



## Genomic-Scale Analysis of DNA Words of Arbitrary Length by Parallel Computation

X.Y. Yang, A. Ripoll, V. Arnau, I. Marín, E. Luque

published in

*Parallel Computing:*

*Current & Future Issues of High-End Computing,*

Proceedings of the International Conference ParCo 2005,

G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),

John von Neumann Institute for Computing, Jülich,

NIC Series, Vol. 33, ISBN 3-00-017352-8, pp. 623-630, 2006.

© 2006 by John von Neumann Institute for Computing

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

<http://www.fz-juelich.de/nic-series/volume33>

## Genomic-scale analysis of DNA Words of Arbitrary Length by Parallel Computation\*

X.Y. Yang<sup>a</sup>, A. Ripoll<sup>a</sup>, V. Arnau<sup>b</sup>, I. Marín<sup>b</sup>, E. Luque<sup>a</sup>

<sup>a</sup>ETSE, Universitat Autònoma de Barcelona Computer Science Department 08193-Bellaterra, Barcelona, Spain

<sup>b</sup>Universidad de Valencia Campus de Burjassot Avd. Vicent Andrés Estellés,s/n. 46100 Burjassot, Valencia

In the post-genomic era, one of the main tasks is deciphering the meaning of the DNA sequences of complex organisms. In order to do so, there is a clear need for biocomputer tools able to extract and order the information of long DNA molecules, such as whole chromosomes or even complete genomes. However, most genomic analyses have been concentrated on the detection and counting of short words having sizes of between 1 and 10 nucleotides. In this paper, we describe parallel algorithms with different complexities that exhaustively determine all words of size  $k$ ,  $k$  being arbitrarily large, in a source DNA sequence. The results shown that our algorithms achieve a high degree of scalability, allowing the detection of DNA words of 64 nucleotides in only 800 seconds.

### 1. Introduction

In genomic analysis, the determination of the DNA words (sequence of nucleotides) found in chromosomes or even of whole genomes is essential in many contexts. Some examples are: 1) Determination of genomic signatures that characterize organisms or species; 2) Characterization of differences among chromosomes for certain specific oligonucleotides, such as the differences in CpG dinucleotides that are specific targets for DNA methylation in many organisms; 3) Characterization of repetitive DNA sequences; or, 4) Finding singular sequences that are much more frequent in certain chromosomes or genomes than in others. However, many of these analyses become unfeasible for long word lengths on standard PC equipment due to memory limitations.

Significant advances in our understanding of genome structure and complexity have been provided by the analysis of oligonucleotide words (reviewed in [7]). However, most of the analyses performed to date have been limited to the detection and counting of short words (of sizes  $k$  between 1 and 10 nucleotides;  $1 \leq k \leq 10$ ) [1][8]. When those analyses are to be extended to longer words, the complex problem emerges of designing algorithms that are able to handle the millions of possible combinations ( $4^k$ ; i.e. for  $k = 15$ , the number of possible words is about  $10^9$ ). One solution is to use complex data pre-processing ([6]) that introduces a high computational requirement.

In this study, we propose two parallel algorithms [3][4][5] for the DNA words frequency analysis: *two-stage* and *k-stage* algorithm. The *two-stage* algorithm is able to achieve high parallel computer-system efficiency to perform DNA exhaustive analysis, extracting the frequency information of every word of a particular length in a DNA sequence. The *k-stage* algorithm is designed for highly frequent DNA word analysis, such as Alu sequences. The *k-stage* is able to find frequency information about extremely long DNA words. In our study, we evaluated our algorithms with human chromosome analysis using a cluster of 16 PCs. Our results show that *two-stage* algorithm achieves up to 76% of

---

\*This work was supported by the MCyT-Spain under contract TIC 2004-03388, TIC 2003-08154-C06-04 and partially supported by the Generalitat de Catalunya- Grup de Recerca Consolidat 2001SGR-00218.

the cluster optimal performance and the  $k$ -stage algorithm is able to analyze DNA words longer than 64 nucleotides.

The remainder of this paper is organized as follows: we show the sequential algorithm definition in section 2. The *two*-stage and  $k$ -stage parallel algorithms are analyzed in sections 3 and 4, respectively. In section 5, we show performance evaluation and indicate our main conclusions are set out in section 6.

## 2. The Sequential Algorithm

In a previous study [2], we described an algorithm that can exhaustively determine all 12-nucleotide-long ( $k = 12$ ) words present in a given DNA sequence, together with their frequencies.

The rationale of the algorithm is as follows: a tree is started that has a root node (level 0) from which four different pointers can be established, corresponding to nucleotides A, C, G or T, that lead to the four possible nodes in the level 1. In the level 2 of the tree, we have 16 nodes, corresponding to 16 different words with 2-nucleotide (AA, AC,...,TT). The solution tree structure is dynamically generated to contain all the possible words. To build the tree at a faster rate, a pointer was used for each level, so that after reading one nucleotide, each pointer indicates a new node determined by the pointer on the previous level and the newly read nucleotide. The final nodes have a different data structure. They have a counter that indicates the frequency of appearances of each word of  $k$  nucleotides.

In that study, we also showed that the algorithm is fast enough to be used on a genomic scale. However, the algorithm is not able to extend the frequency analysis to DNA sequences with more than 12 nucleotides. Given a DNA sequence that is long enough to contain all the possible combinations of words of length  $k$ , the maximum number of nodes ( $N_{max}$ ) of the solutions tree of the sequential algorithm is:

$$N_{max} = \sum_{i=1}^k 4^i = \frac{4^{k+1} - 1}{3} \quad (1)$$

In the implementation, the sequential algorithm requires a value of 32 bits (4 bytes) to reference a node of the tree. Since each node makes 4 references to nodes of the next level of the tree, each node requires, at least, 16 bytes. In a 32-bit machine, the maximum amount of available memory is  $2^{32}$  bytes. In other words, it is able to hold a tree of  $\frac{2^{32}}{16} = 2^{28} = 4^{14}$  nodes. This implies that the sequential algorithm would not be able to solve the problem of searches for words of longer than 13 nucleotides using a 32-bit machine.

Moreover, if the machine has less than  $2^{32}$  bytes (4 Gigabytes) of memory, the solvable space for the problem, using the sequence algorithm, is even smaller. For example, if we have 512 Mbytes of memory, the sequential algorithm is only able to deal with a search problem for words of up to 12 in length. One Gigabyte of memory is not enough to extend the search to a length of 13. The result of this analysis leads us to the unquestionable need to research an algorithm using parallel systems where there is more than one computational element.

## 3. Two-stage Parallel Algorithm

In this section, we show the key ideas of the parallel algorithm. The aim of this algorithm is to be able to extend the initial problem to analyse words of any length.

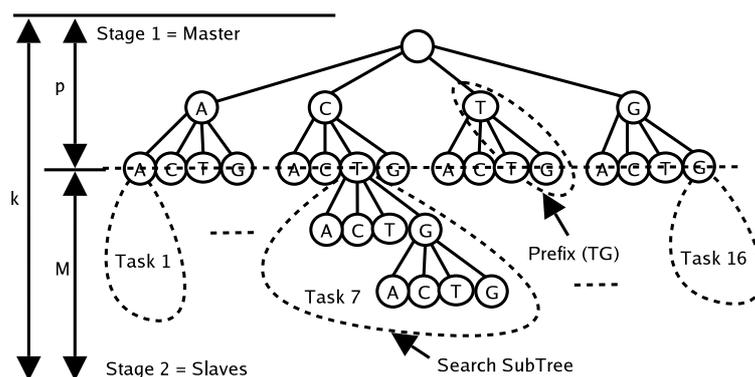


Figure 1. Parallel Task Assignment

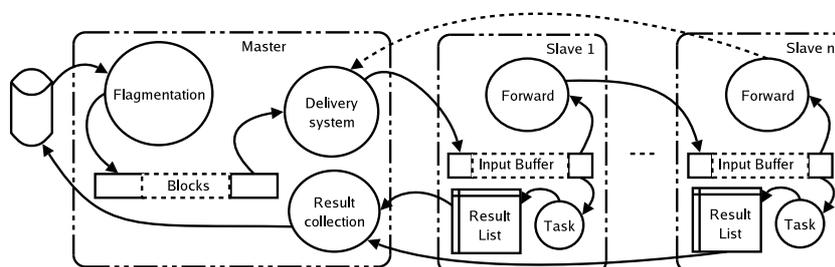


Figure 2. DNA information Delivery Mechanism

### 3.1. Definition of Parallel Tasks

The number of levels of solution tree depends on the length of the words that we want to analyze and is limited to a maximum ( $M$ ) that is determined by the size of the memory of the machine being used. The key idea of the parallel algorithm consists of dividing the process of generating the search tree into two stages in which a partial tree of only the first  $p$  levels is generated for stage one (Figure 1). Stage 2 performs the search process for all of the sub-trees that are derived from the leaf nodes of the partial tree in stage 1. The levels of the sub-trees in stage 2 will be limited by  $M$  and therefore the process can be performed on a single machine. Each of these sub-trees represents one task that should be performed in stage 2 and the number of these is equal to  $4^p$ . The value of  $p$  depends on  $M$  and  $k$  that is the length of the words we want to search for; so that  $p = k - M$ . Figure 1 shows an example of a 4-levels tree for searching for words of length 4 ( $k = 4$ ). In this case, the value of  $M$  is 2 and, therefore,  $p$  is 2.

Since the sub-trees are derived from different leaf nodes of the partial tree in stage 1, the tasks related to these in stage 2 are independent from each other and, therefore, can be run on parallel machines following the Master-Slave model. In stage 1, the master generates the partial tree, which defines different tasks of the stage 2. Each path between the root node and a leaf node of the partial tree defines the *Prefix* of the words that should be analyzed by a task. The slaves generate the sub-trees in stage 2 and represent the different tasks of analysis. Depending on the number of machines available for the parallel system, the master creates a number of slaves and sends them the different prefixes to determine the analysis task they must perform. In the case of Figure 1, the master generates a partial tree of 2 levels and defines the 16 tasks. The 16 tasks are defined by 16 prefixes that are sent from the master to the slaves. For example, the task 7 searches for the frequency of all the words that start with a {CT} prefix.

### 3.2. DNA sequence Delivery and Results Collection Mechanism

As well as the distribution of the task prefixes to slaves, the master sends the DNA sequence to be used for an analysis. Since every slave requires the complete DNA sequence in order to generate the sub-tree, the master has to send the DNA sequence to all of the slaves. An initial solution could be for the master to send the sequence to the slaves using independent communication channels. The major problem with this solution is that it quickly saturates the system's communication network and the parallel algorithm cannot be scaled to every number of machines.

Our design for the DNA sequence delivery is based on the logical chaining of slaves (Figure 2). The master assigns two neighbours to each slave; one neighbour that collects the DNA sequence ( $V_s$ ) and one neighbour that sends the sequence ( $V_d$ ). All of the DNA sequence that has been collected from  $V_s$  is redirected towards  $V_d$  (except the last slave) using the Forward mechanism. In this way, DNA sequence is sent to every slave.

---

#### Algorithm 1 Slave Pseudocode

---

```

1: Receive Control Information from Master  $Prefix \leftarrow \{P_1, P_2, \dots, P_p\}$ ,  $K$  and  $m$ 
2: Create an array of  $(K - m)$  pointers  $P[0..K - m] \leftarrow \{NULL, \dots, NULL\}$ 
3: Create an array of  $(K - m)$  numbers  $Pos[0..K - m] \leftarrow \{1, \dots, 1\}$ 
4: while There is more DNA information do
5:   Receive one Nucleotide ( $NN = 1, 2, 3, 4$ ) from Master
6:   for  $i = 1$  to  $(K - m)$  do
7:     if  $Pos[i] == m$  then
8:        $P[i] \leftarrow$  Next level of Tree according to  $N$ 
9:       if  $P[i] == NULL$  then
10:        Create next level of the Tree
11:       end if
12:       if  $P[i]$  is the last level then
13:        Increase the Frequency
14:         $P[i] \leftarrow NULL$ 
15:         $Pos[i] \leftarrow 1$ 
16:       end if
17:     else
18:       if  $NN == Prefix[Pos[i]]$  then
19:         $Pos[i] \leftarrow Pos[i] + 1$ 
20:       end if
21:     end if
22:   end for
23: end while

```

---

The chaining mechanism works as a Pipeline process, where the stages of the pipeline represent the slaves that perform the searching task and re-send the DNA information. The performance of the pipeline closely depends on the DNA sequence delivery mechanism, which has to guarantee information availability as well as no overflow of the input buffers. The DNA sequence delivery is complemented by the control mechanism in which a message is sent by the last slave (slave  $n$ ) to the master to report the successive arrivals of packets of information. Using the information on the frequency of arrival of these messages, the master adapts its delivery speed to the processing capacity of the different stages.

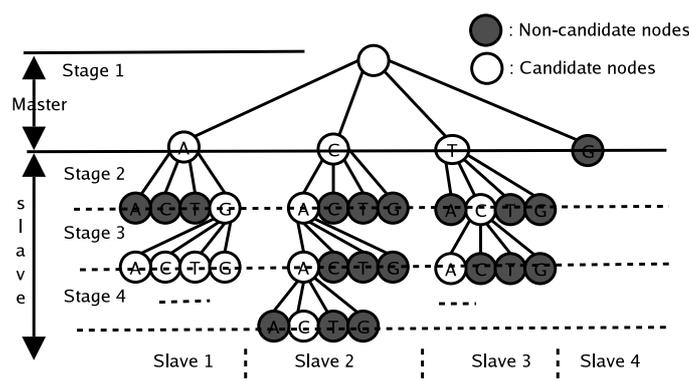


Figure 3.  $k$ -stage Algorithm Solution Tree

In order to deliver DNA sequence on a packet commutation based network, the master divides the DNA into blocks. This is performed in the Fragmentation module. As well as the division, the module also performs the DNA sequence compacting process by assigning 2 bits to each base. As a result of the analysis, the slave generates a list that contains the words found with a frequency higher than the threshold. The slaves' lists are collected by the master to generate the final output of the system.

Algorithm 1 show the slave pseudocodes. In the step 1, the slave receives the control information from the master whileas the DNA sequence is received in step 5.  $P[i]$ 's are memory pointers to scan the tree and  $Pos[i]$ 's indicate the position of the DNA word that each pointer is. The master pseudocode is not shown for space limitation.

#### 4. $K$ -Stage Parallel Algorithm with Dynamic Memory Deallocation

In the solution process of *two*-stage algorithm, a distributed tree is created that includes every word in the input DNA sequence, independently of frequency of appearance. But should we only wish to search for very frequently repeated words, the system does not use memory efficiently, as we will be storing information about uninterested words. For example, in the chromosome 1 of Homo sapiens, there are more than 40000 words with a frequency higher than 200, if the words-length is 12 ( $k = 12$ ), However, less than 10000 of these appear at a frequency of more than 600. These results show that the number of words of interest depends on the threshold frequency and the word length. The higher the threshold frequency, the lower the number of words found. The longer the words are, the fewer there will be that will achieve the threshold frequency.

The aim of the  $k$ -stage parallel algorithm is to achieve better performance in the use of memory and number of machines. The key idea of this algorithm is to perform a progressive and directed analysis. In analysis process, only those parts of the tree that contain significant words (i.e. above the cut-off value) are created.

##### 4.1. Definition of the $k$ -stage Algorithm

The initial problem of searching for nucleotide words can be formulated mathematically as the search for all words of  $\{b_1, b_2, \dots, b_k\}$  such that  $f(\{b_1, b_2, \dots, b_k\}) > U$  where  $f()$  is the function that determines the frequency at which any word appears. Given this formulation, the searching problem follows the axiom: *given a word  $\{b_1, \dots, b_{k-1}, b_k\}$  with  $f(\{b_1, \dots, b_{k-1}, b_k\}) > U$  if and only if  $f(\{b_1, \dots, b_{k-1}\}) > U$ .* This axiom tells us that if a word of length  $k$  is more frequent than  $U$ , then the word formed by the first  $k - 1$  nucleotides should also have a frequency greater than  $U$ .

The  $k$ -stage parallel algorithm is based on the aforementioned axiom in which each stage  $i$  searches

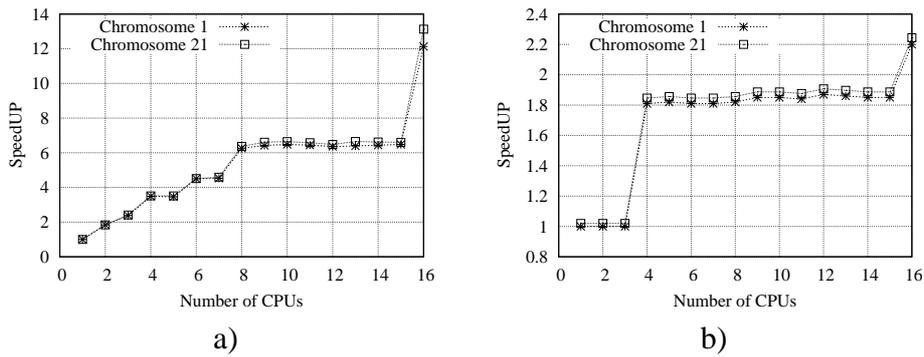


Figure 4. SpeedUP of: a) *two*-stage Algorithm. b) *k*-stage Algorithm.

for the frequency of words of length  $i$  and eliminates all those that are below threshold  $U$ . The tree of stage  $i$  is used to create the tree of stage  $i + 1$ . In this way, only the nodes that contain possible solutions are kept in the memory, achieving higher memory efficiency. Figure 3 represents the tree of solutions created in accordance with a  $k$ -stage algorithm. The number of stages is 4 in this case, where stage 1 is run by the master and the other 3 in the four slaves. Two types of nodes have been defined: 1) candidate nodes, which are those that could contain solutions. 2) non-candidate nodes, which are those with a frequency lower than  $U$  and where therefore there are no longer any possible solutions in these branches of the tree. In the case shown, a single word  $\{C,A,A,C\}$  has been found by slave 2 that has a higher frequency than the threshold. In the case of slave 4, no branch has been created.

## 5. Performance Results

In this section, we describe the experimental results obtained by running the different versions of the algorithm using the parallel system. The different parallel algorithms are implemented using language C with the C+PVM library and the parallel system consists of a cluster of 16 Pentium 4 PCs with 512 MB of memory interconnected using a 100 Mbps Ethernet. There are several points that we are especially interested in measuring: 1) the speed-up that can be obtained using parallel algorithms. 2) the response time of parallel algorithms given a certain number of machines. 3) how the response time of the algorithms varies when we add more machines to the parallel system.

### 5.1. Speed-up of two- and $k$ -stage Algorithms

We calculated the time taken to search for words of length 14 with a cut-off frequency of 1000. We ran the programs that implement the 2- and  $k$ -stage parallel algorithms on 1-16 machines. The  $M$  value of our machines is 12 and therefore in the case of *two*-stages, we found that  $p = 14 - 12 = 2$  and therefore, the program must run  $4^2 = 16$  independent tasks.

Figure 4.a shows the speed-up results in the cases of human chromosomes 1 and 21 using the *two*-stage algorithm. The speedup value increases as more machines are added to the running of the program. The algorithm obtains a speedup of 12.13 with 16 CPUs, which is 76% of the theoretical value.

Between 8 and 15 CPUs, no increase of the speed-up value is observed owing to the existence of non-usage of the CPU in the task distribution process. For example, in the case of 12 CPUs, the algorithm assigns 12 tasks to 12 CPUs in the first distribution, leaving 4 tasks to be run. In the next distribution, the system runs the 4 remaining tasks using only 4 CPUs and there are therefore 8 CPUs that do not do any task.

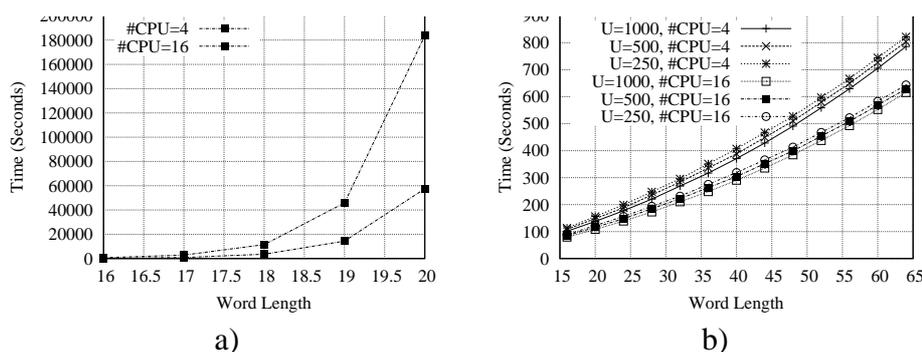


Figure 5. a) Response time of *two-stage*. b) Response time of *k-stage*.

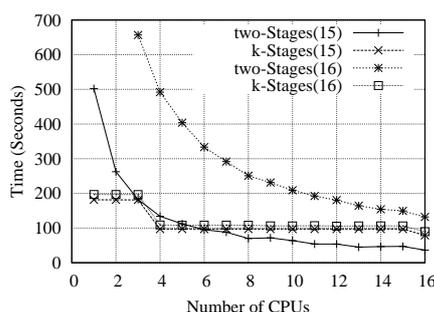


Figure 6. Response time Comparison according with Number of CPUs

Figure 4.b shows the speed-up values using the *k-stage* algorithm. In this case, the speed-up value is lower than the case for *two-stages*. This is because multiplying the number of machines by 4 involves the reduction of one stage of calculation. With 4 machines, we only achieve a speed-up of 1.85 and with 16 CPUs, the speed-up value is only 2.3.

## 5.2. Response Time According With the Number of Machines

We measured the computation time of 2- and *k-stage* algorithms using 4 machines to perform a search for words of lengths between  $16 \leq k \leq 64$  and with a frequency of appearance greater than 250, 500 and 1000.

Figure 5.a shows the response time in seconds using the *two-stage* algorithm. In this case, we have only determined the time for words of a length of up to 20. The time required increases exponentially in accordance with the length. No outstanding differentiations are observed when varying cut-off frequency ( $U$ ).

Figure 5.b shows the times taken in seconds by the *k-stage* algorithm. As we can see, the response time of *k-stage* algorithm increases linearly with value of  $k$ . There is also a slight increase depending on the value of the cut-off frequency ( $U$ ). This is because with a lower cut-off value, there are more words in the final result that should be collected by the master.

## 5.3. Algorithms Response Time Comparison

In order to make a comparison of the performance of the two versions of algorithm, we made a search for words of lengths 15 and 16 with a frequency of appearance greater than 500. In this test, we determined the total time of the two algorithms using from 1 to 16 CPUs.

Figure 6 shows what results were found. Owing to the complexity of the *two-stage* algorithm, the time increases exponentially as the number of available machines decreases. Meanwhile, the *k-stage* algorithm lineally increases the total time as fewer CPUs are used.

There is a point where the *two-stage* algorithm achieves better results than the *k-stage* algorithm.

This is the case for words of length 15. The *two*-stage algorithm achieves better results when there are more than five CPUs in the system. However, the result is not the same in the case of the search for words of 16, where the *k*-stage algorithm is always better than the *two*-stage one when we do not have more than 16 machines in the system.

From observing the data, the point can be calculated where the *two*-stage algorithm is better than the *k*-stage one. The following expression estimates the number of CPUs where the *two*-stage algorithm better the *k*-stage one:

$$\frac{4^{k-M}}{N} \leq k - \log_4 N \quad (2)$$

For example, if we want to find chains of 15 using machines that have  $M = 12$ , the *two*-stage algorithm is better than the *k*-stage one when we have about 5 CPUs. In the case for 16, we can estimate that from around 19 machines, the *two*-stage algorithm is better than the *k*-stage one with regards to response time.

## 6. Conclusions

In this study, we present a parallel application that enables us to determine the frequency of appearance of words of  $k$  nucleotides in long DNA sequences. The proposed parallel algorithm has demonstrated high scalability in accordance with the number of processors and can be used to analyse any length of words in the search problem.

The proposed parallel algorithm presents two implementations. In the first algorithm, the distributed tree of solutions is totally built to obtain an exhaustive analysis. In the second implementation, however, the algorithm eliminate no interested words and only the most frequently repeated words are presented in the solution tree. The second design achieves a high memory efficiency for analysis of extremely large words.

In order to validate the parallel algorithms, a set of tests was performed on various human chromosomes and the most repeated sequences found were highly repetitive sequences typical of our genome, known as Alu sequences plus simple DNA sequences (e.g. poli(A), poli(GA), etc). Those results were as expected when high cut-off values are used. Similar analyses can be performed on less known genomes in order to establish the most frequent highly repetitive sequences found in these.

## References

- [1] A.J.Gentles and S.Karlin. Genome-scale compositional comparisons in eukaryotes. *Genome Res.* 11, pages 540–546, 2001.
- [2] V. Arnau and I. Marín. Fast algorithm for the exhaustive analysis of 12-nucleotide-long dna sequences. applications to human genomics evolution. In *Proceedings of the 17th IPDPS-HiCOMB 2003, Nice (France)*, 2003.
- [3] R. Buyya. *High Performance Cluster Computing (Vols. 1 y 2)*. 1999.
- [4] J. C. Cunha, P. Kacsuk, and S. C. W. Nova. *Parallel Program Development For Cluster Computing: Methodology, Tools and Integrated Environments*. 2001.
- [5] A. G. et alter. Addison Wesley. *Introduction to Parallel Computing, Second Edition*. 2003.
- [6] J. Healy, E. Thomas, J. Schwartz, and M. Wigler. Annotating large genomes with exact word matches. *Genome Res.* 13, pages 2306–2315, 2003.
- [7] S. Karlin, A. M. Campbell, and J. Mrázek. Comparative dna analysis across diverse genomes. *Annu. Rev. Genet.* 32, pages 185–225, 1998.
- [8] R. Roset, S. J.A, and X. Messeguer. MREPEAT: detection and analysis of exact consecutive repeats in genomic sequences. *Bioinformatics*, 19 no. 18:2475–2476, 2003.