Comparing Two Long Biological Sequences Using a DSM System

Renata Cristina F. Melo, Maria Emília Telles Walter,
Alba Cristina Magalhaes Alves Melo, Rodolfo Batista, Marcelo Nardelli,
Thelmo Martins, and Tiago Fonseca

Department of Computer Science, Campus Universitario - Asa Norte, Caixa Postal 4466,
University of Brasilia, Brasilia – DF, CEP 70910-900, Brazil
{renata, mia, albamm, rodolfo, thelmo, tiago,
marcelo}@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. 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 in a DSM system. Our results on an eight-machine cluster presented good speedups, showing that our parallelisation strategy and programming support were appropriate.

1 Introduction

Distributed Shared Memory (DSM) is an abstraction that allows the use of the shared memory programming paradigm in parallel or distributed architectures. The first DSM systems tried to give parallel programmers the same guarantees they had when programming uniprocessors and this approach created a huge coherence overhead [10]. 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 are used every time processes want to restrict the order in which memory operations should be performed. Using this fact, hybrid Memory Consistency Models (MCM) guarantee that processors only have a consistent view of the shared memory at synchronisation time [10]. This allows a great overlapping of basic read and write operations that can lead to considerable performance gains. By now, the most popular MCMs for DSM systems are Release Consistency [2] and Scope Consistency [6].

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 517–524, 2003.

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 made visible to the next process that acquires the same lock. On a synchronisation barrier, however, consistency is globally maintained and all processes are guaranteed to see all past modifications to the shared data.

In DNA sequencing projects, researchers want to compare two sequences to find similar portions of them 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[13], both based on heuristics and used for comparing long sequences. To obtain optimal local alignments, the most widely used algorithm is the one proposed by Smith-Waterman [12], with quadratic time and space complexity.

Many works are known that implement the Smith-Waterman algorithm for long sequences of DNA. Specifically, parallel implementations were proposed using MPI [9] or specific hardware [3]. 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 in a DSM system . Work is assigned to each processor in a column basis with a two-way lazy synchronisation protocol. An heuristic described in [11] was used to reduce the space complexity.

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 organised as follows. Section 2 describes the sequence alignment problem and the optimal algorithm to solve it. In Section 3, DSM systems are presented. Section 4 describes our sequential and parallel algorithm. Some experimental results are discussed in Section 5. Finally, Section 6 concludes the paper.

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 [11]. 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 $sim(s,t)$. In general, there are many alignments with maximal score. Figure 1 gives and example.

Smith-Waterman [12] proposed an algorithm based on dynamic programming. As input, it receives two sequences s, $|s|=m$, and t, $|t|=n$. There are $m+1$ possible prefixes

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= GACGGATTAG* and *t=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)

for *s* and *n+1* prefixes for *t*, including the empty string. An array *(m+1)x(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])*.

Fig. 2 shows the similarity array between *s=AAGC* and *t=AGC*. The first row and column are initialised with zeros. The other entries are computed using Equation 1.

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

In equation 1, *p(i,j) = +1* if *s[i]=t[j]* and *–1* if *s[i]≠t[j]*. If we denote the array by *a*, the value of *a[i,j]* is the similarity between s[1..i] and t[1..j], *sim(s[1..i],t[1..j])*.

		A	G	C
	0	0	0	0
A	0	1	0	0
A	0	1	0	0
G	0	0	2	0
C	0	0	0	3

Fig. 2. Array to compute the similarity between the sequences *s=AAGC* and *t=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*.

```
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 x g
For j ← 0 to n do
  a[0, j] ← j x 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*.

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. An horizontal arrow leaving entry (i,j) corresponds to a column with a space in *s* matched with *t[j]*, a vertical arrow 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. The detailed explanation of this algorithm can be found in [11]. Many optimal alignments may exist for two sequences because many arrows can leave an entry.

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

3 Distributed Shared Memory Systems

Distributed Shared Memory offers the shared memory programming paradigm in a distributed environment where no physically shared memory exists. DSM is often implemented a single paged, virtual address space over a network of computers that is managed by the virtual memory system [8]. Local references usually proceed without the interference of the DSM system and only generate exceptions by protection fault. When a non resident page is accessed, a page fault is generated and the DSM system is contacted to fetch the page from a remote node. The instruction that caused the page fault is restarted and the application can proceed.

In order to improve performance, DSM systems usually replicate pages. Maintaining strong consistency among the copies was the approach used by the first DSM systems but it created a huge coherence overhead.[7]

Relaxed memory models aim to reduce this overhead by allowing replicas of the same data to have, for some period of time, different values [10]. By doing this, relaxed models provide a programming model that is complex since, at some moments, the programmer is 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 [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) [2] 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 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 operations and data accesses that are related to the same synchronisation variable are ordered. The association between shared data and the synchronisation variable that guards them is implicit and depends on program order. Additionally, a global synchronisation point can be defined by synchronisation barriers. JIAJIA [4] is an example of scope consistent software DSM.

JIAJIA implements the Scope Consistency memory model with a write-invalidate multiple-writer home-based protocol. 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.

Each lock is assigned 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 [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 Algorithm to Compare DNA Sequences

To analyse the performance of our parallel algorithm, we implemented a sequential variant of the algorithm described in Section 2 that uses two linear arrays [11]. The bi-dimensional array was not used since, for large sequences, the memory overhead would be prohibitive. In this algorithm, we 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*, $|s|=m$ and *t*, $|t|=n$. First, one linear array 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*. Each entry also 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. 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.

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 [9]. The greater value is considered as the origin of the current entry. If the 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. 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.

The access pattern presented by this variant of the Smith-Waterman algorithm leads to a non-uniform amount of parallelism. The parallelisation strategy that is traditionally used in this case is the “wave-front method” since the calculations that are done in parallel evolve as waves on diagonals.

We propose a parallel version of this variant where each processor p acts on two rows, a writing and a reading row. Work is assigned in a column basis, i.e., each proc-

essor calculates only a set of columns on the same row, as shown in figure 4. Synchronisation is achieved by locks and condition variables provided by JIAJIA [5,6]. Barriers are only used at the beginning and at the end of computation.

In figure 4, p0 starts computing and, when value $a_{1,3}$ is calculated, it writes this value at the shared memory and signals p1, that is waiting on jia_waitcv. At this moment, p1 reads the value from shared memory, signals p0, and starts calculating from $a_{1,4}$. P0 proceeds then calculating elements $a_{2,1}$ to $a_{2,3}$ When this new block is finished, p0 issues a jia_waitcv to guarantee that the preceeding value was already read by p1. The same protocol is executed by every processor pi and processor pi+1.

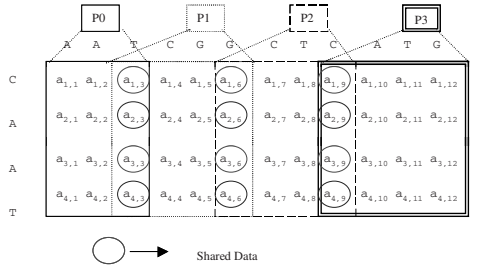

Fig. 4. Work assignment in the parallel algorithm. Each processor p is assigned N/P rows, where P is the total number of processors and N is the length of the sequence.

5 Experimental Results

Our parallel algorithm was implemented in C, using the software DSM JIAJIA v.2.1 on top of Debian Linux 2.1. We ran our experiments on a dedicated cluster of 8 Pentium II 350 MHz, 160 MB RAM connected by a 100Mbps switch. Our tests used real DNA sequences obtained from *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. 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

As can be seen in table 1, 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

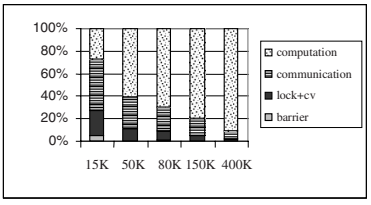

Fig. 5. Execution time breakdown for 5 sequence sizes, containing the relative time spent in computation, communication, lock and condition variable and barrier.

are obtained. This effect can be better noticed in figure 5, 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, FASTA and PipMaker [14]. For this task, we used two 50KB mithocondrial genomes, *Allomyces acrogynus* and *Chaetosphaeridium globosum*.

In table 2, we present a comparison among these four programs, showing the alignments with the best scores found by GenomeDSM. Still in table 2, the second and third best alignments were not found by FASTA. In FASTA, the query sequence had to be broken, since our version of FASTA did not compute sequences greater that 20KB. Thus, the lack of these two sequence alignments can be due to this limitation.

Table 2. Comparison among results obtained by GenomeDSM, BlastN, FASTA and PipMaker

		GenomeDSM	BlastN	FASTA	PipMaker
Alignment 1	Begin	(39109, 55559)	(39099, 55549)	(38396, 55317)	(38396, 54897)
	End	(39839, 56252)	(39196, 55646)	(39840, 56673)	(39828, 56239)
Alignment 2	Begin	(39475, 48905)	(39522, 48952)	-	(39617, 49050)
	End	(39755, 49188)	(39755, 49005)	-	(39756, 49189)
Alignment 3	Begin	(28637, 47919)	(28667, 47949)	-	(28505, 47787)
	End	(28753, 48035)	(28754, 48036)	-	(28756, 48038)

We also developed a tool to visualise the alignments found by GenomeDSM. An example can be seen in figure 6.

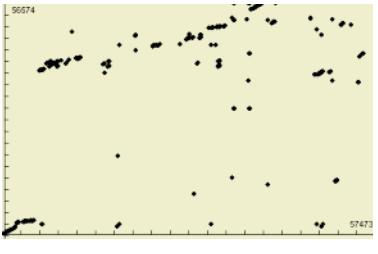

Fig. 6. Visualisation of the alignments generated by GenomeDSM with the 50KB sequences. Plotted points show the similarity regions between the two genomes.

Martins et al. [9] presented a version of the Smith-Waterman algorithm using MPI that ran on a Beowulf system with 64 nodes each containing 2 processors. Speedups

attained were very close to ours, e.g., for 800Kx500K sequence alignment, a speedup of 16.1 were obtained for 32 processors.

6 Conclusions and Future Work

In this paper, we proposed and evaluated a DSM implementation of the Smith-Waterman algorithm that solve the DNA local sequence alignment problem. Work is assigned to each processor in a column basis and the wavefront method was used.

The results obtained in an 8-machine cluster present good speedups which are improved as long as the sequence lengths increase. To compare sequences of 400KB, we obtained a 4.58 speedup on the total execution time, reducing execution time of the sequential algorithm 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 [9] 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.

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. K. Gharachorloo et al, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proc. Int. Symp. On Computer Architecture*, May, 1990, p. 15–24.
3. L. Grate, M. Diekhans, D. Dahle, R. Hughey, *Sequence Analysis With the Kestrel SIMD Parallel Processor* –1998.
4. 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.
5. W.Hu, W.Shi, "JIAJIA User´s Manual", Technical report, CAS – China, 1999.
6. 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.
7. Lamport L., *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, 1979, 690–691.
8. K. Li, "Shared Virtual Memory on Loosely Coupled Architectures", PhDThesis, Yale, 1986.
9. W. S. Martins, et al., *A Multithread Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison*, Proc. SBAC-PAD, 2001, Pirenopolis, Brazil, p.1–8.
10. Mosberger D.: Memory Consistency Models, Operating Systems Review, p. 18–26, 1993.
11. J. C. Setubal, J. Meidanis, *Introduction to Computational Molecular Biology*. Pacific Grove, CA, United States: Brooks/Cole Publishing Company, 1997.
12. T. F. Smith, M. S. Waterman, *Identification of common molecular sub-sequences* – Journal of Molecular Biology, 147 (1) 195-197–1981.
13. W. R. Pearson; D. L. Lipman, *Improved tools for biological sequence comparison*. Proceedings Of The National Academy Of Science USA, v. 85, p. 2444–2448, April 1988.
14. Schwartz Et Al. – *PipMaker – A Web Server for Aligning Two Genomic DNA Sequences* – Genome Research 10:577-586, April 2000 – http://bio.cse.psu.edu.