Local DNA Sequence Alignment in a Cluster of Workstations: Algorithms and Tools

Alba Cristina M. A. Melo, Maria Emilia M. T. Walter, Renata Cristina F. Melo,
Marcelo N. P. Santana, Rodolfo B. Batista
Department of Computer Science
University of Brasilia, Brazil
Campus Universitario - ICC-Norte Subsolo, Asa Norte, Brasilia, Brazil
{albamm, mia, renata, nardellis, rodolfo}@cic.unb.br

Abstract

Distributed Shared Memory systems allow the use of the shared memory programming paradigm in distributed architectures where no physically shared memory exist. Scope consistent software DSMs provide a relaxed memory model that reduces the coherence overhead by ensuring consistency only at synchronization operations, on a per-lock basis. Much of the work in DSM systems is validated by benchmarks and there are only a few examples of real parallel applications running on DSM systems. Sequence comparison is a basic operation in DNA sequencing projects, and most of sequence comparison methods used are based on heuristics, that are faster but do not produce optimal alignments. Recently, many organisms had their DNA entirely sequenced, and this reality presents the need for comparing long DNA sequences, which is a challenging task due to its high demands for computational power and memory. In this article, we present and evaluate a parallelization strategy for implementing a sequence alignment algorithm for long sequences. This strategy was implemented in JIAJIA, a scope consistent software DSM system. Our results on an eight-machine cluster presented good speedups, showing that our parallelization strategy and programming support were appropriate.

1. Introduction

In order to make shared memory programming possible in distributed architectures, a shared memory abstraction must be created. This abstraction is called Distributed Shared Memory (DSM). The first DSM systems tried to give parallel programmers the same guarantees they had when programming uniprocessors. It has been observed that providing such a strong memory consistency model creates a huge coherence overhead, slowing down the parallel application and bringing frequently the system into a thrashing state [10]. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new shared memory behaviors that are different from the traditional uniprocessor one.

In the shared memory programming paradigm, synchronization operations must be used every time processes want to restrict the order in which memory operations should be performed. Using this fact, hybrid Memory Consistency Models guarantee that processors only have a consistent view of the shared memory at synchronization time [10]. This allows a great overlapping of basic read and write memory accesses that can potentially lead to considerable performance gains. By now, the most popular hybrid memory consistency models for DSM systems are Release Consistency (RC) [3] and Scope Consistency (ScC) [6].

JIAJIA is a scope consistent software DSM system proposed by [4] that implements consistency on a per-lock basis. When a lock is released, modifications made inside the critical section are sent to the home node, a node that keeps the up-to-date version of the data. When a lock is acquired, a message is sent to the acquirer process containing the identification of the data that are cached at the acquirer node that are no longer valid. These data are, then, invalidated and the next access will cause a fault and the up-to-date data will be fetched from the home node. On a synchronization barrier, however, consistency is globally maintained by JIAJIA and all processes are guaranteed to see all past modifications to the shared data [4].

A software DSM system can be used to parallelize DNA sequencing algorithms. In DNA sequencing projects, researchers want to compare two sequences to find similar portions of them, that is, they want to search similarities between two substrings of the sequences, and obtain good sequence alignments. This process is divided in two phases.

First, similarity regions between the long sequences are found with a local alignment algorithm. Second, for each similarity region, a global alignment is produced.

In practice, two families of tools for searching similarities between two sequences are widely used - BLAST [1] and FASTA [12], both based on heuristics. To obtain optimal local and global alignments, the most commonly used methods are the ones proposed by Smith-Waterman [15] and Needleman-Wunsh [11], respectively, both based on dynamic programming, with quadratic time and space complexity.

Many works are known which implement the Smith-Waterman algorithm for long sequences of DNA. Specifically, parallel implementations were proposed using MPI [8] or specific hardware. As far as we know, this is the first attempt to use a scope consistent DSM system to solve this kind of problem.

In this article, we present and evaluate a parallelization strategy for comparing two long DNA sequences. A DSM system was used since the shared memory programming model is often considered easier than the message passing counterpart. The strategy has two phases. In the first phase, as the algorithm proposed by [15] to obtain local alignments calculates each matrix element Ai,j by analyzing the elements Ai-1,j-1, Ai-1,j and Ai,j-1, we used the "wavefront method" [13]. The work was assigned to each processor in a column basis with a two-way lazy synchronization protocol. The heuristic proposed by [8] was used to reduce the space complexity to O(n). The similarity regions found in this first phase are sorted by size and placed in a shared queue. In the second phase, similarity regions are assigned to processors using a scattered mapping approach [2]. For each similarity region it is assigned, the processor performs the global alignment algorithm [11]. A previous work [9] discusses only the first phase of this strategy.

The results obtained in an eight-machine cluster with large sequence sizes show good speedups when compared with the sequential algorithm. For instance, to locally align two 400K sequences, a speedup of 4.58 was obtained in the first phase, reducing the execution time from more than 2 days to 10 hours. In the second phase, we obtained a speedup of 7.58 to globally align 1000 similarity regions with an average size of 300x300 bytes.

The rest of this paper is organized as follows. Section 2 briefly describes the local and global sequence alignment problems and the serial algorithms to solve them. In section 3, DSM systems and the JIAJIA software DSM are presented. Section 4 describes our strategy to align two long DNA sequences. Some experimental results are presented and discussed in section 5. Finally, section 6 concludes the paper and presents future work.

2. Smith-Waterman's Algorithm for DNA Sequence Alignment

To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters or substrings from the sequences [14]. We define alignment as the insertion of spaces in arbitrary locations along the sequences so that they end up with the same size.

Given an alignment between two sequences *s* and *t*, a score is associated for them as follows. For each column, we associate *+1* if the two characters are identical, *-1* if the characters are different and *-2* if one of them is a space. The score is the sum of the values computed for each column. The maximal score is the similarity between the two sequences, denoted by *sim(s,t)*. In general, there are many alignments with maximal score. Figure 1 shows the alignment of sequences s and t, with the score for each column. In this case, there are nine columns with identical characters, one column with distinct character and one column with a space, giving a total score *6 = 9*(+1)+1*(-1) + 1*(-2)*.

G	A	$-$	C	G	G	A	T	T	A	G
G	A	T	C	G	G	A	A	T	A	G
$+1$	$+1$	-2	$+1$	$+1$	$+1$	$+1$	-1	$+1$	$+1$	$+1$

$\Sigma = 6$

Figure 1. Alignment between sequences $s = GACGGATTAG$ and $t = GATCGGAATAG$.

For long sequences, it is unusual to obtain a global alignment. Instead, the local alignment algorithm is executed to obtain regions inside both sequences that are similar. For instance, for two 400K DNA sequences, we can obtain around 2000 similarity regions with an average size of 300x300 bytes. Global alignment is executed only for each similarity region. Figure 2 illustrates these concepts.

Smith-Waterman [15] proposed an algorithm based on dynamic programming to solve the local alignment problem. As input, it receives two sequences *s*, with $|s| = m$, and *t*, with $|t| = n$. There are $m + 1$ possible prefixes for *s* and $n + 1$ prefixes for *t*, including the empty string. An array $A_m + 1, n + 1$ is built, where the (i,j) entry contains the value of the similarity between two prefixes of *s* and *t*, sim(s[1..i],t[1..j]). Figure 3 shows the similarity array between *s*=AAGC and *t*=AGC. To obtain local alignments, the first row and column are initialized with zeros. The other entries are computed using equation 1.

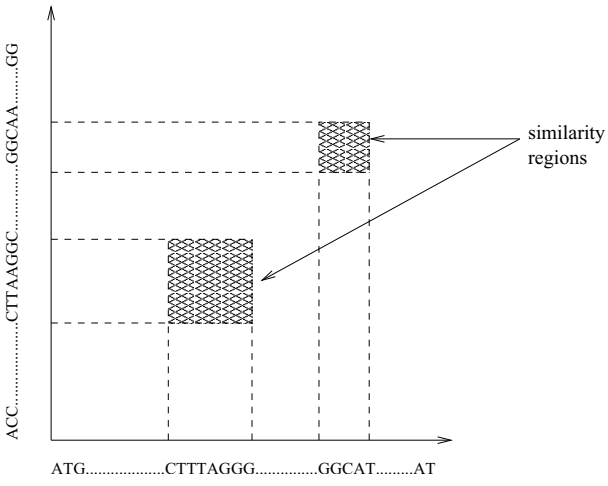

Figure 2. Local alignment of two 400K sequences (s and t), that produced two similarity regions

$$sim(s[1..i], t[1..j]) = \begin{cases} sim(s[1..i], t[1..j-1]) - 2, \\ sim(s[1..i-1], t[1..j-1]) \\ \quad +p(i,j), \\ sim(s[1..i-1], t[1..j]) - 2, \\ 0 \end{cases} \quad (1)$$

In equation 1, *p(i,j) = +1* if $s[i] = t[j]$ and *-1* if $s[i] \neq t[j]$. If we denote the array by A, the value of $A[i,j]$ is exactly $sim(s[1..i], t[1..j])$.

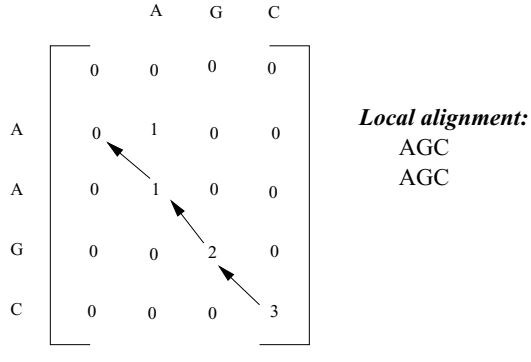

Figure 3. Array to compute the similarity between the sequences s=AAGC and t=AGC, using local alignment

We have to compute the array A row by row, left to right on each row, or column by column, top to bottom, on each column. Finally, arrows are drawn to indicate where the maximum value comes from, according to equation 1. In this case, the score value of the best alignment is in $A[m,n]$.

An optimal local alignment between two sequences can be obtained as follows. We begin in a maximal value in array A, and follow the arrow going out from this entry until we reach another entry with no arrow going out, or until we reach an entry with value 0. Each arrow used gives us one column of the alignment. If we consider an arrow leaving entry (i,j) and if this arrow is horizontal, it corresponds to a column with a space in s matched with $t[j]$, if it is vertical it corresponds to $s[i]$ matched with a space in t and a diagonal arrow means $s[i]$ matched with $t[j]$. An optimal local alignment is constructed from right to left if we have the array computed by the basic algorithm. Many optimal local alignments may exist for two sequences. The detailed explanation of this algorithm can be found in [14].

The time and space complexity of this algorithm is $0(mn)$, and if both sequences have approximately the same length, n, we get $O(n^2)$.

To obtain global alignments, the algorithm proposed by [11] is used. In this algorithm, some minor changes are made to the previously described algorithm. First, negative values are allowed and, thus, entries are still computed using equation 1 but the fourth condition no longer exists. Second, the first row and column of array A are filled with the gap penalty, as shown in figure 4. In the case of global alignment, only one alignment is produced for each pair of sequences.

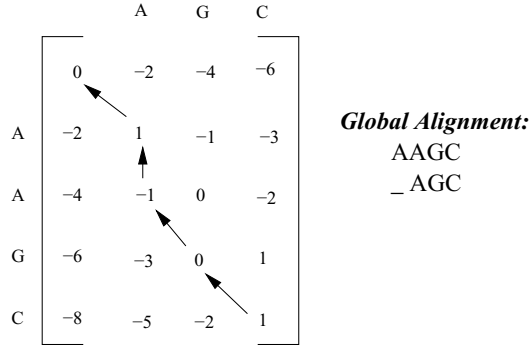

Figure 4. Array to compute the similarity between the sequences s=AAGC and t=AGC, using global alignment

2.1. Sequential Implementation

To obtain local alignments, we implemented a variant of the Smith-Waterman algorithm that uses two linear arrays [14]. The bi-dimensional array was not used since, for large sequences, the memory overhead would be prohibitive. The idea behind this algorithm is that it is possible to simulate the filling of the bi-dimensional array just using two rows in memory, since, to compute entry $A[i,j]$ we just need the

values of $A[i-1,j]$, $A[i-1,j-1]$ and $A[i,j-1]$. So, the space complexity of this version is linear, $O(n)$. The time complexity remains $O(n^2)$.

The algorithm works with two sequences s and t with length $n = |t|$. First, one of the arrays is initialized with zeros. Then, each entry of the second array is obtained from the first one with the Smith-Waterman algorithm, but using a single character of s on each step.

We denote $A[i,j] = sim(s[1..i, 1..j])$ as current score. Besides this value, each entry contains: initial and final alignment coordinates, maximal and minimal score, gaps, matches and mismatches counters and a flag showing if the alignment is a candidate to be an optimal alignment. These information allow us to keep a candidate optimal alignment with a score greater than a certain value. When computing the $A[i,j]$ entry, all the information of $A[i-1,j]$, $A[i-1,j-1]$ or $A[i,j-1]$ is passed to the current entry.

To obtain the above values for each entry, we used some heuristics proposed by [8]. The minimal and maximal scores are updated accordingly to the current score. The initial coordinates are updated if the flag is 0 and if the value of the maximal score is greater than or equal to the minimal score plus a parameter indicated by the user, where this parameter indicates a minimum value for opening this alignment as a candidate to an optimal alignment. If it is the case, the flag is updated to 1, and the initial coordinates change to the current position of the array. The final coordinates are updated if the flag is 1 and if the value of the current score is less than or equal to the maximal score minus a parameter, where the parameter indicates a value for closing an alignment. In this case, this alignment is closed and passed to a queue alignments of the reached optimal alignments and the flag is set to 0.

The gaps, matches and mismatches counters are employed when the current score of the entry being computed comes from more than one previous entry. In this case, they are used to define which alignment will be passed to this entry. We use an expression $(2 * matchescounter + 2 * mismatchescounter + gapscounter)$ to decide which entry to use. In this heuristic [8], gaps are penalized and matches and mismatches are rewarded. The greater value will be considered as the origin of the current entry. These counters are not reset when the alignments are closed, because the algorithm works with long sequences, and the scores of candidate alignments can begin with good values, turn down to bad values and turn again to good values.

If these values are still the same, our preference will be to the horizontal, to the vertical and at last to the diagonal arrow, in this order. This is a trial to keep together the gaps along the candidate alignment [8]. At the end of the algorithm, the coordinates of the best alignments are kept on the queue alignments. This queue is sorted by subsequence size and the repeated alignments are removed.

To obtain the global alignments, the queue alignments is accessed to obtain the begin and end coordinates of sequences s and t which determine the subsequences where the similarity regions reside. For each subsequence of s and t obtained this way, the global alignment algorithm proposed by [11] is executed.

3. Distributed Shared Memory Systems

Distributed Shared Memory has received a lot of attention in the last few years since it offers the shared memory programming paradigm in a distributed or parallel environment where no physically shared memory exists.

One way to conceive a DSM system is by the Shared Virtual Memory (SVM) approach [7]. SVM implements a single paged, virtual address space over a network of computers. It works basically as a virtual memory system. Local references are executed exclusively by hardware. When a non resident page is accessed, a page fault is generated and the SVM system is contacted. Instead of fetching the page from disk, as do the traditional virtual memory systems, the SVM system fetches the page from a remote node and restarts the instruction that caused the trap.

Relaxed memory models aim to reduce the DSM coherence overhead by allowing replicas of the same data to have, for some period of time, different values [10]. By doing this, relaxed models no longer guarantee strong consistency at all times, thus providing a programming model that is complex since, at some instants, the programmer must be conscious of replication.

Hybrid memory models are a class of relaxed memory models that postpone the propagation of shared data modifications until the next synchronization point [10]. These models are quite successful in the sense that they permit a great overlapping of basic memory operations while still providing a reasonable programming model. Release Consistency (RC) [3] and Scope Consistency (ScC) [6] are the most popular memory models for software DSM systems.

The goal of Scope Consistency (ScC) [6] is to take advantage of the association between synchronization variables and ordinary shared variables they protect. In Scope Consistency, executions are divided into consistency scopes that are defined on a per lock basis. Only synchronization and data accesses that are related to the same synchronization variable are ordered. The association between shared data and the synchronization variable (lock) that guards them is implicit and depends on program order. Additionally, a global synchronization point can be defined by synchronization barriers [6]. JIAJIA [4] and Brazos [16] are examples of scope consistent software DSMs.

In JIAJIA, the shared memory is distributed among the nodes in a NUMA-architecture basis. Each shared page has a home node. A page is always present in its home node and

it is also copied to remote nodes on an access fault. There is a fixed number of remote pages that can be placed at the memory of a remote node. When this part of memory is full, a replacement algorithm is executed.

In order to implement Scope Consistency, JIAJIA statically assigns each lock to a lock manager. The functions that implement lock acquire, lock release and synchronization barrier in JIAJIA are jia_lock, jia_unlock and jia_barrier, respectively [5].

Additionally, JIAJIA provides condition variables that are accessed by jia_setcv and jia_waitcv, to signal and wait on conditions, respectively. The programming style provided is SPMD (Single Program Multiple Data) and each node is distinguished from the others by a global variable jiapid [5].

4. Parallel DNA Sequence Alignment

The access pattern presented by the algorithm described in section 2.1 presents a non-uniform amount of parallelism and has been extensively studied in the parallel programming literature [13]. The parallelization strategy that is traditionally used in this kind of problem is known as the "wave-front method" since the calculations that can be done in parallel evolve as waves on diagonals.

Figure 5 illustrates the wave-front method. At the beginning of the computation, only one node can compute value $a[1,1]$. After that, values $a[2,1]$ and $a[1,2]$ can be computed in parallel, then, $a[3,1]$, $a[2,2]$ and $a[1,3]$ can be computed independently, and so on. The maximum parallelism is attained at the main matrix anti-diagonal and then decreases again.

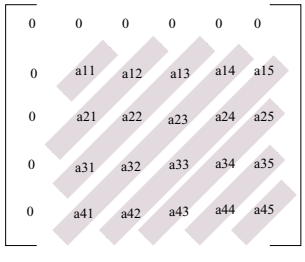

Figure 5. The wave-front method used to exploit the parallelism presented by the algorithm.

We propose a parallel version of the algorithm presented in section 2.1 and, thus, only two rows are used. Each processor p acts on two rows, a writing row and a reading row. Work is assigned in a column basis, i.e., each processor calculates only a set of columns on the same row, as shown in figure 6.

For the sake of simplicity, we represented in figure 6 the whole similarity array. However, each processor works in fact with two rows, as explained in the previous paragraph. When a processor finishes calculating a row, it copies this row to the reading row and starts calculating the next row, which is now the writing row.

Synchronization is achieved by locks and condition variables provided by JIAJIA (section 3). Barriers are only used at the beginning and at the end of computation.

In figure 6, assuming that there are N columns, processor 0 starts computing and, when value $a[1, N/4]$ is calculated, it writes this value at the shared memory and signals processor 1, that is waiting on jia_waitcv. At this moment, processor 1 reads the value from shared memory, signals processor 0, and starts calculating from $A[1, N/4+1]$. Processor 0 proceeds calculating elements $A[2,1]$ to $A[2, N/4]$. When this new block is finished, processor 0 issues a jia_waitcv to guarantee that the preceding value was already read by processor 1. The same protocol is executed by every processor i and processor $i+1$.

At the end of the computation of this first phase, the queue alignments contains the best local alignments found by our algorithm.

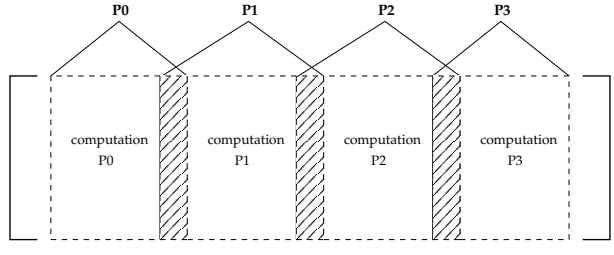

Figure 6. Work assignment in the parallel algorithm. Each processor p is assigned N/P columns, where P is the total number of processors and N is the length of the sequence.

As the average subsequence size obtained in phase 1 is small and hundreds or even thousands of subsequences can be obtained, we propose, for phase 2, a distributed algorithm where each processor calculates S/P global alignments, where S is the total number of subsequence pairs and P is the total number of processors.

In the proposed algorithm, the queue alignments is treated as a vector sorted by subsequence size and we use a scattered mapping approach [2] to assign similarity regions to processors. In this way, processor i is responsible for accessing positions i, $i+P$, $i+2P$, ..., of the vector alignments. This strategy eliminates the need of synchronization operations, such as those provided by locks and condition variables. For each position it accesses, the processor retrieves the begin and end coordinates of the subse-

quences corresponding to the local alignment. After that, it compares the subsequences using the global alignment algorithm described in section 2.

Each processor is responsible for recording the results of the global alignments it performs. These results include: begin and end coordinates of the aligned subsequences, the similarity between them and the globally aligned subsequences. After all global alignments are performed, the processors write their results in a shared vector. Once again, processor i is responsible for accessing the shared vector in positions i, $i + P$, $i + 2P$, ... In this way, no locks or condition variables are used.

5. Experimental Results

The proposed parallel algorithm was implemented in C, using the software DSM JIAJIA v.2.1.

To evaluate the gains of our strategy, we ran our experiments on a dedicated cluster of 8 Pentium II 350 MHz, with 160 MB RAM connected by a 100Mbps Ethernet switch. The JIAJIA software DSM system ran on top of Debian Linux 2.1.

Our results were obtained with real DNA sequences obtained from *www.ncbi.nlm.nih.gov/PMGifs/Genomes*. Five sequence sizes were considered (15K, 50K, 80K, 150K and 400K). Execution times for each n x n sequence comparisons, where n is the size of both sequences, with 1, 2, 4 and 8 processors are shown in table 1. Figure 7 shows the absolute speedups, which were calculated considering the total execution times and thus include times for initialization and collecting results.

Size n x n	Serial	2 proc	4 proc	8 proc
15K	296	283.18	202.18	181.29
50K	3461	2884.15	1669.53	1107.02
80K	7967	6094.18	3370.40	2162.82
150K	24107	19522.95	10377.89	5991.79
400K	175295	141840.98	72770.99	38206.84

Table 1. Total Execution Times (seconds) for 5 sequence sizes

As can be seen in figure 7, for small sequence sizes, e.g. 15K, very bad speedups are obtained since the parallel part is not long enough to surpass the amount of synchronization inherent to the algorithm. As long as sequence sizes increase, better speedups are obtained since more work can be done in parallel. This effect can be better noticed in figure 8, which presents a breakdown of the execution time of each sequence comparison.

We also compared the results obtained by our implementation (denoted GenomeDSM) with BlastN. For this task,

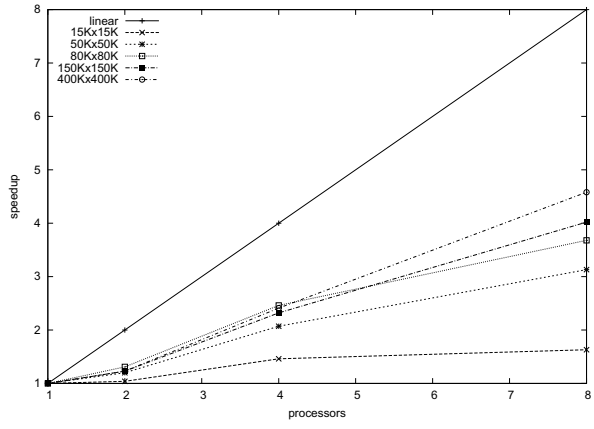

Figure 7. Absolute speedups for DNA sequence comparison.

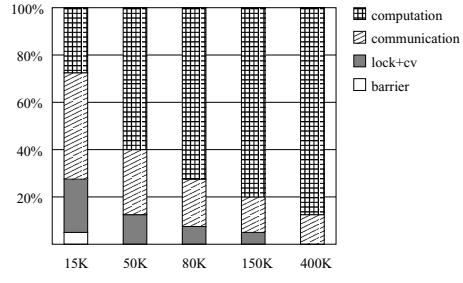

Figure 8. Execution time breakdown for 5 sequence sizes, containing the relative time spent in computation, communication, lock and condition variable and barrier.

we used two 50K mithocondrial genomes, *Allomyces acrogynus* and *Chaetosphaeridium globosum*.

In table 2, we present a comparison between these programs, showing the coordinates of the alignments with the best scores found by them. Still in table 2, we can note that the results obtained by both programs are very close but they are not the same. This can be explained since both programs use heuristics that involve different parameters.

We also developed a tool to visualize the alignments found by GenomeDSM [9]. An example can be seen in figure 9. We note that the user can make zoom in a particular region and obtain more details of the desired alignment.

In phase 2, for each similarity region, global alignments of subsequences are generated. Figure 10 shows the execution times to globally align 100, 1000, 2000, 3000, 4000, and 5000 pairs of subsequences obtained from the similarity regions with 1, 2, 4 and 8 processors. In order to evaluate the results of this second phase, we varied the parameter minimal score, that defines which alignments are

		GenomeDSM	BlastN
Alignment 1	Begin	(39109,55559)	(39099,55549)
	End	(39839,56252)	(39196,55646)
Alignment 2	Begin	(39475,48905)	(39522,48952)
	End	(39755,49188)	(39755,49005)
Alignment 3	Begin	(28637,47919)	(28667,47949)
	End	(28753,48035)	(28754,48036)

Table 2. Comparison among results obtained by GenomeDSM and BlastN

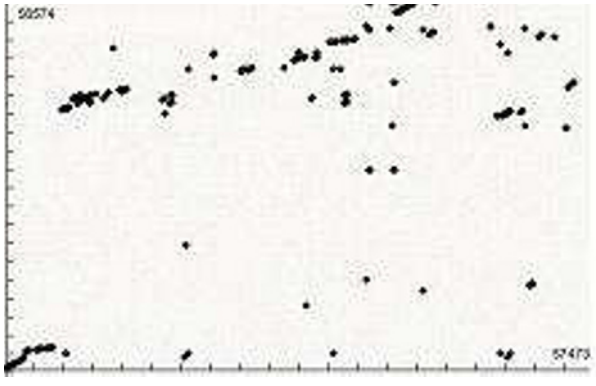

Figure 9. Visualization of the alignments generated by GenomeDSM with the 50K sequences. Plotted points show the similarity regions between the two genomes.

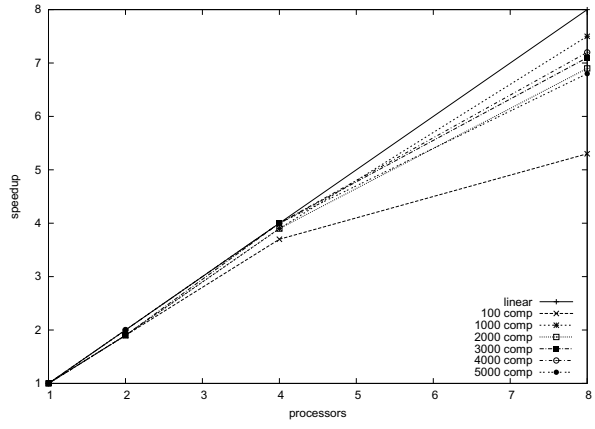

Figure 10. Speedups obtained in phase 2 for 2, 4 and 8 processors for a varying number of subsequence comparisons.

intitial_x: 8553 final_x: 8625
intitial_x: 42400 final_x: 42472
similarity: 39
align_s: G_AG_GTATACAACTTCGCTACAGAGT GGACACTAATGGGTGGCTGTAATGCTGCTTAT GTACAGTCGGTATT
align_t: GTTGTGTGTACAA_T_CGCTACAGACT GGTTCACCGTGGGCCTGAAATGGTGCTTAAT GTACAGTCGAAATT

intitial_x: 11669 final_x: 11751
intitial_x: 42964 final_x: 43046
similarity: 26
align_s: CTGCAAATCCGTTAGT_TACGCCTGTA TCTATCGTGCCTGAGTGGTACTTTTTACC_GT TCTATGCCA_TATTAAGACTATACCA
align_t: CAGCAAATCCAAT_GTCAACACCTGCG CATATAGTACCAGAATGGTATTTTCTACCAGT T_TACGCAATTCTTCGCAG_TATACCA

Figure 11. Global alignment of two subsequences generated in phase 1.

considered relevant. Small values for minimal scores generate more similarity regions and, consequently, more pairs to be compared. In figure 9, 123 similarity regions are represented. In figure 10, this result is labeled as 100 pairs. The average size of the subsequences is 253 bytes. Figure 10 shows the speedups obtained in this process. An example of the results produced by this second phase is illustrated in figure 11.

The distributed algorithm we proposed to globally align subsequences uses a scattered mapping scheme, which is quite effective, since no synchronization is needed to obtain work from the shared queue. For this reason, we were able to obtain very good speedups, e.g., 7.57 to globally align 1000 subsequences with 8 processors. Also, the speedup obtained apparently does not depend on the shared queue size. This can be seen in figure 10. Speedups for 2 and 4 processors are between 2 and 1.91 and between 4 and 3.76, for 100 and 5000 subsequence pairs, respectively. Speedups for 8 processors presented a slightly higher variation, where a speedup of 7.57 was attained for 1000 subsequences pairs.

For 100 and 5000 subsequence pairs, speedups obtained for 8 processors were 5.33 and 6.80, respectively. For 100 comparisons, we also measured a speedup of 5.33 for 7 processors. That indicates that, with a reduced number of comparisons of small subsequences, no benefit is obtained when increasing the number of processors from 7 to 8.

Martins et al. [8] presented a parallel version of the Smith-Waterman [15] algorithm using MPI that ran on a Beowulf system with 64 nodes each containing 2 processors. Speedups attained considering the total execution time were very close to ours, e.g., for 800KBx500KB sequence alignment, a speedup of 16.1 were obtained for 32 processors and we obtained a speedup of 4.58 with 8 processors for 400KB x 400KB sequences. Besides that, the DSM programming model is considered easier for this kind of problem.

6. Conclusion and Future Work

In this paper, we proposed and evaluated one parallel and distributed approach to solve the DNA local and global sequence alignment problem. A DSM system was chosen since, for this kind of problem, DSM offers an easier programming model than its message passing counterpart. The wavefront method was used for local alignment and work was assigned in a column basis. In this phase, synchronization was achieved with locks and condition variables and barriers. In the second phase, a distributed algorithm is proposed that assigns work to processors using a scattered mapping function.

The results obtained to locally align large sequences in an eight machine cluster present good speedups that are improved as long as the sequence lengths increase. In order to compare two sequences of approximately 400KB, we obtained a 4.58 speedup on the total execution time, reducing execution time from 2 days to 10 hours. To globally align 1000 pairs of subsequences obtained in the first phase, we also obtained very good speedups, e.g., 7.57 with 8 processors This shows that our parallelization strategy and the DSM programming support were appropriate to solve our problem.

As future work, we intend to port the algorithm implemented in MPI proposed in [8] to solve the same problem to our cluster and compare its results with ours. Also, we intend to propose and evaluate a variant of our approach, which will use variable block size to take advantage of the non-uniform type of parallelism presented by the wavefront approach.

References

[1] S. F. Altschul et al. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[2] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[3] K. Gharachorloo. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Int. Symp. On Computer Architecture (ISCA)*, pages 15–24. ACM, 1990.

[4] S. Hu, W. Shi, and Z. Tang. Jiajia: An svm system based on a new cache coherence protocol. In *High Performance Computing and Networking (HPCN)*, pages 463–472. Springer-Verlag, 1999.

[5] W. Hu and W. Shi. Jiajia users manual. Technical report, Chinese Academy of Sciences, 1999.

[6] L. Iftode, J. Singh, and K. Li. Scope consistency: Bridging the gap between release consistency and entry consistency. In *8th ACM SPAA'96*, pages 277–287. ACM, 1996.

[7] K. Li. *Shared Virtual Memory on Loosely Coupled Architectures*. PhD thesis, Yale University, 1986.

[8] W. S. Martins, J. B. Del Cuvillo, F. J. Useche, K. B. Theobald, and G. R. Gao. A multithread parallel implementation of a dynamic programming algorithm for sequence comparison. In *Brazilian Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–8, 2001.

[9] R. Melo, M. E. T. Walter, A. C. M. A. Melo, and R. B. Batista. Comparing two long dna sequences using a dsm system. In *Euro-Par 2003*, pages 517–524. Springer-Verlag, 2003.

[10] D. Mosberger. Memory consistency models. *Operating Systems Review*, pages 18–26, 1993.

[11] S. B. Needleman and C. D. Wunsh. A general method applicable to the search of similarities of amino acid sequences of two proteins. *Journal of Molecular Biology*, (48):443–453, 1970.

[12] W. R. Pearson and D. L. Lipman. Improved tools for biological sequence comparison. In *Proceedings Of The National Academy Of Science USA*, pages 2444–2448. NAS, 1988.

[13] G. Pfister. *In Search of Clusters - The Coming Battle for Lowly Parallel Computing*. Prentice-Hall, 1995.

[14] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. Brooks/Cole Publishing Company, 1997.

[15] T. F. Smith and M. S. Waterman. Identification of common molecular sub-sequences. *Journal of Molecular Biology*, (147):195–197, 1981.

[16] E. Speight and J. Bennet. Brazos: a third generation dsm system. In *USENIX/WindowsNT Workshop*, pages 95–106, 1997.