Parallel DNA Sequence Alignment Using a DSM System in a Cluster of Workstations

Renata Cristina Faray Melo, Maria Emília Telles Walter,
Alba Cristina Magalhaes Alves de Melo, and Rodolfo B. Batista

Department of Computer Science, Campus Universitario - Asa Norte, Caixa Postal 4466,
University of Brasilia, Brasilia – DF, CEP 70910-900, Brazil
{renata, mia, albaum, 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 synchronisation 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 parallelisation 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 parallelisation 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[13]. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new shared memory behaviours that are different from the traditional uniprocessor one.

In the shared memory programming paradigm, synchronisation 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 synchronisation

time [13]. 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)[7].

JIAJIA is a scope consistent software DSM system proposed by [5] 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 synchronisation barrier, however, consistency is globally maintained by JIAJIA and all processes are guaranteed to see all past modifications to the shared data [5].

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 local sequence alignments. In practice, two families of tools for searching similarities between two sequences are widely used - BLAST [1] and FASTA, both based on heuristics. To obtain optimal local alignments, the most commonly used method is based on the Smith-Waterman algorithm [17], based on dynamic programming, with quadratic time and space complexity.

Many works are known that aim to efficiently implement the Smith-Waterman algorithm for long sequences of DNA. Specifically, parallel implementations were proposed using MPI [12] or specific hardware [4]. 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 parallelisation strategy for implementing the Smith-Waterman algorithm. A DSM system was used since the shared memory programming model is often considered easier than the message passing counterpart. As the method proposed by [17] calculates each matrix element $A_{i,j}$ by analysing the elements $A_{i-1,j-1}$, $A_{i-1,j}$ and $A_{i,j-1}$, we used the “wavefront method” [14]. In this method, the parallelism is small at the beginning of the calculation, increases to a maximum across the matrix diagonal and then decreases again. The work was assigned to each processor in a column basis with a two-way lazy synchronisation protocol. The heuristic proposed by [12] was used to reduce the space complexity to $O(n)$.

The results obtained in an eight-machine cluster with large sequence sizes show good speedups when compared with the sequential algorithm. For instance, to align two 400KB sequences, a speedup of 4.58 was obtained, reducing the execution time from more than 2 days to 10 hours.

The rest of this paper is organized as follows. Section 2 briefly describes the sequence alignment problem and the serial algorithm to solve it. In section 3, DSM systems and the JIAJIA software DSM are presented. Section 4 describes our parallel algorithm. 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 Local 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 [15]. We define *alignment* as the insertion of spaces in arbitrary locations along the sequences so that they finish with the same size.

Given an alignment between two sequences s and t , an 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 $\text{sim}(s, t)$. In general, there are many alignments with maximal score. Figure 1 illustrates this strategy.

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	= 6

Fig. 1. Alignment of the sequences $s = \text{GACGGATTAG}$ and $t = \text{GATCGGAATAG}$, with the score for each column. There are nine columns with identical characters, one column with distinct character and one column with a space, giving a total score $6 = 9*(+1) + 1*(-1) + 1*(-2)$

Smith-Waterman [17] proposed an algorithm based on dynamic programming. 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 $(m+1) \times (n+1)$ is built, where the (i, j) entry contains the value of the similarity between two prefixes of s and t , $\text{sim}(s[1..i], t[1..j])$.

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

Figure 2 shows the similarity array between $s = \text{AAGC}$ and $t = \text{AGC}$. The first row and column are initialised with zeros and the other entries are computed using equation 1. In this equation, $p(i, j) = +1$ if $s[i] = t[j]$ and -1 if $s[i] \neq t[j]$. Note that if we denote the array by a , the value of $a[i, j]$ is exactly $\text{sim}(s[1..i], t[1..j])$.

		A	G	C
A	0	0	0	0
A	0	1	0	0
G	0	0	0	0
C	0	0	0	3

Fig. 2. Array to compute the similarity between the sequences $s = \text{AAGC}$ and $t = \text{AGC}$.

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. Figure 3 presents the basic dynamic programming algorithm for filling the array a . Notice that the score value of the best alignment is in $a[m,n]$.

Algorithm Similarity

Input: sequences s and t

Output: similarity between s and t

```

m ← |s|
n ← |t|
For i ← 0 to m do
a[i,0] ← i × g
For j ← 0 to n do
a[0,j] ← j × g
For i ← 1 to m do
For j ← 1 to n do
a[i,j] ← max(a[i-1,j]-2, a[i-1, j-1]±1, a[i,j-1]-2, 0)

Return a[m, n]
```

Fig. 3. Basic dynamic programming algorithm to build a similarity array $a[m][n]$.

An optimal 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 alignment is constructed from right to left if we have the array computed by the basic algorithm. It is not necessary to implement the arrows explicitly, a test can be used to choose the next entry to visit. The detailed explanation of this algorithm can be found in [15].

Many optimal alignments may exist for two sequences because many arrows can leave an entry. In general, the algorithms for giving optimal alignments return just one of them, giving preference to the vertical edge, to the diagonal and at last to the horizontal edge.

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

2.1 Sequential Implementation

We implemented a variant of the algorithm described in Section 2 that uses two linear arrays [15]. The bi-dimensional array could not be 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 $|t|$. First, one of the arrays is initialised with zeros. Then, each entry of the second array is obtained from the first one with the algorithm described in Section 2, 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 [12]. 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 ($2 * matches\ counter + 2 * mismatches\ counter + gaps\ counter$) to decide which entry to use. In this heuristic [12], 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, the heuristic adopted is different from the one described in Section 2. 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 [12]. At the end of the algorithm, the coordinates of the best alignments are kept on the queue *alignments*. This queue is sorted and the repeated alignments are removed. The best alignments are then reported to the user.

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 [11]. 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 [13]. 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 synchronisation point [13]. 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) [7] are the most popular memory models for software DSM systems.

The goal of Scope Consistency (ScC) [7] is to take advantage of the association between synchronisation 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 synchronisation and data accesses that are related to the same synchronisation variable are ordered. The association between shared data and the synchronisation variable (lock) that guards them is implicit and depends on program order. Additionally, a global synchronisation point can be defined by synchronisation barriers [7]. JIAJIA [5] and Brazos [16] are examples of scope consistent software DSMs.

JIAJIA is a software DSM system proposed by [5] which implements the Scope Consistency memory model. 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 synchronisation barrier in JIAJIA are `jia_lock`, `jia_unlock` and `jia_barrier`, respectively [6].

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` [6].

4 Parallel Algorithm to Compare DNA Sequences

The access pattern presented by the algorithm described in section 2 presents a non-uniform amount of parallelism and has been extensively studied in the parallel programming literature [14]. The parallelisation 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 4 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.

$$\begin{bmatrix}
 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\
 0 & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\
 0 & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\
 0 & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\
 0 & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5}
 \end{bmatrix}$$

Fig. 4. The wave-front method 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 5. Synchronisation 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 5, processor 0 starts computing and, when value $a_{1,3}$ 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 $a_{1,4}$. Processor 0 proceeds calculating elements $a_{2,1}$ to $a_{2,3}$. When this new block is finished, processor 0 issues a jia_waitcv to guarantee that the preceeding value was already read by processor 1. The same protocol is executed by every processor i and processor $i+1$.

Fig. 5. Work assignment in the parallel algorithm. Each processor p is assigned N/P rows, where P is the total number of processors and N is the length of the sequence.

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 the site www.ncbi.nlm.nih.gov/PMGifs/Genomes. Five sequence sizes were considered (15KB, 50KB, 80KB, 150KB and 400KB). Execution times and speedups for these sequences, with 1,2,4 and 8 processors are shown in Table 1 and illustrated in figure 6. Speedups were calculated considering the total execution time and thus include times for initialisation and collecting results.

Table 1. Total execution times (seconds) and speedups for 5 sequence comparisons

Size	Serial Exec	2 proc Exec /Speedup	4 proc Exec /Speedup	8 proc Exec /Speedup
15K x 15K	296	283.18/1.04	202.18/1.46	181.29/1.63
50K x 50K	3461	2884.15/1.20	1669.53/2.07	1107.02/3.13
80K x 80K	7967	6094.19/1.31	3370.40/2.46	2162.82/3.68
150K x 150K	24107	19522.95/1.23	10377.89/2.32	5991.79/4.02
400K x 400K	175295	141840.98/1.23	72770.99/2.41	38206.84/4.58

Fig. 6. Total execution times (s) for DNA sequence comparison

Fig. 7. Execution time breakdown for 5 sequence sizes, containing the relative time spent in computation, communication, lock and condition variable and barrier.

As can be seen in table 1 and figure 6, 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 synchronisation 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 7, which presents a breakdown of the execution time of each sequence comparison.

Martins et al. [12] presented a parallel version of the Smith-Waterman[1] 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 800Kx500K sequence alignment, a speedup of 16.1 were obtained for 32 processors and we obtained a speedup of 4.58 with 8 processors for 400K x 400K sequences. Besides that, our solution is cheaper and the DSM programming model is easier for this kind of problem.

6 Conclusions and Future Work

In this paper, we proposed and evaluated one parallel algorithm to solve the DNA local 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 and work was assigned in a column basis. Synchronisation was achieved with locks and condition variables and barriers.

The results obtained for 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. This shows that that our parallelisation strategy and the DSM programming support were appropriate to our problem.

As future work, we intend to port the algorithm implemented in MPI proposed in [12] 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, v. 25, n. 17, p. 3389–3402, 1997.
2. J. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. PhD dissertation, Rice University, 1993.
3. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*. Proc. Int. Symp. On Computer Architecture, May, 1990, p15–24.
4. L. Grate, M. Diekhans, D. Dahle, R. Hughey, *Sequence Analysis With the Kestrel SIMD Parallel Processor*. 1998.
5. W. Hu., W. Shi., Z. Tang. *JIAJIA: An SVM System Based on A New Cache Coherence Protocol*. In Proc. of HPCN'99, LNCS 1593, pp. 463–472, Springer-Verlag, April, 1999.

6. W.Hu, W.Shi. *JIAJIA User's Manual*. Technical report, Chinese Academy of Sciences, 1999.
7. Iftode L., Singh J., Li K. *Scope Consistency: Bridging the Gap Between Release Consistency and Entry Consistency*. Proc. Of the 8th ACM SPAA'96, June, 1996, pages 277–287.
8. Keleher, P., Cox, A., Dwarkakas, S., Zwaenenpoel, W. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. Proc. of USENIX, 1994, p.115–132.
9. Lamport L. *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*. IEEE Transactions on Computers, 1979, 690–691.
10. D. Lenosky et al. *The DASH Prototype: Logic Overhead and Performance*. IEEE Transactions on Parallel and Distributed Systems, January, 1993.
11. K. Li. *Shared Virtual Memory on Loosely Coupled Architectures*. PhD Dissertation, Yale University, 1986.
12. W. S. Martins, J. B. Del Cuvillo, F. J. Useche, K. B. Theobald, G. R. Gao. *A Multithread Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison*. Proceedings of the Symposium on Computer Architecture and High Performance Computing, 2001, Pirenopolis, Brazil, p.1–8.
13. Mosberger D. *Memory Consistency Models*. Operating Systems Review, p. 18-26, 1993.
14. G. Pfister., *In Search of Clusters – The Coming Battle for Lowly Parallel Computing*. 1995.
15. J. C. Setubal, J. Meidanis, *Introduction to Computational Molecular Biology*. Pacific Grove, CA, United States: Brooks/Cole Publishing Company, 1997.
16. E. Speight, J. Bennet. *Brazos: a Third Generation DSM System*. Proc. Of the USENIX/WindowsNT Workshop, p.95-106, August, 1997.
17. T. F. Smith, M. S. Waterman. *Identification of common molecular sub-sequences*. Journal of Molecular Biology, 147 (1) 195-197–1981.