Towards speed up of Whole Genome Alignment using Distributed Suffix Tree

Julio Cesar Garcia Vizcaino*[1], and Antonio Espinosa[2]

[1]Computer Architecture & Operating Systems Department (CAOS), Universitat Autònoma de Barcelona,Bellaterra (Barcelona), Spain.
[2]Computer Architecture & Operating Systems Department (CAOS), Universitat Autònoma de Barcelona,Bellaterra (Barcelona), Spain.

Email: Julio Cesar Garcia Vizcaino*- juliocesar.garcia@e-campus.uab.cat; Antonio Espinosa - antonio.espinosa@caos.uab.es;

*Corresponding author

Abstract

Background: This work explores the potential of using high performance computing to accelerate whole genome alignment in MUMmer, a whole genome alignment software. 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 on 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 whole genome alignment 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.

Results: We evaluate several genome sizes in order to test if our novel parallelization reduces the execution time and it produces the same set of matches than the serial execution. A data level parallelism is used in the reference and query genome.

Conclusions: We have set in place a parallelization of whole genome alignment 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.

Background

Today, we know that species are described by DNA, 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 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**.

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.

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.

This paper focuses in the whole genome alignment problem. A whole genome alignment is the process of identifying a mapping from each position in query genome to its corresponding position in the reference genome.

We now describe some notation, give a more detailed introduction to whole genome 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. The goal of sequence alignment is to infer which changes occurred with a mathematical model that abstracts the physical mutation processes.

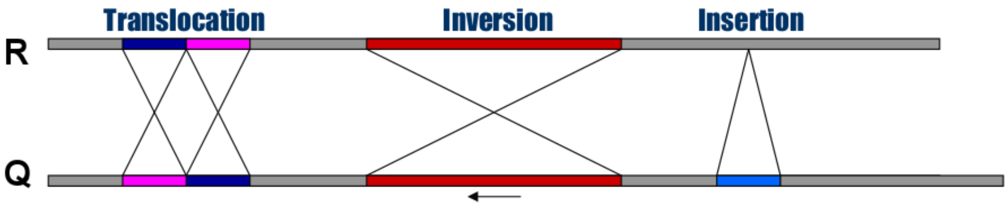


Figure 1: Operations in whole genome alignment

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

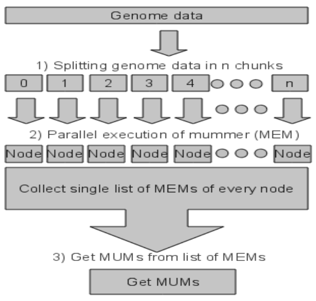
Finding all these changes is a time-intensive operation in whole genome alignment. Moreover, the size of the genome can be an issue when there is not enough memory to store the reference and query genome. The standard algorithms for sequence alignment rely on either dynamic programming or hashing techniques. Naïve versions of dynamic programming use $O(n^2)$ space and time (where n is the length of the shorter of the two sequences being compared), which makes computation simply unfeasible for sequences of size ≥ 4 Mb. Hashing techniques operate faster on average, but they involve a 'match and extend' strategy, where the 'extend' part also takes $O(n^2)$ time. For dynamic programming, it is possible to reduce the required space to $O(n)$ by taking more time; this solves the memory problem but still leaves one with an unacceptably slow algorithm. Faster algorithms can be developed for specialized purposes. More complex dynamic programming methods can be used for alignment when the alignment error is expected to be low. For example, one can align two similar sequences with at most E differences (or errors) in time proportional to E times the length of the longer sequence.

0.1 MUMmer

MUMmer, a widely-used bioinformatic application [1–3], is the tool that we use to solve the whole genome alignment problem. MUMmer assumes the sequences are closely related, and using this assumption can quickly compare sequences that are millions of nucleotides in length. The basis of the algorithm is a data structure known as a suffix tree, which allows one to find, extremely efficiently, all distinct subsequences in a given sequence. However the requirements of space needed to store the suffix tree in main memory make a non-viable structure when the suffix tree to be stored exceeds the available memory. The space complexity of the suffix tree used in MUMmer is $O(17n)$, where n is the length of the reference genome.

Results and Discussion

The following Figure 2 shows the process of our data-level parallelism technique. To verify that our



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 2: Data-level parallelism technique for whole genome alignment.

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

Each genome alignment was executed following these phases:

- Split genome data:
  - Genome size: 115,86 Mbp[1].
  - 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 3.

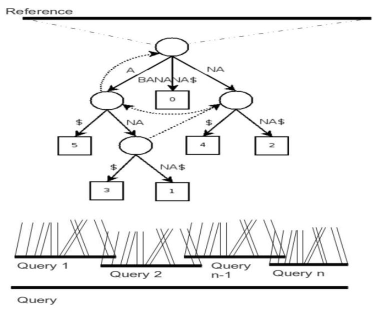


Figure 3: Split query genome.

- Align the whole query genome against every chunk of reference genome, see Figure 4.

[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 [4]

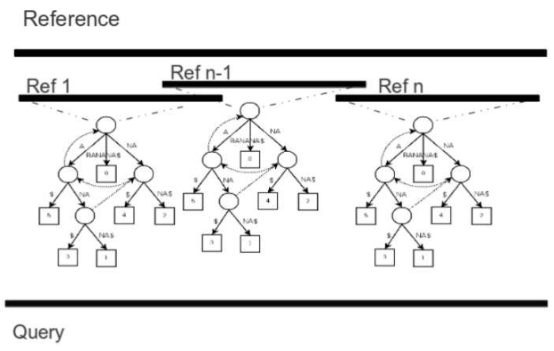
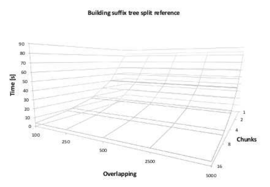


Figure 4: Split reference genome.

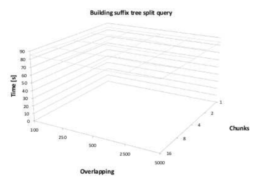
Our was with the human chromosome 13 as reference genome and the chimpanzee chromosome 13 as query genome. The correct size of each genome is:

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

The first parameter to measure was the construction of suffix tree. In Figure 5a is shown how, 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 5b is shown the building time for suffix tree which remains constant. The second parameter to test was the computation time to find the



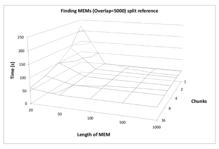
(a) Construction of ST split Human Chr13 reference.



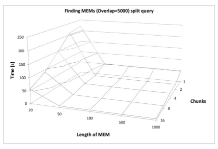
(b) Construction of ST split Chimpanzee Chr13 query.

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. In other words, it might be possible that for some genome sizes our approach can be worse than the sequential version.
In the Figure 5a 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 6b and Figure 6a which shows the high number of hits when a short match is searched.



(a) Finding MEMs split Human Chr13, Overlap=5000bp.



(b) Finding MEMs split Chimpanzee Chr13, Overlap=5000bp.

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 5. 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.

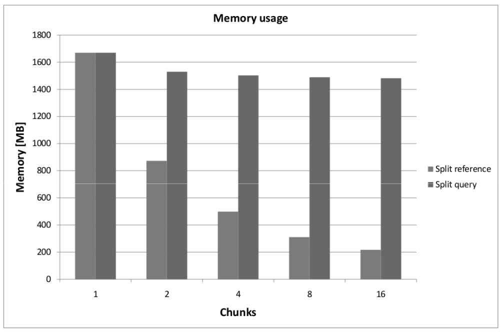
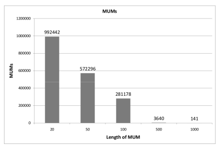
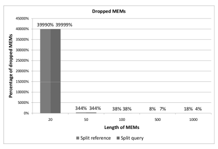


Figure 5: Use of memory when aligning Human Chr13 vs. Chimpanzee Chr13.

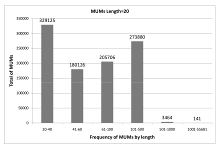
The most important step in our test was to verify that our parallelization can get the same alignment (same list of MUMs from the sequential version), in Figures 6a, 6a, 6b, 6a, 6b and 6a reader can watch the list of MUMs which should be the same when we apply our parallelization technique. In order to know if our approach does the job we filter those matches which are MUMs. To understand this point, the Figure 6b shows the number of matches discarded.



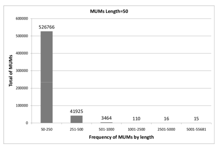
(a) List of MUMS for several sizes of MUM when aligning Human Chr13 vs. Chimpanzee Chr13.



(b) 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.



(a) Frequency of MUMs, length of 20bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.



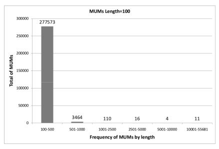
(b) Frequency of MUMs, length of 50bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

Drawbacks

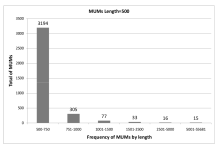
Our novel approach has some limitations which are listed below:

- 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.).

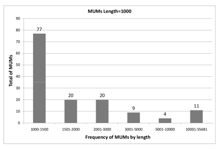
These drawbacks are some opportunities to work with in the near future.



(a) Frequency of MUMs, length of 100bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.



(b) Frequency of MUMs, length of 500bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.



(a) Frequency of MUMs, length of 1000bp, found when aligning Human Chr13 vs. Chimpanzee Chr13.

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.

Conclusions

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.

This work has made some progress, especially in improved pairwise alignment with a heuristic basis, 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 1 *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.

Our technique is a startup to explore the multiple faces of bioinformatics to do a better use of the available computer resources. This work 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.

Methods

The sequential version of the MUMmer's algorithm 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 WGA 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[2] found in the previous phase.

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

[2]Maximal Exact Match

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.
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.
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.

Implementation

This section explains in detail how our proposal is implemented and the modifications in order to be executed in MUMmer. The following diagram, see Figure 6, 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

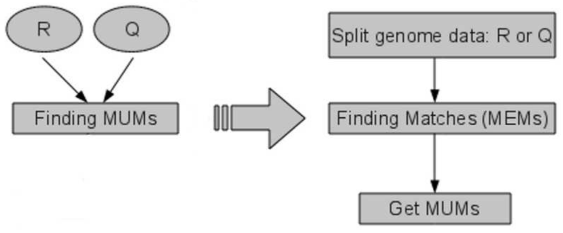


Figure 6: Xipe Totec: proposal for parallelization of whole genome alignment

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

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.
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.

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, 0.2.

0.2 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 7. To get the MUMs we filter those MEMs that

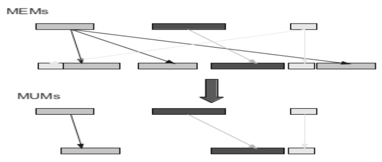


Figure 7: Xipe Totec: Finding the real MUMs

are unique in the list and order them using a modified version of LIS[3] 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
```

[3] Longest Increasing Subsequence

```
        MEM[$i$] is a MUM
      end if
    end for
```

The output of this algorithm gives the MUMs.

Authors contributions

JCGV carried out the design and evaluation of the technique described and drafted the manuscript.

AE helped to draft the manuscript.

Acknowledgements

This work was supported by grant from Ejecución eficiente de aplicaciones multidisciplinares: nuevos desafíos en la era multi/many core, with reference TIN2011-28689-C02-01.

References

1. Delcher AL, Kasif S, Fleischmann RD, Peterson J, White O, Salzberg SL: **Alignment of whole genomes MUMMER**. Nucl Acids Res 1999, **27**(11):2369.
2. Delcher AL, Phillippy A, Carlton J, Salzberg SL: **MUMmer: comparative applications Fast algorithms for large-scale genome alignment and comparison**. Nucleic Acids Research 2002, **30**:2478–2483, [http://www.tigr.org/software/mummer/applications.html].
3. Delcher AL, Salzberg SL, Phillippy AM: **Using MUMmer to identify similar regions in large sequence sets.** Current protocols in bioinformatics editoral board Andreas D Baxevanis et al 2003, **Chapter 10**(1934-340X (Electronic) LA - eng PT - Journal Article SB - IM):Unit 10.3, [http://www.ncbi.nlm.nih.gov/pubmed/18428693].
4. **U.S. National Library of Medicine** [http://www.ncbi.nlm.nih.gov/books/NBK62051/].

Figures

Figure 1 - Sample figure title

A short description of the figure content should go here.

Figure 2 - Sample figure title

Figure legend text.

Tables

Table 1 - Sample table title

Here is an example of a small table in LaTeX using `\tabular{...}`. This is where the description of the table should go.

My Table		
A1	B2	C3
A2	...	..
A3	..	.

Table 2 - Sample table title

Large tables are attached as separate files but should still be described here.

Additional Files

Additional file 1 — Sample additional file title

Additional file descriptions text (including details of how to view the file, if it is in a non-standard format or the file extension). This might refer to a multi-page table or a figure.

Additional file 2 — Sample additional file title

Additional file descriptions text.