Parallelization of whole genome alignment

Julio Cesar Garcia Vizcaino
Department of Architecture & Operating Systems
Universitat Autonoma de Barcelona

A thesis submitted for the degree of
Master of High Performance Computing
July, 2011

Trabajo de investigación
Màster en Computaciò d'Altes Prestacions
Curso 2010-11

Parallelization of whole genome alignment

Autor
Julio César García Vizcaíno

Director
Antonio Espinosa

Departamento Arquitectura de Computadores y Sistemas Operativos
Escola d'Enginyeria
Universitat Autònoma de Barcelona

Firma director   Firma Autor

To you ♡.

Acknowledgements

To my family and friends in Mexico.
To Toni, Porfi & Juan Carlos for their support and advices.
To my new partners in CAOS.

Abstract

With the advent of High performance computing, it is now possible to achieve orders of magnitude performance and computation eficiency gains over conventional computer architectures. This thesis explores the potential of using high performance computing to accelerate whole genome alignment.
A parallel technique is applied to an algorithm for whole genome alignment, this technique is explained and some experiments were carried out to test it. This technique is based in a fair usage of the available resource to execute genome alignment and how this can be used in HPC clusters. This work is a first approximation to whole genome alignment and it shows the advantages of parallelism and some of the drawbacks that our technique has.
This work describes the resource limitations of current WGA applications when dealing with large quantities of sequences. It proposes a parallel heuristic to distribute the load and to assure that alignment quality is mantained.

Contents

List of Figures

Chapter 1

Introduction

1.1 Genome alignment

Today, we know that species are described by DNA, see Figure 1.1, a complex molecule comprised of many smaller molecules called nucleotides. The data describing a single specie,commonly called a genome, can be millions or billions of nucleotides long.

One of the most basic computational tasks that we perform on genomic data is

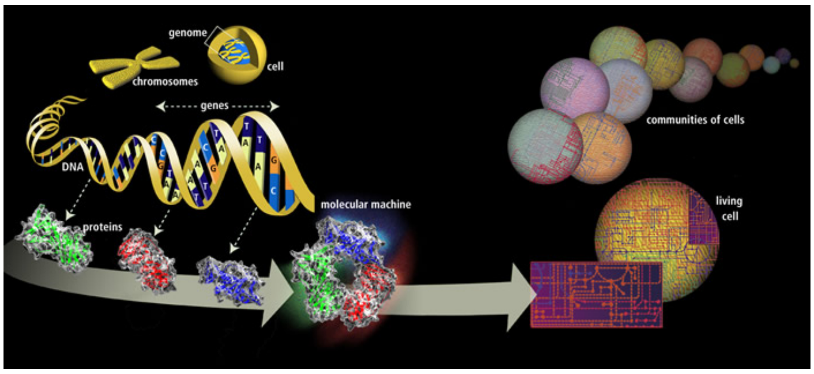


Figure 1.1: Structure of a genome.

identifying the evolutionary relationships between DNA from two or more species. On a smaller scale, we wish to identify which individual nucleotides are unique to species, and which nucleotides share ancestry. On a larger scale, we look to **find entire subsequences that are a common between them**.

1.1.1 Sequence alignment

We now describe some notation, give a more detailed introduction to sequence alignment, and describe the mathematical framework on which we base most of techniques used to perform whole genome alignment. First we define some basic notation. We use R (Reference) and Q (Query) to denote sequences. For sequence R, we refer to position i as R_i and let the first position be R_0.
Sequence alignment is the primary tool for finding evolutionary relationships between DNA sequences. A DNA sequence is a string over four symbols: A, T , C, and G. These symbols represent the nucleotides adenine, thymine, cytosine, and guanine, respectively. As time passes, DNA sequences incur mutations from a variety of physical processes. Thus, DNA sequences from individuals of a specie contain many differences. When aligning DNA sequences from different species, large scale changes, such as long insertions and deletions, duplications, reversals and translocations, are common, see Figure 1.2. The goal of sequence alignment is to infer which changes occurred with a mathematical model that abstracts the physical mutation processes.

Given two sequences, we can interpret an alignment. We stack the aligned

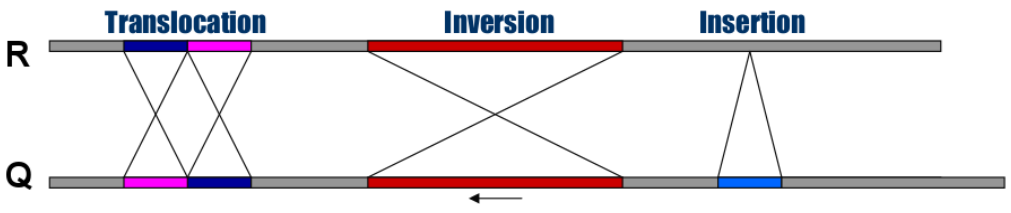


Figure 1.2: Operations in whole genome alignment

sequences on top of one another and look at the content of the every position, we can see the following schema:

Original sequence:	AGGCCTC
Mutations:	AGGACTC
Insertions:	AGGGCCTC
Deletions:	AGG.CTC

When two nucleotide symbols are in the same position, they either match or they mismatch. These positions describe two evolutionary predictions. First, the nucleotides corresponding to each symbol are predicted to have a common ancestor. Second, a match/mismatch column makes a prediction about point mutations having occurred in the time since the original sequences diverged. For example, if the symbols mismatch, then one or both of the nucleotides corresponding to

the mismatching symbols may have incurred a point mutation.
When a nucleotide symbol shares a position with a gap symbol, we refer to this as either an insertion or deletion (indel). Typically, we consider consecutive gap positions as a single unit, corresponding to a single mutation event. Positions containing only gap symbols have no meaning and are thus disallowed. For a pair of sequences, we make no distinction between insertion and deletion events as it is impossible to know the true sequence of the common ancestor. It could be that the subsequence in question was deleted from the common ancestor in one sequence, or that it was never in the common ancestor at all and was instead inserted into the other sequence.

1.2 Why whole genome alignment?

For the reasons previously mentioned, the objective is to find a method that can help us to compare the whole genomes of two species and to extract all the conserved gene pairs they share. One possible way to accomplish the task is to perform **whole genome alignment**.

1.2.1 Score-based alignment

Score-based alignment measures sequence similarity with edit distance. Here, matches, mismatches, and indels are edit operations transforming one sequence into the other. We assign each edit operation a cost, and the sum of the costs allows us to assign a similarity measure to a particular alignment. The goal of sequence alignment in this framework is to find an alignment with maximum score.
One of these algorithms was introduced in 1970 by [Needleman and Wunsch, 1970] and is the foundation for many modern alignment algorithms. The original Needleman-Wunsch alignment algorithm uses dynamic programming to find the optimal alignment in $O(nm)$ time and space for two sequences of length m and n.

1.2.2 Why not use standard pairwise methods?

The standard sequence alignment methods are not used because of the following reasons:

1. Different features that we are looking at

- Standard alignment: only point mutation, insertion and deletion can be observed.
- Genome alignment: we can also learn about transposition, large insertions/deletion, etc.

2. Resource limitation

- Given all the available resources today, it is infeasible to align whole genomes using standard methods. The time and space complexity is too high, so that it takes too much time and space if we employ those methods.

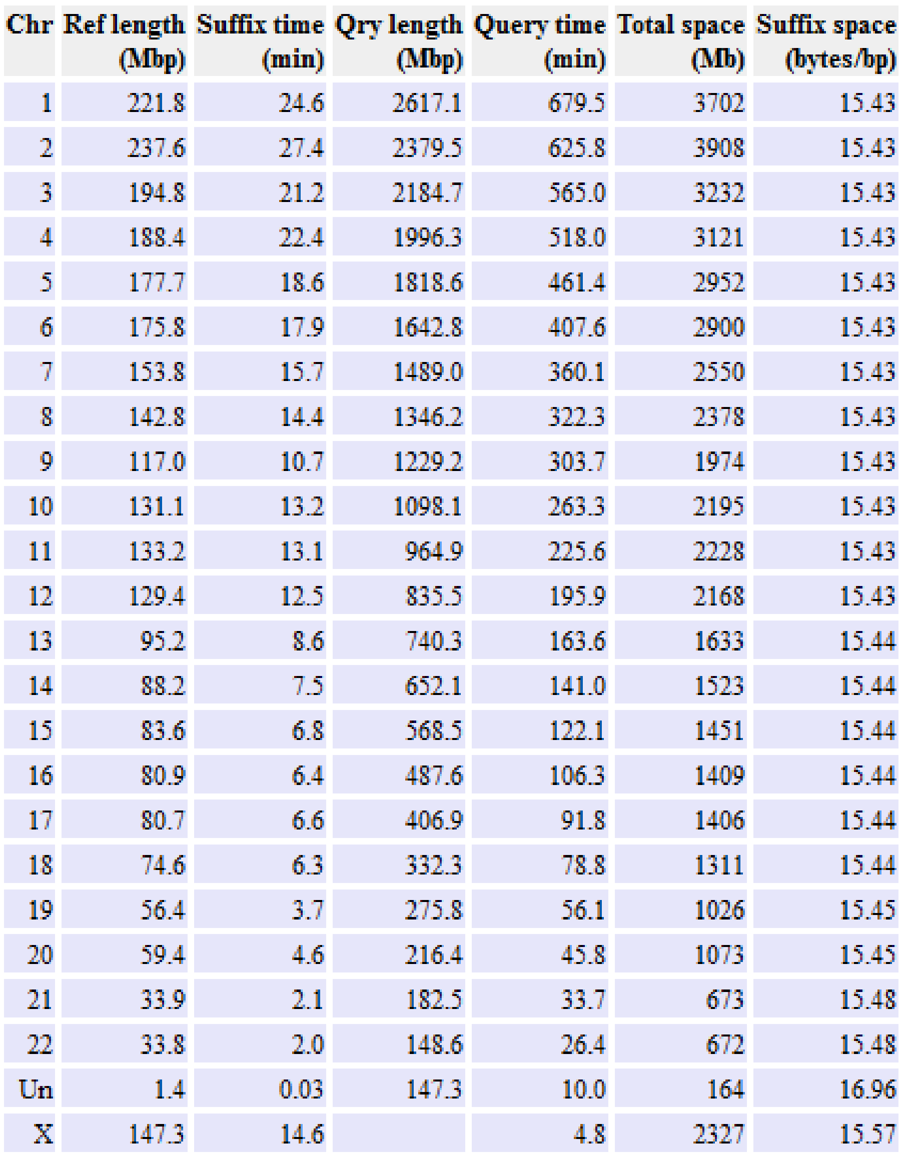
1.2.3 Why is HPC needed?

High performance computing allows to use computers, software, and expertise to solve problems too difficult to solve effectively by other means. Therefore the reason to use HPC is because of an application asking for its performance. Currently the availability of a huge amount of Bioinformatics data (often in the public domain), and on the other hand the need for new and efficient methods and algorithms capable of compute the information contained in the data requires the use of HPC to manipulate it. As a matter of fact, the emphasis of research in Bioinformatics and HPC is shifting from the development of efficient data storing and handling methods, to the one of methods able to extract useful information from data, as in Figure 1.3.

Consequently, the computational demands needed to explore and analyze the data contained in the genome databases is quickly becoming a great concern. To meet these demands, we must use high performance computing, such as parallel computers and distributed networks of workstations.
Although the expertise of both areas: Bioinformatics and HPC is hard to have in a research team. The purpose of this thesis is to provide an improved algorithm to be used in whole genome alignment.

1.3 Related work

A lot of research for aligning two sequences has been done for many years. One of the earliest work includes Needleman and Wunsch [1970] and Smith and Waterman [1981]. The main goal of pioneer research has been on comparing single proteins or genomic DNA sequences containing a single gene. Current algorithms work very well with this data set, however they are ineffective in aligning whole genomes. The problem is when a genome is more than tens of thousands of base pairs, there are whole genomes which are usually millions of nucleotides or larger.



Chr	Ref length (Mbp)	Suffix time (min)	Qry length (Mbp)	Query time (min)	Total space (Mb)	Suffix space (bytes/bp)
1	221.8	24.6	2617.1	679.5	3702	15.43
2	237.6	27.4	2379.5	625.8	3908	15.43
3	194.8	21.2	2184.7	565.0	3232	15.43
4	188.4	22.4	1996.3	518.0	3121	15.43
5	177.7	18.6	1818.6	461.4	2952	15.43
6	175.8	17.9	1642.8	407.6	2900	15.43
7	153.8	15.7	1489.0	360.1	2550	15.43
8	142.8	14.4	1346.2	322.3	2378	15.43
9	117.0	10.7	1229.2	303.7	1974	15.43
10	131.1	13.2	1098.1	263.3	2195	15.43
11	133.2	13.1	964.9	225.6	2228	15.43
12	129.4	12.5	835.5	195.9	2168	15.43
13	95.2	8.6	740.3	163.6	1633	15.44
14	88.2	7.5	652.1	141.0	1523	15.44
15	83.6	6.8	568.5	122.1	1451	15.44
16	80.9	6.4	487.6	106.3	1409	15.44
17	80.7	6.6	406.9	91.8	1406	15.44
18	74.6	6.3	332.3	78.8	1311	15.44
19	56.4	3.7	275.8	56.1	1026	15.45
20	59.4	4.6	216.4	45.8	1073	15.45
21	33.9	2.1	182.5	33.7	673	15.48
22	33.8	2.0	148.6	26.4	672	15.48
Un	1.4	0.03	147.3	10.0	164	16.96
X	147.3	14.6		4.8	2327	15.57

Figure 1.3: Human vs. Human alignment with MUMmer in Kurtz et al. [2004]

Many algorithms have been proposed to solve the whole genome alignment problem, one classification is based on how the WGA is solved:

- Optimal algorithms: consist of an exhaustive search in the whole sequence to align them. It outputs the best alignment but theirs time complexity and space complexity is $O(NM)$ which N and M are lengths of each genome.
- Heuristic algorithms: are based on an heuristic approach to reduce the time complexity and space complexity of optimal algorithms.

The shortcomings in computational efficiency of classical methods motivated the recent development of several whole genome alignment algorithms. Several global alignments systems have been developed:

- MUMmer.
- AVID:

  > "is sensitive in finding homologous regions, but is also specific and avoids the false-positive problem of local alignment programs." Bray et al. [2003]

- LAGAN:

  > "performs anchoring with techniques designed to work well on distant, as well as close organisms. Specifically, LAGAN uses the CHAOS algorithm Brudno et al. [2003a], a highly sensitive method that detects local alignments using multiple short inexact words instead of longer exact words. LAGAN then constructs the global alignment by first applying CHAOS recursively in areas with sparse anchors, so that each consecutive pair of anchors is separated by a distance smaller than a given maximum, and then performing a limited-area dynamic programming algorithm on the area around the anchors." Brudno et al. [2003b]

Most of these methods are based on a *chaining* strategy of first finding and chaining words or local alignments between two genomes, thereby creating a set of anchors; these anchors cut the whole genome alignment problem into many smaller sequences alignments.
Chaining works well in practice in addressing both shortcomings of pure dynamic-programming approaches, namely insufficient speed and intolerance of long gaps. Concerning speed, chaining relies on three steps:

1. Exact word matching or index-based local alignment: which is much faster than the time required for full dynamic programming.

2. Construction of the optimal chain of local hits: a procedure that can be done in time $O(n \log n)$ where n is the number of hits, using an extension of the Longest Increasing Subsequence algorithm.

3. Potential computation of alignments in the regions in between the chained pieces, which give rise to much smaller alignment problems.

Concerning accuracy in dealing with long gaps, chaining relies precisely on the fact that DNA sequences similarity comes in islands of highly conserver (evolutionarily constrained) regions, flanked by less conserved pieces that are typically much longer in higher organisms. This property makes it reasonable to first find the highly similar regions and construct a map based on them, and only later deal with the potentially not aligned regions in-between.

1.4 Objectives

The main objective of this thesis is to use parallelism techniques in whole genome alignment in order to improve the use of CPU and Memory resources. Moreover, we use the SPMD paradigm to achieve this goal and it is related to use in HPC clusters.
This work allows to study data distribution policies for whole genome alignment. Some changes and add-ons are implemented to validate obtained results regardless the parallelization techniques used.
Some minor objectives are also proposed to be achieved with this work:

- Understand how MUMmer's algorithm works to choose which modifications can be made to improve its performance.

- Validate results of our parallel data distribution versus the MUMmer's algorithm.

- Evaluate how to distribute genome data and be able to divide this data.

- Improve memory and execution time of the WGA application by an effective data distribution.

1.5 Structure of this work

This work is divided in five chapters. The first chapter explains the problem to be solved in this work and the state of art of it. The second chapter describes in detail how MUMmer's algorithm works. The third chapter explains our proposal to improve MUMmer in terms of time and memory. Different approaches are studied and it also describes how this technique can be used in conjunction with MUMmer. The fourth chapter describes some tests to check if our proposal works under several scenarios (data level parallelism) and to show the performance of using this technique under HPC clusters. The fifth chapter concludes the study of our approach and it adds which modifications can be done to improve it and its restrictions.

Chapter 2

Whole genome alignment with MUMmer

2.1 The MUM: an heuristic approach

2.1.1 Definition MUM

Although a pair of conserved genes rarely contain the same entire sequence, they share a lot of short common substrings and some of them are indeed unique to this pair of genes. For example the following two sequences, R and Q:

$$\begin{gathered}\text{R=}\underline{\text{ac}}\text{ ga }\underline{\text{ctc}}\text{ a }\underline{\text{gctac}}\text{ t }\underline{\text{ggtcagctatt}}\text{ }\underline{\text{acttaccgc}}\text{\$}\\ \text{Q=}\underline{\text{ac}}\text{ tt }\underline{\text{ctc}}\text{ t }\underline{\text{gctac}}\text{ }\underline{\text{ggtcagctatt}}\text{ c }\underline{\text{acttaccgc}}\text{\$}\end{gathered}$$

It is clear that sequences R and Q have many common substrings, they are:

- ac
- ctc
- gctac
- ggtcagctatt
- acttaccgc

Among those five common substrings, ac is the only substring that is not unique. It occurs more than once in both sequences. You can also observe that actually a, c, t, and g are common substrings of R and Q. However, they are not maximal, i.e. they are contained in at least one longer common substrings. We are only interested in those that are of maximal length.

Our aim is to search for all these short common substrings. Given genomes R and Q, we need to find all common substrings which are unique and of maximal length. Each of such common substrings is known as Maximum Unique Match (MUM). For almost every conserved gene pairs, there exist at least one MUM which is unique to them.
So, a Maximal Unique Match (MUM) is a common substring that have the following two properties:

- It occurs exactly once in both genomes R and Q (unique to both R and Q)
- It is not contained in any longer MUMs (it is of maximal length)

A formal definition of a MUM is given below:

Definition 1. *Maximal Unique Match(MUM) substring is a common substring of the two genomes that is longer than a specific minimum length d such that it is maximal, that is, it cannot be extended on either end without incurring a mismatch and it is unique in both sequences.*

For example, assuming d = 3, sequences R and Q in the previous example has four MUMs: ctc, gctac, ggtcagctatt, acttaccgc. Substring ac is not an MUM because its length is smaller than the value of d and it is not unique to both sequences.

R=~~ac~~ ga ~~ctc~~ a gctac t ggtcagctatt acttaccgc$
Q=~~ac~~ tt ~~ctc~~ t gctac ggtcagctatt c acttaccgc$

The concept of MUM is important in whole genome alignment because a significantly long MUM is very likely to be part of the global alignment.

2.1.2 How to find MUMs

In this section, we present two methods to find MUMs of a pair of sequences.

2.1.2.1 Brute-force approach:

The idea of this method is to first find all common substrings of the two sequences, then for each substring, we check whether it is longer than d and unique in both sequences. The algorithm is as follows:

Input: Two genome sequences R[1..n] and Q[1..m]

```
for $i := 1$ to $n$ in R do
  for $j := 1$ to $m$ in Q do
    Find the longest common prefix P of R[i..n] and Q[j..m]
    if $|P| \geq d$ and P is unique in both genomes then
      Report it as a MUM
    end if
  end for
end for
```

This solution requires at least $0(nm)$ and is not used in practice.

2.1.2.2 Finding MUMs in a suffix tree

The key idea in this method is to build a suffix tree for genome R, a data structure which allows finding, extremely efficiently, all distinct subsequences in a given sequence.
Any string of length m can be degenerated into m suffixes, and these suffixes can be stored in a suffix tree, for instance Figure 2.1 shows the suffix tree for the word BANANA. Creating this structure requires time $O(m)$ and searching for a pattern in it requires time $O(n)$, where n is the length of the pattern. These two properties make the suffix tree an appealing structure for a diverse range of bioinformatics applications including: multiple genome alignment Hohl et al. [2002]; selection of signature oligonucleotides for DNA arrays Kaderali and Schliep [2002]; and identification of sequence repeats Kurtz [1999].
The MUM search algorithm consists of three steps, they are as follows:

```
Build a suffix tree for R
for Every suffix of Q do
  Compare it by traversing the suffix tree of R
  Mark every edge found in common between R and Q
  For each marked path, suppose it represents the i-th suffix of R and the j-th
  suffix of Q.
  if $R[i-1] \neq Q[j-1]$ then
    The path label of this marked node is a MUM
  end if
end for
```

2.1.2.3 Complexity analysis

- **Step 1:** Building a suffix tree can be done in $O(n)$ time using McGreight's algorithm [McCreight, 1976].

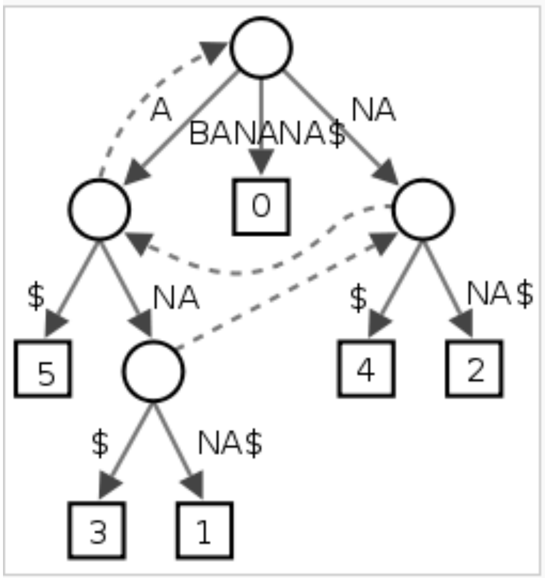


Figure 2.1: Suffix tree for word BANANA.

- **Step 2:** Marking internal nodes takes $O(n)$ time.
- **Step 3:** Comparing R[i-1] and Q[i-1] for each marked nodes takes $O(n+m)$ time as the number of marked nodes is at most $n+m-2$. By the same reasoning, traversing all internal nodes to extracting MUMs also takes $(n+m)$ time.
- In total,this algorithm takes $O(n+m)$ time to find all MUMs of the input sequences.
- The space complexity of this method is $O\left(n \log n\right)$ bits as we need to store the suffix tree of the input sequence.

Based on some experiments, it is found that MUMs can cover 100% of the known conserved gene pairs. Moreover, finding all MUMs can be done in linear time.

2.2 MUMmer1

As mentioned earlier, the idea of applying the concept of longest common subsequence(LCS) in algorithm MUMmer1 arose from a key observation that the ordering of *most of the conserved genes would be preserved* in two related species.

In 1999, [Delcher et al., 1999], invented the MUMmer1.
Hence, in MUMmer1, it computes the longest common subsequence of MUMs contained in both sequences and report only the MUMs in the LCS.
Compared to the brute force approach mentioned in the earlier section, MUMmer1 can successfully reduce the number of false positives reported at the cost of reducing the coverage.

Definition 2. *Let $R = (r_1, r_2, \ldots, r_n)$ and $Q = (q_1, q_2, \ldots, q_n)$ be the order of the n MUMs in two genome sequences R and Q, respectively. A sequence $S = (s_1, s_2, \ldots, s_m)$ is a subsequence of R if there exists indices $(i_1, i_2, \ldots, i_m)$ such that $i_1 < i_2 < \ldots < i_m$ and $s_j = r_{i_j}$ for all j. S is a common subsequence of R and Q if C is a subsequence of both R and Q.*

It is found that two closely related species should preserve the ordering of most conserved genes. For example, if we compare mouse chromosome 16 and human chromosome 16, we shall see that the ordering of 30 conserved gene pairs (out of 31 gene pairs) is preserved, see Figure 2.2.

Definition 3. *Given two sequence R and Q, we say that a sequence S is a Longest Common Subsequence (LCS) of R and Q if S is the longest possible subsequence of both R and Q. Note that the Longest Common Substring is contiguous while the Longest Common Subsequence need not be.*

2.2.1 Description

As the output of the brute force approach contains too many irrelevant subsequences, MUMmer1 was constructed to decrease the number of false positives by outputting only MUMs contained in the LCS of both sequences. In additional to obtaining the MUMs of the two subsequence, we will now be required to determine the LCS of these MUMs, and then to output the MUMs in the LCS.

2.2.1.1 Solution of LCS Problem

The naive way to solve large common subsequence problem is to use dynamic programming. Suppose there are K MUMs found, a dynamic programming algorithm requires $O(K^2)$ time and space to solve the problem. This approach quickly becomes infeasible as the size of the sequences increases.
Later, another method was derived based on a fact that all MUMs are distinct(**by its definition**). As the consequence of this fact, we can label each MUM by different character and we can limit the domain of the LCS problem to a set of sequences consisting of distinct characters, i.e. every single character occurs exactly once in the sequence. Using this approach,the LCS problem can be solved in $O(K \log K)$ time.

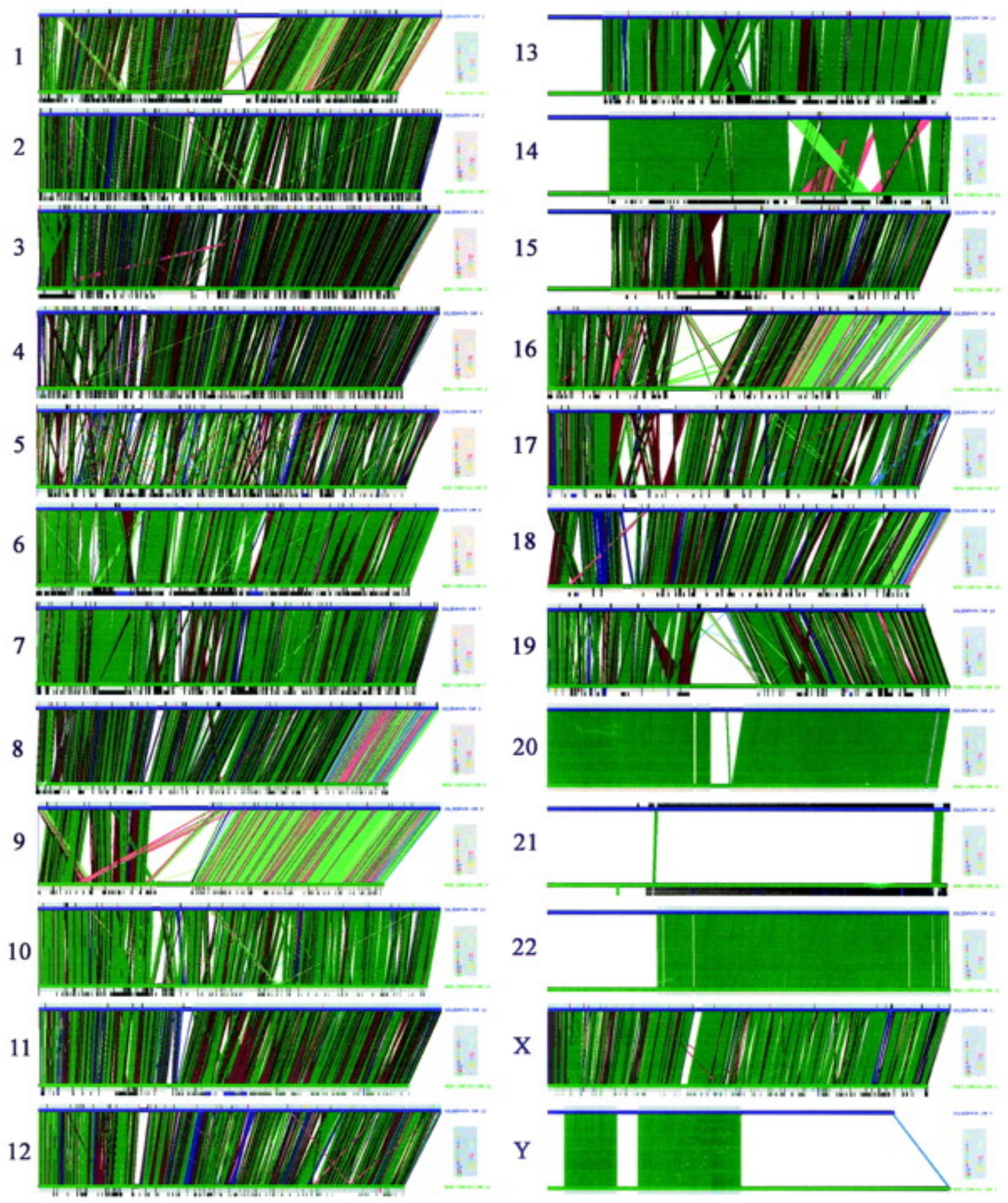


Figure 2.2: Whole genome alignment of two different human genomes.

2.2.1.2 Dynamic programming algorithm in $O(K^2)$

The crux of this algorithm lies in the fact that the LCS of R[1..i] and Q[1..j] is related to the LCS of R[1..(i-1)] and Q[1..(j-1)] by comparing the last characters of subsequence R and Q.
Let us first define $S_i[j]$ as the length of the longest common subsequence of R[1..i] and Q[1..j]. Also define $\delta(i)$ as the index of the character in Q such that $R[i] = Q[\delta(i)]$.
Recall that $S_i[j]$ is the length of the longest common subsequence of R[1..i] and Q[1..j]. There are two possibilities for the character i of S(S[i]) and the character j of Q(Q[j]):

1. $R[i] \neq Q[j]$, in which case the LCS is the same as the LCS of R[1..(i-1)], Q[1..j]
2. $R[i] = Q[j]$, in which case we recursively obtain the LCS of R[1..(i.1)] and Q[1..(j-1)]+1, and select the larger of either 2.

With those observations, we can obtain:

$$S_i[j] = \begin{cases} S_{i-1}[j] & j < \delta(i) \\ max(S_{i-1}[j] + C_{i-1}[\delta(i) - 1] & j \geq \delta(i) \end{cases}$$

The first case corresponds to the case the Q[1..j] does not contain R[i], so R[i] will not be included in the LCS. In the second case, we have the option to either include R[i] or not to, so we can take the maximum over the two cases. Hence, we can obtain the LCS of string R and Q from $S_n[n]$. This is solved in $O(K^2)$ time.

2.2.2 Analysis of complexity

- **Step 1:** Two sequences of length n and m respectively, all MUMs can be computed in $O(n + m)$ time.
- **Step 2:** After finding all the MUMs, we sort them according to their position in Genome R, and employ the LCS algorithm to find the largest set of MUMs whose sequences occur in the same order contained in both Genome R and Q. As mentioned previously, this step requires $O(K \log K)$ time, where K is the number of MUMs. In general, K is much smaller than m and n.
- **Step 3:** Once a global alignment is found, we can deploy several algorithms for closing the local gaps and completing the alignment. A gap is defined as an interruption in the MUM alignment.

2.2.3 Strength and limitation

The strength of MUMmer1 lies on the speed at which it processes long alignments and that MUMmer1 produces much less false positives than its predecessors. The limitation of this algorithm is that this approach assumes that there exist a single long alignment, which may not be always true, as they may be a huge chain of reversals between two genomes.

2.3 MUMmer2

Three significant technical improvements in the core algorithms of MUMmer2 are listed in the following:

2.3.1 Reducing memory usage

By employing techniques described by [Kurtz, 1999], the amount of memory used to store the suffix tree is reduced to at most 20 bytes/base pair (or amino acid, or other character). The maximum memory usage occurs when each internal node in the suffix tree has only two children. In practice, however, many nodes have more than two children, which reduces the actual memory requirement.

2.3.2 A new alternative algorithm

The MUMmer1 [Delcher et al., 1999] employs an algorithm that builds a generalized suffix tree for two input sequences to find all MUMs. Although this algorithm is still available in the MUMmer2 system, a new alternative algorithm was introduced as the default algorithm to find all MUMs. Instead of storing the generalized suffix tree for the two input sequences, this new algorithm stores only one input sequence in the suffix tree. This sequence is called the reference sequence, R. The other sequence, which is called the query, Q, is then streamed against the suffix tree, exactly as if it were being added without actually adding it. This technique was introduced by [Chang and Lawler, 1991] and is fully described in [Gusfield, 2007].
Using this streaming process, we can identify where the query sequence would branch off from the tree, thus we can find all matches to the reference sequence. Whenever a branch occurs at an internal node with just a single leaf beneath it, the matching subsequence is unique in the reference sequence. Note that these matches are not necessarily unique in the query sequence. Then, by checking the character immediately preceding the start of this match, we can determine whether it is maximal. As the consequence of streaming through the query, i.e. outputting matches as we find them, we do not know what sequence will occur

later in the query.
By using this new algorithm, all maximal matches between query sequence and a unique substring of the reference sequence can be identified in time proportional to the length of the query sequence. The advantage of this method is that only one of the two sequences is stored in the suffix tree, reducing the memory requirement by at least half. Furthermore, because of the streaming nature of the algorithm, once the suffix tree has been built for an arbitrarily long reference sequence, multiple queries can be streamed against it. Therefore, we do not have to re-build the suffix tree if one of the input sequences is changed. In fact, [Delcher et al., 2002] have used these programs to compare two assemblies of the entire human genome (each approximately of 2.7 billion characters long), using each chromosome as a reference and then streaming the other entire genome past it.

2.3.3 Clustering matches

Biologists observed that a pair of conserved genes are likely to correspond to a sequence of MUMs that are consecutive and close in both genomes. At the same time, this sequence of MUMs generally has a sufficient length. The set of such MUMs is called a cluster.
For example, as shown in Figure 2.3, for the given two sequences, there are two clusters: 123 and 56. MUMmer1 presumed that two complete sequences were

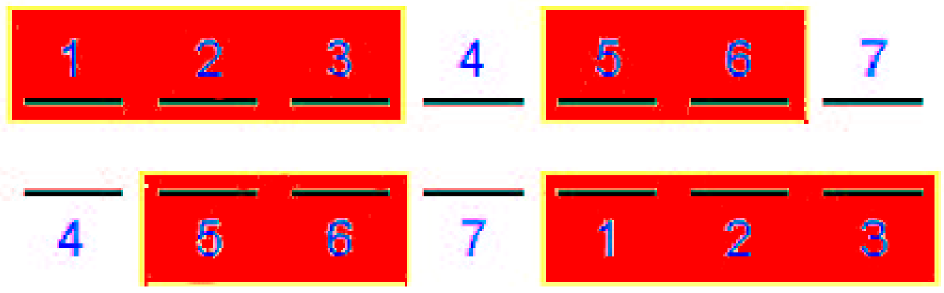


Figure 2.3: Cluster of MUMs [Kin, 2006]

to be aligned, and that no major rearrangements would have occurred between them. Hence, it computed a single longest alignment between the sequences. In order to facilitate comparisons involving unfinished assemblies and genomes with significant rearrangements, a module has been added in MUMmer2. This module first groups all the close consecutive MUMs into clusters and then finds consistent paths within each cluster. The clustering is performed by finding pairs of matches that are sufficiently close and on sufficiently similar diagonals in

an alignment matrix (using thresholds set by the user), and then computing the connected components for those pairs. Within each component, a longest common subsequence is computed to yield the most consistent sequence of matches in the cluster.
As the result of the additional module, the system outputs a series of separate, independent alig nment regions. This improvement is the main key for MUMmer2 to achieve a better coverage.

2.3.4 Applications

MUMmer2 has been applied to solve alignment of incomplete genome and comparative genome annotation problems. Two new solvers with MUMmer as the core algorithm are developed to solve this two problems, they are NUCmer (nucleotide MUMmer) and PROmer (protein MUMmer). Experiments showed that MUMmer is useful to solve the two problems stated above. Detailed description for these two applications can been found in [Delcher et al., 2002].

2.4 MUMmer3

Basically, the structure of MUMmer3 is pretty similar to that of MUMmer2, but some amendments were done to further improve its time and space complexity. In terms of space, MUMmer3 is slightly more efficient as it uses approximately 16 bytes only to store a base pair of the reference sequence in the suffix tree. The process of streaming the query sequence past the reference suffix tree is maintained, so that the memory requirements do not depend on the size of the query sequence at all.
In MUMmer3, the concept of MUM is slightly relaxed that it need not to be unique in both input sequences. We can opt to find all non-unique maximal matches, all matches that are unique only in reference sequence, or all matches that are unique in both sequences. This feature was added as it was observed that the uniqueness constraint would prevent MUMmer from finding all matches for a repetitive substring.
The most critical improvement in MUMmer 3 is a complete re-write of the core suffix tree library, based on the compact suffix tree representation. It was implemented by Stefan Kurtz [Kurtz, 1999] and explained in his various publications. MUMmer3 requires approximately 25% less memory than MUMmer2 and it runs slightly faster. Compared to MUMmer1, MUMmer3 is more than twice faster and uses less than half of the memory. Another additional feature that is only available in MUMmer3 system, is a new Java viewer, DisplayMUM. It is a new graphical output program to generate images in fig-format or PDF, showing the

alignment of a set of contigs to a reference chromosome.

Chapter 3

Xipe Totec: parallelization of whole genome alignment

3.1 Parallelism technique

General-purpose, commodity CPUs currently have SIMD (Single Instruction Multiple Data) functional units and corresponding SIMD instructions. This kind of CPUs allow to be used in several ways of parallel techniques, such as data-level parallelism.
In addition to using SIMD technology, the availability of HPC clusters makes possible to execute the same task with a different kind of data.
The sequential version of the MUMmer's algorithm, as presented in Whole genome alignment with MUMmer, trades extra computation for memory and high computation time when executed in a single machine.
Previously the algorithm for sequence alignment was described in detail. Now our own proposal of a parallelization of whole genome alignment with MUMmer is explained, Xipe Totec. There are two resources to improve in this algorithm:

- Memory usage.
- Running time.

To improve the performance of the algorithm a data-level parallelism technique is deployed in advance to genome alignment.
Our technique is divided in three phases following:

1. Splitting genome data according to the number of available cores.
2. Parallel execution of as many instances of mummer as available cores.
3. Get the list of MUMs from the whole set of MEMs[1] found in the previous

[1]Maximal Exact Match

phase.

The following Figure 3.1 shows the process of our data-level parallelism technique.
The division of genome data was used using the paradigm of data-level parallelism which consists of a generation of chunks of a sequence with a fixed size and a fixed overlap. One issue arises when a genome is splitted because of the heuristic used in the algorithm is affected. So that, a longer sequence is more likely to have a better finding of MUMs while a smaller one can produce MUMs which are not effective MUMS.
Another consideration was the genome structure, because a genome is build from a finite alphabet $\Sigma = \{a, g, c, t, n\}$ but according to the MUMmer's algorithm each alignment is made using only the nucleotide base pairs, this means that letter "n" in a biological way can mean anything: (a,g,c,t). A complex structure of a genome can have a huge impact of our data level parallelism.
The main idea in MUMmer is its heuristic, Maximal Unique Match (MUM), based on this concept it is possible to cover a huge region of a genome when reference and query genomes are very closely related. However to get a MUM, it requires an important feature its uniqueness. Uniqueness can only be found when a whole genome is checked. In other words, after finding MUMs within a chunk it is not possible to determine if the MUM found is or not a "unique" MUM, globally in the genome, because these MUMs are unique only in the chunk that has been read, the rest of the genome it is not known.
One solution to solve this problem is to drop the MUMmer's heuristic in order to be able to find the correct MUMs when we apply our data-level parallelism. The new approach is to find a Maximal Exact Match (MEM), a MEM allows to drop the uniqueness of a match but with a high computational cost: a brute-force approach.
A MEM as shown in Figure 3.2 has the advantage of getting the true MUMs regardless a division of genome data. In figure 3.2 R stands for reference genome and Q stands for query genome, this example shows that blue and red matches have exact matches but turn out not to be unique, then not a valid MUM; a true MUM is the green match.
Nevertheless, the major problem with the use of MEM is that the number of occurrences increases exponentially when shorter MEM are used. Moreover, the high ratio of MEMs requires to save them in some place, memory or disk, that means a heavy use of resources in both CPU and Memory. This drawback is the main disadvantage because the hits of MEMs are increasing with the size of the genome. A new phase has to be designed to reduce the use of resources when a short MEM is searched.

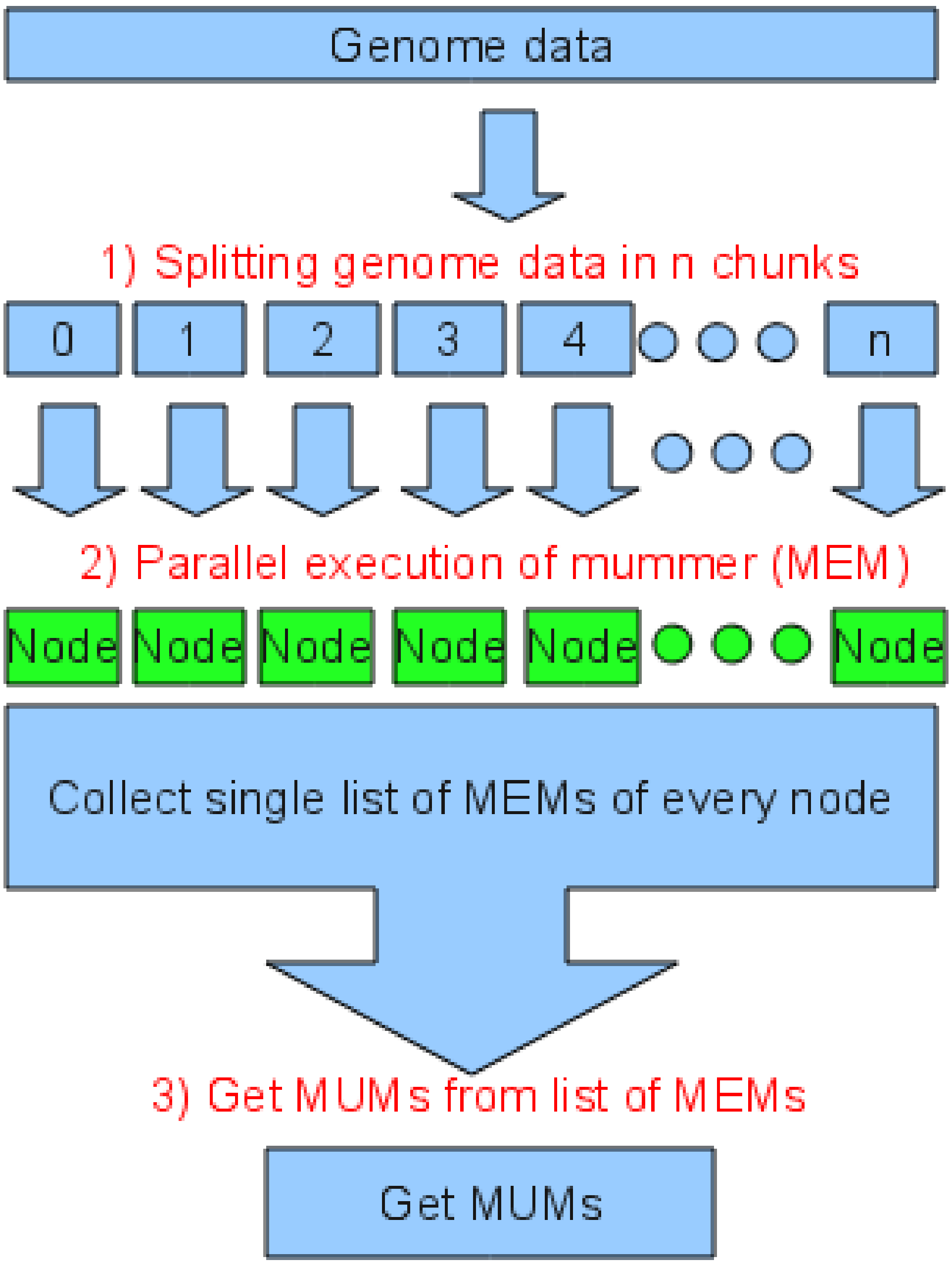

Genome data
1) Splitting genome data in n chunks
0
1
2
3
4
n
2) Parallel execution of mummer (MEM)
Node
Node
Node
Node
Node
Node
Collect single list of MEMs of every node
3) Get MUMs from list of MEMs
Get MUMs

Figure 3.1: Data-level parallelism technique for whole genome alignment

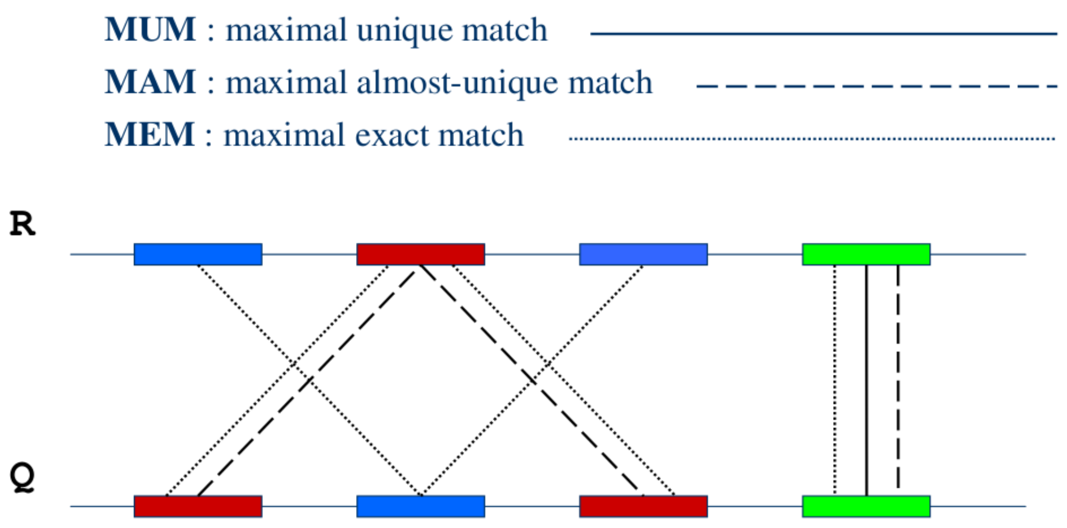

MUM : maximal unique match
MAM : maximal almost-unique match
MEM : maximal exact match
R
Q

Figure 3.2: Types of matches

3.1.1 Reducing memory usage

In order to reduce the high memory usage of MUMmer's sequential version a division of the genome is proposed, as it is explained in 3.2. However, it must be pointed out that an "aggresive" division of the genome can lead to a wrong alignment. This is because of the data structure used, suffix tree. Suffix tree has got a property which can save every suffix of a given string (genome) but when there is a division the new trees can be completely different to a single suffix tree.

3.1.2 Reducing computation time

One goal in any algorithm is to be fast and accurate, one way to accomplish this in MUMmer was to use data-level parallelism so that a chunk of genome data can be aligned by a processor meanwhile other chunks are computed in the same way: a divide and conquer technique. Although, we can improve the execution time we have to get the same output regardless the use of parallelization. So that, instead of looking for MUMs we now, in our proposal, look for MEMs in order to get the same output. Since this change can affect the performance of our approach a set of experiments have to be carry out to know how big a genome can be aligned with the technique which is explained in the following section.

3.2 Implementation

In this section we explain in detail how our proposal is implementedi and the modifications in order to be executed in MUMmer. The following diagram, see Figure 3.3, shows the add-ons of our approach, one is executed before MUMmer and the another after MUMmer. The following sections explain how the split of

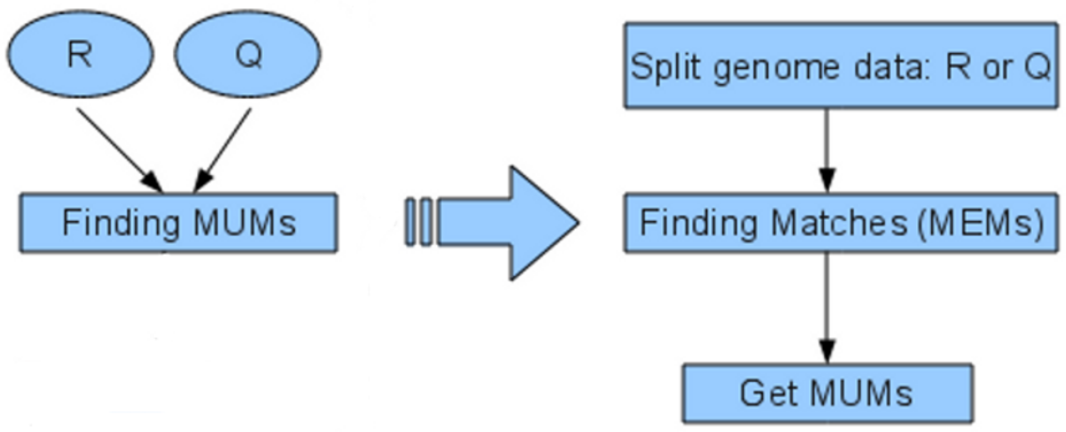


Figure 3.3: Xipe Totec: proposal for parallelization of whole genome alignment

genome data, search of MEMs and get MUMs are carried out in our proposal.

3.2.1 Split genome data

As it was previously explained, the approach is to use a fixed size division of genome data in as many chunks as many available cores. Xipe Totec needs to know how many cores will be used in order to divide the genome data.
One key aspect of Xipe Totec is the way to split a genome. To align a genome requires a reference genome so that, there are two ways of using Xipe Totec:

- Splitting reference genome.
- Splitting query genome.

Both of them can reduce the computation time or memory usage. There is a restriction in Xipe Totec while handling files of each genome, these files must be in FASTA format and they have to be single Fasta files. So that only one entry must exist in each file.
A sequence or genome in FASTA format begins with a single-line description,

followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (>) symbol at the beginning. It is recommended that all lines of text be shorter than 80 characters in length. An example sequence in FASTA format is:

```
>gi|114796141|ref|NC_006492.2|NC_006492 Pan troglodytes chromosome Y,
reference assembly (based on Pan_troglodytes-2.1)
GAATTCAATGGAATAGAATGCAACGCAATGGAAAGTTGACATGTAATGTGAGCTGAGATTGTGCCACTGC
ACTCCAGCCTGGGTGACACAGTGATATCCTGTCGAAAGAAAGGAATGGAATGCAGTGGAGTGAAATGGCA
TGGAATGAAATGGAATGGAGTGACGAGGACTGGAGTGGAATGGATTGGAGTGGAGTGGAGTGGAGAGGAG
TGGAGTGGAATGGAATGGTGAAATGAAATGTGAGGTGAGATTGTGCCTCTGCGCTCCAGTGCACAATAAT
TATCAATTATTAACTATTAACTTATTAATTAATTATTAATTGATATTTATTAGTTAATATGAATTAATAT
AATTAATATTATAGAAATTGTAATATGTTATCATTTTCTATAATATTAATATGTTGTGAAAATATCCTTA
CTTTCTTATTTGTCCAAGGTTCAATATTTTGCAGTCTCTACCTCACCCTGTGAAGCATAAAGATTTTACA
TGCTGTAAAAATAATACATACTTTTTGTGCATAGCGATTGCACAGCTTTTATTTGGCTGACAATAGCTAA
TGTTTACTTCTTCATTTTCTATTTACTGATTTTTCTTCATTTAGTATATACTACTTTATCATAAAAATAA
```

Blank lines are not allowed in the middle of FASTA format. The first step to split a genome is get its length. The second step is divide the genome with chunks of equal size and a fixed overlap. This overlap allows to to have a better finding of MEMs and longer prefixes. The number of chunks is ruled by the number of available cores and the overlap is fixed by the user.
To split genome data a perl script was coded to do this task, the script requires the following arguments:

- Genome data to be splitted.
- Number of chunks.
- Overlap size.
- Location to save the genome data.

3.2.2 Finding MEMs

This phase requires the mummer program, one of the several small programs in the MUMmer suite. mummer has several options. In order to get a correct alignment, our proposal requires to compute MEMs instead of MUMs, so that mummer is executed with:

- -n: to match only nucleotides.
- -maxmatch: option to get MEMs.

- -l length: to find MEMs of a some minimum length.

List of MEMs is saved to a file which has the following format:

```
>Information about the sequence
Position_in_R Position_in_Q Length_of_MEM
```

This list has to be joined with the output of every chunk computed and then the whole list is manipulated in the following phase, 3.2.3.

3.2.3 Getting MUMs

This is the most important phase in our approach because it outputs the final list of MUMs those that are the same to the serial execution of mummer.
This phase needs the list of MEMs to process them and find those matches that are unique. The following diagram shows the basic idea behind this phase, see Figure 3.4 To get the MUMs we filter those MEMs that are unique in the list and order them using a modified version of LIS[1] algorithm. The algorithm is shown below:

```
Input: List of MEMs: Position in R, position in Q, length of MEM
Sort MEMs by increasing position and decreasing length in R
for $i := 0$ to $n$ in Total_MEMs do
    if MEM[$i$] is not unique then
        Sort this subset by increasing position in Q and pick up the first MEM and drop the rest
    else
        MEM[$i$] is a MUM
    end if
end for
Sort MEMs by increasing position and decreasing length in Q
for $i := 0$ to $n$ in Total_MEMs do
    if MEM[$i$] is not unique then
        Sort this subset by increasing position in R and pick up the first MEM and drop the rest
    else
        MEM[$i$] is a MUM
    end if
end for
```

The output of this algorithm gives the MUMs.

[1]Longest Increasing Subsequence

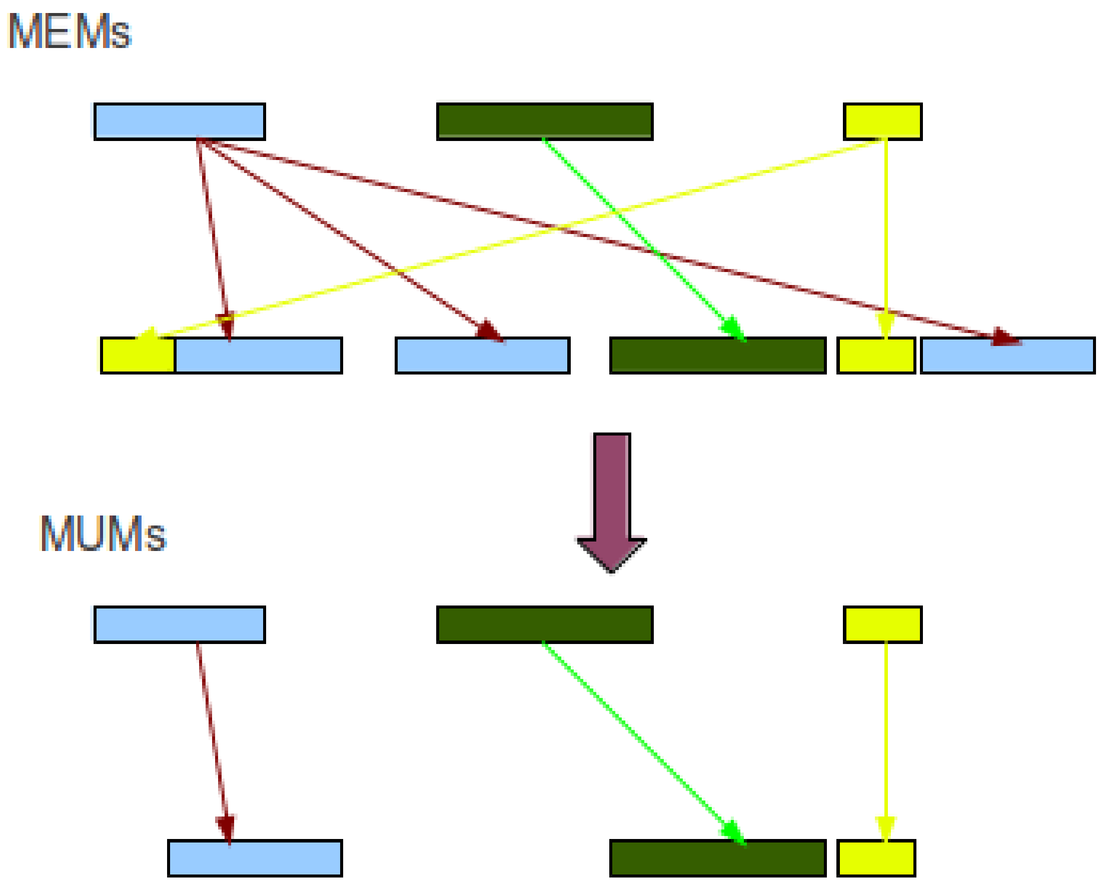


Figure 3.4: Xipe Totec: Finding the real MUMs

Chapter 4

Experiments and results

4.1 Testing environment

To verify that our approach, Xipe Totec, can align a whole genome a set of tests were analyzed. These tests were carried out in a cluster with the following features:

- Hardware:
  - Processor Dual-Core Intel(R) Xeon(R) CPU 5160 @ 3.00GHz 4MB L2 (2x2)
  - Number of processors: 2
  - RAM: 12 GB Fully Buffered DIMM 667 MHz
- Software:
  - Linux Kernel 2.6.16.46-0.12-smp x86_64 GNU/Linux
  - gcc 4.3.2
  - MUMmer 3.22
  - Perl 5.8.8

4.2 Results

Several experiments were carried out to check if this parallelization technique can align genomes of several sizes. To verify its correct output an hypothesis is proposed:

A whole genome alignment can be performed with a parallel distribution of data and reducing the execution time and the need of memory.

In previous chapter Xipe Totec was described and two ways of execution can be performed. So that each genome alignment was executed following these phases:

- Split genome data:
  - Genome size: from 4,6 Mbp[1] up to 115,86 Mbp.
  - Division of genome in 2, 4, 8, 16 chunks.
  - An overlapping size of 5000bp.
- Test for several size of MEMs:
  1. 20 bp
  2. 50 bp
  3. 100 bp
  4. 500 bp
  5. 1000 bp

Then each way of parallelization is executed:

- Align the query genome by splitting it against a global whole reference genome, see Figure 4.1.
- Align the whole query genome against every chunk of reference genome, see Figure 4.2.

4.2.1 First test: Ecoli bacteria

Our first test was with a small genome, two variants of Ecoli bacteria:

- EcoliK12: 4,63 Mbp
- EcoliO157H7: 5,49 Mbp

[1]Two nitrogenous bases paired together in double-stranded DNA by weak bonds; specific pairing of these bases (adenine with thymine and guanine with cytosine) facilitates accurate DNA replication; when quantified (e.g., 8 bp), refers to the physical length of a sequence of nucleotides of Medicine [2011]

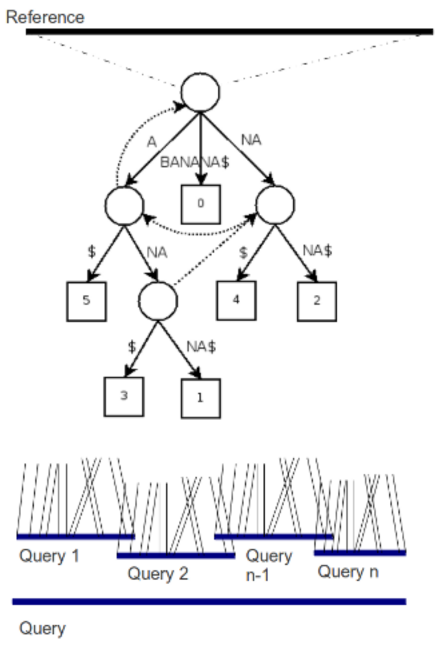


Figure 4.1: Split query genome.

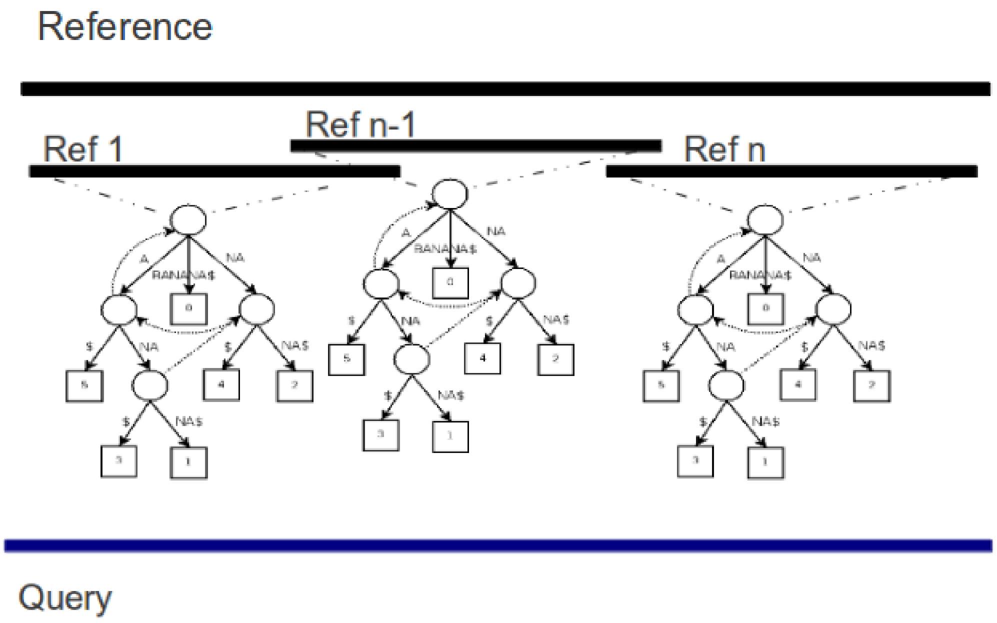


Figure 4.2: Split reference genome.

The first parameter to measure was the construction of suffix tree. The time used to build a suffix tree was proportional to the size of the reference genome, so that we had better times (smaller) when we split the reference genome. In Figure 4.3 is shown that with several configurations of overlapping the construction time is quite similar. In the other hand the construction time is not improved when the query genome is divided, see Figure 4.4, it is because the reference genome is always the same.

The second parameter to test was the computation time when finding MEMs,

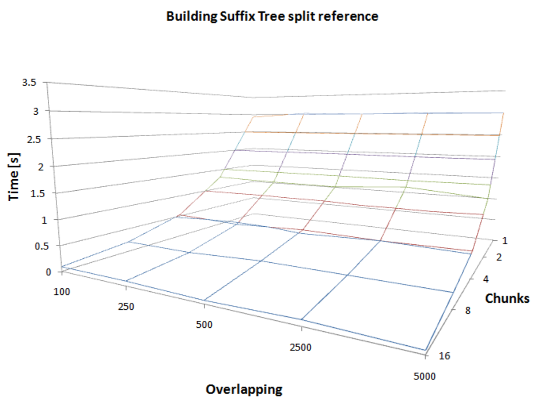


Figure 4.3: Build of Suffix tree split EcoliK12 reference.

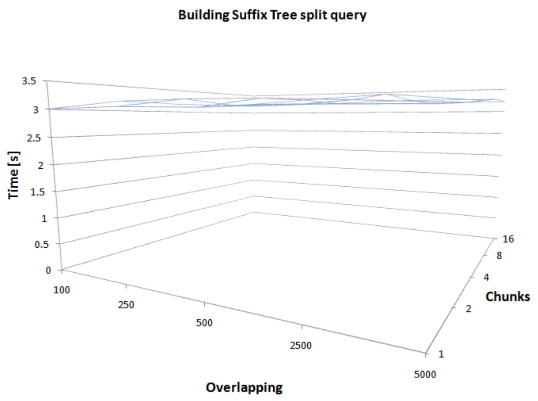


Figure 4.4: Build of Suffix tree split EcoliO157H7 query.

this is important because our approach needs MEMs in order to get the MUMs and have the same alignment. In the Figure 4.5 is shown several configurations for testing purposes. Times are within the same range and we got better times when the genome size was divided in 16 chunks meanwhile with 2 and 4 chunks the gain was less than 10% of the serial execution.

On the other hand, when a division is made for query genome the computation time was greatly improved, in a linear way. In the Figure 4.6 is shown several configurations. Times are within the same range and performance is linear when more chunks are processed.

In addition to computation time, another parameter was measured: memory usage. A fair use of memory allows to have better overall performance. As it is shown in Figure 4.18 a small genome needs more than 18 times its own size. Moreover a better reduction in the use of memory can be achieved when the reference genome is split. That is because the most memory-consuming item in MUMmer is the suffix tree, so that a shorter reference genome reduces the memory usage. On the other hand Figure 4.18 also shows that there is no enough reduction in the use of memory when the query genome is split.

The final parameter to test was to check if the alignment was the same to the

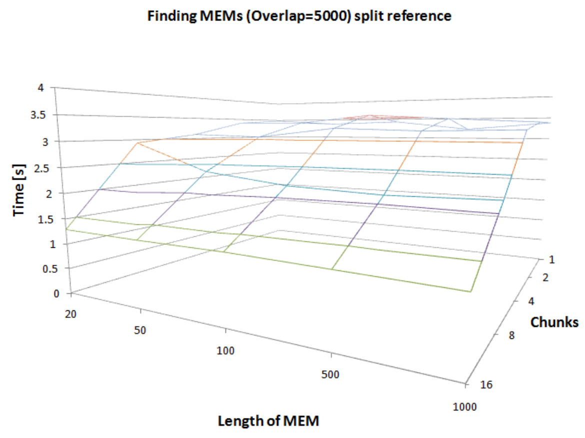


Figure 4.5: Finding MEMs split EcoliK12 genome, Overlap=5000bp.

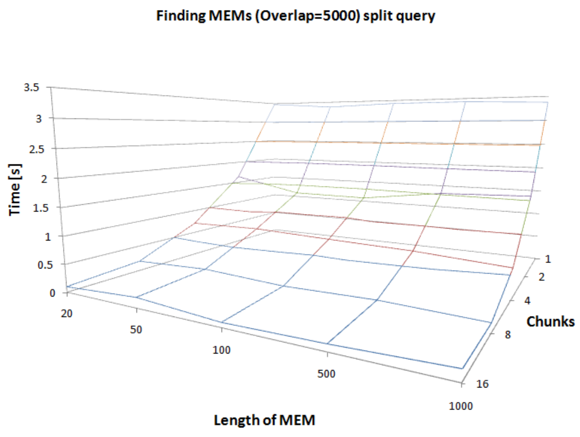
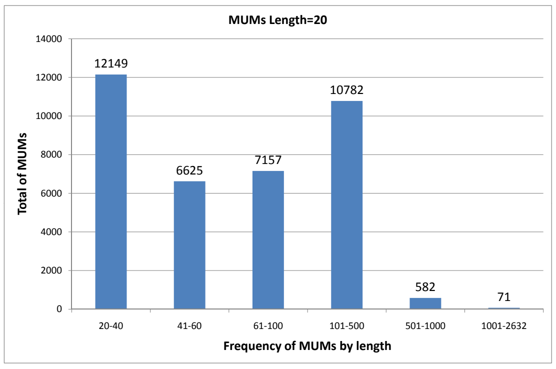


Figure 4.6: Finding MEMs split EcoliO157H7 genome, Overlap=5000bp.

sequential version of MUMmer. Since we compute MEMs instead of MUMs the algorithm described in 3.2.3 we had to check that our list of MUMs is the same to the serial execution of MUMmer. In the Figure 4.12 is shown the total of valid MUMs for several sizes of MUM.
Hence, the following list of Figures 4.7, 4.8, 4.9, 4.10 and 4.11 gives a whole idea of our set of MUMs that have to be the same regardless the division used.



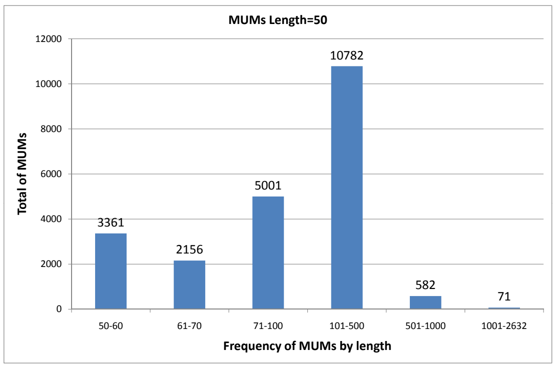


Figure 4.7: Frequency of MUMs, length of 20 bp, found when aligning EcoliK12 vs. EcoliO157H7.

Figure 4.8: Frequency of MUMs, length of 50 bp, found when aligning EcoliK12 vs. EcoliO157H7.

Our first experiment has got the same whole genome alignment to the sequential version of MUMmer. To clarify our result, the following Figure 4.13 shows the percentage of MEMs which were discarded by our algorithm and this output is

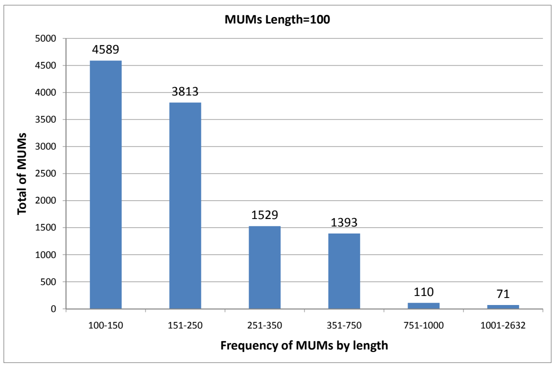


Figure 4.9: Frequency of MUMs, length of 100 bp, found when aligning EcoliK12 vs. EcoliO157H7.

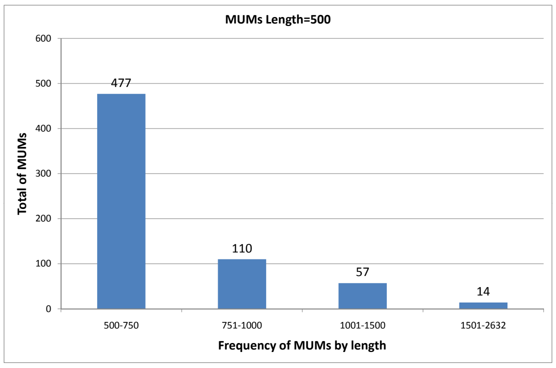


Figure 4.10: Frequency of MUMs, length of 500 bp, found when aligning EcoliK12 vs. EcoliO157H7.

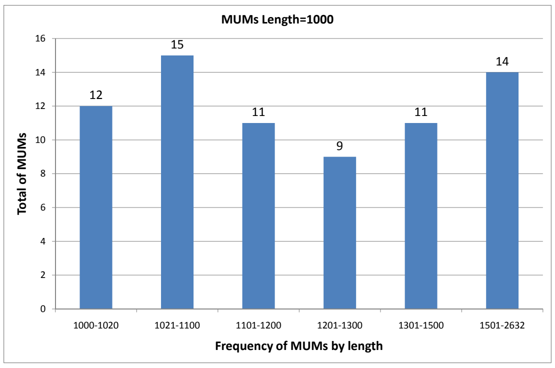


Figure 4.11: Frequency of MUMs, length of 1000 bp, found when aligning EcoliK12 vs. EcoliO157H7.

the same list of MUMs to that computed in the sequential version.

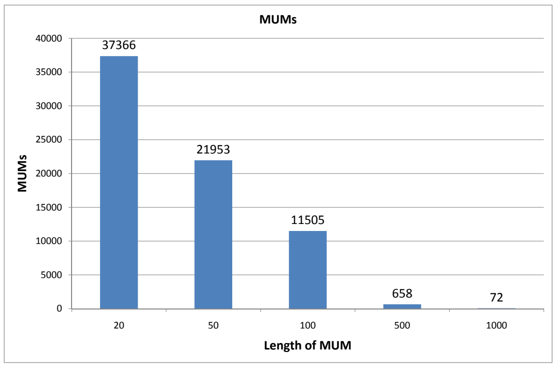


Figure 4.12: Total of valid MUMs for EcoliK12 vs. EcoliO157H7 alignment.

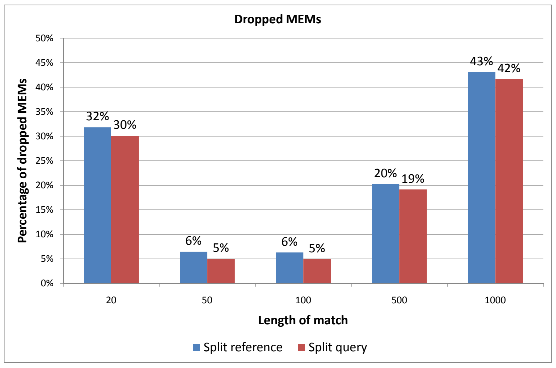


Figure 4.13: Percentage of dropped MEMs in order to get the same list of MUMs for the sequential version of MUMmer, when aligning EcoliK12 vs. EcoliO157H7.

4.2.2 Second test: primate genome

Our second experiment was with a genome 20 times bigger than Ecoli, the human chromosome 13 as reference genome and the chimpanzee chromosome 13 as query genome. The correct size of each genome is:

- Human chromosome 13: 95.78 Mbp
- Chimpanzee chromosome 13: 115,86 Mbp

The first parameter to measure was the construction of suffix tree. Since this genome is much bigger than our first test, the memory requirements are bigger than the previous set of genomes. In Figure 4.14 is shown how, again, a division of reference genome improves the building time of suffix tree, up to 27 times when the reference is divided in 16 chunks. Though a quick build of suffix tree is achieved when the reference is split the but this improvement is not possible when the query genome is split, in Figure 4.15 is shown the building time for suffix tree which remains constant. The second parameter to test was the computation time to find the MEMs. As it was explained in the previous chapter we follow a brute-force approach to get the same list of MUMs when our technique is used.

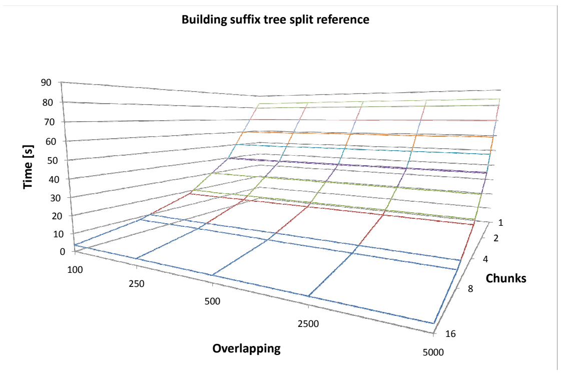


Figure 4.14: Construction of Suffix tree split Human Chr13 reference.

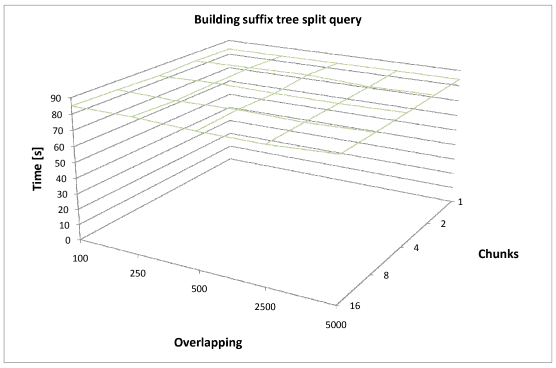


Figure 4.15: Construction of Suffix tree split Chimpanzee Chr13 query.

In other words, it might be possible that for some genome sizes our approach can be worse than the sequential version.
In the Figure 4.16 is shown the several configurations tested when the reference genome is split.
It is important to point out that this test shows that for some configurations our approach does not work at all: when the Match length is very short, like 20 bp. The answer to this problem is that when a small match is searched in a big genome the number of matches can become exponentially. This undesirable feature can be seen in Figure 4.21 and Figure 4.20 which shows the high number of hits when a short match is searched.
On the other hand when the query genome is split the reduction of time to find MEMs is only of the magnitude of 20%.
Memory usage is a big issue when a genome goes from a few Mbp to hundreds of Mbp, our reference genome is almost 100Mbp and the requirement for memory can be seen in Figure 4.19. For a genome of 95Mbp, the algorithm requires more than 1.6GB, from our experiments we can show that if we divide genome reference in 16 chunks, memory requirement reduces to only 12.5% of the sequential version use of memory.
The most important step in our test was to verify that our parallelization can get the same alignment (same list of MUMs from the sequential version), in Figures 4.20, 4.22, 4.23, 4.24, 4.25 and 4.26 reader can watch the list of MUMs which should be the same when we apply our parallelization technique. In order to know if our approach does the job we filter those matches which are MUMs. To understand this point, the Figure 4.21 shows the number of matches discarded.

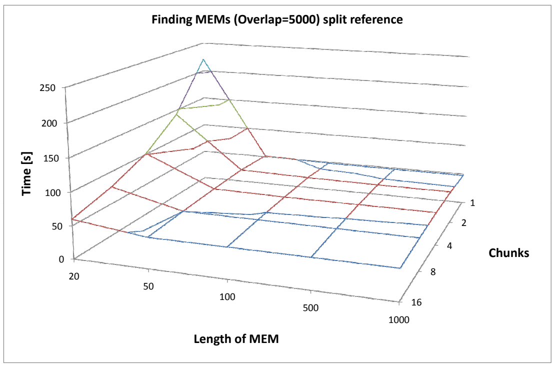


Figure 4.16: Finding MEMs split Human Chr13, Overlap=5000bp.

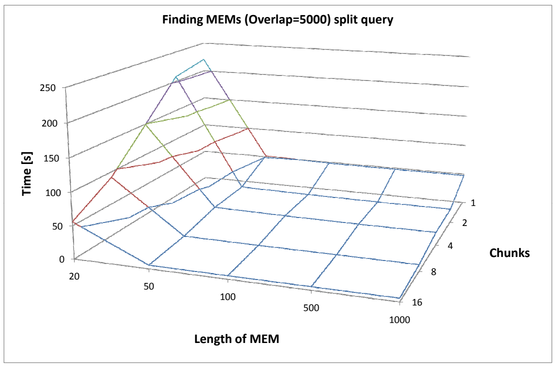


Figure 4.17: Finding MEMs split Chimpanzee Chr13, Overlap=5000bp.

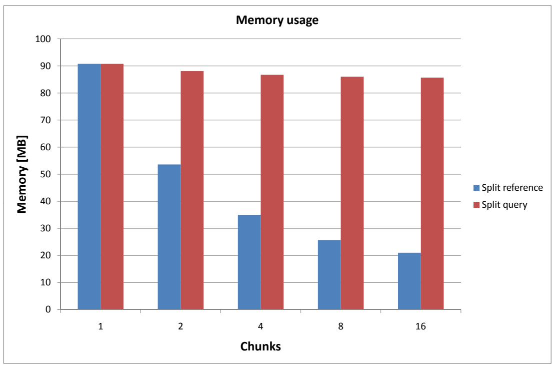


Figure 4.18: Use of memory when aligning EcoliK12 genome vs. EcoliO157H7 genome.

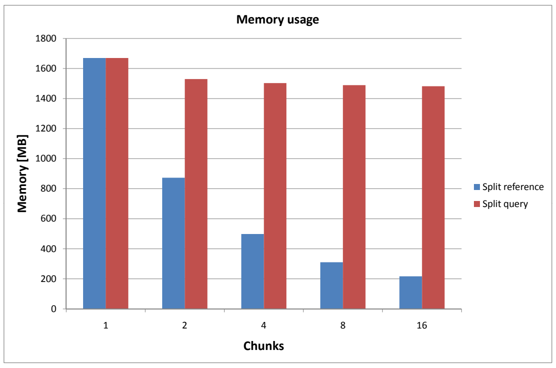


Figure 4.19: Use of memory when aligning Human Chr13 vs. Chimpanzee Chr13.

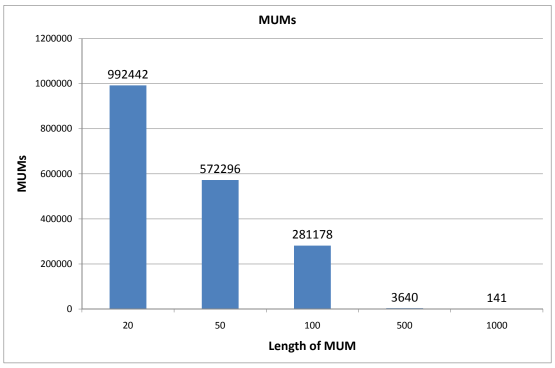


Figure 4.20: List of MUMS for several sizes of MUM when aligning Human Chr13 vs. Chimpanzee Chr13.

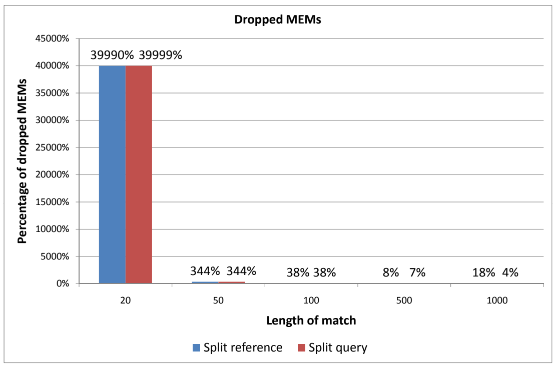


Figure 4.21: Percentage of dropped MEMs in order to get the same list of MUMs for the sequential version of MUMmer, when aligning Human Chr13 vs. Chimpanzee Chr13.

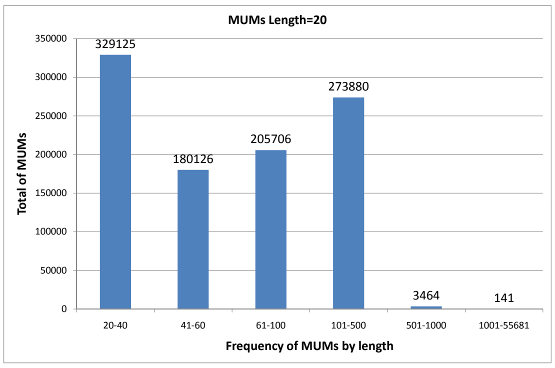


Figure 4.22: Frequency of MUMs, length of 20bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

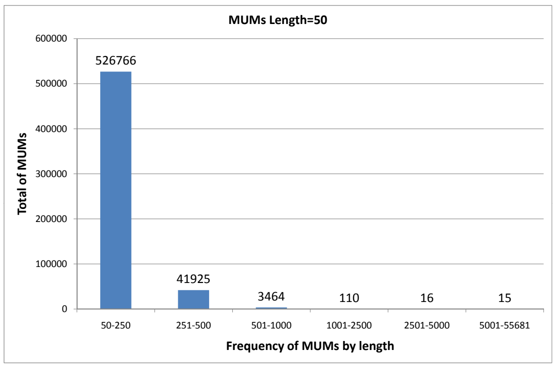


Figure 4.23: Frequency of MUMs, length of 50bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

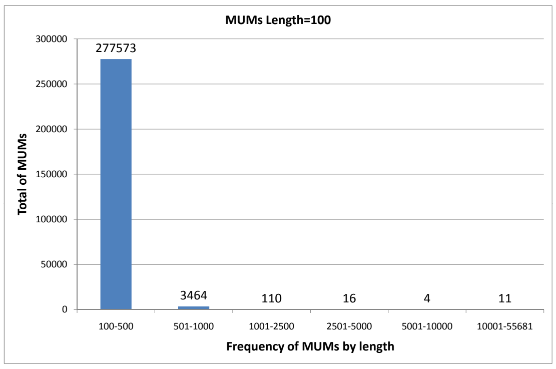


Figure 4.24: Frequency of MUMs, length of 100bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

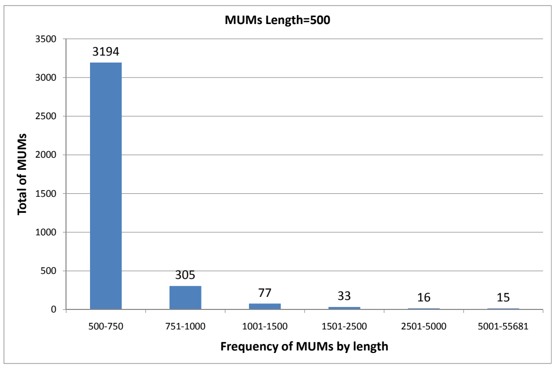


Figure 4.25: Frequency of MUMs, length of 500bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

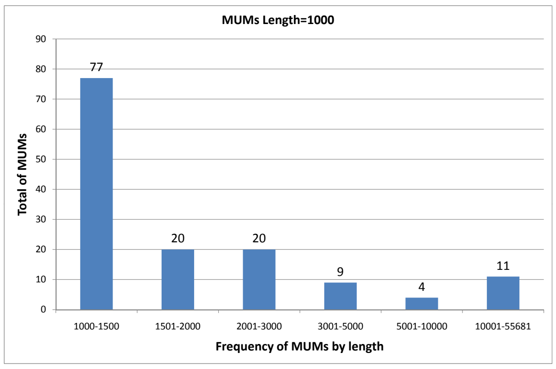


Figure 4.26: Frequency of MUMs, length of 1000bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

Chapter 5

Conclusions

5.1 Achievements

We have set in place a parallelization of whole genome alignment, Xipet Totec, and used MUMmer to evaluate the alignment of two genomes. We have found that a data level parallelism allows an approximation of a correct alignment.
We view this work as a starting point toward the goal of completely automating alignment parallelization. In addition to split data genome, we will also need to automatically choose seeding strategies and thresholds.
The next few years will provide a unique challenge and opportunity to alignment research. One of the immediate goals of the genomics field is to sequence and align a large number of species in order to study their biology and evolution through comparative whole genome alignment. In addition, the enormity of genome data will be dizzying, and alignment will be a key area of genomics - finding all interesting connections between diverse sequences.
A few years ago several unsolved problems in alignment were proposed, which fell into five topics:

1. Improved pairwise alignment with a heuristic basis.

2. Effective multiple genomic sequence alignment.

3. Better alignment browsers.

4. Rigorous methods for evaluating the accuracy of alignment.

This work has made some progress, especially in direction 1 which has shown the efficiency to handle memory management and the parallel distribution of genome data in order to get a quicker whole genome alignment. Additionally, a better heuristic should be used to allow the full use of clusters and multicore architectures. This new heuristic may improve the finding of matches so that every process can handle its chunk and it could notice if a match is unique or not by checking the other chunks. In this way a new concept of distributed MUM is defined:

Definition 4. *Distributed Maximal Unique Match(DMUM) substring is a common substring of the two genomes that is longer than a specific minimum length d such that it is maximal, that is, it cannot be extended on either end without incurring a mismatch and it is unique in any of the chunks of the genome.*

DMUM works by checking first if it is a MUM locally and then asking to the other chunks if this MUM is a real MUM in the whole genome. The DMUM has the advantage of having less intermediate data (MEMs) but with an overhead of communications between each chunk and the computation time to check if the match is unique or not.
Directions 2, 3 and 4 are more resistant: sequence evolution is complicated and difficult to model and therefore rigorous yet practical heuristic methods for whole genome alignment, and objective criteria for evaluation of alignments, are really hard to obtain.
Our technique is a startup to explore the multiple faces of bioinformatics to do a better use of the available computer resources. This thesis shows that a novel parallelization and the knowledge of genome data can improve the run time and memory.
Several configurations were tested and some drawbacks were found. The main restriction to implement this technique is the limited memory resource in our environment test.

5.2 Drawbacks

Our novel approach has some limitations which are listed below:

- It can work well with genome data of size less than 100Mbp.
- The use of a short match length can heavily impact in performance of our approach.
- This technique does not provide any feature of checkpoint.
- Our approach does not take advantage of any parallel programming language (OpenMP, OpenMPI, etc.).

These drawbacks are some opportunities to work with in the near future.

5.3 Future work

Looking to the next few years, the following are just some of the problems that are significant to address:

- Methods to evaluate alignment accuracy. This goes at the core of the whole genome alignment problem: which regions are alignable?, and what is a correct alignment?
- Methods that can align neutrally evolving DNA between multiple distant species, whenever the sequences have not diverged enough in principle to look non-orthologous.
- A definition of alignability - at what point is no longer possible to do meaningful sequence alignment. Or rather, at what point can one conclude that two sequences are no longer related?
- Aligning under different models in different parts of the sequences, according to existing annotations of features such as genes, or by using automatic methods that recognise the evolutionarily constrained, neutral and plain unalignable parts.
- A rigorous way to identify breakpoints of rearrangement events in the context of multiple genome alignment.
- Methods to analyse, navigate and query multiple genome alignments in order to identify the alignable parts, the conserved parts.

In addition, there are more issues related to computational tasks like a fair usage of available resources to do whole genome alignment. Among the desirable attributes any whole genome alignment algorithm should have are:

- Time: in order to be able to process whole genomes.
- Space: a clever use of data structures to not run out of memory.

Both of these attributes were enhanced in MUMmer to provide a first approach to whole genome alignment in HPC clusters.

References

Bray, N., Dubchak, I., and Pachter, L. (2003). AVID: A global alignment program. Genome Research, 13(1):97–102. 6

Brudno, M., Chapman, M., Göttgens, B., Batzoglou, S., and Morgenstern, B. (2003a). Fast and sensitive multiple alignment of large genomic sequences. BMC Bioinformatics, 4(66). 6

Brudno, M., Do, C. B., Cooper, G. M., Kim, M. F., Davydov, E., Green, E. D., Sidow, A., Batzoglou, S., and (2003b). LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. Genome Res, 13(4):721–731. 6

Chang, W. I. and Lawler, E. L. (1991). Sublinear expected time approximate string matching and biological applications. Technical Report UCB/CSD-91-654, EECS Department, University of California, Berkeley. 16

Delcher, A. L., Kasif, S., Fleischmann, R. D., Peterson, J., white, O., and Salzberg, S. L. (1999). Alignment of whole genomes. Nucleic Acids Research, 27(11):2369–2376. 13, 16

Delcher, A. L., Phillippy, A., Carlton, J., and Salzberg, S. L. (2002). Fast algorithms for large-scale genome alignment and comparison. Nucleic Acids Research, 30(11):2478–2483. 17, 18

Gusfield, D. (2007). Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge Univ. Press. 16

Hohl, M., Kurtz, S., and Ohlebusch, E. (2002). Efficient multiple genome alignment. Bioinformatics, 18(suppl_1):S312–320. 11

Kaderali, L. and Schliep, A. (2002). Selecting signature oligonucleotides to identify organisms using DNA arrays. Bioinformatics, 18(10):1340–1349. 11

Kin, K. S. W. (2006). Lecture 5: Whole Genome Alignment - Sept 22, 2006. ix, 17

Kurtz, S. (1999). Reducing the space requirement of suffix trees. Software Practice and Experience, 29:1149–1171. 11, 16, 18

Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004). Versatile and open software for comparing large genomes. Genome Biology, 5(R12):R12.1–R12.9. ix, 5

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. J. ACM, 23:262–272. 11

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology, 48(3):443–453. 3, 4

of Medicine, U. N. L. (2011). Base pair. 30

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. Journal of Molecular Biology, 147:195–197. 4