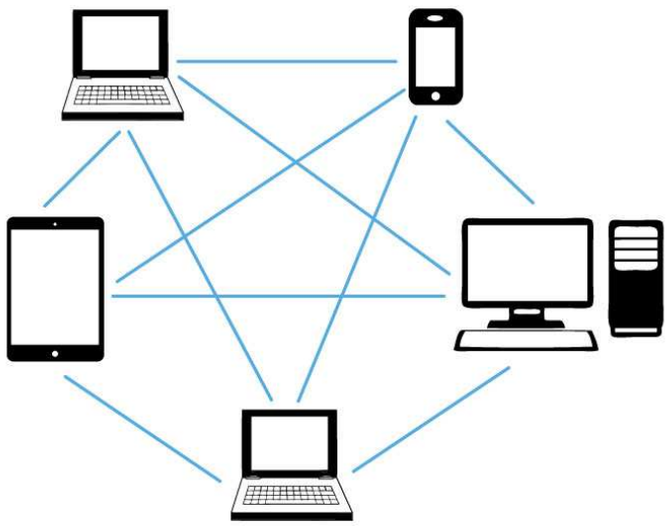

Figure 1.5: P2P computing architecture.

### 1.2.5 Software

Different kinds of software technologies are available in order to use the aforementioned hardware. Most algorithms are implemented with C or C++, mainly because contrary to other languages, they are the ones that can deliver the best efficiency and optimizations on the final binary versions. Implementations on other languages are also available, but they may have important penalty issues over the computation time, making them worse in efficiency than C/C++ ones.

- **System fork**: This technique is used in order to distribute the workload into different system processes. Communication may be performed by sockets, shared memory regions, system pipes or temporary files. It is discouraged its usage, as has some penalty issues on efficiency.

- **Threading**: Similar to system fork, but quite different, it consists on executing a function or set of functions in parallel, known as running a thread. Threads are less resource intensive than launching a new process. Using a threading library such as pthreads, allows us to fully parallelize

the code, benefiting with fully shared memory regions. It is more efficient to system forking, but bad handling of the shared memory regions may incur in synchronization issues.

- **OpenMP**: It is a set of extensions over the C language, that add a set of fully parallel instructions, these instructions allow to write fully parallel SIMD or MIMD code, that will expand over all available cores on the node, the combination of processors and co-processors installed on it.

- **CUDA** and **OpenCL**: These languages are specific to write fully SIMD code to run specifically over a set of one or more GPUs. We can send and receive data over the system bus, by using a set of instructions exposed by its C API.

- **SIMD Multimedia**: Some implementations directly go to assembler mode in order to fully optimize the code by using the multimedia SIMD instructions, although it is not necessary as SIMD C libraries are available, abstracting the lower level layer, to fully support these extensions.

- **MPI** and **PVM**: These libraries expose a set of functions to communicate different parallel processes running over different nodes. They offer all required synchronization and communication services in order to fully write a parallel program for a distributed memory system.

## 1.3 Bioinformatics

The term *bioinformatics* as a science field, is used to englobe all techniques and methods related to applying software based solutions to understand, process and display any kind of biological data. This combines the fields of biology, mathematics, statistics, engineering and computer science. As seen on the previous section 1.2, we are going to apply the usage of HPC systems to the bioinformatics field. In the following subsections, a summary of elementary concepts will be presented in order to understand the basics of sequence alignment.

### 1.3.1 Preliminary concepts

In bioinformatics, a *sequence* is a character string that codifies the different elements of a biological structure such as nucleic acids (DNA, RNA) or a protein.

A *nucleic acid* is one of the most important biological molecules in life [SZ05]. The function of this structure is to store information. This information contains vital instructions to build other vital components such as proteins. These sequences are formed by a chain of nucleic bases, each nucleic base has a different chemical structure. There are two kinds of nucleic acids: The deoxyribonucleic acid (DNA) who is responsible of the long-term storage of the information, and the ribonucleic acid (RNA) who is responsible in the catalysis of reactions and the synthesis of structures. The differences between DNA and RNA are:

- DNA is structured by two paired chains in a double helix, while RNA is only one single chain.
- The chemical structure is different, DNA contains deoxyribose $H-(C=O)-(CH_2)-(CHOH)_3-H$ while RNA contains ribose $C_5H_{10}O_5$.
- A DNA sequence may be formed by four different bases (Table 1.1): Adenine (A), Cytosine (C), Guanine (G) and Thymine (T), while on a RNA sequence the Thymine base is not present, and instead there is another one called Uracil (U). These bases are paired Adenine with Thymine/Uracil, and Cytosine with Guanine A-T/U, C-G.

Table 1.1: ADN/RNA structural bases

| | | | |
|---|---|---|---|
| A | Adenine | G | Guanine |
| C | Cytosine | T<br>U | Thymine (ADN)<br>Uracil (ARN) |

*Proteins* are vital compounds that have an important biological function in the cell. Proteins are formed by a set of different amino acids. *Amino acids*

Table 1.2: Amino acids

| | | | |
|---|---|---|---|
| A | Alanine | B | Aspartic acid or Asparagine |
| C | Cysteine | D | Aspartic acid |
| E | Glutamic acid | F | Phenylalanine |
| G | Glycine | H | Histidine |
| I | Isoleucine | K | Lysine |
| L | Leucine | M | Methionine |
| N | Asparagine | O | Pyrrolysine |
| P | Proline | Q | Glutamine |
| R | Arginine | S | Serine |
| T | Threonine | U | Selenocysteine |
| V | Valine | W | Tryptophan |
| Y | Tyrosine | Z | Glutamic acid or Glutamine |

are critical and basic molecular structures to life, so a protein is formed by a chain of different amino acids. In the nature there are a total of twenty-two amino acids, which only twenty of these are encoded in the genetic code. Other mechanisms are used to incorporate the remaining two into proteins. In addition, there are a few more amino acids not found in proteins, but that are part of other biological processes. Table 1.2 shows the standard codification used to identify each amino acid.

The ADN contains the information of how to build the different proteins. The bases are grouped in triplets called *codons*, each codon represents a different amino acid. For example the codon formed by the bases CAG codifies the amino acid Glutamine. A group of different codons, forms a protein sequence. Finally, a group of codons who is the responsible of the codification of a protein is known as a *gene* (Figure 1.7).

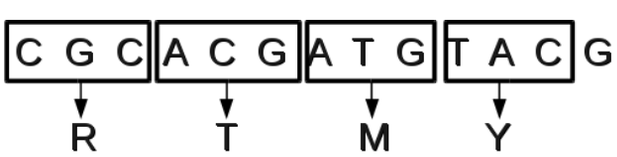


Figure 1.7: Codification of an amino acid sequence from a DNA sequence.

### 1.3.2 Sequence alignment

The sequence alignment problem, consists on finding which patterns have in common two given sequences. The goal is to obtain two sequences, where the different matching patterns are paired. For example, given the following sequences:

```
Seq1: CGCACGATGTACG
Seq2: CGGAGATATTACG
```

The resulting aligned sequences will be:

```
Seq1: CGCACGATGT-ACG
Seq2: CGGA-GATATTACG
```

The character '-' is used to represent a gap, an space in a sequence where the other sequence has a base not present in the origin sequence, thus this base cannot be matched. A character of a sequence is known as a residue.

While aligning sequences, there are three situations that can occur:

- **Insertion**: A residue not present in the origin sequence is present in the second one:

  ```
  Seq1: CGCACGATGT-ACG
  Seq2: CGGA-GATATTACG
  ```

  The underlined *T* in *Seq2* is an example of insertion.

- **Deletion**: A residue present in the origin sequence is missing in the second one:

  ```
  Seq1: CGCACGATGT-ACG
  Seq2: CGGA-GATATTACG
  ```

  The underlined *C* in *Seq1* is an example of deletion.

- **Substitution**: The residue from the origin sequence is different in the second one:

```
Seq1: CGCACGATGT-ACG
Seq2: CGGA-GATATTACG
```

The underlined *G* in *Seq1* and *A* in *Seq2* are an example of substitution.

In order to align two specific sequences, there are two possible strategies:

- **Global alignment**: It attempts to align all the residues of the sequence. It works better for similar sequences of the same size. For example, the following two sequences are aligned globally.

```
Seq1: CGCGTGCTCC-GTCGTC
Seq2: C--GTGC-CCA---GTC
```

- **Local alignment**: It is more focused in aligning similar regions inside the sequence. For example, the following two sequences are aligned locally.

```
Seq1: CGCGTGCTCC-GTCGTC
Seq2: --CGTGC-CCAGTC---
```

In the previous examples, we have seen the alignment between two sequences. According to the number of sequences involved, the sequence alignment problem can be divided in two main cases:

- **Pair-wise alignment**: Only two input sequences are compared and aligned. This is computationally more simple to calculate, and in the literature we can find several proposed algorithms. The most commonly used ones are the Smith-Waterman [SW81] algorithm, and the Needleman-Wunsch [NW70] algorithm. These algorithms are presented in detail in the following section.

- **Multiple sequence alignment**: Three or more sequences have to be aligned. The problem complexity increases. This problem is explained in detail throughout this document.

## 1.4 Sequence alignment algorithms

The sequence alignment problem, consists on searching coincidences or matches between two or more biological sequences. These sequences are character strings, with a biological meaning. A set of aligned sequences helps biologists to study the evolution of the species along other applications. Furthermore, it is a computational problem with high demanding of system resources. In the present section we are going to present the concept of distance and the two most used algorithms for aligning two sequences.

### 1.4.1 Levenshtein distance

The Levenshtein distance, is an algorithm that determines the minimal number of modifications (insertions/deletions) that have to be performed to one of the compared strings to be exactly like the other one. The general purpose of the algorithm is to calculate a distance and not to perform an alignment of the comparing strings. The resulting distance value can be used to have an idea of how similar are two given sequences, and can be used to sort sequences by similarity.

Given two string sequences: $Seq1 = a_1a_2a_3...a_i...a_n$ and $Seq2 = b_1b_2b_3...b_j...b_m$ The matrix $H_{i,j}$ is defined as: $H_{i,0} = i$ for $i \leq n$ and $H_{0,j} = j$ for $j \leq m$. If $a_i = b_j$, then $H_{i,j} = H_{i-1,j-1}$ for $1 \leq i \leq n, 1 \leq j \leq m$ else

$$H_{i,j} = min \begin{cases} H_{i-1,j-1} + 1 & Substitution \\ H_{i-1,j} + 1, & Deletion \\ H_{i,j-1} + 1, & Insertion \end{cases} \qquad 1 \leq i \leq n, 1 \leq j \leq m$$

For the given the sequences:

```
Seq1: AGCCUCGCG
Seq2: AUGCCAUUGG
```

We can build the scoring matrix shown in Table 1.3. The last cell of the table, $H_{n,m}$, contains the Levenshtein distance value.

The resulting Levenshtein distance between both sequences is four, thus we have to change four characters of the origin string in order to obtain the

Table 1.3: Levenshtein scoring matrix.

| | - | A | G | C | C | U | C | G | C | G |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| A | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| U | 2 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 6 | 7 |
| G | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 5 | 6 |
| C | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 5 |
| C | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 |
| A | 6 | 5 | 4 | 3 | 2 | 2 | 3 | 4 | 5 | 6 |
| U | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 |
| U | 8 | 7 | 6 | 5 | 4 | 3 | 3 | 4 | 5 | 6 |
| G | 9 | 8 | 7 | 6 | 5 | 4 | 4 | 3 | 4 | 5 |
| G | 10 | 9 | 8 | 7 | 6 | 5 | 5 | 4 | 4 | **4** |

second string.

### 1.4.2 Needleman-Wunsch

The Needleman-Wunsch algorithm [NW70] is used to perform a global sequence alignment between two sequences. In this case, the algorithm looks at the full sequence. It is a dynamic programming algorithm, it builds up a scoring matrix between both sequences. It obtains the best alignment by backtracking the optimal path from the cell with the biggest score.

Given two molecular sequences $Seq1 = a_1a_2a_3...a_i...a_n, Seq2 = b_1b_2b_3...b_j...b_m$, a matrix is build, using the Seq1 elements as the columns and the Seq2 elements as the rows. The matrix is filled from the last row and column, until the first one. For each pair $a_i, b_j$, a score is assigned. An example simple function to assign a score, is to assign one for matches and zero to mismatches. More complex implementations are using a scoring matrix which assigns different scoring values depending on the residues to compare, these matrices will be discussed at the end of this section. For building the matrix, the initial value is zero, and it will be assigned to all elements that don't match. If there is an assigned score in a previous column or row, then the value used to fill the matrix is this one, plus the addition of the value returned by the function used

Table 1.4: Needleman-Wunsch Scoring Matrix

| | A | G | C | C | U | C | G | C | G |
|---|---|---|---|---|---|---|---|---|---|
| A | **7** | 5 | 4 | 4 | 3 | 3 | 2 | 1 | 0 |
| U | 6 | 5 | 4 | 3 | 4 | 3 | 2 | 1 | 0 |
| G | 5 | **6** | 4 | 3 | 3 | 2 | 3 | 1 | 1 |
| C | 4 | 4 | **5** | 4 | 3 | 3 | 2 | 2 | 0 |
| C | 3 | 3 | 4 | **4** | 2 | 3 | 1 | 2 | 0 |
| A | 4 | 3 | 3 | 3 | 2 | 2 | 1 | 1 | 0 |
| U | 3 | 3 | 3 | 3 | **3** | 2 | 1 | 1 | 0 |
| U | 2 | 2 | 2 | 2 | 3 | **2** | 1 | 1 | 0 |
| G | 1 | 2 | 1 | 1 | 1 | 1 | **2** | 1 | 1 |
| G | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | **1** |

for comparing residues.

As an example, the algorithm is applied to the given the sequences:

```
Seq1: AGCCUCGCG
Seq2: AUGCCAUUGG
```

The obtained scoring matrix is shown in Table 1.4.

The path which marks the alignment is marked in bold. The resulting alignment is:

```
Seq1: A-GCC-UCGCG
Seq2: AUGCCAUUG-G
```

Finally, to calculate the score of the alignment, we have to calculate the sum of the score of each pair $a_i, b_j$ of the alignment. For each gap, we subtract a penalty value. For example, using a gap penalty value of one, the score will be: $S(A,A)+(-1)+S(G,G)+S(C,C)+(-1)+S(U,U)+S(C,U)+S(G,G)+(-1)+S(G,G) = 1-1+1+1-1+1+0+1-1+1 = 3$

In order to increase the quality of the alignments, a substitution matrix is used to determine the score of each pair of residues. These matrices contain all the possible combinations of all residues. By default we are using the identity matrix, as seen in Table 1.5, the diagonal of the matrix contains the scores

Table 1.5: Example of substitution matrix for DNA

| | A | T | C | G |
|---|---|---|---|---|
| A | 1 | 0 | 0 | 0 |
| T | 0 | 1 | 0 | 0 |
| C | 0 | 0 | 1 | 0 |
| G | 0 | 0 | 0 | 1 |

for matching residues. In literature, there are different studies, that contemplate the assignation of different weighted scores for different combinations. Consequently, better alignments are obtained by these matrices. For example BLOSUM62 (BLOcks of Amino Acid SUbstitution Matrix) [SJ92] is a matrix commonly used for amino acids.

One particular fast optimized implementation of the Needleman-Wunsch algorithm is the Hirschberg's alogrithm, that reduces the computation time required to align two sequences. The original Needleman-Wunsch algorithm has a cost in time of $O(nm)$ and in space of $O(nm)$. This implementation maintains the same cost in time, but reduces the space cost to $O(min\{n, m\})$.

### 1.4.3 Smith-Waterman

The Smith-Waterman algorithm [SW81] is used to perform a local sequence alignment between two sequences. It is also a dynamic programming algorithm, in which a scoring matrix is built with the two input sequences, and a optimal path is computed by assigning different scores on each cell of the matrix. The biggest score on the matrix is the beginning of the path, where the algorithm will perform backtracking to determine the best alignment.

Given two molecular sequences: $Seq1 = a_1a_2a_3...a_i...a_n$ and $Seq2 = b_1b_2b_3...b_j...b_m$

The matrix $H_{i,j}$ is defined as: $H_{k,0} = H_{0,l} = 0$ for $0 \leq k \leq n$ and $0 \leq l \leq m$.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + S(a_i, b_j) & Match \\ H_{i-k,j} - W(k), k \geq 1 & Deletion \\ H_{i,j-1} - W(l), l \geq 1 & Insertion \end{cases} \quad 1 \leq i \leq n, 1 \leq j \leq m$$

$S(a_i, b_j)$ returns a similarity coefficient between $a_i$ and $b_j$. For example, if $a_i = b_j$ then the returned value could be 2, and if $a_i \neq b_j$ the value could be -1.

$W(k)$ is a constant or function, that returns a penalty value to be applied when $a_i \neq b_j$. For example it can be defined as as $1 + 1/3 * k$, where k is the horizontal or vertical distance between two cells in the matrix. For example for $H_{i,j}$ and $H_{i-3,j}$ $k$ is 3. In some implementations $W_k$ is a constant value, defined as 1.

Given the following sequences:

```
Seq1: AGCCUCGCG
Seq2: AUGCCAUUGG
```

and applying the algorithm with $S(a_i, b_j) = \{2 \text{ if } a_i = b_j, -1 \text{ if } a_i \neq b_j\}$, and defining $W_k = 1, k = 1$ , the scoring matrix shown in Table 1.6 is built. To initialize the matrix, we put each sequence as the heading of the columns and rows. The first row and the first column of the matrix will be always initialized to zero. Finally, each cell contains the value of applying the already seen algorithm. The path with the biggest scores is marked in bold. The resulting alignment that we obtain from the path is:

```
Seq1: A-GCC-UCGCG
Seq2: AUGCCAUUG-G
```

There are implementations of this algorithm available in several languages and with several optimizations. Implementations are available for CPU, GPU and FPGA. One implementation available is FASTA [WD88], it is a set of tools used for generating the alignment of two sequences, and also they can search a given sequence in a database of sequences. FASTA provides two optimal implementations of the Smith-Waterman algorithm capable of using SSE instructions [A97] to speed the alignment process.

## 1.5 Multiple Sequence Alignment

The Multiple Sequence Alignment problem (MSA), is a specific case of the Sequence Alignment, were instead of aligning only two sequences, we have to align

Table 1.6: Smith-Waterman scoring matrix.

| | - | A | G | C | C | U | C | G | C | G |
|---|---|---|---|---|---|---|---|---|---|---|
| **-** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | **2** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| U | 0 | **1** | 1 | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| G | 0 | 0 | **3** | 2 | 1 | 1 | 1 | 3 | 2 | 2 |
| C | 0 | 0 | 2 | **5** | 4 | 3 | 3 | 2 | 5 | 4 |
| C | 0 | 0 | 1 | 4 | **7** | 6 | 5 | 4 | 4 | 4 |
| A | 0 | 2 | 1 | 3 | **6** | 6 | 5 | 4 | 3 | 3 |
| U | 0 | 1 | 1 | 2 | 5 | **8** | 7 | 6 | 5 | 4 |
| U | 0 | 0 | 0 | 1 | 4 | 7 | **7** | 6 | 5 | 4 |
| G | 0 | 0 | 2 | 1 | 3 | 6 | 6 | **9** | **8** | 7 |
| G | 0 | 0 | 2 | 1 | 2 | 5 | 5 | 8 | 8 | **10** |

a set of three or more sequences. These input sequences are assumed to have an evolutionary relationship. The generated alignment will show shared subsequences from a common ancestor, with insertions, deletions or different bases representing alterations like mutations or single nucleotide polymorphisms.

In Figure 1.8, we can see the output of a MSA of some sequences, the displayed sequences are a portion of a long output file. These matching characters are aligned on the same column. Gaps, insertions and/or deletions of characters may be performed to enforce the alignment of the sequence. A gap is a space between two characters, on a insertion one or more characters are added, and on the deletion one or more characters are deleted. On this example, the alignment quality is represented in a different color tonality, from light to dark, representing the worst to the bests values respectively.

## 1.6 Objectives and contents

The present work is focused on the MSA problem. One of the main demands of the biotechnology researchers for algorithms that solve MSA, is that they should be able to align as many sequences as possible, thousands if possible, with good quality alignments in biological meaning. In this framework, we

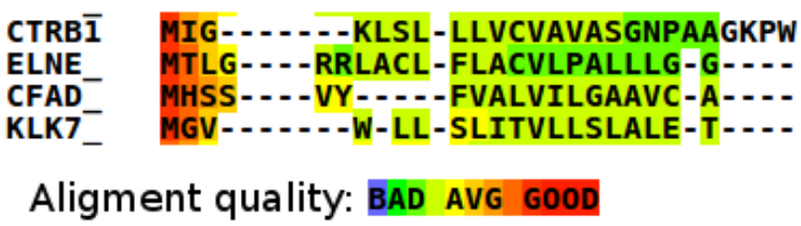


Figure 1.8: Example of a multiple alignment of a set of sequences.

focus our main efforts in a specific implementation of MSA, named T-Coffee, as it is broadly used among the scientific community. Our goal is to provide T-Coffee with the ability to execute in the current high performance computing systems with effectiveness, in order to be able to align thousands of sequences. To achieve this goal, we are proposing the following specific objectives to develop in the present work of thesis:

- To analyse the main steps involved in the MSA process and to measure their complexity.
- To study the most commonly used implementations of MSA and to discern their similarities and differences.
- To analyse the performance and quality of the different MSA implementations compared with T-Coffee.
- To propose some alternatives to improve T-Coffee to be executed in current computing systems with an acceptable performance.
- To propose new implementations to solve the steps inside MSA that have higer complexity and, as a consequence, those that take more computation time.
- To apply load balancing and resource utilization techniques to distribute the MSA work in distributed computing platforms in an efficient way.

These proposed objectives are developed throughout the present document with the following chapters:

- *Chapter 2*: It introduces the sequence alignment and multiple sequence alignment problems, different implementation found on literature, preliminary concepts and related work.

- *Chapter 3*: It focuses on a specific MSA implementation, T-Coffee, and performs a deep efficiency and scalability study of different solutions, by comparing them.

- *Chapter 4*: It starts by presenting our contribution to improve T-Coffee, by rewriting parts of the T-Coffee code, so it can start using threads rather than fork/join. It also presents our proposed sequence alignment solution, with its design discussed and the obtained results compared with the original MSA application.

- *Chapter 5*: It presents an optimal method for determining the most efficient amount of computing resources to solve the MSA problem with optimal efficiency. It also presents a set of experiments and results in order to validate our model.

- *Chapter 6*: Finally, chapter 6 outlines the main conclusions of the work, and future work.

- *Appendices*: A set of additional resources related with the solutions implemented, is found on the last pages of this document.

CHAPTER 2

# Related Work

Sequence alignment and the multiple sequence alignment, when more than two sequences are in action, constitutes a computational a problem, on which several authors, over the past decades, have proposed different kind of algorithms and implementations with the aim of solving it in an efficient way, offering an important value of quality for their alignments. During the last decade, authors have started to explore new ways to solve the problem using HPC systems. These chapter explores in detail the different implementations for sequence alignment that we can find in the literature. Starting with the most common and used implementations, to the least ones, we will briefly explore and name some of its main characteristics.

## 2.1 Implementations overview

According to their implementations Multiple Sequence Alignment algorithms, can be classified in different ways. In one way, we can classify algorithms according on the inner parallelism level of its implementation in the following

categories:

- **Sequential**: This is the most traditional and simple solution, where these implementations run one single thread of code, so they are not optimized to use the capabilities of newer processors. Processor runs instructions sequentially one by one. Most implementations are mostly sequential. Example of this implementations are: Clustal [LBB+07], Kalign [LS05], MAFFT [KT08], MUSCLE [Edg04], Dialign-TX[SKM08].

- **Multi-Threading**: These implementations benefit of capabilities offered by multi-core processors and many-core co-processors. The execution of code runs in parallel, where multiple instructions run simultaneously on the same node, over the same data or different data, as they usually implement a shared memory model. They exploit the execution in parallel through the cores with different approaches:

  1. **Forking**: On this technique a new process is created after the fork operating system call. When this occurs, the operating system will clone the process requesting to fork, by performing a copy on write of the complete process memory, thus only modified data structures after the fork will be copied. In this case, each process, although is a copy of the original one, it will not have access to the memory of the other processes. These processes will run in parallel and will require a messaging queue service in order to intercommunicate. This can be provided by a messaging passing API such as MPI, by using sockets, shared memory regions or by using temporary files. The master process may join the execution of its created child processes, by checking the state of them, until they terminate its execution. This technique, in general has higher memory and computational costs in comparative with other techniques. For example T-Coffee [NHH00] implements this technique.
  2. **Threads**: Threading, is a programming technique that allows to run multiple instances of code simultaneously in parallel. Although the idea is similar to the forking methodology, threads are lighter processes than fork based processes, as they don't perform any kind

of copy of code or data. Threads will follow a complete shared memory model, were the process memory is the same across al threads. A thread will start running a piece of new code when it starts. Created threads will and need to be joined. This technique is more efficient than forking, and benefits on a considerably lighter memory consumption and processor resources. On the contrary, when dealing with shared memory, whom should be very careful to write into data structures that are being processed by a different thread. Thus bad programming patterns may influence in synchronization errors and software malfunctioning.

3. **OpenMP**: OpenMP is a language extension to the C programing language, that offers a set of preprocessor instructions that will expand a fully parallel code. One can write code that will expand to a set of parallel instructions in a specific region of the code, to be run fully in parallel. These parallel regions work very similar to creating a set of worker threads. An example using this paradigm is Clustal Omega [SWD+11].

- **SIMD instructions or GPU**: The single instruction and multiple data paradigm can by exploited by two different techniques:

  1. **Multimedia SIMD**: Most processors come with a set of multimedia based scalar instructions aimed to be used for video and audio processing. These instructions allow to operate over larger sets of data in parallel with the same code. We can observe [Far06] as a parallel implementation of the Smith-Waterman pair-wise sequence alignment algorithm.
  2. **GPU**: The graphical processor unit is a complete SIMD system, capable of running several threads in parallel with different sets of data. In order to operate with this processor, two languages to interface with them are available. CUDA by Nvidia, is a proprietary language specific to the Nvidia vendor based GPU cards, in order to write code that will run on these processors. The program that will run on the GPU can be loaded by a interface available to the C lan-

guages, that also offers operations to copy, read and write into the GPU memory from the main system memory. The second language available is OpenCL, that is more accepted by the community, as it is established as an standard, thus it has implementations available on more platforms than CUDA. OpenCL can benefit of ATI Radeon, and NVIDIA based GPUs. An example using such implementation can be seen on [LMS09] as a parallel implementation for GPU of the Smith-Waterman pair-wise algorithm.

- **Distributed computing**: This paradigm, based on a distributed memory model, works by running several processes on any kind of distributed system. Each process can be either sequential or use any combination of the previously seen paradigms. Thus the most efficient one will combine most of them by fully using all available computing resources. For the communications between each node, we can use plain networking, being the fastest the better, or other hardware specifically designed devices such as InfiniBand. Several implementations can be used for the communications between each node, such as message passing interface (MPI) and parallel virtual machine (PVM). Examples of these implementations are Parallel-T-Coffee [ZYRA07], Clus-T-Coffee [Y09] and ClustalW-MPI [Li03] [CDP+03].

## 2.2 MSA algorithms

The multiple sequence alignment process consists mostly on set of different steps. These steps may be different depending on the implementation, algorithms, heuristics, and mechanism used to achieve the final multiple sequence alignment. We can describe most MSA implementation, seen on Figure 2.1 in 3 main specific steps: Pair-wise alignment, building of the phylogenetic guide tree and progressive alignment. The functionality of each step is the following:

- *Pairwise alignment*: Given a set of input sequences, most implementations build a primary library containing information about all the possible pairs of sequences, with the goal of obtaining a distance matrix

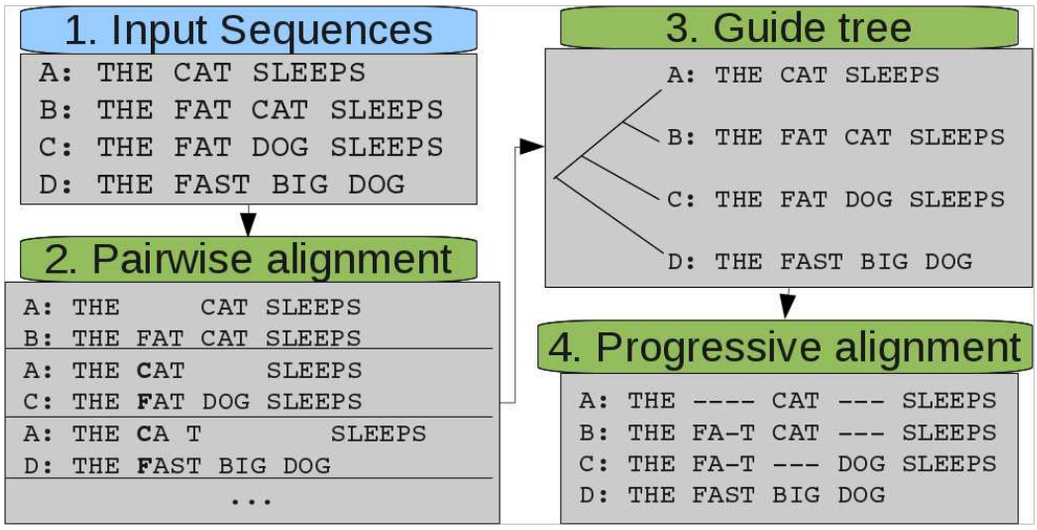


Figure 2.1: Steps of the multiple sequence alignment process.

with all this pairs of sequences, which its corresponding scores. All this computed data, will have an important role in the following steps of the algorithm. In order to build this library, algorithms seen on the previous chapter, such as Smith-Waterman or Needleman-Wunsch between others can be used.

- *Phylogenetic guide tree*: The given implementation may use different techniques to build a phylogenetic guide tree, according to the scoring information that relates sequences between them. The final tree, has the purpose to relate all possible sequences between them as close as a possible and to indicate the order in which sequences have to be aligned in the following step.

- *Progressive alignment*: From the given sequences in the guide tree, a progressive alignment step is performed, by following all nodes in the tree, we are able to build the final MSA alignment, containing all possible sequences.

The most used sequential implementations of the MSA process that can be found in the literature are the following: Clustal, Kaling, MAFFT, MUSCLE,

Dialign-TX and T-Coffee. Next we expose a brief description of their operation.

- **Clustal** [LBB+07] It is able to align hundreds of sequences in a couple of hours. It performs the previously seen three steps:

  1. Pair-wise alignment of all input sequences using the Wilbur and Lipman algorithm.
  2. A speculative evolutionary tree is built, using neighbour-joining.
  3. The tree is used to perform the multiple alignment.

  Two versions of clustal are available, ClustalW is the most known and mature version available, and they have recently published the beta version of Clustal Omega [SWD+11]. While the traditional ClustalW implementation does not benefit on any kind of paralellization optimizations, Clustal Omega is a new implementation that uses OpenMP primitives for its standalone based parallelization in order to use more efficiently a multiprocessor. Finally, a MPI implementation of ClustalW, known as ClustalW-MPI [Li03] [CDP+03] is also available. This later implementation achieves parallelization by dividing into tasks each step, in order to collaborative achieve the final alignment.

- **Kalign** [LS05] is another fast MSA serial implementation. It concentrates on local regions. It uses a dynamic programming standard progressive method for its sequence alignment and it performs the following steps:

  1. Performs the pair-wise alignment of all input sequences using the Wu-Manber algorithm.
  2. Constructs a guide tree using the UPGMA or neighbour-joining algorithms.
  3. The tree is used to perform the multiple alignment.

  There is not known parallel version of this algorithm implemented.

- **MAFFT** [KT08] is a another MSA, with the characteristic of being implemented using a threading library to speed up its alignments. It

implements different algorithms that can be switched in order to obtain different results. These alignment strategies are classified as: Progressive methods, iterative refinement method using the weighted sum of pairs score and the iterative refinement methods using both the weighted sum of pairs and consistency scores. The following steps are done depending of the configured method:

- Progressive methods:
  1. Builds distance matrix.
  2. Constructs a guide tree.
  3. Performs progressive alignment.
  4. Re-constructs the tree.
  5. Re-aligns the sequences to obtain the final alignment.
- Iterative refinement method using the weighted sum of pairs score:
  1. Builds distance matrix.
  2. Constructs a guide tree.
  3. Performs progressive alignment.
  4. Re-constructs the tree.
  5. Re-aligns the sequences.
  6. Iterative refinement.
- Iterative refinement methods using both the weighted sum of pairs and consistency scores:
  1. Builds distance matrix.
  2. Constructs a guide tree.
  3. Performs progressive alignment.
  4. Iterative refinement.

The distance matrix is built using a fast Fourier transform method that is described on [KT08]. The guide tree is built using a custom algorithm based on UPGMA. On the progressive method, the guide tree is built two times, with the achievement of obtaining a better quality based

alignment. The second method, adds an additional step to an additional refinement algorithm, to adjust the final alignment sequences taking into account the sum of pairs score. On the final method, the tree is only build one time, as the consistency scores are used in this case.

- **MUSCLE** (multiple sequence comparison by log-expectation) [Edg04]. This implementation claims to be the fastest with the best quality in comparison with the other implementations. Similar to previous implementations Muscle performs the following steps:

  1. Distance matrix and guide tree estimation.
  2. Pairwise profile alignment, that will be used for the progressive alignment and finally for performing a refinement alignment.

- **Dialign-TX** [SKM08]: Another MSA serial implementation. It uses a different strategy that differs from traditional MSA algorithms. It uses a straight-forward greedy technique to get the multiple sequential alignment from a previous local pairwise step.

- **T-Coffee** [NHH00]: T-Coffee MSA uses a traditional method for solving the problem. Our work focus on this algorithm, by extending or improving it. The following steps are performed by it:

  1. Builds the primary library by performing the pairwise of all sequences.
  2. Extends the library, to give more accuracy to its alignments.
  3. Builds the guide tree.
  4. Performs the progressive alignment obtaining the final alignment.

  Parallel versions of T-Coffee addressing different issues have been proposed as Parallel-T-Coffee [ZYRA07] and ClustalW-MPI [Li03], being the first one a parallel aproach of T-Coffee and the second one a complete different mechanisms were performs a clusterization of the input sequences in order to reduce the problem in smaller ones.

As we have seen along this chapter, in order to to improve the performance of these MSA algorithms, some authors have proposed distributed memory solutions based on message passing (MPI), including Parallel-T-Coffee [ZYRA07] and ClustalW-MPI [Li03] [CDP+03] that have considerably improved their speedup. They coincide in the need to align several pairs of sequences. Thus, they are focused on solving the problem using only a distributed memory paradigm and lack the benefits of a hybrid system (distributed/shared memory).

While the previous implementations will run several instances of the Smith-Waterman algorithm in parallel in order to align as many sequences as possible, there are also parallel implementations focused on the Smith-Waterman algorithm itself using shared memory such as [Far06], which describes an accelerated version using SIMD processor extensions, and [LMS09] which describes an accelerated version running on GPU. These implementations are more specific in aligning a single pair of sequences as fast as possible rather than several pairs of them. Using both methods may benefit from a better hybrid implementation that would exploit and use the modern parallel systems, made up of nodes with multi-core CPUs and GPUs, more efficiently.

Once have seen the overview of the main algorithms existing in the literature to carry out MSA, and their improvements to increase performance, we focus in the present work in T-Coffee algorithm. The reason is that it is the one that provides the best quality results in the biological meaning of their alignments, so that it is very useful for the biotechnology community and it lacks of efficient parallel implementations that combine the use of shared and distributed memory that permit to efficiently exploit nowadays systems.

CHAPTER 3

# Analysis of MSA based on the T-Coffee implementation

In this chapter, we carry out a deep analysis of T-Coffee, which, as we have seen in the previous chapter, is one of the existing algorithms to solve the MSA problem. We expose the different implementations of T-Coffee and we make a deep analysis of usage of system resources, where the following variables are analysed: computing time, memory usage and inter-node communications in the case of the parallel implementations.

## 3.1 MSA based on T-Coffee

Currently, T-Coffee (Tree-based Consistency Objective Function For alignment Evaluation) [NHH00] [DTMX+11] is the most extensively MSA implementation used among biologists. It can be used to align Protein, DNA and RNA sequences, and it can also combine the output generated by other MSA implementations like Clustal, Muscle, Mafft, etc.. into one unique alignment using

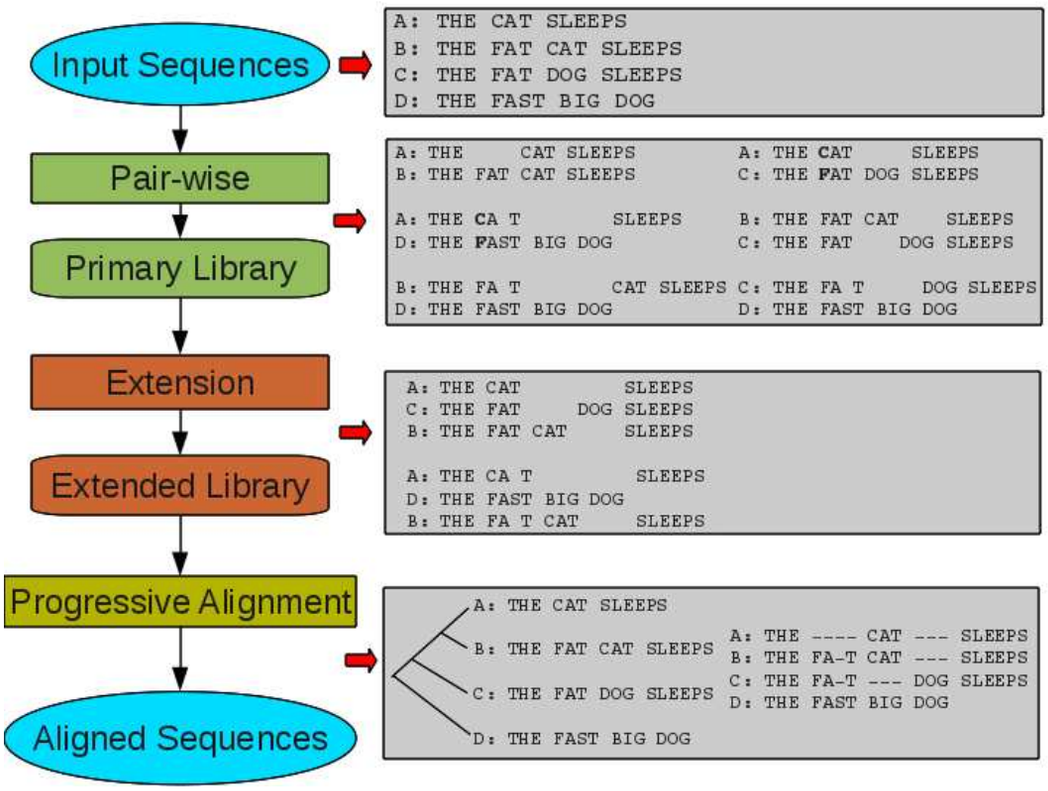


Figure 3.1: Steps and example of the T-Coffee process.

another mode of operation called M-Coffee. Input text files containing unaligned sequences are processed by it, generating the resulting alignment of all the involved sequences. It tries to improve the accuracy of the results taking into account biological information obtained from pair-wise alignments.

The original standalone version of T-Coffee is mostly a sequential implementation, although some parts of it are parallel using the operating systems primitive fork calls. Thus, T-Coffee will launch a set of processes to divide part of the computing work set. However, it cannot be directly used in a cluster based environment. The other implementations presented in the next sections, use the message-passing library MPI, to parallelize the different stages of the algorithm.

The process carried out by T-Coffee, illustrated in Figure 3.1, has the following three main stages, whose operation is exposed next.

- **Pairwise alignment**: Construction of the primary library, containing all the pair-wise alignments from all input sequences. In our example, all the pairs of A, B, C and D sequences are aligned obtaining 6 pair-wise alignments. Information about $N(N-1)/2$ pairs is stored in this library, where N is the number of input sequences. Alignments are represented as a list of matching pairs of residues. Each of these correlations is considered by T-Coffee as a restriction. A weight value from a biological scoring scheme is assigned to each restriction. The computing cost of this phase is very high because it is necessary to obtain the pair-wise alignments for all input sequences. Defining $L$ as the average length of the sequences to align, the alignment of two sequences in particular has a computing complexity in space of $O(3L)$, as we have to store the original sequences and the aligned one. Furthermore, the cost in time is $O(L^2)$. Aligning N sequences requires $N(N-1)/2$ combinations, thus this is equivalent to a global complexity of $O(N^2)$. Given these premises the whole process will have a complexity in space of $O(N^2 * 3L)$ and in time of $O(N^2L^2)$.

- **Extension**: The extension is the heuristic used by T-Coffee to adjust the restrictions group of the primary library into multiple sequence alignments. The analysis in the extension is based on taking from the primary library each pair of aligned residues in two sequences and check whether these are aligned with the same residue of the third sequence. The weight of the aligned residue in the third sequence in the primary library is added to the weight assigned to the pair of residues in the primary library. The process in the extension is made for each pair of residues with all the input sequences. In our example, we can see how the aligned pair of sequences A and B, are aligned against a third sequence, first with C then with D. This will be repeated with all possible sequences and pairs. The extension is designed to generate a list of restrictions used to evaluate the construction of the final multiple sequence alignment. As described in the original article of T-Coffee [NHH00] the complexity in time is about $O(N^3L)$ and the same in space. The generated library is called Extended Library (EL).

- **Progressive Alignment**: In order to obtain the progressive alignment it is necessary to generate a distance matrix from all input sequences, taking into account the scores generated by the primary library and its extension. A guide tree is built with the values of similarity between pairs and triplets of sequences. The guide tree is built as a Directed Acyclic Graph [YA99] and its precedences are used to establish the order to be followed in the progressive alignment. In the example, the order established by the guide tree indicates a first alignment of sequences A and B, next with C and finally with D. The complexity for building the guide tree is $O(N^3)$ and the complexity time for performing the progressive alignment is $O(NL^2)$.

T-Coffee is capable on producing alignments with high quality in the biological sense, with a considerable good score in comparative with other algorithms, but as a main drawback, T-Coffee is the slowest implementation, taking more time to reach the final alignment. Given situations where whom wants to get a quick alignment, whom may sacrifice a bit of quality to get it fast. When a good quality alignment is an important achievement, then it is necessary to dedicate important computing resources to it. As a consequence of this limitations on the current implementation of T-Coffee, authors have proposed some solutions that parallelize, some or all parts of the T-Coffee algorithm, in order to achieve the same result in less time. Other implementations have also been implemented using parallel languages.

### 3.1.1 Parallel T-Coffee

To overcome the problems in the use of resources that became evident for T-Coffee, the Parallel-T-Coffee (PTC) [ZYRA07] is glimpsed as a possible alternative to align a greater number of sequences. PTC is a parallel MSA implementation using a message passing library, specifically MPI, with the aim of aligning a greater number of sequences. PTC, based in T-Coffee version 3.79, was implemented from scratch for a cluster based environment. The constraints library is distributed across the different tasks, and performs alignment operations in parallel using dynamic scheduling techniques. As seen in Figure

3.2, it performs the same process as T-Coffee but parallelized across a set of different message-passing tasks implemented in MPI whose computation can be distributed among different nodes. The parallelized steps are the following:

- Library Generation: In PTC the library generation is computed in a distributed form. First, the pairwise computation is distributed among the nodes, then the constraints are grouped and re-weighted. Library extension is not performed as in original T-Coffee, it is postponed and done on the fly in the progressive alignment phase. Finally, a 3d lookup table is built from the library. Caching techniques are applied on this table.

- Progressive Alignment: Is the most difficult step to parallelize. Computations in progressive alignment follow a tree order, thus its parallelization is reduced to a DAG (Directed acyclic graph) scheduling problem [YA99], where the tasks of sequence alignment are distributed among system nodes taking into account the precedences marked by the DAG.

### 3.1.2 Parallel T-Coffee using similarity based clusters of sequences

According to the performance problems that had been seen in the previous MSA implementations of T-Coffee, in our group we developed a new message passing (MPI) implementation that is based in a divide and conquer technique: Clus-T-Coffee (CTC) [Y09] is based on version 7.81 of T-Coffee and goes one step further in the way it solves the MSA problem. Clus-T-Coffee tries to apply the divide and conquer problem by reducing considerably the combinatory explosion of pairs of sequences to align, by running separate independent T-Coffee processes with smaller inputs. These inputs are determined by grouping and dividing all input sequences on clusters of similar sequences.

In CTC, see Figure 3.3, a previous phase of clustering is introduced that builds a set of different clusters (groups) of sequences. The number of generated clusters and the amount of sequences depends on a cut-off value that determines the similarity of the sequences for being grouped in each cluster. As

the initial task to compute the clusters, CTC needs to compute the pair-wise alignments between all input sequences. In the following phase, each node will

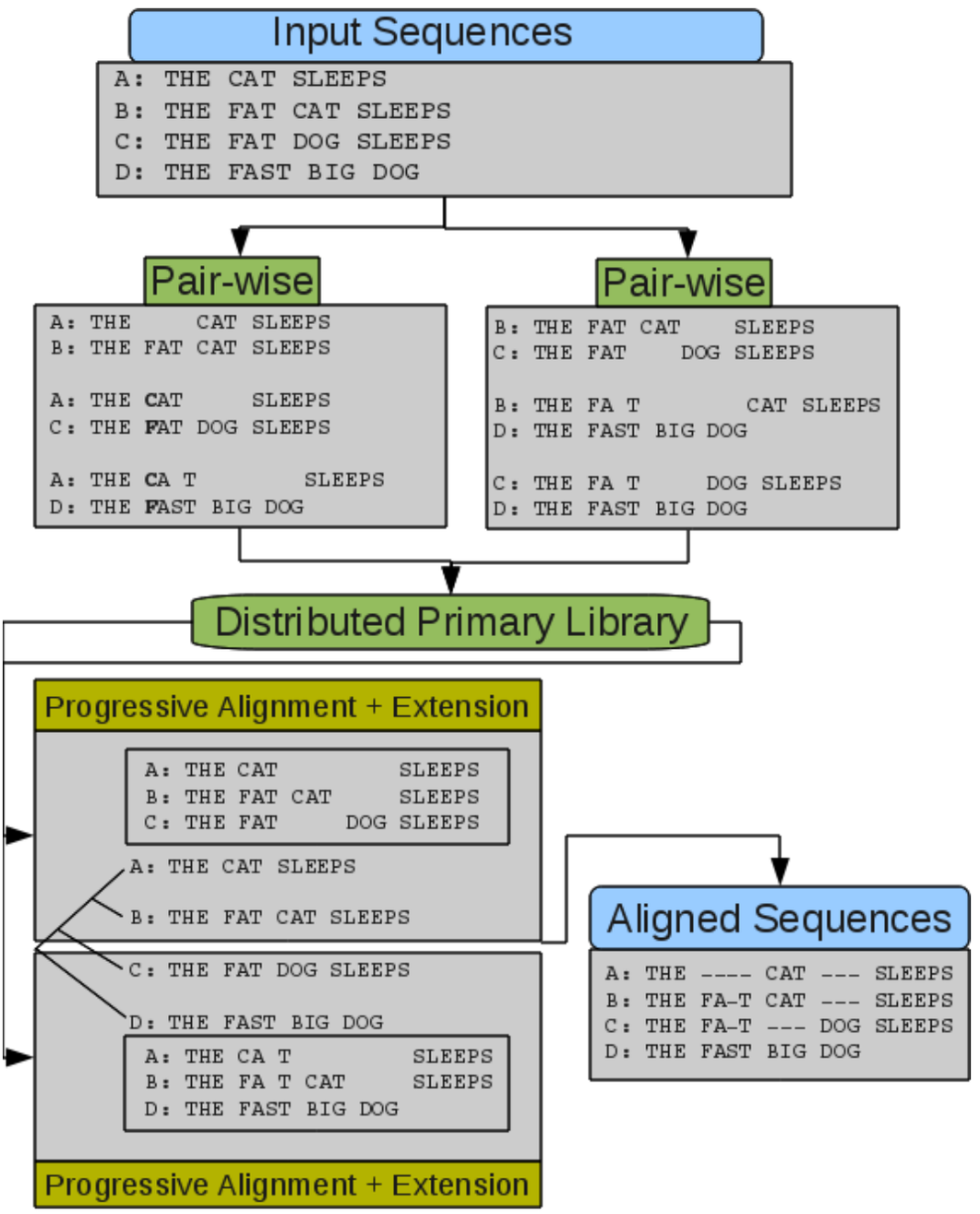


Figure 3.2: Steps of Parallel-T-Coffee (PTC) process.

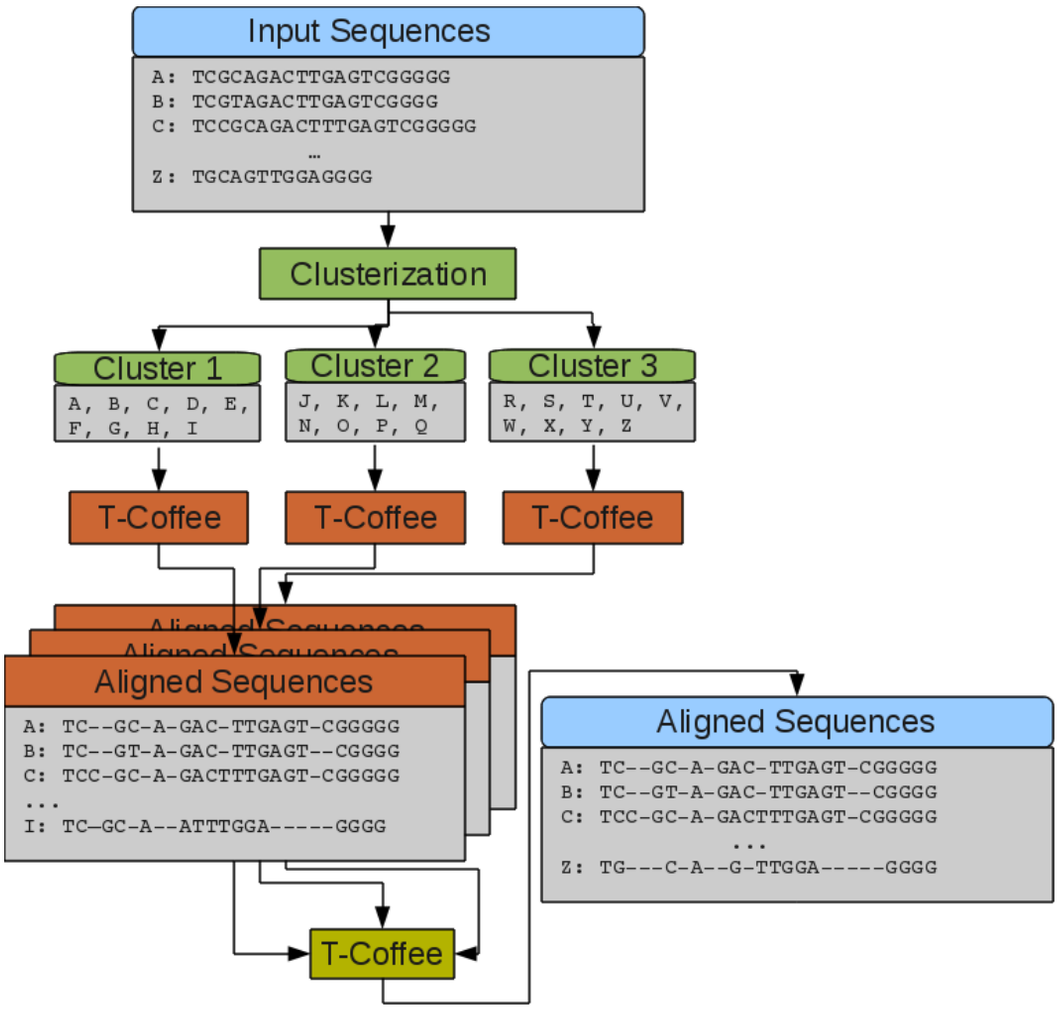


Figure 3.3: Steps of Clus-T-Coffee (CTC) process.

receive a precomputed cluster and will run the whole T-Coffee algorithm on it. At the end it will generate a partial alignment. Finally, all partial alignments are merged on a last invocation of T-Coffee.

## 3.2 Performance analysis of T-Coffee

One of the basic goals of biologists researching on MSA, is to be able to align as much sequences as possible with an acceptable quality level. On this section we are presenting a deep study of the experimentation performed in order to test

the scalability and overall performance of the implementations presented of T-Coffee, in order to study the consumption of memory usage and computing time. This experimentation is an important key in order to know the behaviour of these implementations, so we can determine how can they be improved.

### 3.2.1 Experimentation framework

In order to perform the experimentation, a set of system monitoring tools were used to extract information about CPU and memory usage. The first tool was *vmstat*, shipped by default on most operating systems. It displays average information about the global CPU and memory usage. The second tool used is *sar*, it displays more detailed information about the system, like network usage and CPU usage, detailed per core. All tests were scripted in python, thus all data generated were also processed by python scripts.

The systems used to perform all the experimentation and tests were: one of the clusters of the research group to carry out parallel tests, and one stand-alone machine to execute the serial implementations, with the following characteristics:

- The cluster used had twenty-four nodes and one frontal. All nodes were Intel Core 2 Quad Q6600 at 2.4GHz processor with four Gigabytes of RAM and one Gbit of network speed.

- The stand-alone machine was a Intel Core 2 Duo E6850 at 3GHz processor with four Gigabytes of RAM.

### 3.2.2 Generating sequences

In order to validate and test the different implementations, we need to provide us of an adequate set of input data for the MSA aligners.

Two methods were used to generate sequences:

- The first method consisted in extracting sets of biological sequences from known benchmarking libraries, thus generating a vast set of test input sequences with different properties.

- The second method consisted on random generation of sequences trying to imitate the way in which biological sequences have mutated over the history.

For the first method, we have generated a set of different input files containing from one hundred to one thousand sequences. The origin data used to generate these datasets comes from different BAliBASE sequence files. BAliBASE [TPP99b], is a benchmarking library which contains several input sequences divided in a different set of benchmarks. There is not a fixed sequence length on these tests, as different lengths were concatenated. BAliBASE is used to test the quality level of a set of precomputed MSAs. Its benchmarking tool gives an score of how good is an alignment. In our study, we are only using BAliBASE sequences for generating a set of input sequences.

For the second method, a small application *SeqGen* was written. See Appendix A for usage information. This application generates random sequences trying to mimic the mutations found in real biological sequences. Thus, the generated sequences, as being random, do not have any kind of biological meaning and are only useful for testing the scalability of the applications. The generation of sequences follows the following steps:

1. A pristine sequence is generated, with the settings defined by the user and stored in the input array.

2. A sequence is randomly extracted from the input array (on initialization, the input array will contain only the pristine sequence), and it is used for the duplication.

3. A $\delta$ number of sequences are generated by duplication of this pristine sequence.

   3.1 On the duplication process, a weight table is randomly selected for each sequence.

   3.2 Each sequence is duplicated residue by residue applying the weight table. Each weight determines the probability of the following situations:

- Normal copy: The residue is copied one to one. No mutation present.
- Duplication: The residue from the origin sequence is duplicated in the new sequence.
- Insertion: A new residue is inserted between the previous one and the current one.
- Alteration: The origin residue is randomly swapped with another residue.
- Deletion: The origin residue is deleted and then, it is not present in the destination sequence.
- Subsequence insertion: A new small subsequence is inserted at this point. This case has very low probability, but might happen.

3.3 The new sequences are stored in the input array.

4. The origin sequence is stored in the array of generated sequences.

5. Repeat the process from step 2, until the array of generated sequences contains as many sequences as the user requested.

6. Randomize the order of the generated sequences array.

For the different width of performed experiments on the described systems, different kind of sequences, with determined sequence lengths were generated. These sequences are summarized on Table 3.1, and were generated as follows:

- *Set A*: This set is formed by 20, protein based, input sequences in total for sequential analysis. The average sequence length is about 100 residues. The first pair of input test files is about 100 sequences, by incrementing its amount by 100 sequences, we generate various files, being the last ones up to 1000 sequences.

- *Set B*: It contains 280 input files, which were generated for sequential analysis containing from 10 to 140 (three different protein samples, one RNA sample and one DNA sample) incremented by 10 sequences, and using lengths of 50, 100, 150, and 200 residues on average per sequence.

- *Set C*: Formed by 396 input files for sequential analysis, containing from 100 to 1000 sequences (five different protein samples, two RNA samples and two DNA samples) incremented by 100.

- *Set D*: It contains only 36 input files. Formed each one by 2000 sequences, the lengths used were 50, 100, 150, and 200 residues on average per sequence.

Table 3.1: Summary of all input sequences used for the study

| Set | Total samples | Sample | | | Sequence length | Number of samples |
|---|---|---|---|---|---|---|
| | | First | Last | Δ | | |
| A | 20 | 100 | 1000 | 100 | 100 | 2 prot. |
| B | 280 | 10 | 140 | 10 | 50, 100, 150, 200 | 3 prot., 1 RNA, 1 DNA |
| C | 396 | 100 | 1000 | 100 | 50, 100, 150, 200 | 5 prot., 2 RNA, 2 DNA |
| D | 36 | 2000 | 2000 | 0 | 50, 100, 150, 200 | 5 prot., 2 RNA, 2 DNA |

In addition, BAliBASE input sequences were used to perform a comparative study in the quality of the alignment generated by each MSA implementation.

With these sets of sequences and configurations we carried out an experimental performance study that is reported in the following sections. A set of scripts were designed to automatically generate, configure and launch MSA tests for each implementation on a single machine or in the cluster. In the cluster was a bit more tricky, as the scripts had to purge and process data as they were generating new qsub scripts on the fly for new tests while controlling the available quota.

## 3.3 Comparative of T-Coffee with other MSA implementations

In this section, we present a comparative study of different MSA algorithms to focus on the quality of the generated alignments. The BAliBASE benchmarking package was used for such study. BAliBASE has six different groups of

benchmarks (RV11, RV12, RV20, RV30, RV40, RV50), each one with a set of reference aligned sequences. The generated alignment by each implementation is compared with the reference alignment by the BAliBASE benchmarking tool [TPP99a], and two scores are generated by each alignment:

- **Sum-of-pairs score**: This score is determined by assigning one point to each pair of residues from all pair-wise alignments, and zero points to the incorrect ones. The resulting score is the average, being one point the best score possible. This score determines if the implementation is capable to align at least some sequences from all the input set.

- **Column score**: This score is determined by assigning one point to each column which all residues are correctly aligned, zero if only one or more are unaligned. The resulting score is the average, being also one point the best score. In this case, it determines the ability by the implementation to align all sequences from the input set.

The implementations, presented in the previous chapter, used in the study along with the corresponding version are the following:

- T-Coffee version 8.99
- Clustal Omega version 1.0.2
- ClustalW version 2.1
- Kalign version 2
- MAFFT version 6.857
- MUSCLE version 3.8.31

All benchmarks were run on the Dual Core stand-alone machine.

Table 3.2 displays the sum-of-pairs score of all implementations. From this table we can observe, that the best implementation with the best scores is T-Coffee, it is followed by Clustal Omega, MUSCLE, Kalign, MAFFT and ClustalW. Considering all the sequences in all the computations, the last row, shows the proportional amount of sequences with the biggest score obtained

by each implementation. Note that the addition of all amounts does not sum one hundred percent because multiple implementations are performing alignments with the same score. We can observe from this table, that despite the computing time required by T-Coffee to achieve this score, it is the one that bypasses most in quality in comparative with the other ones.

Table 3.2: BAliBASE sum-of-pairs score comparative

| RV | T-Coffee | ClustalO | ClustalW | Kalign | MAFFT | MUSCLE |
|---|---|---|---|---|---|---|
| RV 11 | 0.62 | 0.51 | 0.49 | 0.54 | 0.44 | 0.55 |
| RV 12 | 0.89 | 0.85 | 0.82 | 0.85 | 0.82 | 0.86 |
| RV 20 | 0.88 | 0.86 | 0.82 | 0.85 | 0.84 | 0.85 |
| RV 30 | 0.79 | 0.77 | 0.68 | 0.73 | 0.74 | 0.75 |
| RV 40 | 0.80 | 0.80 | 0.70 | 0.77 | 0.76 | 0.76 |
| RV 50 | 0.79 | 0.75 | 0.67 | 0.70 | 0.72 | 0.73 |
| Average | **0.79** | **0.75** | **0.70** | **0.74** | **0.72** | **0.75** |
| Best | 64% | 17% | 5% | 7% | 1% | 9% |

Table 3.3 displays the column score for all implementations. In this case, the scores are smaller due to the difficulty by the implementation to align correctly all sequences of the input set. If we observe and compare the results obtained from this table with the previous one, we can observe that the results keep the same line of behaviour. T-Coffee continues to be the one with the high score values. This, taking into account both scores, we can confirm that T-Coffee offers a significant higher quality in their alignments compared to other algorithms.

In addition to the quality, we also carried out a comparative of the average amount of computation time and memory usage required by each implementation under study. The results are shown, in logarithmic scale, on Figure 3.4. From these results, we can observe that T-Coffee is the implementation that more memory and time needs to solve the MSA problem. On the contrary the other implementations require less time for computing the alignments. For example, the fastest implementations are Kalign and MAFFT followed by MUSCLE, Clustal Omega and ClustalW. From these implementations only Clustal Omega, MAFFT and T-Coffee present some kind of stand-alone par-

Table 3.3: BAliBASE column score comparative

| RV | T-Coffee | ClustalO | ClustalW | Kalign | MAFFT | MUSCLE |
|---|---|---|---|---|---|---|
| RV 11 | 0.39 | 0.27 | 0.24 | 0.29 | 0.20 | 0.31 |
| RV 12 | 0.74 | 0.68 | 0.64 | 0.67 | 0.63 | 0.70 |
| RV 20 | 0.36 | 0.34 | 0.26 | 0.29 | 0.27 | 0.30 |
| RV 30 | 0.39 | 0.39 | 0.25 | 0.31 | 0.30 | 0.31 |
| RV 40 | 0.42 | 0.43 | 0.30 | 0.37 | 0.34 | 0.33 |
| RV 50 | 0.42 | 0.36 | 0.27 | 0.28 | 0.32 | 0.33 |
| Average | **0.46** | **0.41** | **0.33** | **0.37** | **0.34** | **0.38** |
| Best | 62% | 31% | 11% | 11% | 8% | 13% |

allelizations (OpenMP, pthreads and fork respectively). In conclusion, when most aligners are capable of producing sequence alignment solutions in a fastest way than T-Coffee, by using less resources, we have to insist in the importance of the alignment quality, being justified the price that is paid by T-Coffee on resources, in order to achieve the quality that we have observed, as it is a key characteristic for being useful for biologists.

### 3.3.1 Scalability of T-Coffee fork model

In the previous section we have observed and compared T-Coffee with other MSA implementations, now we want to perform a deep study of the scalability of T-Coffee. In order to determine how many sequences can be aligned. Thus, we have run a set of tests in serial mode and using the *fork/wait* model. All tests were run on the stand-alone Core 2 Duo machine using a set of input sample sequences as described on section 3.2.2.

We tested the memory usage and the computation time that were necessary to be used for different number of input sequences. On an initial benchmarking, we tried to obtaining the maximum amount of sequences that T-Coffee was able to process before it exhausted all available system memory. We were able to successfully align up to 300 hundred sequences of an average length of 100 residues. Finally, given the limitation of resources, we decided to ran T-Coffee with an input data set from 10 to 140 sequences of lengths from 50

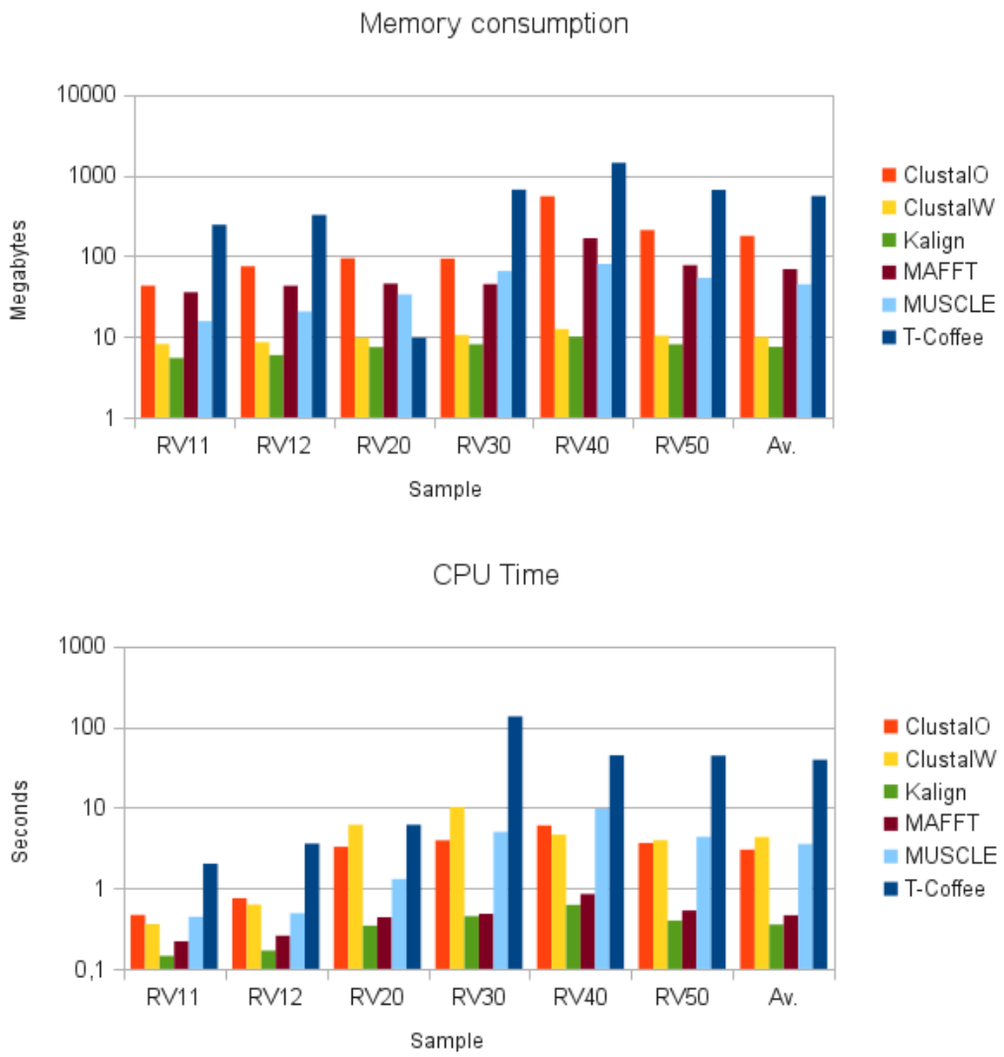


Figure 3.4: Computation time and memory consumption for MSA implementations.

to 200 residues. The results are shown in Figures 3.5 and 3.6, were it can be observed that memory usage and computation time grow in a exponential way, as expected, due to the exponential nature of the algorithm. Besides, given the seen experimental results, we can estimate that the maximum scalability of T-Coffee will be limited by the memory available in the system for a determined problem, for example we were able to align only up to 300 sequences before the system reached the four Gigabytes memory limit.

Finally, for the last test, we run both T-Coffee in serial mode (1 thread) and T-Coffee with fork/wait mode (4 processes). We fixed the number of sequences to be aligned in this test to 100 sequences, and we increased the sequence length from 50 to 200 residues. Table 3.4 shows the improvement of T-Coffee fork versus the serial execution for two cores. The second column, shows the average speed improvement, the third one shows the speed up average, and the last one the efficiency. From this table, we can observe that the *fork/wait* based implementation gives greater improvements for higher sequence lengths, as we are able to achieve better speed up and efficiency results. These results show that T-Coffee can scale quite well while increasing sequence length, and this is due to the nature of the algorithm. Adding more sequences will always increase the number of pairs of sequences to align, increasing considerably the costs of the problem, but increasing the size of these sequences will only affect on the memory usage by the system, and does not affect negatively to the parallelization of the process.

Table 3.4: Improvement of fork/wait versus serial in T-Coffee.

| Seq. Length | ΔSpeed | Speed Up | Efficiency |
|---|---|---|---|
| 50 | 15% | 1.18 | 0.59 |
| 100 | 27% | 1.38 | 0.69 |
| 150 | 34% | 1.5 | 0.75 |
| 200 | 38% | 1.6 | 0.8 |

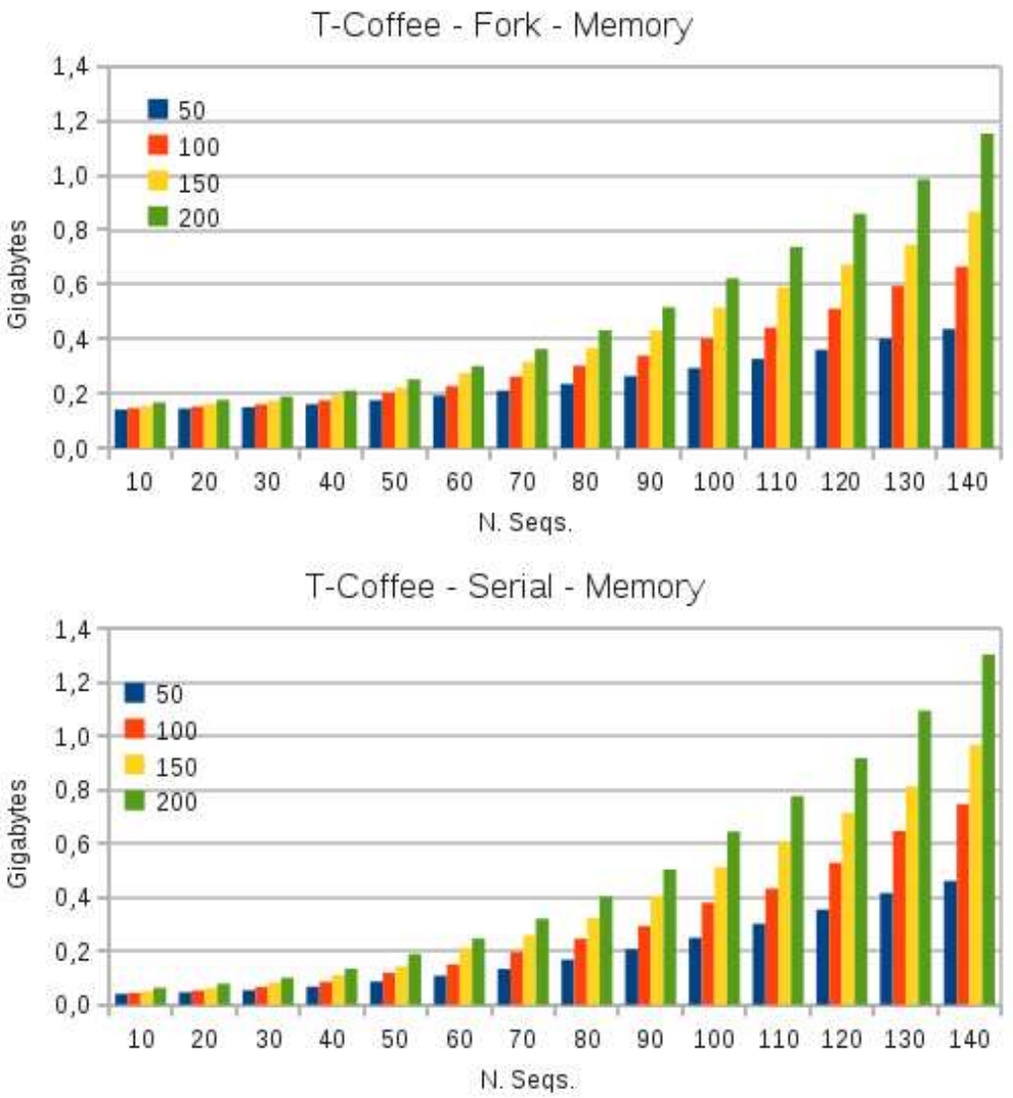


Figure 3.5: Memory required by TC to align n sequences of different lengths.

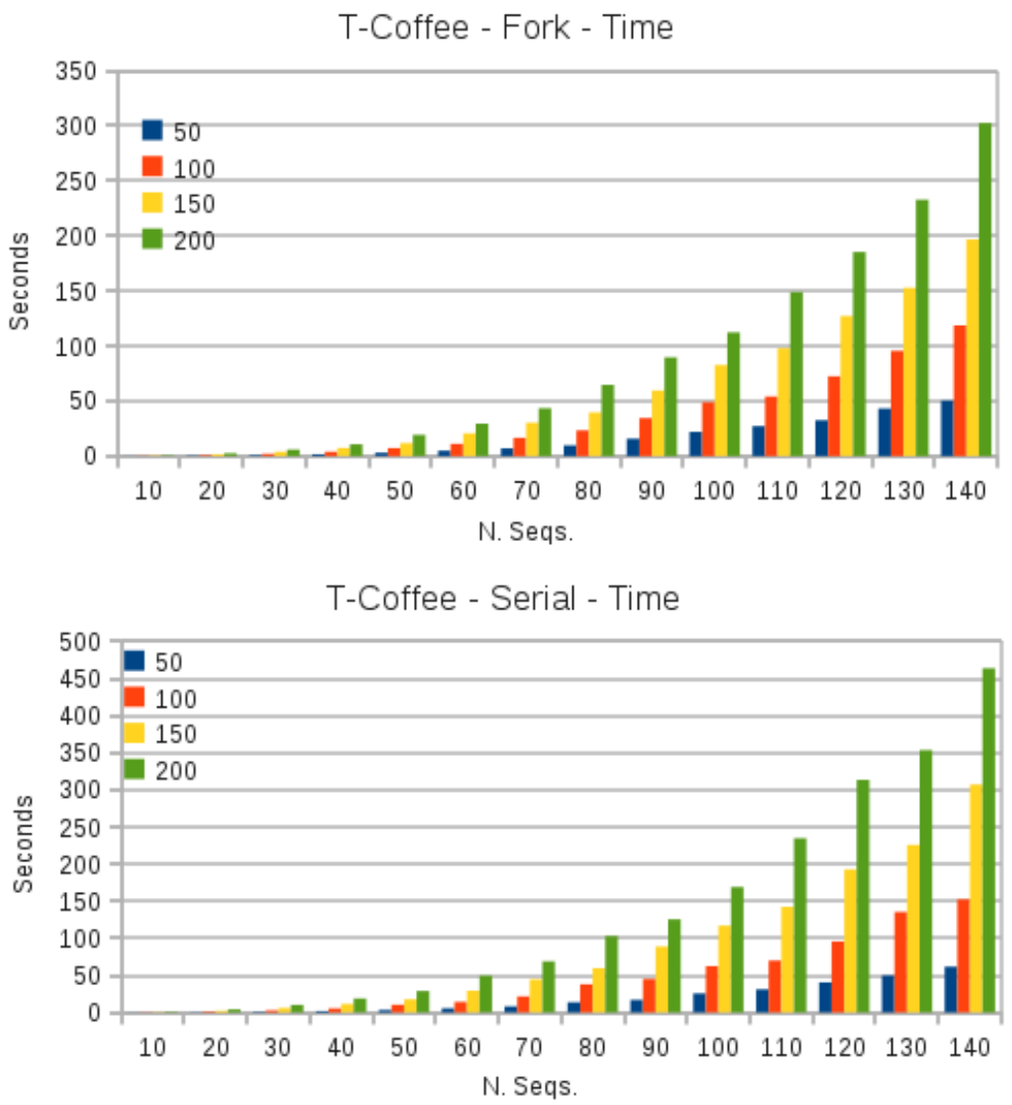


Figure 3.6: Time required by TC to align n sequences of different lengths.

## 3.4 Parallel-T-Coffee

To overcome the problems in the use of resources that became evident for serial T-Coffee, Parallel-T-Coffee (PTC) is glimpsed as a possible alternative to align a greater number of sequences.

The analysis of performance of PTC was carried out with different MPI processes in three different configurations:

- One node with four MPI processes per node (PTC 1n4p)
- Four nodes with one MPI process per node (PTC 4n1p)
- Two nodes with four MPI processes per node (PTC 2n4p)

It has to be noted that for this experimentation the nodes were limited to 1 and 2 because on the initial benchmarking tests, we found that the communications penalty time was to hight, that it negatively affected by increasing the overall computation time so much, it was not possible to solve the requested input files in a limited amount of time.

In order to determine the maximum amount possible of sequences to process, we ran an initial simple test, using an input data set of several sequences with a fixed sequence length of 50 residues. Using this input data set we were able to execute a maximum amount of 1000 sequences for the one node based configuration (PTC 1n4p) and 900 sequences for the remaining two configurations. This limitation is mainly caused by the nature of the intensive communications requirement of the implementation. Additionally the fact that we were using a 100Mbps inter-communications network introduced and important penalty, making a greater amount of sequences to be network intensive, requiring more computation time than its sequential version. Furthermore, higher amounts of sequences failed due to the exhaustion of available memory in the system. Thus the required computing time to solve the problem was higher than 48 hours.

For the second test, we used several samples of 100 to 2000 sequences (with a sequence length from fifty to two hundred residues). On this test, we have observed a curious different treatment of the DNA/RNA sequences from the protein based sequences. PTC is processing more faster DNA/RNA sequences

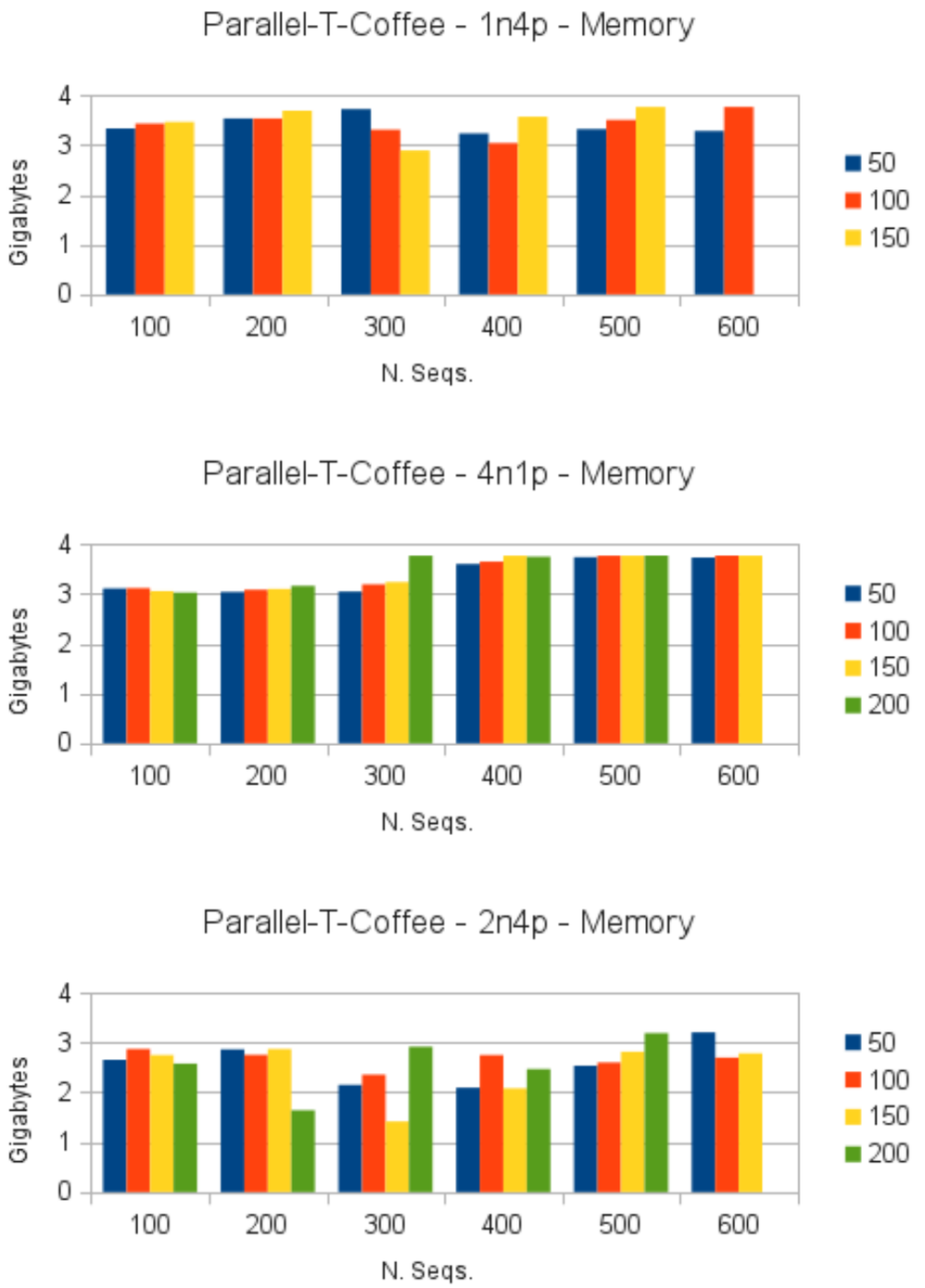


Figure 3.7: Average memory usage per node for Parallel-T-Coffee with protein sequences.

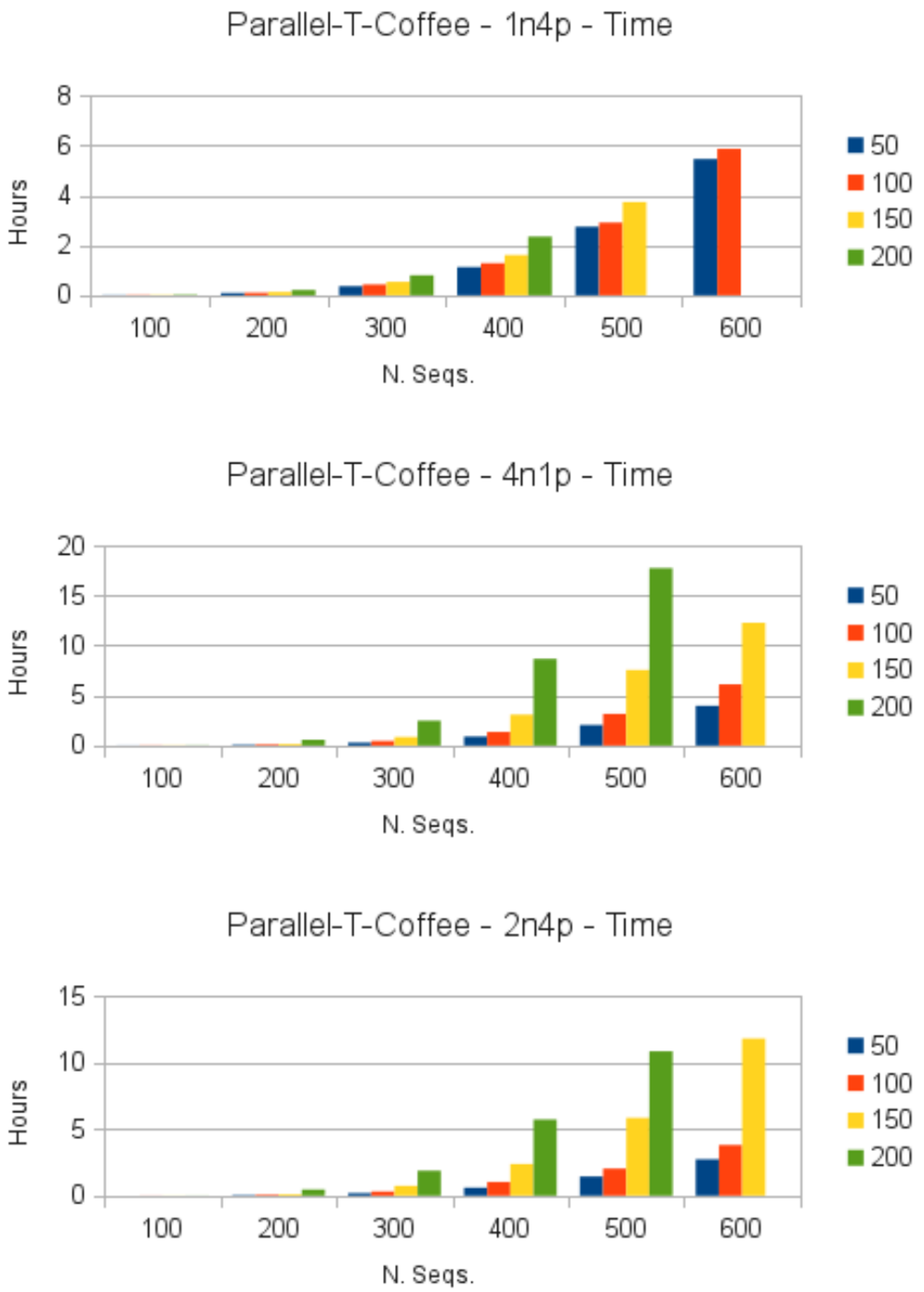


Figure 3.8: Average computation time per node for Parallel-T-Coffee with protein sequences.

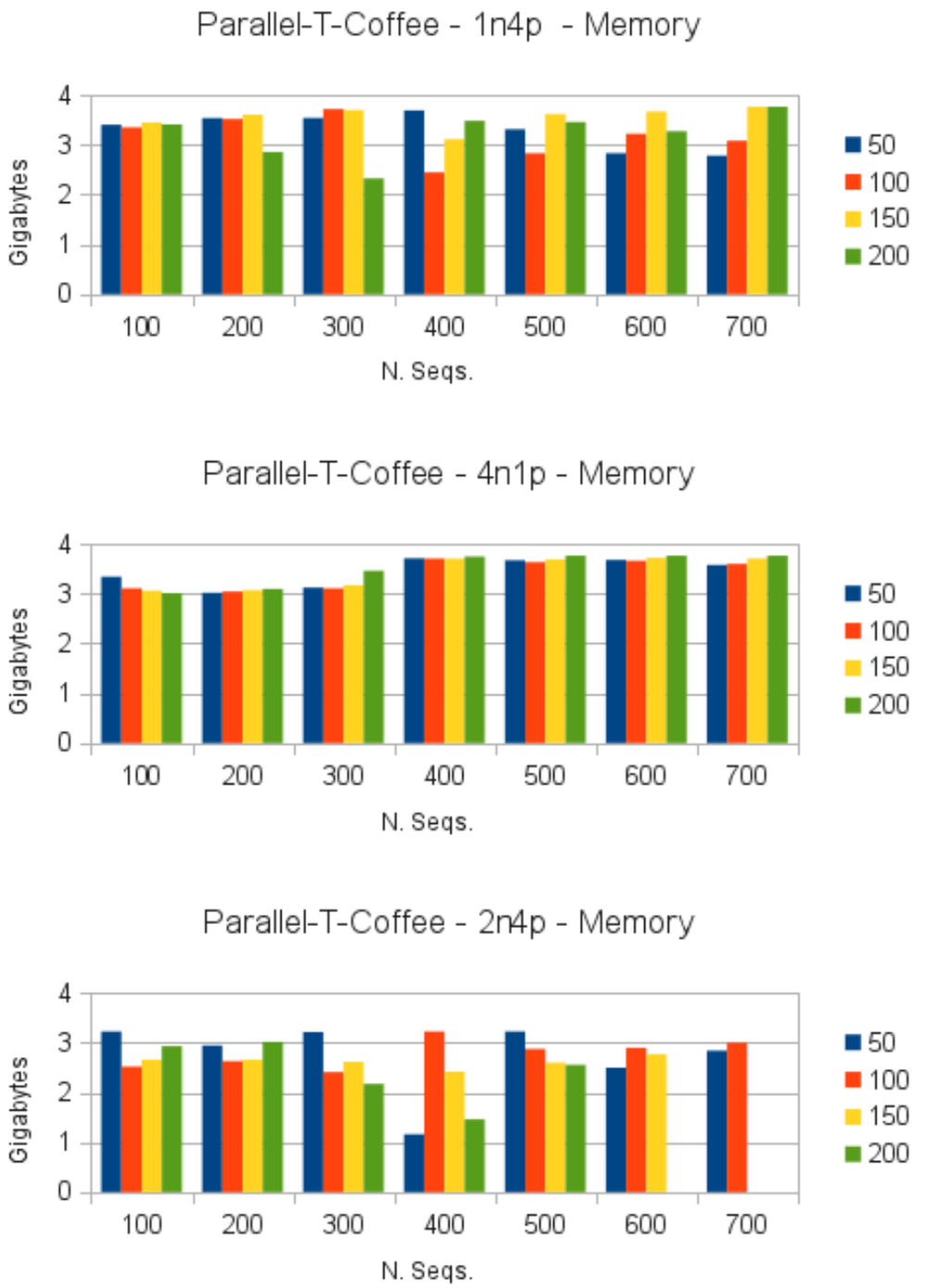


Figure 3.9: Average memory usage per node for Parallel-T-Coffee with DNA/RNA sequences.

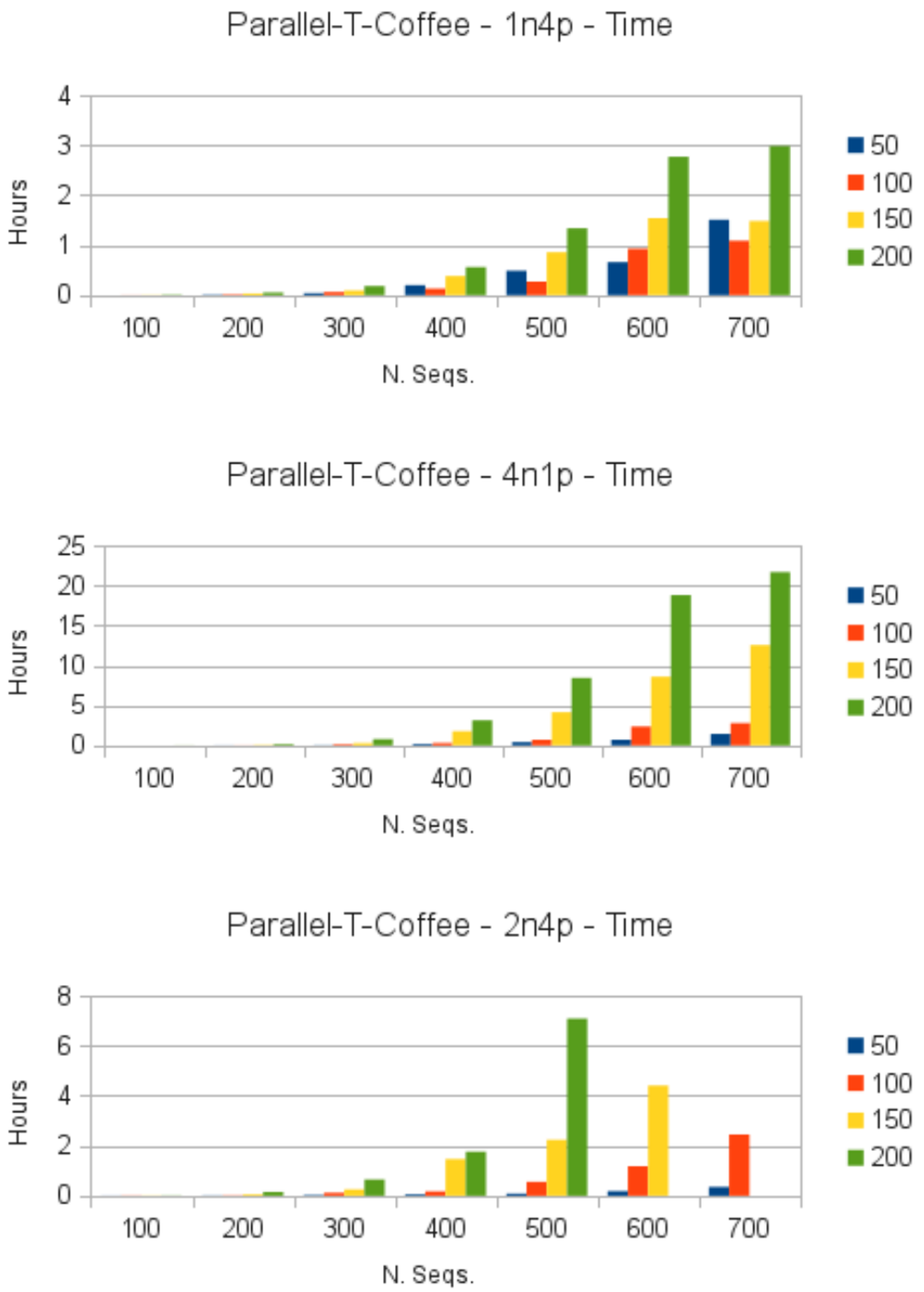


Figure 3.10: Average computation time per node for Parallel-T-Coffee with DNA/RNA sequences.

than protein sequences of the same length, while T-Coffee takes exactly the same amount of time for both types of sequences. We can conclude that PTC is compressing the four pair based sequences, or transforming them into protein sequences. Thus, by performing this operation PTC reduces in a proportion of 1/3 the DNA/RNA sequence length.

Figures 3.7 and 3.8 show the average memory consumed and the computation time for protein based sequences of different sequence lengths (from fifty to two hundred residues), and Figures 3.9 and 3.10 show the corresponding metrics for DNA/RNA sequences. In these results, we can see that memory usage maintains a stable pattern, while the computation time grows in a exponential way, due to the exponential nature of the algorithm. In addition, it is possible to see a drop in performance when comparing the computation time with one node based execution with the two and four nodes executions. This is due to the communications intensive nature of PTC (see Figure 3.2), thus the performance goes down due to the exchange of data between nodes along the different steps of the MSA process. In addition, it can be observed that PTC is capable to process more sequences than T-Coffee with the same amount of memory, due to the different implementation and memory management. Unfortunately, this implementation is not maintained any more by its authors and it is now out of date from T-Coffee current development branch.

We can conclude, that even PTC is capable of processing more sequences than T-Coffee, this implementation does not scale very well, and its computation time is higher than the sequential version, thus we obtain a very bad efficiency and, as a consequence, new more efficient version of parallel algorithms for T-Coffee have to be studied.

## 3.5 Clus-T-Coffee

The clustering version, CTC, developed in our group, was also run using four MPI processes in a single node (CTC 1n4p), four processes with one node per processor in the second test (CTC 4n1p) and four processes in two nodes in the last test (CTC 2n4p). In Figures 3.11 and 3.12, the graph shows also a exponential growth of average computation time, but memory consumption

Table 3.5: Computation time required by CTC for 2000 sequences.

| Seq. Length | Time (hours) | | |
|---|---|---|---|
| | 1n4p | 4n1p | 2n4p |
| 50 | 10.8 | 8.7 | 5.5 |
| 100 | 12.6 | 10.3 | 6.4 |
| 150 | 15.2 | 13 | 7,6 |
| 200 | 17.7 | 15.7 | 9.3 |

seems to maintain an average stable level independently of the number of sequences.

Furthermore, we can observe a slight performance gain when the four MPI processes are run on separated nodes rather than in one stand-alone node, or when we scale the execution tests to run in eight processes. On the contrary to PTC, CTC does not use communications at all for its parallelization technique, thus they don't affect performance. Finally, CTC had been able to process more sequences in less time than the other implementations, as we were able to process up to 2000 sequences, see Table 3.5, The division of the input sequences in separate clusters, reduces considerably the combinatory explosion of sequences in the pair-wise step of the process.

Although, with Clus-T-Coffee we were able to process more sequences with a reasonable amount of computing time, its implementation lacks a multicore implementation, and we think that its performance can be improved by optimizing the parallel code for a multicore based architecture. Thus, in the following chapter we propose a new implementation that should benefit from multicore systems.

## 3.6 Analysis of results

To sum up all the performed tests, Table 3.6 shows a global summary of our study. The simple test, consists on the first fast benchmark that was done on all implementations in order to find the maximum amount of sequences that can be processed. The deep test, consists on the specific study that was performed

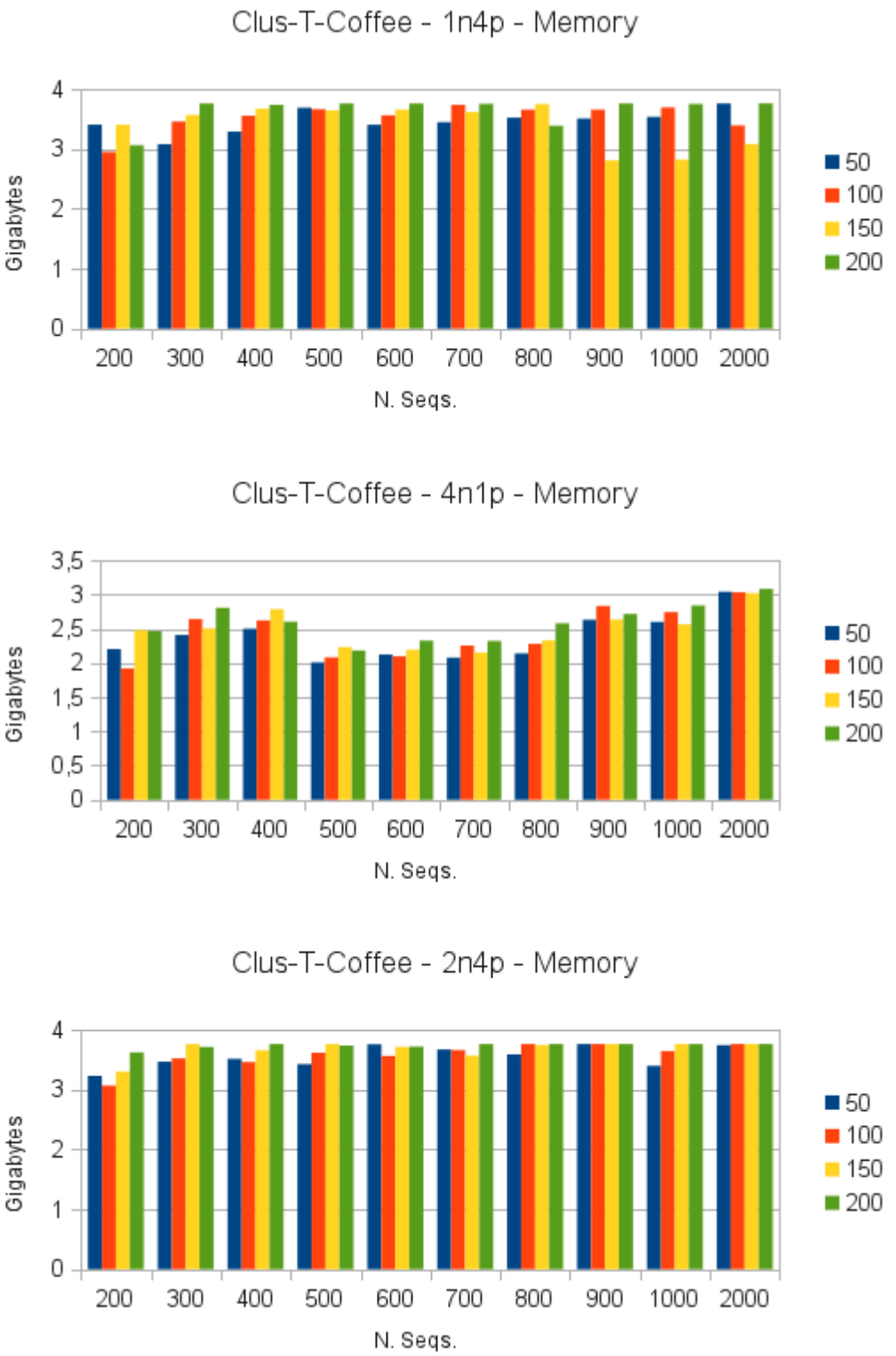


Figure 3.11: Average memory usage per node for Clus-T-Coffee.

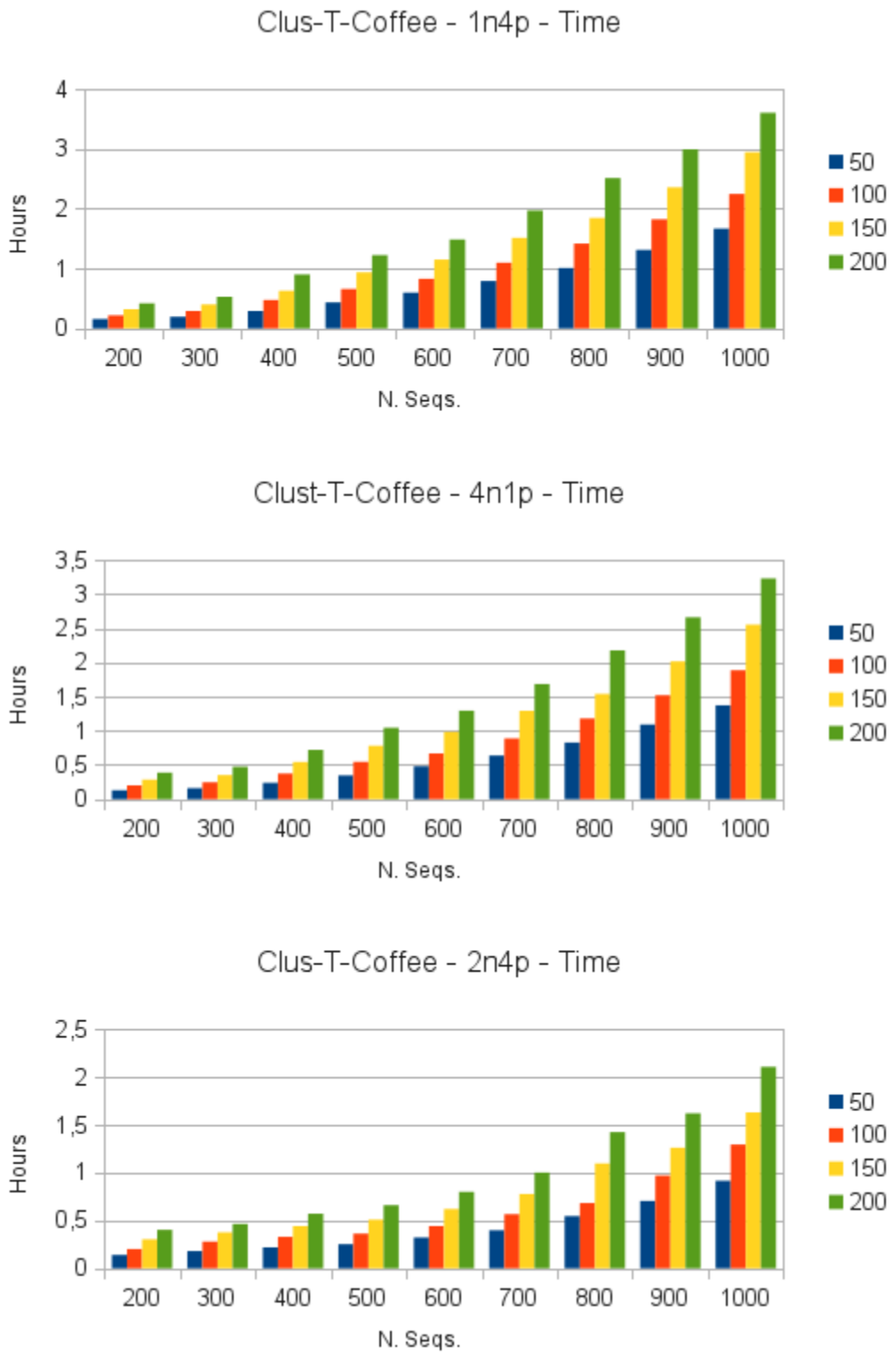


Figure 3.12: Average computation time for Clus-T-Coffee.

on each implementation, thus more sequences were generated and processed for it, and it consists on data gathered from the different tests sets described on section 3.2.2. For each test it is shown the number of MPI processes that were launched, the system configuration expressed in terms of number of nodes and number of processors per node, and the maximum number of sequences that we were able to execute for each test. Tests with a higher number of sequences failed mainly due to complete exhaustion of all the system memory, or due to exhaustion of the imposed time limit of two-three days.

As can be observed in the table, the sequential implementation of T-Coffee is able to process a maximum of 300 sequences. As we saw in the previous sections, although this number is not displayed in the resulting graphics obtained from the second deep analysis, we were able to align this amount of sequences on the first simple test. On the other way, on the deep test we only processed up to 140 sequences due to the limitation of time and resources that we had during the development of the experimentation, but more sequences can be achieved and will be achieved in future tests. The parallel versions PTC and CTC are able to increase this limit up to different values depending on the implementation. These differences between the parallel implementations are caused by two reasons. First, on PTC implementation, as seen before, memory handling of the constraints library allows a bit more of space for processing more sequences. Second, when it runs with more than one node, the penalty of the message passing communications is so high that it increases considerably its processing time, causing the expiration of the time limit that we set on all tests. CTC handling of memory and communications is different than in PTC, due to the way the problem is divided into a subset of different parallel problems, but its limitation to process sequences is due to some stability problems in the current implementation. Thus stability of CTC decreases when the number of processes is increased.

In order to improve the memory usage, and to reduce the overhead of the current available parallelizations, we propose to optimize them using a multithreading library such as pthreads as we expose in the next chapter.

Table 3.6: Relation of the different tests that we performed

| Test | N. Procs. | Test | Max. Seqs. |
|---|---|---|---|
| Simple | 1 | TC | 300 |
| Deep | 1 | TC | 140 |
| Simple | 4 | PTC 1n4p | 1000 |
| Deep | 4 | PTC 1n4p | 700 |
| Simple | 4 | CTC 1n4p | 2000 |
| Deep | 4 | CTC 1n4p | 2000 |
| Simple | 4 | PTC 4n1p | 900 |
| Deep | 4 | PTC 4n1p | 700 |
| Simple | 4 | CTC 4n1p | 2000 |
| Deep | 4 | CTC 4n1p | 2000 |
| Simple | 8 | PTC 2n4p | 900 |
| Deep | 8 | PTC 2n4p | 700 |
| Simple | 8 | CTC 2n4p | 2000 |
| Deep | 8 | CTC 2n4p | 2000 |

CHAPTER 4

# Pairwise alignment

After the deep study performed on T-Coffee MSA and other implementations based on it, on this chapter, we are presenting a couple of solutions in order to improve its scalability. We will start by analysing the principal issues and bottlenecks of T-Coffee, and two different solutions are presented and discussed. Firstly we will discuss about rewriting the T-Coffee based fork model into a pure threading model. Subsequently we will present a new aligner based on T-Coffee which combines the threading model with a distributed memory model.

## 4.1 Implementation issues of T-Coffee

As we have seen in the previous chapter, one of the main problems of the current T-Coffee implementation is the limitation of the amount of sequences that can be processed imposed by the system memory limit. Thus, the most obvious solution path is to increase the system memory. Furthermore, we have seen that parallel implementations are sometimes able to process more sequences

in one single machine than in several machines, due to the different memory management implementation used in PTC, or the different approach used to solve the problem seen in CTC. However, these parallel implementations of T-Coffee present an important drawback due to the overhead of communications of the message-passing implementation. This makes no benefit in having the parallel solution. Thus, in the present work, we faced the challenge of defining and deploying a new parallel implementation of T-Coffee based on a combination of the message-passing and the multi-threading paradigms.

In order to migrate the current T-Coffee *fork/wait* based implementation, we carried out a deep study of the internal code of T-Coffee in order to identify possible issues that, a priory, are limiting its performance. We studied a set of issues related with the thread safety of the code. At the first glance, we could see that T-Coffee is not thread safe, due to several issues:

- Usage of non-reentrant system calls. When more than one thread performs the same call simultaneously, a synchronization issue arises, as non-reentrant system calls are used to use static memory assignment. When this happens, multiple threads may modify the same static variable at the same time, consequently the program will enter into an unpredictable state where it may give wrong results or crash due to a memory access violation error.

- Usage of static variables. In the beginning, T-Coffee was designed for a sequential usage, thus a lot of variables with an static scope are used in all core functions of the process. The purpose is mainly the performance impact that such technique offers to T-Coffee, so this kind of optimizations are for letting the sequential version to be more efficient. A static variable, is a variable that remembers the last value that it had from a previous call to the function, thus when it is used in a parallel environment, where multiple threads access to the same function at the same time, the value stored by one thread my be altered by another one, when the first thread still expects its original value. Thus, this technique is discouraged in multi-threaded programming.

- Shared memory synchronization issues. Each child instance heavily uses

the same global memory, thus in the *fork/wait* model each child process has its own copy of all the data structures, T-Coffee uses file based I/O operations to communicate with the child processes. On a threaded environment all data structures are shared and accessed simultaneously causing serious synchronization issues between threads.

As for Parallel-T-Coffee and Clus-T-Coffee, we have seen that parallelization is achieved by using the message passing library MPI, thus on a multi-core machine the different cores are used by multiple MPI processes. On all these parallel implementations we are not using a shared memory model such as threads. By this way, the overhead generated by the sub-process creation and by the interprocess communications should be reduced by applying a different model, more suitable for a multi-core system. Thus, we proposed a reorientation of the implementation of the parallel versions of T-Coffee in such a way that they rely on the best of the studied implementations by combining a message passing technology such as MPI with a threading API such as pthreads.

Regarding communications usage in Parallel-T-Coffee, as we have seen in the experimentation, it suffers a considerable penalty in efficiency when comparing a one node execution with an execution in multiple nodes endangering its scalability, thus Parallel-T-Coffee may work better with higher numbers of sequences. At the same time that we have seen that Clus-T-Coffee does not work for smaller amounts of sequences (less than one hundred).

Finally, Clus-T-Coffee has several stability issues on the memory management, increasing the probability to crash when the number of nodes is increased.

In the following sections we will discuss how these issues have been addressed.

## 4.2 Introducing multithreading

The purpose is to benefit from the current *fork/wait* parallelization in T-Coffee, by introducing a threaded shared memory model in order to use more efficiently a multicore system [MRH+11] [MnRG+13]. Besides, we want to combine this

proposal with the usage of a message passing (MPI) implementation. By running T-Coffee in a cluster based environment, we increase considerably the average memory available in the system, thus we can process more sequences.

The gain in efficiency justifies the usage of a threaded model in substitution of the present model. Some authors have performed some efficiency comparative studies of the application usage of the threaded model versus the fork based model, and they coincide that the threaded model achieves better performance than the other one [KC99]. Although this does not report anything new to the T-Coffee algorithm, thus it is a reimplementation, the new contribution is a new prototype implementation that will run more faster than the original one.

In order to migrate the current implementation, we have to address the thread safety issues enumerated in the previous section:

- Usage of non-reentrant system calls: Most non-reentrant system calls offer a reentrant implementation available in the standard C library. The migration of these calls is a simple trivial operation.

- Usage of static variables: The most extended issue on the code, is the usage of static variables as global ones in all its core functions, thus a complete rewrite of the code would be needed. As a possible solution for this issue, we have substituted the static variables with a mechanism that mimics the same functionality: a memory container. All static variables have been contextualized inside this container, in order to keep track of the state of the variable associated to a specific thread. On all affected functions, the static variables are substituted by normal ones, and its state is recovered from the container at the beginning of its execution, and it is stored back at the end of the function.

- Memory management: We have found a lot of synchronization problems due to incorrectly referred memory locations accessed from the wrong threads. T-Coffee has its own memory management system, mainly for debugging purposes. The memory management code has been modified in a way that each memory allocation keeps track of its own thread caller identification. Then, the code performs checks over the memory usage

in a way that it avoids and reports illegal access to memory regions of other threads. This verification code, causes a considerable penalty on the efficiency on all the memory allocation code. Finally, as we come from a fork based implementation, the default implementation relies on the automatic release of allocated memory on child process termination, but the threaded implementation does not. In order to address this issue, we keep track of the memory allocated by each thread, and we automatically release this memory on thread termination.

At current time, all these issues have been addressed only in the first stage of the T-Coffee process, on the functions related with the pair-wise alignment step of T-Coffee, as it is the most time consuming one. For the study, we have compared our modified version of T-Coffee, which its usage is described on Appendix B, with the original one without any modifications. The only modification performed on the original code was to set a timer in order to measure the time consumed by the pair-wise step.

To study our implementation, we have run several tests using different generated sequences:

- Starting with 10 sequences, and by incrementing in amounts of 10 sequences until reaching an amount of 140.
- For each sample, we generated 4 sets, with 5 sequences each one, of different lengths, being 50, 100, one 150 and 200 residues.
- The total amount of generated sequences was: $14 * 4 * 5 = 280$ sequences.

For this test the scoring was ignored, as the generated alignments by the threaded version will be the same than the ones from the original one, thus we are not performing changes to the algorithm at all. As a safe guard consistency check, all generated alignments are compared against the original alignments generated by T-Coffee, ensuring that they remain the same and have not been corrupted by our implementation improvements. The tests were run on the Core 2 Duo stand-alone machine as described on section 3.2.1. We measured the execution time for T-Coffee in serial mode, the fork based model and our threaded prototype. Figure 4.1, shows the average time required to perform

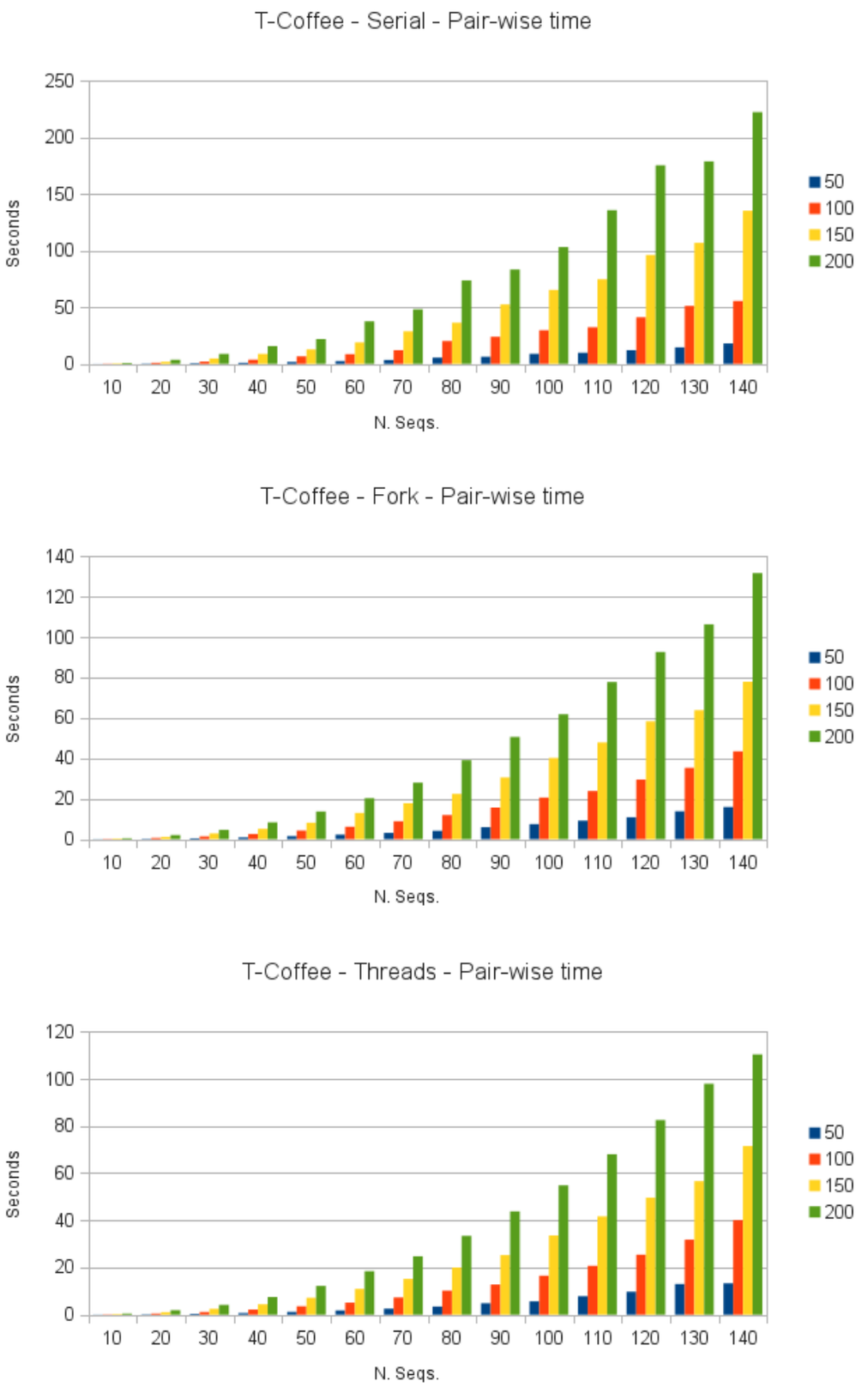


Figure 4.1: Computing time required to process from 10 to 140 sequences in the pair-wise step by different implementations.

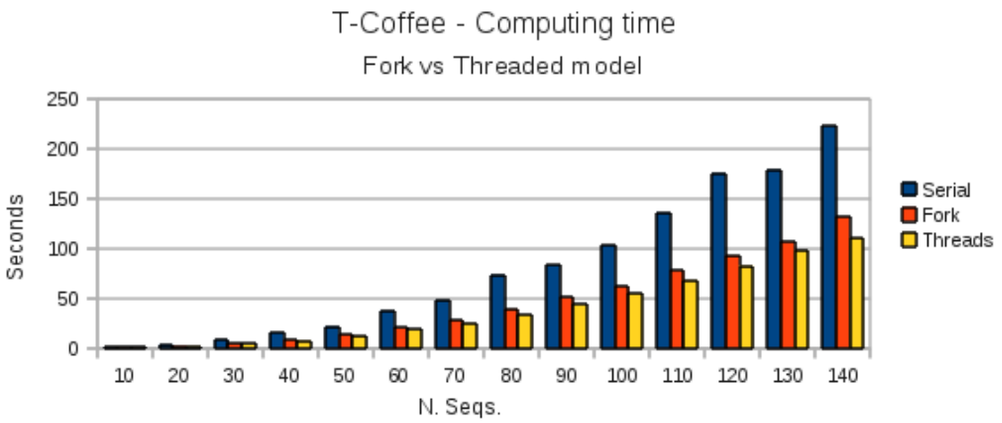


Figure 4.2: Computing time required to process n sequences in the pair-wise step.

the pair-wise alignment step for sequences of different lengths by each implementation (the different sequence lengths are represented by different column colors, thus we have four columns of different lengths per sample). As we saw on the previous chapter, all graphs have the same tendency, showing the scalability of T-Coffee, and how it scales better with higher sequence lengths. On the graphics we can see the overall computation time used by each implementation, and how the threads version achieves better timings than the fork and serial versions.

Besides, Figure 4.2, displays the comparison of all implementations for the specific sequence length of 200 residues. There is a slight improvement in the computing speed, as our implementation, for 200 residues, runs 10% faster than the T-Coffee fork model and 49% faster than the serial T-Coffee, when the fork implementation runs only 43% faster. The ideal speed, as we are in a dual core machine, would be an improvement of the 50%. So, we are next to this ideal.

Table 4.1, shows the Speed Up and Efficiency of fork T-Coffee and our threaded implementation for different sequence lengths. From this table, we can conclude that our implementation is being able to achieve a 0.98 efficiency in a dual core machine in contrast with the 0.88 efficiency achieved by the default *fork/wait* version of T-Coffee. This demonstrates that a multithreaded

based implementation has better efficiency than the other model, and that we are able to obtain a quite good efficiency.

Table 4.1: SpeedUp and Efficiency of T-Coffee implementations in the pair-wise step.

| Seq. Len. | Fork | | | Threads | | |
|---|---|---|---|---|---|---|
| | ΔSpeed | Speed Up | Efficiency | ΔSpeed | Speed Up | Efficiency |
| 50 | 10.3% | 1.13 | 0.56 | 27.9% | 1.4 | 0.7 |
| 100 | 30.6% | 1.45 | 0.72 | 41.2% | 1.71 | 0.86 |
| 150 | 38.3% | 1.62 | 0.81 | 46.2% | 1.86 | 0.93 |
| 200 | 43.3% | 1.77 | 0.88 | 49% | 1.96 | 0.98 |

## 4.3 Introducing a new pair-wise implementation

In order to solve the issues that we had with the development of the threaded version of T-Coffee, we studied the possibility of implementing a new version from scratch on the pair-wise step based on the T-Coffee algorithm and source code. With this new implementation we can concentrate and focus more on performing optimizations for a fully parallel version for current HPC systems.

The overall computation cost of MSA is exponential[WT94], being the pair-wise alignment step the part which is determinant in this cost. Studying the scalability of MSA helps us to determine how much we can obtain from parallelizing MSA and to which limit we can take this parallelization into account. Thus, the generation of an efficient parallel implementation of the MSA process necessarily implies an improvement of the global pairwise alignment step.

Since the pair-wise step, is one of the most compute intensive steps of the MSA process, we are going to be focused on the parallelization of the construction of the primary library, the pair-wise initial step of T-Coffee. The algorithm proposed to carry out pairwise sequence alignment is focused on the exploitation of current distributed systems such as clusters, where several nodes coexist, each with multiple cores. Thus, the implementation combines the message-passing paradigm to exploit distributed memory among nodes

with thread programming to use the shared memory between the internal cores of each node.

In order to implement our distributed version of the pair-wise aligner [MnRH13], we are using a message passing library (MPI), in conjunction with Linux phtreads library. Pthreads provides a fast an efficient way to parallelize in a multicore based machine, while MPI is one of the most spread and used paradigms for parallelizing applications in cluster based environments. For the programming language, we need to use a fast an efficient language capable of running all the computation operations as fast as possible. Scripted and or virtual machine based languages such as Python and Java would have a considerable penalty in the calculations, that is the main justification of the usage of C or C++ by the programmers in the development of the studied MSA in this work. We want our code to be easily interoperable with T-Coffee, since T-Coffee is written in C, we have also written our prototype implementation in C.

## 4.4 Parallel pairwise alignment implementation

Our implementation of the parallelized pairwise alignment is based on the Smith-Waterman [SW81] basic algorithm, as it is an extensively used implementation. Furthermore, we use the MPI message-passing library to distribute the work across an arbitrary sized cluster. In a later step the algorithm spawns as many threads as each node is capable of running, which is determined by its hardware capabilities. As it is said, the threads are implemented using the standard pthreads library, as it provides an efficient and simple way to parallelize on multi-core machines.

Figure 4.3 shows the basic steps of our implementation, whose functionality is presented below:

1. Input sequences are parsed and loaded into the memory by a master task, and the system is queried to determine the total number of nodes, maximum memory available and number of available cores per node.

2. It is created one MPI process per node and all possible pairs of sequences

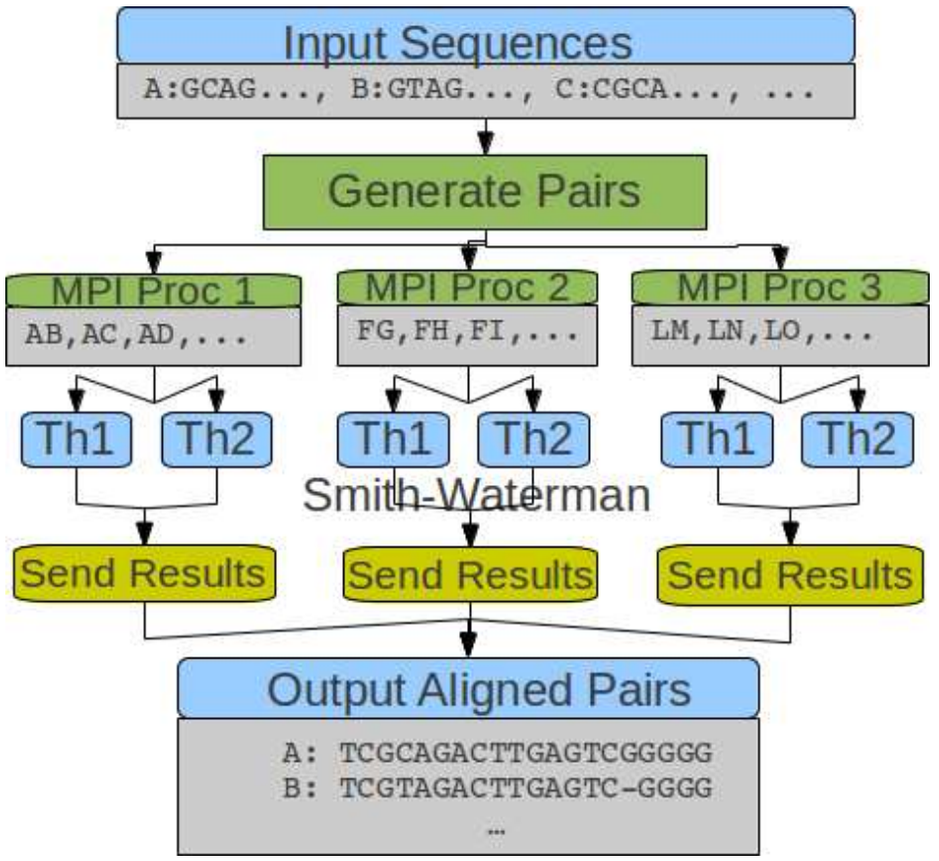


Figure 4.3: Parallel pairwise alignment steps.

are generated and distributed across them.

3. Each process spawns a worker pool of $c$ threads, where $c$ is the total number of cores available in a given node.

4. Each thread carries out the Smith-Waterman algorithm for each pair of sequences that has been assigned to.

5. All threads and processes are joined and all aligned pairs are merged into a final list.

The implementation of the algorithm that generates all possible pairs assumes that each pairwise computation requires similar computation times. This way, we can assume that all threads should process the same number of pairs. Besides, we should send the same number of pairs to each node to be

processed. Thus, we propose a method to assign the right number of sequences to each node and each core in order to balance the computation among them.

The model that we are going to use in our prototype is a message passing (MPI) model with a threading library, thus we will have two levels of parallelization:

- First level: Distributed memory parallelization using MPI.
- Second level: Shared memory parallelization using pthreads.

Figure 4.4 shows the parallelization levels used in our implementation, on the first level we will run up to $n$ MPI processes, and in the second level we will run up to $c$ threads per node. The current threading level is determined on each MPI process, so if a mixed environment is found, each node will spawn as many threads as possible, but this will cause that nodes with an high threading level to terminate earlier than the other ones, and this situation is not yet contemplated in the code.

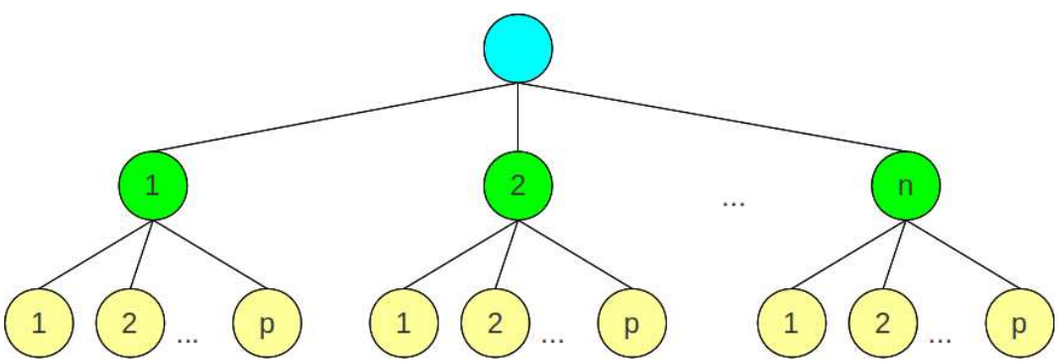


Figure 4.4: Tree representing the parallelization levels used in the prototype.

The problem that we want to solve is the pair-wise alignment of all possible pairs from the input file. Although parallel versions of most pair-wise algorithms are available, in our problem we need to parallelize hundreds of pairs, so we are using a sequential implementation of the pair-wise algorithm and performing in parallel the pair alignment of multiple pairs at the same time. In this way to solve the problem, we need to perform communications between the different nodes and the master only at the beginning and at the end of the

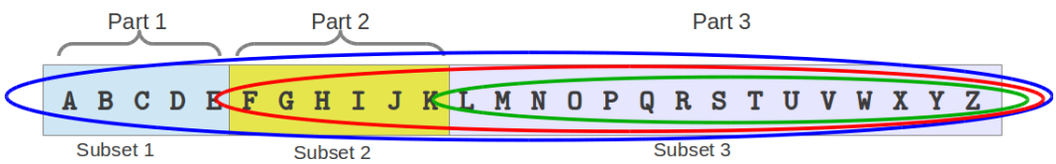


Figure 4.5: Pair generation algorithm

job. In the first layer, a master/worker model was implemented, where the master process distributes all input sequences to the workers to be aligned, at the end the workers sent back all aligned pairs to the master. For the second level of parallelization, each worker will split up the work in a pool of worker threads. The amount of worker threads will be determined by the maximum number of processes that can be run in parallel in the machine.

Our proposed method consists of dividing the input list of sequences into different parts in such a way that they generate subsets with similar numbers of pairs of sequences to be aligned. A subset is created with all combinations of sequences from part $i$ with all the remaining sequences. We illustrate the division method with 26 sequences A-Z in Figure 4.5, distributed into three parts. The sequences assigned to each part are: $part1 = A, B, ..., E$, $part2 = F, G, ..., K$ and $part3 = L, M, ..., Z$. Then the worker process of $part1$ will compute the subset of pairs made up of all combinations from A to E with all sequences A-Z. For $part2$, it will compute all combinations from F to Z (the A-E sequences have already been computed by the previous subset). Finally, the last subset will consist of all remaining L-Z sequences. With this partition, there will be three subsets with 115, 105 and 105 pairs of sequences.

Given $n$ sequences, the total number of pairs generated will be: $\frac{n^2-n}{2}$ pairs. In order to distribute all these sequences, while trying to minimize the data transmission of the sequences as far as possible, while keeping the same average computation time per thread, we propose the following distribution method:

- Let $f$ be the parallelization level for our problem, which means the number of parts our problem will be divided into. Under optimal conditions, each node will process up to $\frac{n^2-n}{2f}$ pairs of sequences.

- Given $n$ sequences, we calculate the size of each part, called $p$, that will

balance the number of pairs in each subset. Considering the $n$ sequences ordered in a list, the first one has to be aligned with $n-1$ sequences in order to generate all the possible combinations. The second has to be aligned with $n-2$ sequences and so on. Thus, considering a size of $p$ sequences, the corresponding subset will have a number of pairs that can be calculated as:

$$\Sigma_p n - p = \frac{p(2n-1) - p^2}{2} \tag{4.1}$$

- By assigning (4.1) the total number of pairs, and solving the equality (4.2), we obtain (4.3)

$$\frac{p(2n-1) - p^2}{2} = \frac{n^2 - n}{2f} \tag{4.2}$$

$$p = \frac{-2n + 1 \pm \sqrt{(2 * n - 1)^2 - 4(\frac{n^2-n}{f})}}{2} \tag{4.3}$$

From the two possible solutions, only the natural number smaller than $n$ will be considered valid. This $p$ indicates the number of sequences to be assigned to each worker task for a given f.

For example, for a parallelization level $f = 4$ nodes, and a problem with size $n = 100$ sequences, we obtain 4095 pairs to compute, composed by total of 1237 pairs per node with an addition of 2 remaining pairs.

Thus the parallelization partition size for the first node will be: $p = \frac{-199 \pm \sqrt{(199)^2 - 4(\frac{9900}{4})}}{2} = 14$ sequences, the corresponding amount of pairs would be 1295. For the second node, applying the same equation, but with a parallelization level of $f-1$ and a new problem size of $n - partitionsize$, we obtain a new partition size of 16 sequences with 1240 pairs, for the third node we obtain a partition size of 21 sequences and 1239 pairs, and for the last node the partition size would be 49 with 1176 pairs. This means that the master node will have to send to the worker nodes only up to 86, 70 and 49 sequences to each node.

Using this partition schema, the master task will only send up to 100, 86, 70 and 49 sequences to the corresponding 4 workers, thus minimizing the communications and balancing the computation cost of worker tasks.

As another example, for $n = 1000$ and $f = 8$ we obtain: 499500 total pairs distributed in 62437 pairs per node plus 4.

- Node: 1 (master), partsize: 65, send: 1000, numpairs: 62855
- Node: 2, partsize: 70, send: 935, numpairs: 62965
- Node: 3, partsize: 76, send: 865, numpairs: 62814
- Node: 4, partsize: 84, send: 789, numpairs: 62706
- Node: 5, partsize: 95, send: 705, numpairs: 62415
- Node: 6, partsize: 112, send: 610, numpairs: 61992
- Node: 7, partsize: 146, send: 498, numpairs: 61977
- Node: 8, partsize: 352, send: 352, numpairs: 61776

An additional aspect that is worth considering is how tasks are created in our method compared with the method used in the pairwise alignment step of classic MSA implementations. A job list with all possible pairs of sequences is precomputed sequentially in an early stage. This computation has a cost of $O(n^2)$. This cost is reduced to $O(p)$ with our proposed partition based algorithm, where $p$ is the total number of threads that can be expanded by the system.

By default the prototype, which its usage is described in Appendix C, reads and parses all sequences form a plain text file, the parsing process of the input file has a cost that increases considerably with greater amounts of sequences, at the same time the uncompressed input files can use up to ten and more megabytes of disk space (11 Megabytes for 20000 sequences). As soon as the prototype has parsed all sequences, it will perform a compressed binary dump of all these sequences in an indexed binary file, described in Appendix D, for future invocations of the program.

### 4.4.1 Pair-Wise implementation and validation

Currently the prototype is using the Smith-Waterman algorithm for computing all the pair-wise alignments, T-Coffee uses a combined method were a global and a local alignment algorithm is used together in order to determine the best score. For validating the implementation, it is necessary to compare the resulting alignments with a tool which generates similar using the same algorithm such as FASTA, in this case the FASTA toolset is used to analyse and validate the correctness of these alignments.

In order to test the viability of our implementation, we performed different tests to determine the impact of a pure distributed memory model on performance, compared with our hybrid distributed/shared model. A different set of experiments was performed to test and evaluate the computation time and memory use. We obtained a selection of input files with numbers of sequences varying from 400 to 1400 with sets of sequence lengths ranging between 100 and 200 residues.

The environment used consisted of the previously defined 24 nodes cluster 3.2.1. Moreover, the configuration used for each execution consisted of using different combinations of the hybrid distributed-shared memory parallelization model (MPI+threads) always using all the cores of each node. Thus, in a Quad Core node with four cores available we always ran a total of four processes/threads. This allowed us to configure three scenarios in each node:

- p4t1: 4 MPI processes with one thread each.
- p2t2: 2 processes with 2 threads each.
- p1t4: a 4-thread process.

In all the samples, the average processing time was calculated by measuring the total time required by the implementation. The total memory usage on each node was measured by measuring all memory allocations. The results show the total system memory by adding the memory measures of each node.

Firstly, we studied the impact of the level of message passing processes used per node for a different set of sequences. As a representative case, Figure 4.6

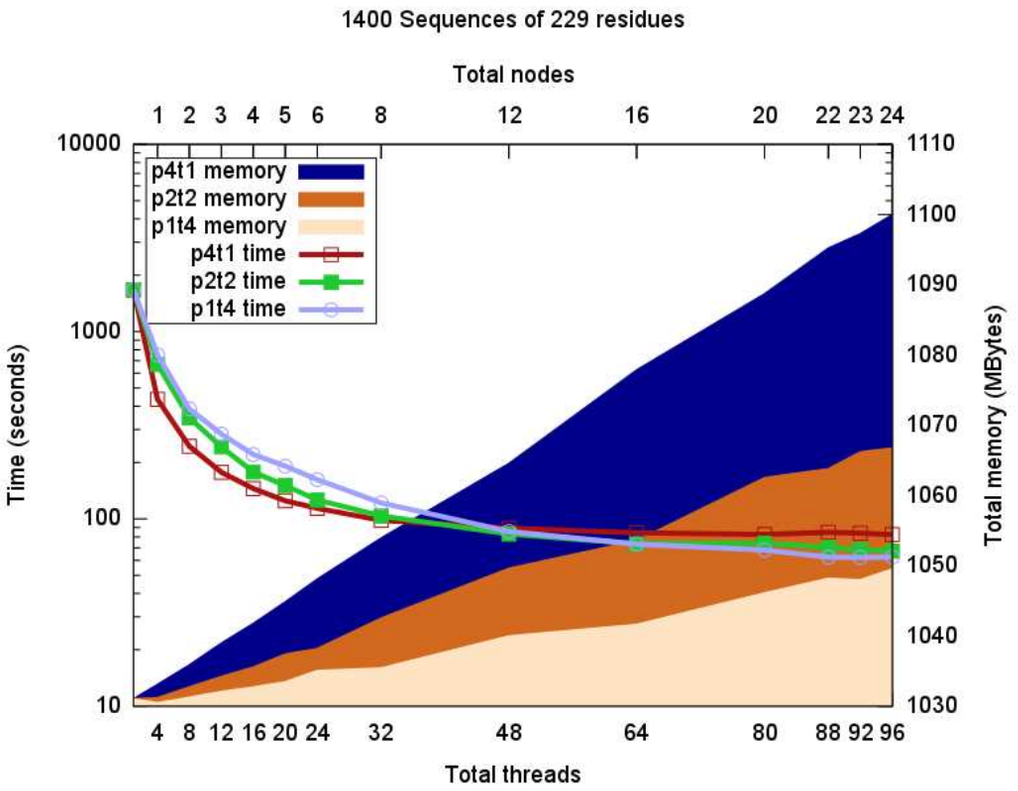


Figure 4.6: Memory usage and processing time for 1400 sequences with 229 residues.

shows the average total memory usage and processing time required to compute 1400 sequences with 229 residues in length for different levels of parallelization.

The graph shows the total number of threads (bottom) launched in the cluster and the nodes that were used (top). Four processes/threads were always run on each node, as the nodes had four cores. As can be observed, in all the cases, the computing time decreased as both the threads and MPI processes were increasing. However, it reached a limit, in this example at 12 nodes, where the time tends to stabilize and decreases insignificantly. Moreover, it is possible to observe how the computation time graphs are inverted after this point. It is observed how the p4t1 goes from having the best computation time to become the worst after 32 threads. This was caused by the overhead of the increased number of MPI processes.

Regarding the total average memory it can be seen that it grew in line with the number of expanded threads, and the implementation with the highest consumption was the one running four MPI processes per node. Otherwise, running a full threaded version (p1t4) saves around 50 Megabytes of memory compared with p4t1. This difference is not very high for current systems, but will increase with the rise in the number of processes and sequences.

Figure 4.7 shows the corresponding SpeedUp from the obtained results for a deeper analysis of the computing time. These allow us to see the limit up to which it is interesting to expand more threads and MPI processes using this parallel pairwise method. Thus, for this test of 1400 sequences, is was not necessary to use more than 12 nodes (48 threads), as that is the limit where the SpeedUp stabilizes. The use of more nodes would cause an inefficient use of system resources. Additionally, it can be observed that until this point of 48 threads, the implementation p4t1 is the one that provides more efficiency.

Other experiments consisted of studying the average computation time and memory usage for different sets with a fixed length, together with the total number of threads. As a representative threading level, in Figure 4.8, we display the results corresponding to launching two MPI processes per node (p2t2), with two threads per process, for sets with different number of sequences from 400 to 1400 sequences. Computation time is directly related to the number of input sequences and grows exponentially with the rising number of sequences. Furthermore, the initial computation time in all tests drops considerably when the number of threads is increased until it stabilizes after a certain point. This method achieves reductions in computation time. For example, we can see an improvement from 1000 seconds to 100 seconds for 1400 sequences. With regard to memory usage, it is possible to observe how this grows exponentially in proportion to the number of sequences. Besides, the usage stays stable despite the number of threads and the linear growth is caused by the overhead of running extra processes.

According to the results, we can conclude that the parallel version developed for the pair-wise alignment step of T-Coffee is able to fully use the whole cluster without scalability problems. Additionally, we are able to significantly decrease the computer time, as the level of parallelism is increasing. However,

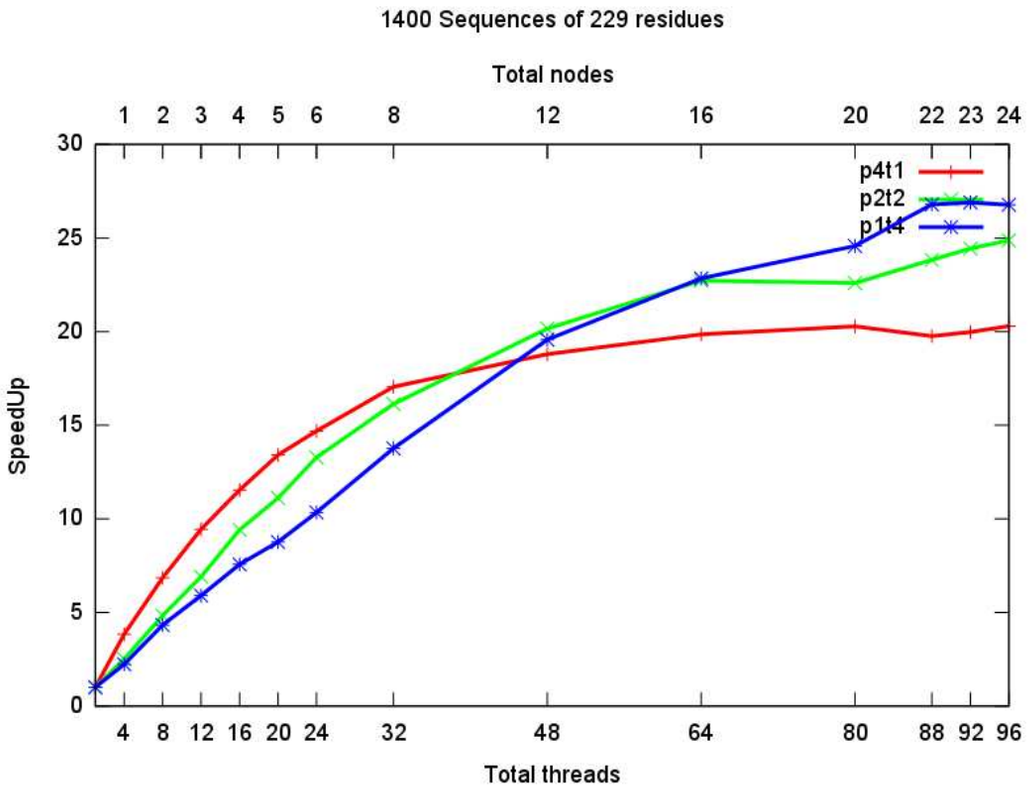


Figure 4.7: SpeedUp for 1400 sequences each with 229 residues.

there is a certain point, at which the increase in the amount of resources produces only slight improvements in computation time. In the next chapter, we carry out an analytical study to identify this point, previous to execution, in order to use the system in a efficient way.

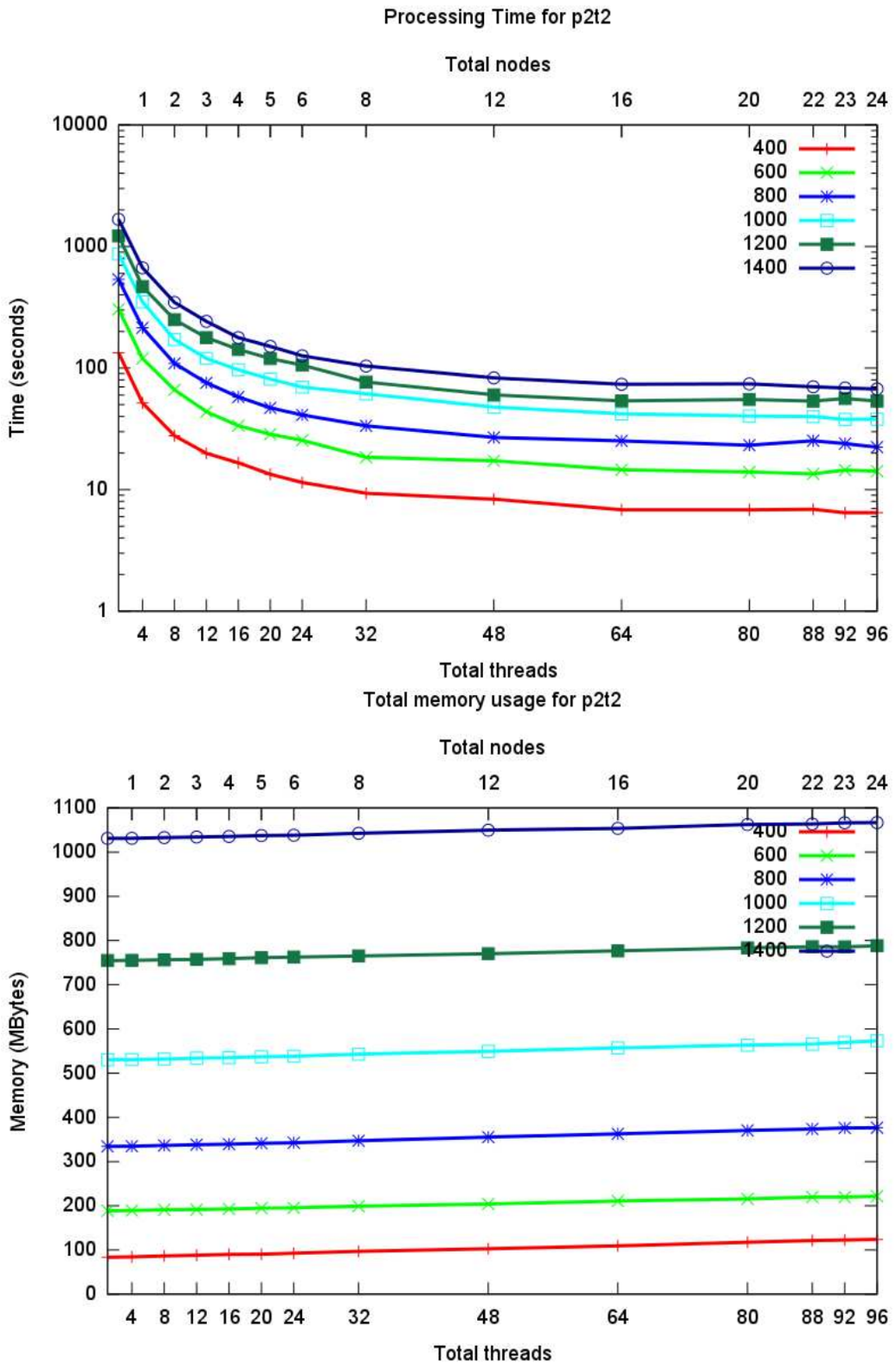


Figure 4.8: Average processing time (seconds) and total memory usage (megabytes) for set of sequences from 400 to 1400 of a fixed length of 229 residues. Using 2 processes with 2 threads per node.

CHAPTER 5

# Resource scheduling

In this chapter, we are going to define a method for resource scheduling and usage of the cluster by the sequence alignment algorithms. This method will aid in determining the correct amount of resources that will be used by a determined problem input, and will aid in determining the best cluster configuration that should be used in order to obtain the best performance possible.

## 5.1 Motivation

When one needs to solve a problem of a determined complexity, a set of several questions will arise on such as which amount of memory and computation resources should be given to the application. Providing the suitable amount of resources will ensure the efficient use of the available infrastructure, assuring that no more than necessary is spent in the MSA process.

In this chapter we develop a method that estimates the average amount of resources needed to solve the pair-wise alignment problem [MRH14b] [MRH14a]. Given a set of several input sequences, by changing different input parameters,

we study the scalability and sensibility of computation usage and how the overall resources are used. As seen in previous chapters, the implementation always reaches a scalability limit where by adding more resources to the system the overall computation time remains stable. Thus, our goal is to find the maximum amount of resources to give to the application in order to ensure an effective usage of them.

To do this, we propose a method to estimate the number of computation resources needed to solve the problem, given a set of input sequences, in such a way that an efficient use of resources could be achieved. Finally, we compare the experimental data with the estimated prediction given by our proposed model.

## 5.2 Resource scheduling methodology

In this section, we are going to present an analytical method used to determine the amount of resources required for its optimal use when solving the pair-wise alignment into the global MSA.

We assume a computing platform based on a hybrid architecture where cores in the same node have a shared memory access while different nodes act in a distributed memory fashion through message-passing. The proposed method distributes the pairwise alignment work by mapping to each available core and node a partition of sequences from all possible combinations, by distributing the total computation time in the best uniform way across the system.

We consider a number of $n$ sequences of similar length to be aligned. Then, the number of pairwise alignments to carry out will be $\frac{n^2-n}{2}$, which corresponds to the total number of pairs. Assuming a balanced distribution of all pairwise alignments into the cores of the system and considering that a single pair alignment is entirely done in a core, the total pairwise computation ($t_{pwc}$) can be calculated with expression (5.1), where $pwt$ is the time needed to carry out a single pairwise operation and $f$ is the total number of cores.

$$t_{pwc} = pwt \times \frac{n^2 - n}{2f} \tag{5.1}$$

The expected communications overhead ($t_{comm}$) of the message passing stack, for the distribution of their sequences to all nodes and the gathering of all the aligned pairs, can be calculated with expression (5.2), where $\ell$ is the average sequence length in bytes, $bw$ the transmission speed in bytes per second and $k$ is the number of nodes of the system.

$$t_{comm} = \frac{\ell \times n \times k}{bw} + \frac{\ell \times (n^2 - n)}{bw} \tag{5.2}$$

The total computation time $t$, given in (5.3), corresponds, at maximum, to the sum of total pairwise computation and the communication time. It has also to be added a value $k_o$ that corresponds to the remaining application overhead used to activate the pairwise tasks and reading sequences from disk. The communication time between the cores in the same node is considered to be negligible.

$$t = t_{pwc} + t_{comm} + k_o \tag{5.3}$$

Regarding the memory, the expected total amount of memory ($mem$) that will be used by our implementation can be defined by (5.4), were $k_{sm}$ is the average size of required data structures for one pair of sequences and $k_{th}$ the overhead penalty for each additional thread. In average, we consider the size in memory of two aligned sequences as $3\ell$, taking into account that the aligned sequences can add gaps inside. Additionally it has to be added the storage of the $n$ original sequences for each of the $k$ nodes.

$$mem = \frac{n^2 - n}{2} \times (3 \times \ell + k_{sm}) + f \times k_{th} + n \times \ell \times k \tag{5.4}$$

From equation (5.3), by making $k = f = 1$, we can calculate the speedup as the sequential computation time ($pwt \times \frac{n^2-n}{2}$) divided by the parallel computation time $t$. According to this, the efficiency is calculated with expression (5.5) as the obtained speedup with respect to the optimum one. This permits us to determine the expected number of resources needed for a specific input problem.

$$\textit{Efficiency} = \frac{pwt \times bw + 2 \times \ell}{pwt \times bw + 2 \times \ell \times f} \tag{5.5}$$

This means that the efficiency directly depends on the length of the input sequences and the computing time required to align one pair of sequences. From this expression, it can be calculated the recommended number of cores to achieve the desired efficiency.

As seen on the previous chapter, depending on the size of the problem, we saw that the sequence alignment algorithm, reaches a limit of scalability, were adding more and more resources, will not have any notable effect on the required computation time. The capability of looking for the effective amount of computing and memory resources, in order to obtain the maximum efficiency, is going to be vital to correctly configure the HPC system, to obtain this maximum efficiency with the correct amount of resources. Using less resources may require more computing time to obtain the desired result, when using more will not have any effect. It is going to be a key, in order to minimize the economical costs of running the aligner on HPC based systems.

## 5.3 Experimental results

Different tests were performed in order to analyse the behaviour of our presented method, thus proving the correctness of the prediction model used to evaluate and determine the best parameters for a specific input. In these experiments we are measuring the total computation time and memory usage of the application and comparing it to the model that defines the expected behaviour of the experiments. Different sequence sets from 400 to 1400 sequences of lengths varying from 94 to 229 residues were used.

The first step is to get a set of input sequences of specific lengths used for the evaluation of the algorithm. The method used to obtain these sequences is to extract them from sets of sequences available from different public databases. The PFAM [PCE+12] database was used as the main input of sequences. Different sequence sets from the PFAM database were randomly selected for the input data. On these sets, sequence lengths varies from 94 to 229 residues, and contains thousands of sequences. Furthermore, using these sets as our source for sequences, we got sub-sets from 400 to 1400 sequences for being used as the input sets of our implementation.

The platform used to perform executions consists of the 24 node cluster previously described on section 3.2.1. Moreover, the configuration used for each execution consists in running one MPI process per node, expanding as many threads as available cores on the node. For example, one execution may consist on expanding 5 MPI processes on 5 nodes with 4 threads each one, thus achieving a total number of 20 threads.

The total memory usage is measured on each node by measuring all memory allocations by providing replacements to the standard allocation library. The displayed results show the total system memory by adding the memory measures of each node.

Figure 5.1 shows the memory usage for different input sets of sequences, from 400 to 1400 of size 229 residues varying the number of threads/nodes used in the execution. The first observation, is the considerable increase of memory usage in function of the amount of sequences to be processed. Additionally, it can be observed that in all the cases, incrementing the number of threads have an impact of increasing the memory about 50MB from the sequential execution (one thread) to the 96 threads one. As we can see the memory usage is quite constant indifferently on the amount of nodes and threads expanded, its increase is only due to the parallellization code overhead term $k_{th}$ described on 5.4. Thus in the following example we will depreciate this term.

In order to validate the correctness of the previously seen expression (5.4), we are going to perform a comparative of the previously seen graphic with the table 5.1. The average size of the data structures used to allocate the sequences is about 400 bytes, thus, we can consider that $k_{sm} = 400\ bytes$. As previously seen from the experimental results, we can depreciate the impact of the $k_{th}$ term, as it will grow very little in comparison with the other terms, as it is not dependent in any way on the size of the problem. This permits us to estimate quite close the required amount of memory that will be used by any amount of configured threads. We will compare the expected memory usage with the memory consumed with 1 node running 4 threads. On the obtained resulting table, we can compare the results, and conclude that the experimental results are quite close to the results theoretically obtained.

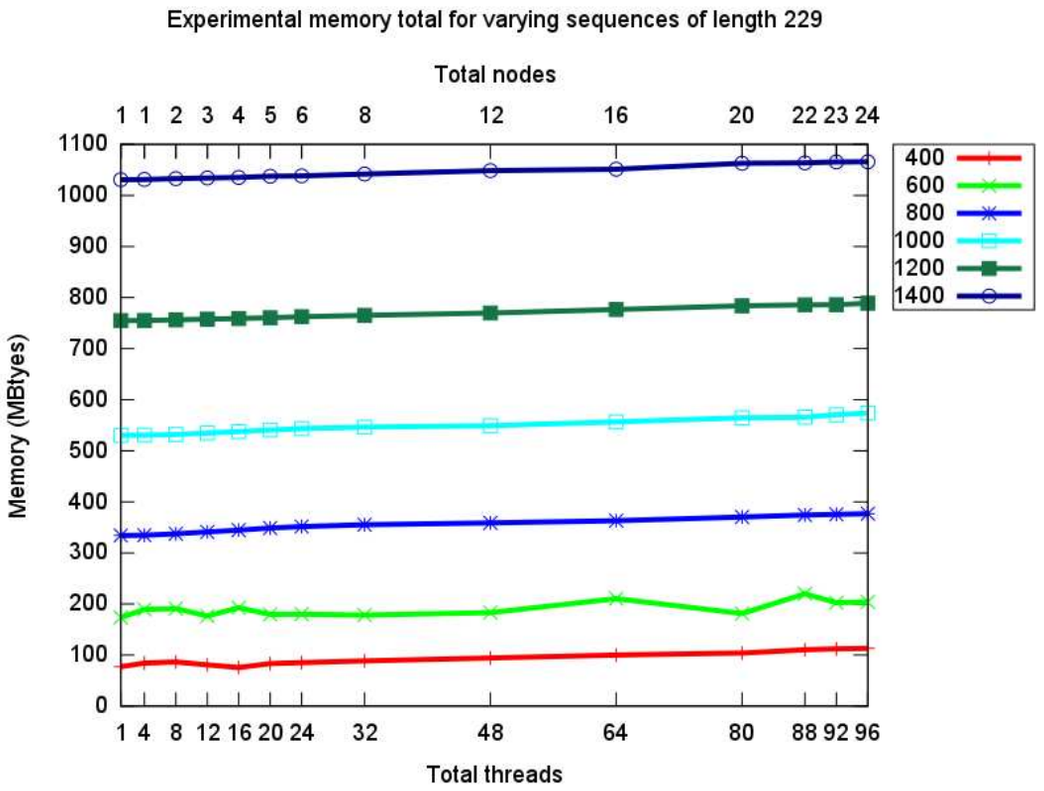


Figure 5.1: Memory usage for different amounts of sequences.

In all samples, the average processing time is calculated by measuring the total time required by the application since it loads the sequences from disk until all the results are written back to disk by the master process.

Figure 5.2 shows in logarithmic scale the execution time, per 1400 sequences with 229 residues varying the number of threads and measured in seconds, with the following graphs: $t_{pwc}$: the execution of Smith-Watermanm, $t_{comm}$: the communications time, and $total$: the total time. The predicted graphic is generated from data gathered from the prediction method using the aforementioned equation models. As it can be observed, the simulated graphs are a good prediction of the behaviour of the algorithm. The predicted graphs is quite close to the experimental one, being very little the difference seen between the graphics. Communications overhead stays stable on both graphs, being constant in the simulation because the time to transfer $\frac{n^2-n}{2}$ pairs of sequences is fixed indifferently on the number of nodes. The computation time

Table 5.1: Comparison of experimental memory usage versus the predicted one

| Num seq. | Memory MB | |
|---|---|---|
| | Experimental | Predicted |
| 400 | 94 | 92 |
| 600 | 196 | 196 |
| 800 | 343 | 341 |
| 1000 | 539 | 527 |
| 1200 | 762 | 755 |
| 1400 | 1038 | 1025 |

tends to decrease with the number of nodes, but the total computation time tends to stabilize to a common value.

Figures 5.3 and 5.4 shows in logarithmic scale the computation time, in seconds, corresponding to the execution of the Smith-Waterman step in each core. It is also shown the corresponding SpeedUp. On the first two graphs, we varied the number of sequences and fixed the length of them to 229 residues, and on the second ones, we varied the length of them from 94 to 229 with a fixed number of 1400 sequences. As it can be observed the computation time increases in proportion of the number of sequences to be aligned and the amount of residues. In all the cases, increasing the number of threads entails a reduction of computation time. As we can see in the SpeedUp graph, it shows how the algorithm correctly scales, meaning the computation balance is correctly distributed among the nodes.

In the same way, Figure 5.5 shows the computation time and the corresponding SpeedUp for a fixed number of sequences ($n = 1400$), varying the length of them from 94 to 229 residues. As can be observed, the length has a similar impact, in computation time, since time increases with increasing length, and it decreases when the number of threads and nodes increase. The SpeedUp, shows how the sequence length also scales for longer sequences. But, it also tends for the stabilization.

Figure 5.6 shows the average time invested in the communications layer of MPI, for different numbers of sequences, varying the number of nodes. It

can be seen that the time invested in communications is proportional to the number of sequences as it is also established in equation (5.2). However, it can be observed in the experiment that this time is stabilized from 8 nodes to 24. Indifferently of the number of nodes, this time tends to remain stable and depends on the problem size, as it will determine the amount of data to transfer. This is displayed as constant line on the right graphs.

Finally, Figure 5.7 shows the sum of total computation time required by the expression (5.3). We can see that the time stabilizes to a certain point were adding more nodes will not improve the overall time, thus we can see that the impact of the communications are affecting negatively to the Efficiency endangering the scalability of the algorithm. From the prediction expression (5.5), with a target efficiency of 0.5, we determined that the best number of nodes for 400 sequences is 20, between 600 and 1000 is 24, for 1200 is 28, and lastly for 1400 is 32 cores. Thus, we can see that experimental results are not so far from the expected ones.

This experimentation, with its comparison with the methodology used for predicting the best amount of resources, should perfectly validate that the methodology is closely matched by the real obtained results. This methodology, can be used in conjunction with the sequence alignment process, in order to determine how many resources should be used for an specific problem, in order to solve it. Thus, we would need to run a set of initial calibration/benchmarking tests on the target cluster, in order to calculate the different cluster specific constants, and from this data we can lately configure the required amount of resources to run the requested algorithm.

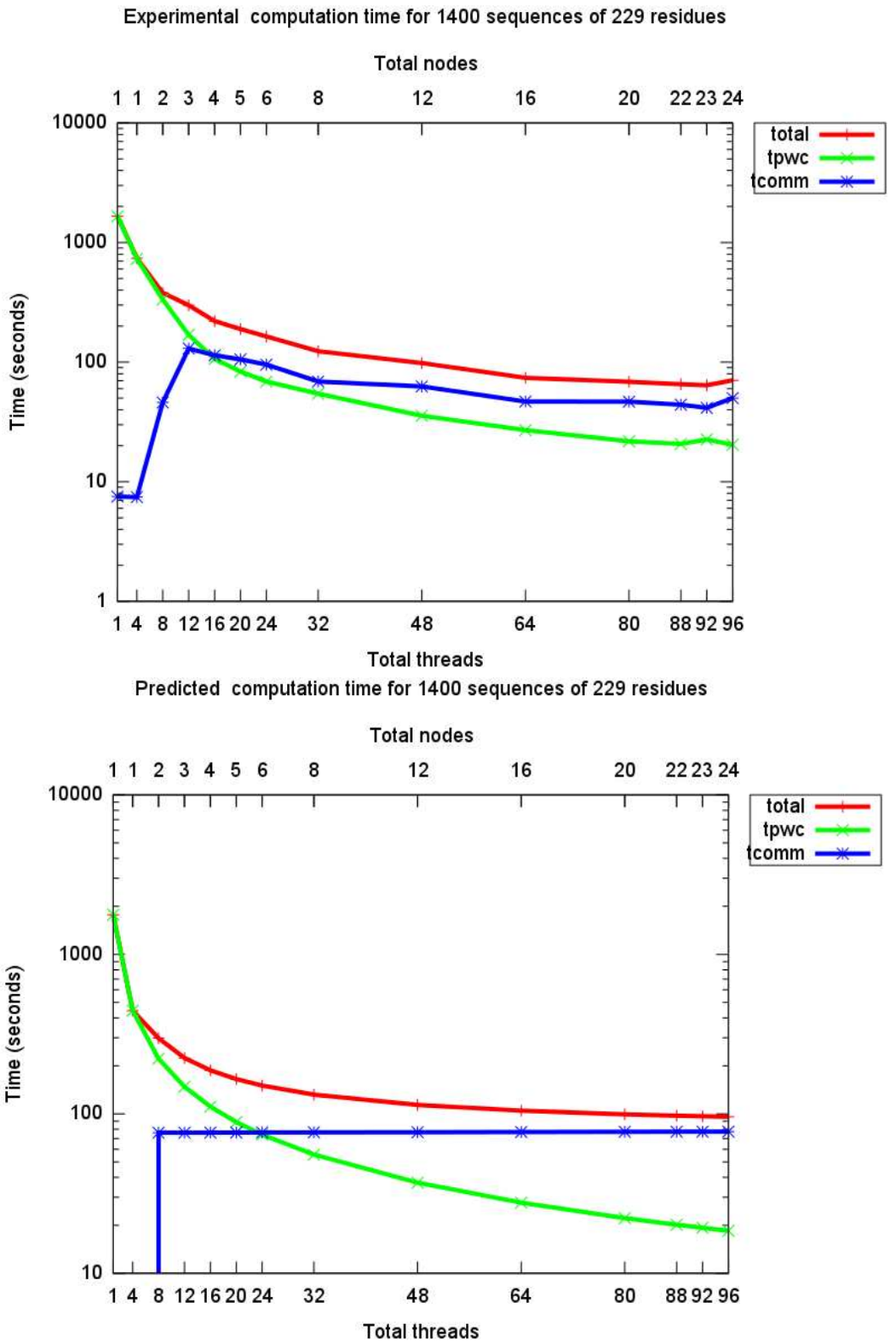


Figure 5.2: Experimental and simulated execution time.

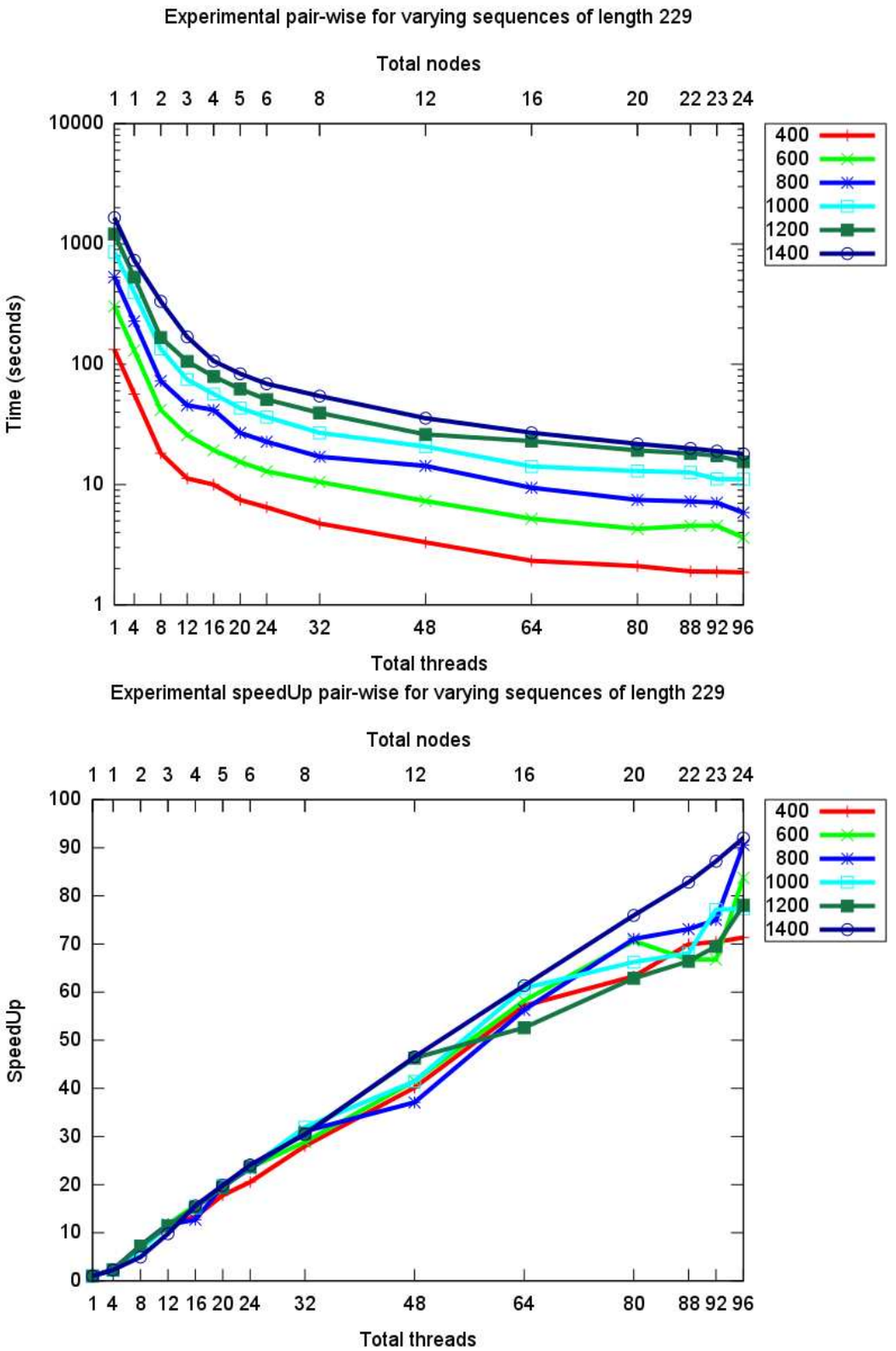


Figure 5.3: pairwise time and SpeedUp varying the number of sequences.

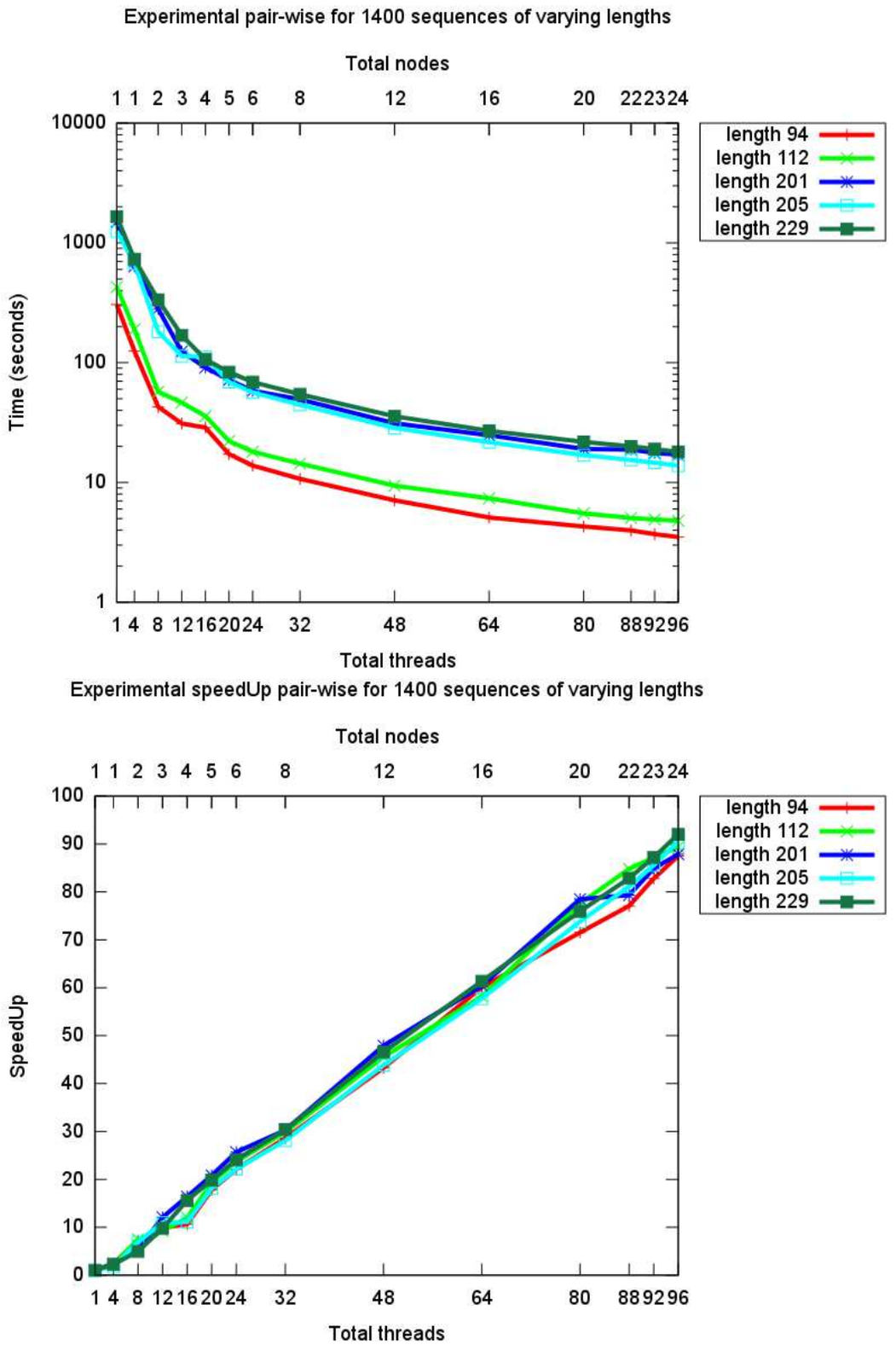


Figure 5.4: pairwise time and SpeedUp varying the sequence lengths.

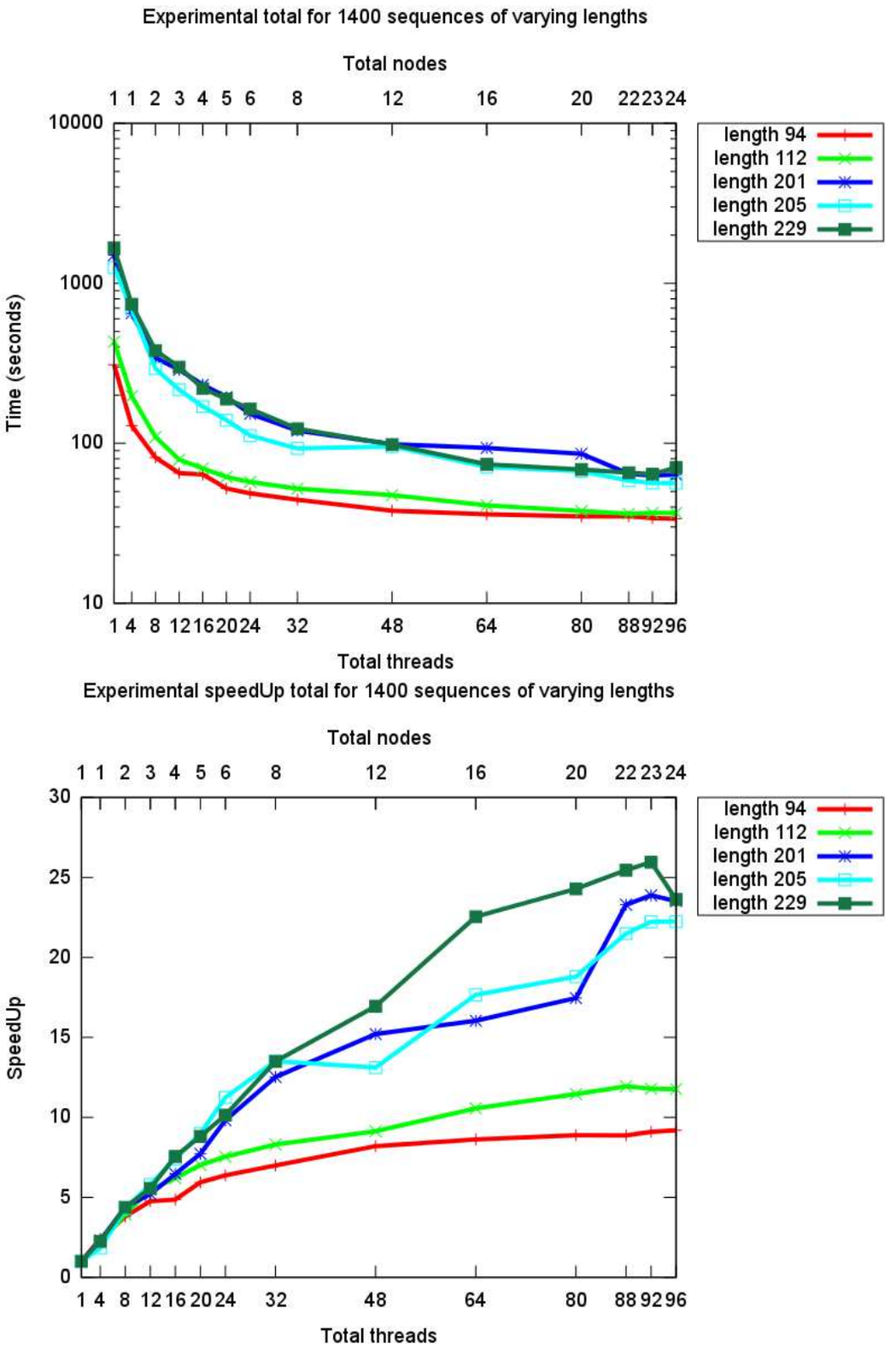


Figure 5.5: Total computation time and SpeedUp varying the sequence lengths

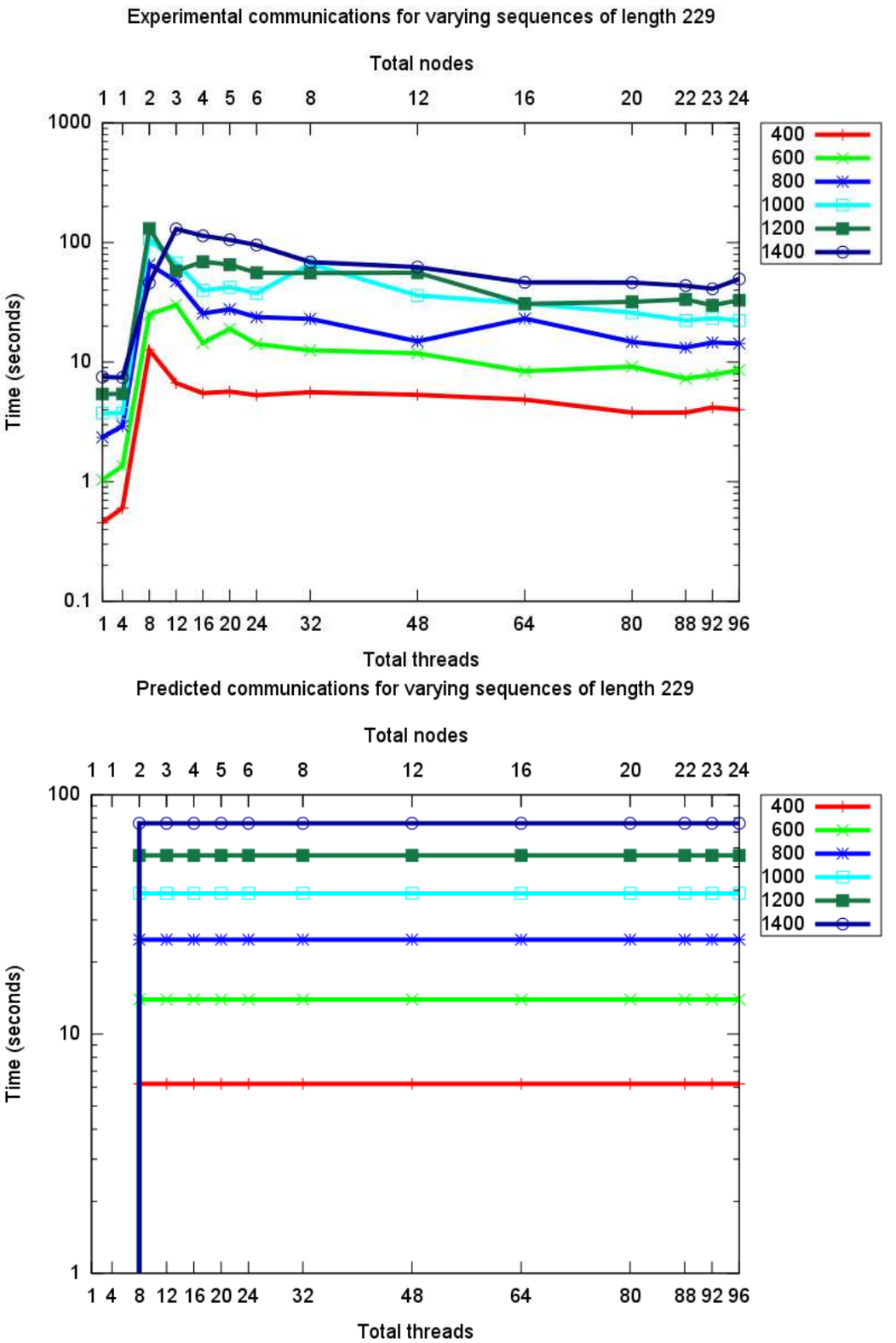


Figure 5.6: Communications overhead varying the number of sequences

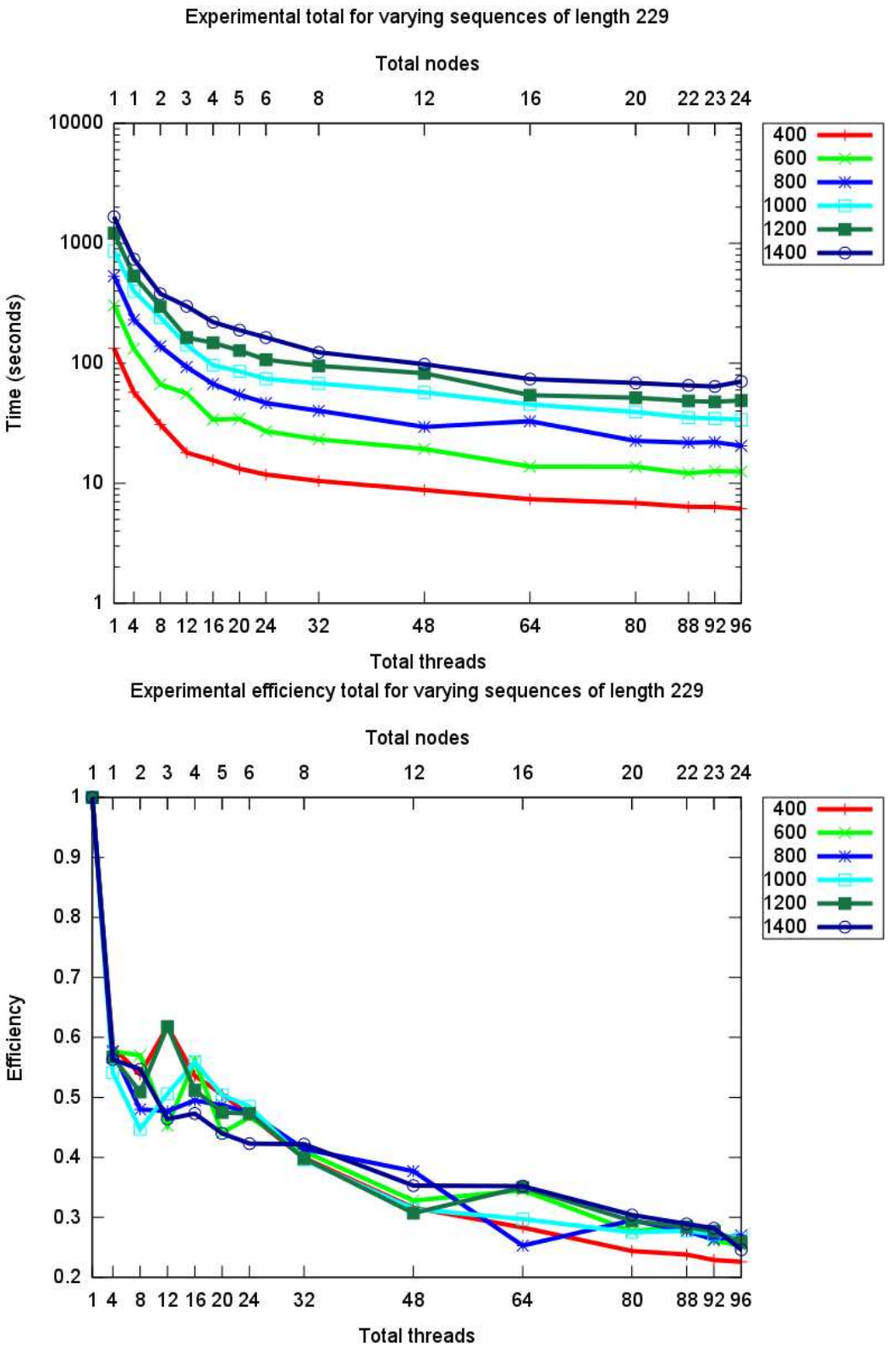


Figure 5.7: Total pairwise time and Efficiency varying the number of sequences

CHAPTER 6

# Conclusions

The main goal of this thesis had been to contribute in the sequence alignment problem by providing new proposals and solutions in order to increase the overall efficiency of these algorithms on High Performance Computing HPC based systems. The Multiple Sequence Alignment (MSA) problem, is a high computing demanding process, in the biology field. Increasing the amount of sequences that can be processed by also obtaining a good quality alignment is the first goal of this problem. The complexity of the problem increases in an exponential way, thus limiting us the amount of sequences to be processed, due to the grow in memory consumption and the required computation time.

One of the first tasks performed in the present work, consisted on carrying out an analysis of performance and scalability of the most common MSA applications used by the biologists community. First, we performed a study of the serial implementations T-Coffee, MAFFT, Kalign, Clustal Omega, Clustal W and MUSCLE. During this study, we validated the quality of the alignments generated by each implementation using the BAliBASE benchmarking tool and the memory and computation time were measured for a comparative

study. In the results, we observed that T-Coffee is the implementation that was able to achieve the best quality scores, but at the same time, it is the one that takes a significant greater amount of system resources to generate an alignment, thus limiting the maximum amount of sequences that can be aligned in comparison with the other implementations.

As T-Coffee is one of the most used implementations among the scientific community, as it performs the best alignments in comparison with the others, in the present work we focused in this application and we made a deep analysis of its behaviour and its code implementation. From the study, we saw that T-Coffee is only capable to align up to three hundred sequences before it runs out of available resources.

Furthermore, two different parallel implementations based on T-Coffee, using a message passing library, have been analysed, to determine their scalability. One is Parallel-T-Coffee (PTC) that parallelizes the internal steps and then gathers their results to generate a final global alignment. The other parallel implementation is Clus-T-Coffee (CTC) that adds an initial step to divide the input sequences in sets (clusters), where sequences are joined by similarity.

Several files of input sequences were tested, varying their amount from one hundred to two thousand with different lengths. With the different tests, we could verify that these parallel implementations were able to process more sequences than than the serial one. However, there are still some issues to confront with the communication costs, as when the message-passing tasks were spread among up to three or four nodes, the global processing time started to increase.

As a solution of the aforementioned problems, we have proposed to introduce a multi-threading solution in one of the current T-Coffee implementations in order to increase its efficiency, reduce the consumed memory and reduce the overhead of using a message-passing library for computation on a multi-core machine. These optimizations were added to the pair-wise step of the process in the sequential T-Coffee. The intensive usage of static variables in all core functions of T-Coffee are the main reason to make any threaded implementation very hard, as these functions have to be rewritten again so them can be executed in a concurrent way. Moreover, on the tests that we have executed on

our proposed prototype are showing us a better efficiency and speed up with our threaded implementation than with the original implementation based on the *fork/wait* model. These achievements and research has been presented in the following publications:

- Alberto Montañola, Concepció Roig, Porfidio Hernández, Toni Espinosa, Yandi Naranjo, and Cedric Notredame. Towards an efficient execution of multiple sequence alignment in multi-core systems. In *International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*, pages 823–835, 2011.

- Alberto Montañola, Concepció Roig, Fernando Guirado, Porfidio Hernández, and Cedric Notredame. Performance analisys of computational approaches to solve multiple sequence aligment. *Journal of Supercomputing*, 64:69,78, 2013.

However, the implemented prototype is very difficult to maintain as it requires a lot of time to audit and rewrite lot of code of T-Coffee, by this way, it was more faster to start writing a new implementation from scratch. Thus, we implemented a new threaded and MPI based implementation capable of generating all possible pair of sequences from the input file.

In order to address these issues, we presented a hybrid implementation using a threading library combined with a message passing one to exploit the available resources as best as possible. A new implementation based on the Smith-Waterman local pairwise algorithm was presented alongside some experimental results that prove its effectiveness with a shared and distributed memory hybrid implementation. We can reach the following conclusions from the experimentation performed: The proposed algorithm is capable of optimizing the average computation time and exploiting the use of the system distributed and shared memory resources. A fully threaded implementation saves memory resources by efficiently using the shared memory paradigm, while a distributed memory only version will have a penalization due to replication of data structures in multi-core systems. The results obtained show how the optimal parallelization level can be evaluated in order to obtain efficient use of the resources. These research was presented on:

- Alberto Montañola, Concepció Roig, and Porfidio Hernández. Pairwise sequence alignment method for distributed shared memory systems. *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 0:432–436, 2013.

The experimentation carried out with the new implementation of the pairwise step, showed us that for all the sets of input sequences, increasing the number of threads and nodes entails a decrease in the computation time. However, it was also observed that this decrease become very slight from a certain number of nodes due to the overhead of communications. This implies that identifying this number of nodes is a key point in order to have an efficient use of computing resources. With this in mind, we developed an analytical method to estimate the appropriate number of resources needed to solve the pair-wise alignment in order to obtain a given efficiency in the speed-up. To validate the effectiveness of our predicting method, we checked the predicted resources with the values obtained experimenting in a cluster with shared and distributed memory. The results showed that we were able to determine the appropriate number of nodes in order to achieve a given efficiency. This research was presented in the following publications:

- Alberto Montañola, Concepció Roig, and Porfidio Hernández. Optimizing multiple pairwise alignment of genomic sequences in multicore clusters. In *PACBB*, pages 121–128, 2014.

- Alberto Montañola, Concepció Roig, and Porfidio Hernández. Efficient mapping of genomic sequences to optimize multiple pairwise alignment in hybrid cluster platforms. *Journal of Integrative Bioinformatics (JIB)*, 2014.

# Future work

There are a lot of open research lines for this work that would contribute to a greater increase of scalability of the program:

- Using GPUs. Nowadays most computers are shipped with a Graphics Processor Unit (GPU), while executing MSA on these systems, only the main CPU would be used leaving the GPU idle, there is the possibility to send some work also to the GPU. This presents a new challenge, thus two different implementations of the pair-wise code have to be provided. Furthermore, another challenge is predicting how much work can process the GPU and the CPU in order to sent the correct sized batches of sequences to each one.

- SSE. We have seen as FASTA is using the SSE instruction set in the Smith-Waterman algorithm in order to achieve better performance than with the normal implementation. We could benefit from the usage of this implementation.

- OpenMP. OpenMP is another way to parallelize a problem, this technology is based in a pure data parallelization. Clustal Omega is using this technology in their implementation to improve the speed of the algorithm.

In addition of these improvements, it would be interesting to offer mechanisms to easily plug-in other pair-wise alignment implementations, and scoring

mechanism, and expand the current implementation with more plugins, so a set of different techniques could be used for obtaining a final alignment. The current code, is still missing more work on building a pylogenetic tree, and finally performing the progressive alignment, thus this tasks at current time can be sent to other MSA implementation, in our case with T-Coffee. We would like also to validate the proposed prediction model on more up to date cluster based systems, and introduce to this model more challenges such as GPU processing between others.

# Appendices

APPENDIX A

# Sequence Generator description

The sequence generator *SeqGen.py*, can be used to generate arbitrary random sequences of determined length. The arguments recognized by the generator are:

- -type [protein, rna, dna]: Sets the type of sequence to generate, whether it is DNA, RNA or a protein sequence.
- -len [n]: Sets the length of the pristine sequence to the n value. The generated sequences will have a length approximately around this value.
- -seed: Sets the random seed, in theory you can generate the same sequences reusing the same seed, as far as the provability tables have not been modified.
- [$x$]: Sets the amount of sequences to generate

You may modify the provability tables in order to achieve different distributions of the mutations for the sequences.

An example execution would be:

```
./SeqGenerator.py -len 200 -type protein 100
```

This will generate one hundred protein based sequences with an average of two hundred residues per sequence. The generated sequences are written in FASTA format to the standard output.

In order to generate a batch of sequences from a python script, you can use:

```
#!/usr/bin/python

from SeqGenerator import SeqGen

if __name__ == "__main__":

    tests = [["protA","protein"],
            ["protB","protein"],
            ["protC","protein"],
            ]
    lengths = [50,100,150,200]
    sizes = range(10,150,10)

    path = "inputs/generated"

    for size in sizes:
        for length in lengths:
            for test in tests:
                fname = "%04i_%03i_%s.tfa" %(size,length,test[0])
                print fname
                fout = file(path + "/" + fname,"w")
                sg = SeqGen(test[1],length)
                sg.generate(size,fout)
                fout.close()
```

APPENDIX B

# T-Coffee Threads usage

New input parameters were added to the threaded T-Coffee implementation, this appendix describes the usage of these parameters.

- -n_core = [n]: Sets the number of threads to spawn, default implementation already has this parameter but it was ignored by the implementation and the only way to define the number of threads was by the environment variable *NUMBER_OF_PROCESSORS*, by default it tries to guess the number of threads that you can optimally run on the system by reading the */proc* filesystem.
- -mcheck: Enables memory checking (must be the first parameter).
- -dbg_single: Ignores all parallelization, threads are spawned but only one runs at a time forcing a pure serial execution. Only affects the threaded implementation.
- -mth = [pthreads, disabled]: Sets the parallelization implementation to use. By default, it will use threads, by setting it to disabled, it will disable the threading code, thus it will run as the original T-Coffee.

Example T-Coffee execution would be:

```
./t_coffee -mcheck input.fasta -n_core=4 -mth=pthreads
```

It will run T-Coffee, spawning four threads and using the pthreads implementation. It will run also some memory checking affecting considerably the overall performance.

APPENDIX C

# Caffee usage

The distributed pair-wise aligner Caffee has the following input parameters:

- -mcheck / -nomcheck: Enables or disables the memory checking, by default it will be enabled or not depending on the configuration performed in compilation time. Normally memory should not be checked by default, but debugging builds of the binary may enable this checking by default, thus you can disable by passing the correct parameter. It must be the first parameter passed.
- -n_thd = [n]: Sets the number of threads to spawn, by default it tries to guess from the operating system by reading the proc filesystem.
- -dbg_single: Threads are spawned but only one runs at a time forcing a pure serial execution in the node, this will not affect the message passing implementation. Only affects the threaded implementation.
- -ff: Force FASTA, it will not reuse a previous generated BCaffee file, and it will parse again the FASTA file.

- -nbc: Disable the generation of the BCaffee file.
- [*SequencesFile*]: Input sequence file in either FASTA format or BCaffee format. If the -ff switch is not present, and there is already a corresponding BCaffee file present in the same input directory, the FASTA file will be ignored and the last one will be used instead.

Example Caffee execution would be:

```
mpirun -n 4 ./caffee -n_thd=2 input.bcaf
```

It will run caffee, spawning four MPI processes and two threads.

APPENDIX D

# BCaffee file format

This appendix describes the binary file format used by Caffee to store sequences.

```
//Little-Endian
[HEADER]
U32 magic = 0xeeffca0f; //File magic header
Byte version = 0x01;    //Version 1
Bytes compression;      //Compression used,
                        //data chunk compressed in zlib format
                        // 0 - No compression
                        // 1 - Full compression
                        // 2 - Sequence based compression
Byte seqtype;           //Sequence type
                        // 0 - Undefined
                        // 1 - DNA
                        // 2 - RNA
                        // 3 - Protein
```

```
Byte reserved = 0x00;   //Must be 0
If compression == 0x01 then
    U32 iSize;          //Size of uncompressed index
    U32 cSize;          //Size of compressed index
    Byte binary[cSize]; //Compressed INDEX
Else
    INDEX
Endif


[INDEX]
U32 nSeqs;              //Number of sequences
U32 offsets[nSeqs];     //Address of sequence in data chunk
                        //relative to this offset
SEQUENCE[nSeqs];


[SEQUENCE]
U32 sSize;              //Sequence size
U32 nSize;              //Sequence name size
If compression == 0x02 and sSize & 0x80000000 then
    U32 csSize;         //Compressed sequence size
Endif
If compression == 0x02 and nSize & 0x80000000 then
    U32 cnSize;         //Compressed sequence name size
Endif
//Sequences will be stored as compressed if the field is present
Byte Sequence[SeqSize]; //Sequence
Byte SeqName[SeqNameSize]; //Sequence name
```

# Bibliography


[A97] Wozniak A. Using video-oriented instructions to speed up sequence comparison. *Comput. Appl. Biosci*, (13):145–150, 1997.

[CDP+03] James Cheetham, Frank Dehne, Sylvain Pitre, Andrew Rau-Chaplin, and PeterJ. Taillon. Parallel CLUSTAL W for PC Clusters. In Vipin Kumar, MarinaL. Gavrilova, ChihJengKenneth Tan, and Pierre L'Ecuyer, editors, *Computational Science and Its Applications — ICCSA 2003*, volume 2668 of *Lecture Notes in Computer Science*, pages 300–309. Springer Berlin Heidelberg, 2003.

[CLHH09] Sao-Jie Chen, Guang-Huei Lin, Pao-Ann Hsiung, and Yu-Hen Hu. *Hardware Software Co-Design of a Multimedia SOC Platform*. Springer, 2009.

[D05] Geer D. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[DTMX+11] Paolo Di Tommaso, Sebastien Moretti, Ioannis Xenarios, Miquel Orobitg, Alberto Montañola, Jia-Ming Chang, Jean-François Taly, and Cedric Notredame. T-coffee: a web server for the multiple sequence alignment of protein and rna sequences using structural information and homology extension. *Nucleic Acids Research, 39(Web Server issue)*, page W13–W17, 2011.

[EB06] Robert C Edgar and Serafim Batzoglou. Multiple sequence alignment. *Current opinion in structural biology*, 16(3):368–373, 2006.

[Edg04] R.C. Edgar. MUSCLE: Multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 5(32):1792–1797, 2004.

[Far06] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 2(23):156–161, November 16 2006.

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[KC99] Bhupesh Kothari and Mark Claypool. PThreads Performance. *Computer Science Technical Report Series. Worcester Polytechnic Institute. Massachusetts*, 1999.

[KT08] K Katoh and H Toh. Recent developments in the MAFFT multiple sequence alignment program. *Briefings in Bioinformatics*, 9(4):286–298, 2008.

[LBB+07] M.A. Larkin, G. Blackshields, N.P. Brown, R. Chenna, P.A. McGettigan, H. McWillian, F. Valentin, I.M. Wallace, A. Wilm, R. Lopez, J.D. Thompson, T.J. Gibson, and D.G. Higgins. ClustalW and ClustalX version 2. *Bioinformatics*, 21(23):2947–2948, 2007.

[Li03] Kuo-Bin Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[LMS09] Y Liu, DL Maskell, and B Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, page 2:73, 2009.

[LS05] T Lassmann and E.L.L Sonnhammer. Kalign–an accurate and fast multiple sequence alignment algorithm. *BMC Bioinformatics*, (6):298, 2005.

[MM08] Hill M.D and Marty M.R. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[MnRG+13] Alberto Montañola, Concepció Roig, Fernando Guirado, Porfidio Hernández, and Cedric Notredame. Performance analisys of computational approaches to solve multiple sequence aligment. *Journal of Supercomputing*, 64:69,78, 2013.

[MnRH13] Alberto Montañola, Concepció Roig, and Porfidio Hernández. Pairwise sequence alignment method for distributed shared memory systems. *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 0:432–436, 2013.

[MRH+11] Alberto Montañola, Concepció Roig, Porfidio Hernández, Toni Espinosa, Yandi Naranjo, and Cedric Notredame. Towards an efficient execution of multiple sequence alignment in multi-core systems. In *International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*, pages 823–835, 2011.

[MRH14a] Alberto Montañola, Concepció Roig, and Porfidio Hernández. Efficient mapping of genomic sequences to optimize multiple pairwise alignment in hybrid cluster platforms. *Journal of Integrative Bioinformatics (JIB)*, 2014.

[MRH14b] Alberto Montañola, Concepció Roig, and Porfidio Hernández. Optimizing multiple pairwise alignment of genomic sequences in multicore clusters. In *PACBB*, pages 121–128, 2014.

[NHH00] C Notredame, DG Higgins, and J Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–17, September 8 2000.

[NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 3(48):443–53, 1970.

[PCE+12] M. Punta, P.C. Coggill, R.Y. Eberhardt, J. Mistry, J. Tate, C. Boursnell, N. Pang, K. Forslund, G. Ceric, J. Clements, A. Heger, L. Holm, E.L.L. Sonnhammer, S.R. Eddy, A. Bateman, and R.D. Finn. The Pfam protein families database. *Nucleic Acids Research*, 40:D290–D301, 2012.

[PM10] Timothy Prickett Morgan. Intel pushes workhose xeons to six cores. `http://www.theregister.co.uk/2010/03/16/intel_xeon_5600_launch`, 2010. [Online; accessed 5-July-2015].

[SJ92] Henikoff S and Henikoff JG. Amino acid substitution matrices from protein blocks. *roc Natl Acad Sci U S A*, 89(22):10915–9, 1992.

[SKM08] Amarendran R Subramanian, Michael Kaufmann, and Burkhard. Morgenstern. DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment. *Algorithms for Molecular Biology*, pages 3–6, 2008.

[SW81] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[SWD+11] Fabian Sievers, Andreas Wilm, David Dineen, Toby J Gibson, Kevin Karplus, Weizhong Li, Rodrigo Lopez, Hamish McWilliam, Michael Remmert, Johannes Söding, Julie D Thompson, and Desmond G Higgins. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular Systems Biology*, 7(1):539, 2011.

[SZ05] Kim Sneppen and Giovanni Zocchi. *Physics in Molecular Biology*. Cambridge University Press, 2005.

[top15] Top 500, June 2015 List. `http://www.top500.org/lists/2015/06/`, 2015. [Online; accessed 5-July-2015].

[TPP99a] J.D Thompson, F Plewniak, and O Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Res.*, 27(13):2682–90, 1999.

[TPP99b] J.D Thompson, F Plewniak, and O Poch. BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *BIOINFORMATICS -OXFORD-*, 15(1):87–88, 1999.

[WD88] Pearson WR and Lipman DJ. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8):2444–8, 1988.

[WT94] Lusheng Wang and Jiang Tao. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(3):337–348, 1994.

[Y09] Naranjo Y. Alineamiento múltiple de secuencias con T-Coffee: una aproximación paralela. *master thesis, UAB*, 2009.

[YA99] Kwok Y.K and I Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, (59):381–422, 1999.

[ZYRA07] J Zola, X Yang, S Rospondek, and S Aluru. Parallel T-Coffee: A parallel multiple sequence aligner. *In Proc. of ISCA*, pages 248–253, 2007.