DNA sequence alignment: hybrid parallel programming on a multicore cluster

Rucci Enzo, De Giusti Armando E., Chichizola Franco, Naiouf R. Marcelo, De Giusti Laura C.

Instituto de Investigación en Informática LIDI (III-LIDI) – Facultad de Informática – Universidad Nacional de La Plata. Argentina
{erucci, degiusti, francoch, mnaiouf, ldgiusti}@lidi.info.unlp.edu.ar

Abstract. DNA sequence alignment is one of the most important operations of computational biology. In 1981, Smith and Waterman developed a method for sequences local alignment. Due to its computational power and memory requirements, various heuristics have been developed to reduce execution time at the expense of a loss of accuracy in the result. This is why heuristics do not ensure that the best alignment is found. For this reason, it is interesting to study how to apply the computer power of different parallel platforms to speed up the sequence alignment process without losing result accuracy.
In this article, a new parallelization strategy (*HI-M*) of Smith-Waterman algorithm on a multi-core cluster is presented, configuring a pipeline with a hybrid communication model. Additionally, a performance analysis is carried out and compared with two previously presented parallel solutions.
Finally, experimental results are presented, as well as future research lines.

Keywords: bioinformatics, sequence alignment, parallel algorithms, multicore cluster, pipeline.

1 Introduction[1]

The study of distributed and parallel systems is one of the most active research lines in Computer Science nowadays [1][2]. In particular, the use of multiprocessor architectures configured in clusters, multiclusters, grids and clouds, supported by networks with different characteristics and topologies, has become general, not only for the development of parallel algorithms but also for the execution of processes that require intensive computation and the provision of concurrent web services [3][4][5][6].

The term cluster is applied to "sets of computers built with standard hardware components that act as if they were an only computer" [7] [8]. Cluster technology has evolved to support activities that go from supercomputing applications and mission-critical software, to web services and high-performance databases.

Clusters' computing is the result of the convergence of several current trends, including the availability of cheap high-performance processors and high-speed networks, the development of software tools for high-performance distributed computation, and the growing need for computer power for the applications that require it [9].

The technological change caused by energy consumption and heat generation problems that appear when scaling processor speed has caused the appearance of multicores. This type of processors is formed by the integration of two or more computer cores within the same chip, and increases application performance by dividing computing work among all available cores [10][11].

The incorporation of this type of processors to conventional clusters gives birth to an architecture that combines shared and distributed memory, known as multicore cluster [12][13].

1.1 DNA sequence comparison on a multicore cluster

One of the areas of greatest interest and growth in the last few years within the field of parallel processing applications is that of the treatment of large volumes of data such as DNA sequences. The extensive comparison processing required to analyze genetic patterns demands a significant effort in the development of efficient parallel algorithms [14].

The center for all bioinformatics operations and analyses is partly held by Sequence Alignment, both for pattern searching among amino acid and nucleotide sequences, and for the search of phylogenetic relationships among organisms. The Smith-Waterman algorithm for local alignment is one of these methods; it focuses on similar regions only in part of the sequences, which means that the purpose of the algorithm is finding small, locally similar regions. This method has been used as the basis for many subsequent algorithms and is often used as basic pattern to compare different alignment techniques. If the length of the sequences involved are N and M, the complexity of the algorithm is $O(NxM)$. Thus, the problem is scaled as the square of sequence size [15].

[1] The final authenticated version is available online at http://dl.acm.org/citation.cfm?id=2047950.2047978

Taking into account that sequences can have up to 10^9 nucleotides each, the time and memory required to solve this problem with a sequential algorithm is impracticable. This leads to the parallelization of the algorithm over powerful parallel architectures.

In this paper, a new parallel solution for the Smith-Waterman algorithm on a multi-core cluster (*HI-M*) is proposed. The solution is compared with two previously developed versions (*PM and HI*) [16] that use different resolution schemes and communication models.

In Section 2, the Smith-Waterman algorithm is explained, together with the sequential and the parallel solutions used in this paper. In Section 3, the experimental work carried out is described, whereas in Section 4, the results obtained are presented and analyzed. Section 5 presents the conclusions and future lines of work in relation to this paper.

2 Smith-Waterman Algorithm definition

This method allows aligning two DNA sequences by inserting gaps (if necessary) that are used to detect locally similar regions that may indicate the presence of a relation between both sequences, which is done by assigning a similarity score. If gaps are inserted, that is, certain elements of the sequences are not aligned to achieve a better overall alignment, a penalization is applied.

The algorithm calculates a similarity score between two sequences and then, if necessary, employs a backwards alignment process for an optimal result [14].

The following paragraphs explain the operation of the algorithm to find a similarity score between two DNA sequences.

Given two sequences: $A = a_1a_2a_3...a_M$ and $B = b_1b_2b_3...b_N$, a matrix H of $(N+1)x(M+1)$ is built, in such a way that the nucleotide bases that form sequence A label the rows (starting with 1), and those from sequence B label the columns (starting with 1). The following steps are applied to calculate the values of H that will yield the similarity score between A and B:

a. Start row 0 and column 0 of H with 0, as indicated in Equation 1.

$$H_{i0} = H_{0j} = 0 \qquad \text{for } 0 \le i \le N \text{ and } 0 \le j \le M \tag{1}$$

b. Calculate the value of H_{ij}, $\forall i \in [1,..,N]$ and $\forall j \in [1,..,M]$ by means of Equation 2. This value indicates the maximum similarity between two segments ending in a_i and b_j, respectively.

$$H_{ij} = \max \begin{cases} 0 \\ H_{i-1,j-1} + V(a_i, b_j) \\ C_{ij} \\ F_{ij} \end{cases} \tag{2}$$

- $V(a_i, b_j)$ is the matching function that indicates the score obtained for matching a_i with b_j. It is based on a table of values called *substitution matrix* that describes the probability of a nucleotide base from sequence A at position i to occur in sequence B at position j. The most common matrix is the one that rewards with positive value when a_i and b_j are identical, and punishes with a negative value otherwise.
- C_{ij} is the score in column j considering a gap, and is calculated with Equation 3.

$$C_{ij} = \max_{1 \le k \le i} \{H_{i-k,j} - g(k)\} \tag{3}$$

- F_{ij} is the score in row i considering a gap, and is calculated with Equation 4.

$$F_{ij} = \max_{1 \le l \le j} \{H_{i,j-l} - g(l)\} \tag{4}$$

- $g(x)$ is the penalization function for a gap of length x, and is obtained with Equation 5, q being the penalization applied for opening a gap and r the penalization for prolonging it.

$$g(x) = q + rx \qquad (q \ge 0; r \ge 0) \tag{5}$$

c. The similarity score is obtained as shown in Equation 6.

$$G = \max_{(0 \le i \le N)(0 \le j \le M)} \{H_{ij}\} \tag{6}$$

d. Based on the position in matrix H where the value G was found (representing the end of the highest-scoring alignment between both sequences), a backwards process is performed to obtain the pair of segments with maximum similarity, until a position whose value is 0 is reached, this being the starting point of the segment.

2.1 Sequential solution of Smith-Waterman algorithm

In this section, the sequential solution of Smith-Waterman algorithm is analyzed with the purpose of determining the similarity score between two DNA sequences. This means that the backwards process is not taken into account when obtaining the segment that represents the optimal alignment (step *d* of the algorithm explained in the previous section is not performed).

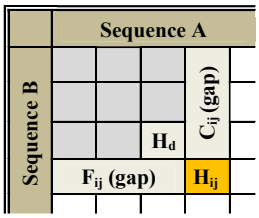

Fig. 1. Data dependency scheme.

Figure 1 shows the data dependency that exists for calculating matrix values. To obtain $H_{i,j}$, the result of $H_{i-1,j-1}$ (H_d in Figure 1) is required, and the score must be known when considering a gap in row *i* and another one in column *j*. This restriction allows calculating *H* values from top to bottom and left to right (H_{11}, H_{12}, H_{13}, …H_{21}, H_{22}, H_{23}, …..).

Taking into account that step *d* of the algorithm is not carried out, matrix *H* does not have to be stored in full, all that is needed is:

- A vector *h* of length *M+1* that at each position keeps the value obtained in the last processed row over that column.
- An element *e* to temporarily store the last value calculated in the row that is being processed. In Figure 1, $e = H_{i,j-1}$.
- A vector *c* of length *M+1* that at each position keeps the maximum score considering a gap in that column.
- An element *f* that keeps the maximum score considering a gap in the row that is being processed. In the example shown in Figure 1, $f = F_{i,j-1}$.

2.2 General parallel solution of Smith-Waterman algorithm

The data dependency mentioned in the previous section causes the problem to be solved following a pipeline scheme where *S* stages perform the same work over various consecutive nucleotide subsets of the first sequence (*A* in Figure 1). In each cycle, stage s_i (for $i \in [1, S-1]$) receives a data block from s_{i-1}, solves part of its work, and then sends these results to s_{i+1} (except for the last stage which does not need to send its results to any other stage). The first stage (s_0) only performs its work by sending partial results (corresponding to a block) to its successor.

An important aspect of this solution is selecting the number of elements (*BS*) from sequence *B* that form the data blocks that are sent from one process to another, taking into account that:

- Pipeline parallelism is exploited to its maximum capacity only after *S-1* cycles have been processed. That is, when all stages have received work to do. The larger the *BS*, the longer the time required to fill the pipe, and therefore, the lower its exploitation. From this point of view, *BS* should tend to 1.
- If the size of *BS* is very small, the stages spend more time communicating partial results than actually processing information. From this point of view, *BS* should tend to *N*.

A suitable block size should be found, so that data communication and data processing can be done simultaneously. The optimal size does not only depend on the architecture used, but also on the communication model implemented.

In previous works, a procedure for calculating the optimal value of *BS* based on architecture characteristics and sequence size has been established [17].

2.2.1 Message Passing as communication model

In this case, each pipeline stage is carried out by a different process p_i (for $i \in [0, S-1]$), and partial results are communicated by sending messages between consecutive processes. The first sequence (*A* in Figure 1) is distributed by p_0 among the *S* processes that form the pipeline.

2.2.2 Shared Memory as Communication Model

In this case, each pipeline stage is carried out by a different thread t_i (for $i \in [0, S-1]$). Instead of communicating partial results through message passing, these are kept in the shared memory as a single structure (as in the sequential algorithm). Consecutive threads are synchronized to indicate that work with a new data block can begin.

2.2.3 Combination of message passing with shared memory as communication model

When each process p_i begins (for $i \in [0, P\text{-}1]$) it generates $T\text{-}1$ threads ($S = PxT$) to jointly solve the data blocks corresponding to the different cycles. Thus, there are PxT threads (all P processes plus all $T\text{-}1$ threads generated by each of them), which means that the set of nucleotides from the first sequence is equally distributed among PxT threads.

The threads that are in the same node synchronize and communicate through shared memory, whereas those that are in different nodes do so through message passing.

Figure 2 shows a scheme of the pipeline formed by algorithm *HI-M* assuming an architecture of P blades with T cores each.

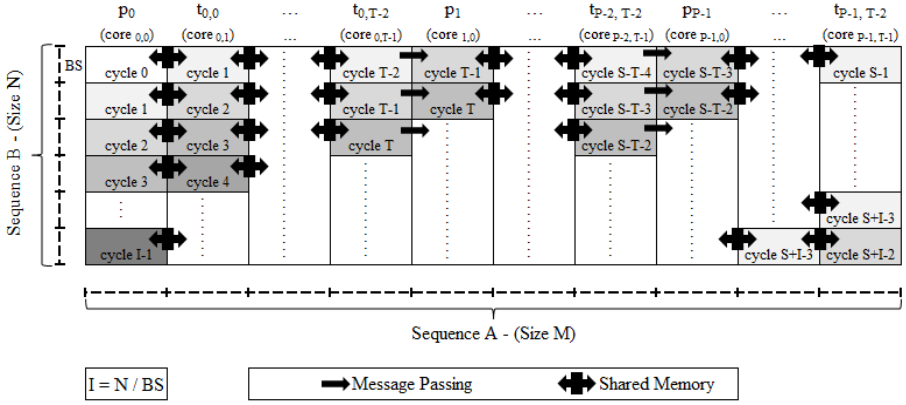

Fig. 2. Schematic representation of the pipeline formed in algorithm *HI-M*.

3 Experimental Work

In this paper, language C is used with OpenMPI and Pthreads libraries for message passing and the handling of threads, respectively.

3.1 Architecture used

To analyze the behavior of the algorithms, tests were carried out on a *multi-core cluster*: Blade with eight blades, each with two quad core Intel Xeon e5405 2.0 GHz processors. Each blade has 2 Gb RAM memory (shared between both processors) and 2 x 6Mb L2 cache for each pair of cores [18][19].

3.2 Algorithms used

The algorithms used in this experiment are described below:

- ***PM*:** this solution is based on using a pipeline of $S = P$ stages as the one described in Section 2.2.1, where P is the number of cores used [16].
- ***HI*:** this hybrid solution is based on the use of a pipeline of $S = P$ stages as the one described in Section 2.2.1, using in each stage a pipeline of T phases as the one detailed in Section 2.2.2, where P is the number of processors used and T is the number of cores in each processor [16].
- ***HI-M*:** this new hybrid solution is based on using a pipeline of $S = PxT$ stages as the one described in Section 2.2.3, where P is the number of blades used and T is the number of cores in each blade.

3.3 Tests carried out

The algorithms were tested using all the cores with different numbers of blades: two, four and eight, which means that 16, 32 and 64 cores were used, respectively. Sequence sizes of various lengths (*65536; 131072; 262144; 524288; 1048576*) were also taken. Table 1 shows the optimal block size (*BS*) used in each test in accordance with the function described in previous works [17].

Table 1. Block size *BS* used in each test run on the multicore cluster.

Algorithm	Cores	Sequence size				
		65536	131072	262144	524288	1048576
PM	16	48	48	48	48	48
	32	47	47	47	47	47
	64	47	47	47	47	47
HI	16	1024	2048	2048	2048	2048
	32	512	1024	2048	2048	2048
	64	512	1024	1024	1024	2048
HI-M	16	48	48	48	48	48
	32	47	47	47	47	47
	64	47	47	47	47	47

4 Results

To assess the behavior of the algorithms developed when scaling the problem and/or the architecture, the *efficiency* of the tests carried out are analyzed [1][3][9].

Figure 3 shows the efficiency achieved by the algorithm *H1-M* when using two, four and eight blades of the architecture for different problem sizes (N). The sizes of the data blocks used were detailed in Table 1.

The chart shows that the efficiency increases when the length of the sequences (problem size) increases. On the other hand, the efficiency decreases when the number of cores used increases, although this reduction is less significant as problem size increases.

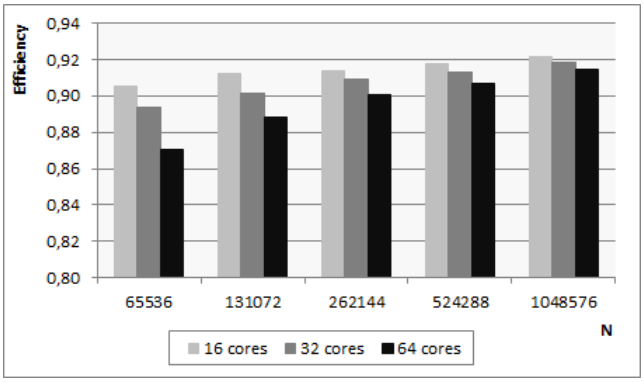

Fig. 3. Efficiency achieved by algorithm *HI-M* when using 16, 32 and 64 cores of the architecture for different problem sizes (*N*).

The efficiency achieved by the new algorithm (*HI-M*) and *H1* is compared using two, four and eight blades of the architecture for different sequence lengths (*N*). Figure 4 shows the percentage of the relative difference between the efficiencies of both algorithms (*rpdHI*), calculated by means of Equation 7.

$$rpdHI = \frac{efficiency\ (a\lg.\ HI-M) - efficiency\ (a\lg.\ HI)}{efficiency\ (a\lg.\ HI)} \times 100 \qquad (7)$$

Better efficiency achieved by algorithm *HI-M* versus *HI* can be clearly seen. This is mainly due to the resolution scheme (pipeline of pipelines) used by algorithm *HI*, where processes and threads are idle most of the time. The sizes of the data blocks used are detailed in Table 1.

This chart shows that the difference between the efficiencies achieved by *HI* and *HI-M* decreases as problem size increases, and inversely, it increases as the total number of nodes increases.

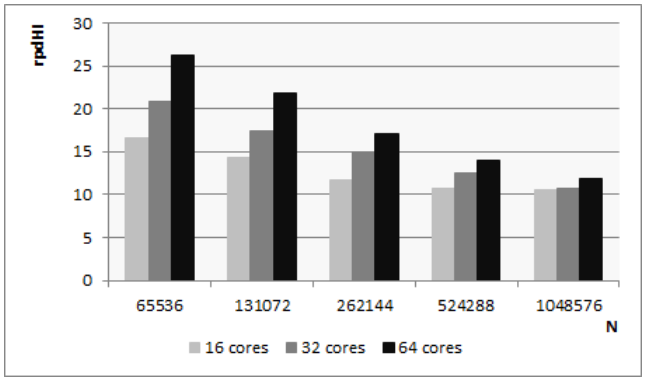

Fig. 4. *rpdHI with 16, 32 and 64 cores for different sequence lengths (N).*

The efficiency achieved by the new algorithm (*HI-M*) and *PM* is also compared using two, four and eight blades of the architecture for different sequence lengths (*N*). Figure 5 shows the percentage of the relative difference between the efficiencies of both algorithms (*dprHIM*), calculated by means of Equation 8.

$$rpdHIM = \frac{efficiency\ (a\lg .\ PM) - efficiency\ (a\lg .\ HI - M)}{efficiency\ (a\lg .\ HI - M)} \times 100 \qquad (8)$$

The chart shows that the performance of both algorithms is similar, although *PM* has a slight advantage over *HI-M*. The sizes of the data blocks used were detailed in Table 1.

The difference between the efficiencies achieved by *PM* and *HI-M* is not significant, being less than 2% in all cases. It can be observed that the difference decreases as the size of the problem increases, and inversely, it increases as the total number of cores increases. The similarity of the results is favored by the use of the same resolution scheme by both solutions. The minimum difference between the efficiencies is due to two factors. First, both algorithms have reduced memory requirements, which does not allow exploiting the benefits of using shared memory. The second factor is the optimization of current message passing libraries to work in shared memory environments. The combination of these two factors improves the performance of *PM* versus *HI-M.*

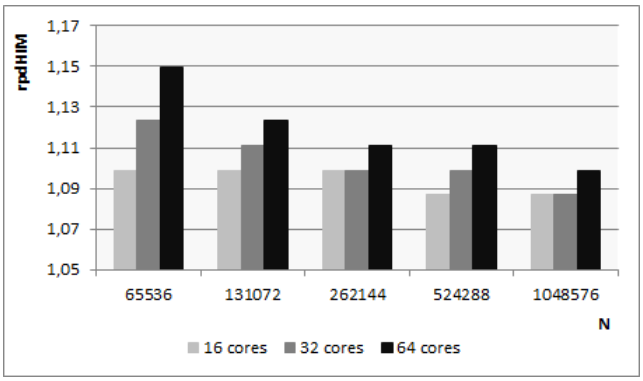

Fig. 5. *rpdHIM with 16, 32 and 64 cores for different sequence lengths (N).*

5 Conclusions and Future Works

In this paper, a new parallelization strategy for the Smith-Waterman algorithm for DNA sequence alignment is proposed. Also, its performance is analyzed and compared with that of two other parallel solutions to the same problem that were previously presented. The architecture used for the experiments is a multi-core cluster (eight blades with eight cores each).

The two first solutions presented, *PM* and *HI*, use different resolution schemes (single pipeline the first one, and pipeline of pipelines the second one) and different communication models (message passing and a hybrid combining message passing with shared memory, respectively). The strategy proposed combines characteristics of both previous solutions, since it uses the pipeline resolution scheme from *PM* and the hybrid communication model from *HI*.

The algorithms were tested using various work and architecture sizes. The result obtained was that *HI-M* achieves a performance that is much better than that of *HI* and very similar to that of *PM*. The significant difference with *HI* is explained by the overhead generated by the idle time of the processes and threads of this solution; whereas the significant similarity between *PM* and *HI-M* is due to the use of the same resolution scheme (single pipeline), and the slight

advantage of *PM* over *HI-M* is due to the low memory requirements of both algorithms together with the optimization of current message passing libraries.

As future lines of work, we plan to study the behavior of these algorithms in heterogeneous multi-core clusters, and analyze them from the point of view of energy efficiency on different architectures [20][21].

References

1. Grama A., Gupta A., Karypis G., Kumar V., "An Introduction to Parallel Computing. Design and Analysis of Algorithms. 2nd Edition". Pearson Addison Wesley. 2003.
2. Ben-Ari, M. "Principles of Concurrent and Distributed Programming, 2/E". Addison-Wesley, 2006.
3. Dongarra J., Foster I., Fox G., Gropp W., Kennedy K., Torczon L., White A., "The Sourcebook of Parallel Computing". Morgan Kauffman Publishers. Elsevier Science. 2003.
4. Juhasz Z. (Editor), Kacsuk P. (Editor), Kranzlmuller D. (Editor), "Distributed and Parallel Systems: Cluster and Grid Computing". Springer; First Edition. 2004.
5. Di Stefano M., "Distributed data management for Grid Computing". John Wiley & Sons Inc. 2005.
6. Miller M., "Cloud Computing: Web-Based applications that change the way you work and collaborate online". QUE Publishing. 2008.
7. Grid Computing and Distributed Systems (GRIDS) Laboratory - Department of Computer Science and Software Engineering (University of Melbourne), "Cluster and Grid Computing". 2007. http://www.cs.mu.oz.au/678/.
8. Zoltan J., Kacsuk P., Kranzlmuller D., "Distributed and Parallel Systems: Cluster and Grid Computing". The International Series in Engineering and Computer Science. Springer; 1st ed., 2004.
9. Wilkinson B, Allen M, "Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers", 2da ed., Pearson Prentice Hall, 2005.
10. AMD, "Evolución de la tecnología de múltiple núcleo". 2009. http://multicore. amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution.
11. Burger T. W., "Intel Multi-Core Processors: Quick Reference Guide". http://cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf
12. Mc Cool M., "Programming models for scalable multicore programming". 2007. http://www.hpcwire.com/features/17902939.html
13. Chai L., Gao Q., Panda D. K., "Understanding the impact of multi-core architecture in cluster computing: A case study with Intel Dual-Core System". IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGRID 2007), pp. 471-478. 2007.
14. Attwood T. K., Parry-Smith D. J., "Introducción a la Bioinformática". Pearson Educación S.A. 2002.
15. Zhang F., Qiao X., Liu Z., "A Parallel Smith-Waterman Algorithm Based on Divide and Conquer". Proceeding of the Fifth International Conference on Algorithms and Architecture for Parallel Processing. 2002.
16. Rucci E., De Giusti A., Chichizola F., Naiouf M., De Giusti L.,. "Comparación de modelos de comunicación/sincronización en Programación Paralela sobre Cluster de Multicores". XVI Congreso Argentino de Ciencias de la Computación, pp. 201-210. 2010.
17. Chichizola Franco, "Estudio analítico de TB óptimo en base a caracteristicas del cluster". Reporte técnico III-LIDI. 2011.
18. HP, "HP BladeSystem". http://h18004.www1.hp.com/products/blades/components/c-class.html.
19. HP, "HP BladeSystem c-Class architecture". http://h20000.www2.hp.com/bc/docs/support /SupportManual/c00810839/c00810839.pdf.
20. Feng, W.C., "The importance of being low power in high-performance computing". Cyberinfrastructure Technology Watch Quarterly (CTWatch Quarterly). 2005.
21. Balladini J., Grosclaude E., Hanzich M., Suppi R., Rexachs D., Luque E., "Incidencia de los modelos de programación paralela y escalado de frecuencia de CPUs en el consumo energético de los sistemas de HPC". XVI Congreso Argentino de Ciencias de la Computación, pp. 172-181. 2010.