2002

Parallel Syntenic Alignments

Natsuhiko Futamura
Iowa State University

Srinivas Aluru
Iowa State University, aluru@iastate.edu

Xiaoqiu Huang
Iowa State University, xqhuang@iastate.edu

Parallel Syntenic Alignments

Abstract
Given two genomic DNA sequences, the syntenic alignment problem is to compute an ordered list of subsequences for each sequence such that the corresponding subsequence pairs exhibit a high degree of similarity. Syntenic alignments are useful in comparing genomic DNA from related species andin identifying conservedgen es. In this paper, we present a parallel algorithm for computing syntenic alignments that runs in $O(mn/p)$ time and $O(m + n/p)$ memory per processor, where m and n are the respective lengths of the two genomic sequences. Our algorithm is time optimal with respect to the corresponding sequential algorithm and can use $O(n/log\ n)$ processors, where n is the length of the larger sequence. Using an implementation of this parallel algorithm, we report the alignment of human chromosome 12p13 andit s syntenic region in mouse chromosome 6 (both over 220, 000 base pairs in length) in under 24 minutes on a 64-processor IBM xSeries cluster.

Disciplines
Biomedical | Computational Biology | Electrical and Computer Engineering | Genetics and Genomics | Theory and Algorithms

Comments
This is a manuscript of a proceeding published as Futamura N., Aluru S., Huang X. (2002) Parallel Syntenic Alignments. From the 9th International Conference on High-Performance Computing, Bangalore, India, December 18-21, 2002. doi: 10.1007/3-540-36265-7_40. Posted with permission.

Parallel Syntenic Alignments

Natsuhiko Futamura, Srinivas Aluru*and Xiaoqiu Huang†
Department of Electrical and Computer Engineering
Department of Computer Science
Iowa State University
email: {nfutamur, aluru, xqhuang}@iastate.edu

Abstract

Given two genomic DNA sequences, the syntenic alignment problem is to compute an ordered list of subsequences for each sequence such that the corresponding subsequence pairs exhibit a high degree of similarity. Syntenic alignments are useful in comparing genomic DNA from related species and in identifying conserved genes. In this paper, we present a parallel algorithm for computing syntenic alignments that runs in $O\left(\frac{mn}{p}\right)$ time, where m and n are the respective lengths of the two genomic sequences. Our algorithm is time optimal with respect to the corresponding sequential algorithm and can use $O\left(\frac{n}{\log n}\right)$ processors, where n is the length of the larger sequence. The space requirement of the algorithm is $O\left(m + \frac{n}{p}\right)$ per processor. Using an implementation of this parallel algorithm, we report the alignment of human chromosome $12p13$ and its syntenic region in mouse chromosome 6 (both over $220,000$ base pairs in length) in under 24 minutes on a 64-processor IBM xSeries cluster.

1 Introduction

Sequence alignments are fundamental to many applications in computational biology, and comprise one of the best studied and well understood problem areas in this discipline. Much of the early pioneering work concentrated on two types of alignments − 1) global alignments, which are intended for comparing two sequences that are entirely similar [7, 14, 15], and 2) local alignments, which are intended for comparing sequences that have locally similar regions [10, 18]. In general, these problems can be solved in time proportional to the product of the

*Research supported by NSF Career under CCR-0096288 and NSF EIA-0130861.
†Research supported by NIH Grants R01 HG01502-05 and R01 HG01676-05 from NHGRI.

lengths of the sequences and in space proportional to the sum of the lengths of the sequences. Research has also been conducted in developing parallel algorithms for solving global and local alignment problems [1, 5, 6, 9, 13].

It is widely recognized that evolutionary processes tend to conserve genes. Along a chromosome, genes are interspersed by large regions known as 'junk DNA'. A gene itself is comprised of alternating regions known as *exons* and *introns*, and the introns are intervening regions that do not participate in the translation of a gene to its corresponding protein. Homologous DNA sequences from related organisms, such as the human and the mouse, are usually similar over the exon regions but different over other regions. Because the different regions are much longer than similar regions, conserved sequences cannot be identified through global alignment. This results in the problem of aligning two sequences where an ordered list of subsequences of one sequence is highly similar to a corresponding ordered list of subsequences from the other sequence. We refer to this problem as the *syntenic alignment* problem. This is an important computational problem in the emerging field of comparative genomics.

A number of fast comparison algorithms have been developed for comparing syntenic genomic sequences [3, 4, 12, 16]. Generally, these methods perform fast identification of significant local similarities, and narrow further consideration to such regions. Because of this, such methods tend to work well on sequences with highly similar regions. Recently, Huang [11] developed a dynamic programming based solution to the syntenic alignment problem. This method guarantees finding an optimal solution and is capable of detecting weak similarities. However, the run-time of this scheme is quadratic, making it difficult to apply over long sequences.

In this paper, we present a parallel syntenic alignment algorithm based on the sequential dynamic programming solution developed by Huang [11]. The algorithm runs in $O\left(\frac{mn}{p}\right)$ time and $O\left(m + \frac{n}{p}\right)$ space, where m and n are the lengths of the two genomic sequences ($m \leq n$). The algorithm is time optimal with respect to the sequential algorithm and can use up to $O\left(\frac{n}{\log n}\right)$ processors. We implemented the parallel algorithm using C and MPI, and demonstrate its scalability using an IBM xSeries cluster. Using this software, we report the alignment of human chromosome $12p13$ and its syntenic region in mouse chromosome 6 (both over 220kb in length) in 23.32 minutes using 64-processors.

The rest of the paper is organized as follows: In Section 2, we describe the syntenic alignment problem and describe a dynamic programming solution for solving it. In Section 3, we present our parallel algorithm. Experimental results are presented in Section 4. Section 5 concludes the paper.

2 Problem Formulation

An alignment of two sequences $S = s_1 s_2 \ldots s_k$ and $T = t_1 t_2 \ldots t_l$ over an alphabet Σ is obtained by inserting gaps in chosen positions and stacking the sequences such that each character in a sequence is either matched with a character in the other sequence or a gap. The quality of an alignment is computed as follows: A scoring function $f : \Sigma \times \Sigma \rightarrow I\!R$ specifies the score for matching a character in one sequence with a character in the other sequence. Gaps are penalized by using an affine gap penalty function that charges a penalty of $h + gr$ for a sequence of r maximal gaps. Here, h is referred to as gap opening penalty and g is referred to as gap continuation penalty. An optimal alignment of S and T is an alignment resulting in the maximum possible score over all possible alignments. Let $score(S, T)$ denote the score of an optimal alignment. In line with the tradition in molecular biology, we use *sequence* to mean string and *subsequence* to mean substring.

Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two sequences. A subsequence A' of A is said to precede another subsequence A'' of A, written $A' \prec A''$, if the last character of A' occurs strictly before the first character of A'' in A. An ordered list of subsequences of A, $(A_1, A_2, \ldots, A_k)$ is called a chain if $A_1 \prec A_2 \prec \ldots A_k$. The syntenic alignment problem for sequences A and B is to find a chain $(A_1, A_2, \ldots, A_k)$ of subsequences in A and a chain $(B_1, B_2, \ldots, B_k)$ of subsequences in B such that the score

$$\left\{ \sum_{i=1}^{k} score(A_i, B_i) \right\} - (k-1)d$$

is maximized (see Figure 1).

The parameter d is a large penalty aimed at preventing alignment of short subsequences which occur by chance and not because of any biological significance. Intuitively, we are in-

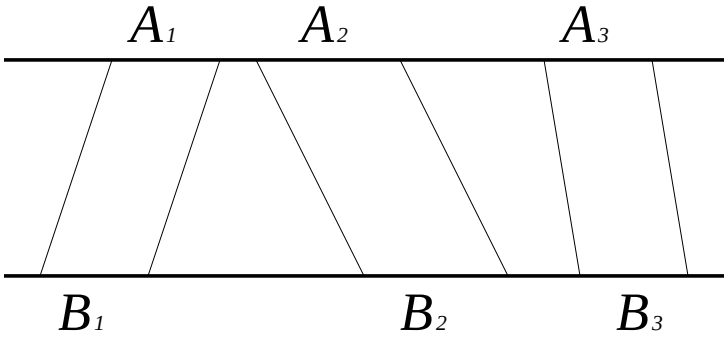

Figure 1: An illustration of the syntenic alignment problem.

terested in finding an ordered list of matching subsequence pairs that correspond to conserved exons. One can think of unmatched subsequence pairs that lie between consecutive matched subsequences, i.e. the subsequence between A_i and A_{i+1} and the subsequence between B_i and B_{i+1}. The penalty d can be viewed as corresponding to an unmatched subsequence pair. For a small alphabet size, given a character in an unmatched subsequence, there is a high probability of finding the same character in the corresponding unmatched subsequence. In the absence of the penalty d, using these two characters as another matched subsequence pair would increase the score of the syntenic alignment. The penalty d serves to avoid declaring such irrelevant matching subsequences as part of the syntenic alignment, and its value should be chosen carefully.

Based on the problem definition, the syntenic alignment of two sequences $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$ can be computed by dynamic programming. Basically, we compute the syntenic alignment between every prefix of A and every prefix of B. We compute 4 tables C, D, I and H of size $(m+1) \times (n+1)$. Entry $[i, j]$ in each table corresponds to the optimal score of a syntenic alignment between $a_1 a_2 \dots a_i$ and $b_1 b_2 \dots b_j$, subject to the following conditions:

- In C, a_i is matched with b_j.
- In D, a_i is matched with a gap.
- In I, gap is matched with b_j.
- In H, either a_i or b_j is part of an unmatched subsequence.

It follows from these definitions that the tables can be computed using the following recurrence equations:

$$C[i,j] = f(a_i, b_j) + \max \begin{cases} C[i-1, j-1] \\ D[i-1, j-1] \\ I[i-1, j-1] \\ H[i-1, j-1] \end{cases} \quad (1)$$

$$D[i,j] = \max \begin{cases} C[i-1, j] - (g+h) \\ D[i-1, j] - g \\ I[i-1, j] - (g+h) \\ H[i-1, j] - (g+h) \end{cases} \quad (2)$$

$$I[i,j] = \max \begin{cases} C[i, j-1] - (g+h) \\ D[i, j-1] - (g+h) \\ I[i, j-1] - g \\ H[i, j-1] - (g+h) \end{cases} \quad (3)$$

$$H[i,j] \;=\; \max \begin{cases} C[i-1,j]-d \\ I[i-1,j]-d \\ C[i,j-1]-d \\ D[i,j-1]-d \\ H[i-1,j] \\ H[i,j-1] \end{cases} \tag{4}$$

Prior to computation, the top row and left column of each table should be initialized. These initial values can be directly computed. After computing the tables, the optimal score of a syntenic alignment is given by the maximum score in $C[m,n]$, $D[m,n]$, $I[m,n]$, or $H[m,n]$. Thus, the problem can be solved in $O(mn)$ time and space. If we draw links from each table entry to an entry which gives the maximum value in equation (1), (2), (3) or (4), the optimal syntenic alignment can be retrieved by tracing backward in the tables starting from the largest $[m,n]$ entry and ending at $C[0,0]$. Using the now standard technique of space-saving, introduced originally by Hirschberg [8], the space required can be reduced to $O(m+n)$, while increasing the run-time by at most a factor of 2.

3 Parallel Syntenic Alignment Algorithm

Let p denote the number of processors, with id's ranging from 0 to $p-1$. Without loss of generality, assume that $m \leq n$. We compute the four tables C, D, I and H together in parallel. We use a columnwise decomposition to partition the tables to the processors. For simplicity, assume m and n are multiples of p. Processor i receives columns $i\frac{n}{p}+1, \ldots, (i+1)\frac{n}{p}$ of each table, and is responsible for computing the table entries allocated to it. The tables are computed one row at a time, in the order C, D, H and I.

Consider computing the i^{th} row of the tables. The recurrence relation for D uses entries from the already computed $(i-1)^{th}$ row and in the same column. These are readily available on the same processor. In computing C, entries that are in the previous row and previous column are needed. These are available on the same processor, except in the case of the first column assigned to each processor. After computing the $(i-1)^{th}$ row, each processor sends the last entry it computed in each of the four tables to the next processor. This is sufficient to compute the next row of C, and requires communicating just four entries per processor irrespective of the problem size. Next, we compute the i^{th} row of H. Let

$$v[j] = max(C[i-1,j]-d,\ I[i-1,j]-d,\ C[i,j-1]-d,\ D[i,j-1]-d,\ H[i-1,j])$$

Because the i^{th} rows of C and D are already computed, the vector v can be computed directly in parallel using the information available within each processor. Then, $H[i,j]$ can

be written as

$$H[i,j] = \max \begin{cases} v[j] \\ H[i,j-1] \end{cases}$$

It is easy to see that the computation of $H[i,j]$ can be done using the *parallel prefix*[1] operation with 'max' as the binary associative operator.

Now, let us turn to the computation of the i^{th} row of table I. Let

$$w[j] = \max \begin{cases} C[i,j-1] - (g+h) \\ D[i,j-1] - (g+h) \\ H[i,j-1] - (g+h) \end{cases} \tag{5}$$

Then,

$$I[i,j] = \max \begin{cases} w[j] \\ I[i,j-1] - g \end{cases} \tag{6}$$

Let

$$\begin{aligned} x[j] &= I[i,j] + gj \\ &= \max \begin{cases} w[j] + gj \\ I[i,j-1] + gj - g \end{cases} \\ &= \max \begin{cases} w[j] + gj \\ I[i,j-1] + g(j-1) \end{cases} \\ &= \max \begin{cases} w[j] + gj \\ x[j-1] \end{cases} \end{aligned}$$

Let

$$z[j] = w[j] + gj \tag{7}$$

Then,

$$x[j] = \max \begin{cases} z[j] \\ x[j-1] \end{cases} \tag{8}$$

Since the $z[j]$'s can be easily computed from the i^{th} row of C, D, and H, $x[j]$'s can be computed using parallel prefix with 'max' as the binary associative operator. In turn, $I[i,j]$ can be computed from $x[j]$ by simply subtracting gj from it.

[1]Given $x_1, x_2, \ldots, x_n$ and a binary associative operator $\otimes$, parallel prefix is the problem of computing $s_1, s_2, \ldots, s_n$, where $s_i = x_1 \otimes x_2 \otimes \ldots \otimes x_i$ (or equivalently, $s_i = s_{i-1} \otimes x_i$). This is a well-known primitive operation in parallel computing, and is readily available on most parallel computers. For example, the function MPI_Scan computes parallel prefix.

As mentioned before, processor i is responsible for computing columns $i\frac{n}{p}+1$ through $(i+1)\frac{n}{p}$ of the tables C, D, I and H. Distribution of sequence B is trivial because b_j is needed only in computing column j. Therefore, processor i is given $b_{i\frac{n}{p}+1} \ldots b_{(i+1)\frac{n}{p}}$. Each a_i is needed by all the processors at the same time when row i is being computed. Sequence A is stored in each processor. It remains to be described how the traceback procedure is performed in parallel to retrieve the optimal syntenic alignment. However, we defer this as the scheme presented so far cannot be used directly due to the unreasonably large amount of memory required.

Run-time and Space Analysis: Each processor computes $\frac{n}{p}$ entries per row of each of the four tables. The run-time is dominated by parallel prefix, which takes $O\left(\frac{n}{p}+\log p\right)$ time. To achieve optimal $O\left(\frac{n}{p}\right)$ run-time, the number of processors used should be $O\left(\frac{n}{\log n}\right)$. To enable using as large a number of processors as possible, and more importantly because practical efficiencies are better when the problem size per processor is large, we choose the larger sequence to represent the columns of the table (i.e., $n \geq m$). The parallel run-time for computing all the tables is $O\left(\frac{mn}{p}\right)$, optimal with respect to the sequential algorithm. The space required is also $O\left(\frac{mn}{p}\right)$.

The space required by the algorithm presented can be prohibitively large for syntenic alignments. For example, consider the alignment of two sequences of length one million each, on 100 processors. Assuming each entry of the table requires two memory words (one for the value and one for the pointer), the space required per processor can be estimated as $\frac{1}{100} \times 10^6 \times 10^6 \times 2 \times 4 \approx 80GB$!

Note that complete storing of the tables is required only because of the necessity to traceback to retrieve the optimal alignment. If only the optimal score is required, we only need the entry $[m, n]$ in each of the four tables. In computing row i of the tables, only the previous row of the tables is required. Thus, one can compute the tables by keeping track of at most two rows at a time (this can be actually reduced to one row plus constant storage). This will immediately reduce the storage to $O\left(\frac{n}{p}\right)$, but it would not allow traceback.

Parallel Space-Saving Algorithm

Define p special columns C_k $(0 \leq k \leq p-1)$ of a table to be the last columns of the parts of the tables allocated to each processor, except for the last processor, i.e., $C_k = (k+1) \times \frac{n}{p}$. If the intersections of an optimal path with the special columns are identified, the problem can be split into p subproblems (see Figure 2), to be solved one per processor using the sequential

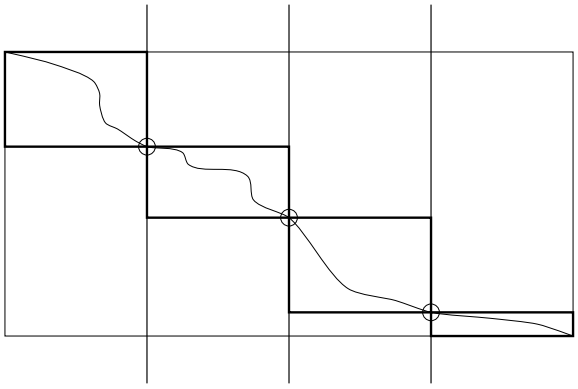

Figure 2: Problem decomposition in parallel space-saving algorithm

space-saving algorithm. The solutions of the subproblems are then concatenated to get the total alignment. Each subproblem receives exactly $\frac{1}{p}^{th}$ of sequence B but an undetermined portion of sequence A. Since the total length of the sequence A is m, each subproblem can be solved sequentially in $O\left(\frac{mn}{p}\right)$ time and $O\left(m + \frac{n}{p}\right)$ space. This memory requirement can be easily satisfied even if the input sequences span entire chromosomes. Memory permitting, multiple special columns per processor can be used, resulting in smaller subproblems and decreased overall run-time. Thus, there is a memory vs. run-time tradeoff.

It remains to be described how the intersection of an optimal path with the special columns can be computed. We only store information on the special columns of a table. In addition, we store the most recently computed row of a table in order to compute the next row using parallel prefix. This gives a space bound of $O\left(m + \frac{n}{p}\right)$. For each entry of a table, the 'value' of the entry and the $\langle$*table number*, *row number*$\rangle$ tuple of the entry in the closest special column to the left that lies on an optimal path from $C[0,0]$ to the entry are computed. Call such a tuple a *pointer* to the previous special column. This essentially gives the ability to perform a traceback through special columns, without considering other columns. The values in a row of each table are computed as before. The pointers for tables C and D can be copied from the entry in the previous rows of the four tables that is responsible for the value chosen by the max operator. For H and I, the pointer is similarly known if it results from one of the known entries but is not known if it results from the previous entry in the same row of the table being computed. Therefore it is initially set to u (undefined), unless $j-1$ is a special column. If so, $\langle H, i\rangle$ (or $\langle I, i\rangle$ when computing I) is taken to be the pointer. The undefined entries can then be filled using parallel prefix and the following

operation:

$$x \oplus y = \begin{cases} x, & y = u \\ y, & y \neq u \end{cases}$$

In fact, the parallel prefix for establishing the pointers can be avoided altogether. This is because the last column of the table allocated to each processor is a special column and the pointer value in an entry next to the special column is already known. Therefore, a sequential prefix computation within each processor is enough to determine the pointers.

A sequential traceback procedure along the special columns can be used to split the problem into p subproblems in $O(p)$ time. This does not significantly affect the run-time of $O\left(\frac{mn}{p}\right)$ provided $p^2 = O(mn)$. While this is a reasonable assumption in practice, time-optimality can be retained even if this is not true.

The idea is to parallelize the traceback procedure itself using parallel prefix. Each element on a special column contains a pointer to the element on the previous special column in one of the four tables. It is required to establish a pointer from each element on the last special column of each table to an element on every other special column in one of the four tables following the chain of pointers leading to it. Consider the special columns $C_{p-1}, C_{p-2}, \ldots, C_0$ of all the tables. To operate on the special columns on the four tables at once, special columns with the same column numbers are concatenated together and considered as an array of size $4(m+1)$, and pointer tuples $\langle \textit{table number}, \textit{row number} \rangle$ stored at each special column be adjusted accordingly. Define the operator $\oplus$ such that

$$(A \oplus B)[i] = B[A[i]]$$

where A, B and $A \oplus B$ are arrays of length $4(m+1)$ representing array of pointers on special columns. Partial sums $s_{p-1}, s_{p-2}, \ldots, s_0$, where

$$s_k = C_{p-1} \oplus C_{p-2} \oplus \ldots \oplus C_{k+1}$$

can be computed using parallel prefix. As applying this operator takes $O(m)$ time, such a parallel prefix takes $O(m \log p)$ time. As we can take $O\left(\frac{mn}{p}\right)$ time, up to $O\left(\frac{n}{\log n}\right)$ processors can be utilized.

It remains to be described how the data required for the subproblems is moved to the respective processors. Sequence B is already distributed appropriately. Distribute sequence A uniformly across all the processors. While better methods can be designed, the following suffices to prove the required time complexity. Perform p circular shift operations on sequence A such that the entire sequence passes through each processor. Each processor retains as much of sequence A as it needs. If there is sufficient memory on each processor, data

movement can be avoided by storing the entire sequence A throughout the computation, without violating our space bound of $O\left(m + \frac{n}{p}\right)$.

4 Experimental Results

We implemented the parallel syntenic alignment algorithm in C and MPI and experimentally evaluated its performance using an IBM xSeries cluster. The cluster consists of 64 Pentium processors each with a clock rate of 1.26GHZ and 512MB of main memory, connected by Myrinet, supporting peak bidirectional communication rates of 2Gb/sec. The parallel syntenic alignment algorithm consists of a problem decomposition stage, followed by a local computation stage: In the decomposition stage, the tables are computed in parallel, storing entries only on the special columns. This is followed by a traceback procedure to split the problem into p subproblems. In the local computation stage, the subproblems are solved independently on each processor. The time spent in the decomposition stage depends only on the size of the tables. Even though the time spent in the local computation stage is worst-case optimal ($O\left(\frac{mn}{p}\right)$), the actual time spent depends upon how evenly the problem splits into subproblems, which in turn depends upon the structure of the optimal alignment. In the best case where all subproblems have equal size, the run-time for the local computation stage is only $O\left(\frac{mn}{p^2}\right)$. The worst-case corresponds to a single processor receiving a problem of size $m \times \frac{n}{p}$, which translates to all conserved exons confined to $\frac{1}{p}^{th}$ of an input sequence allocated to the same processor. This situation is highly unlikely, and the actual performance is expected to be closer to the best-case.

To study the scalability of the algorithm, the program is run using sequences of the same length and varying the number of processors. Note that the communication required in computing a row depends only on the number of processors and is independent of the problem size. Thus, it is interesting to determine the smallest problem size per processor (*grain-size*) that gives good scaling results. This can be used to calculate the largest number of processors that can be beneficially used to solve a given problem. On the IBM cluster, we determined that the grain-size required for efficient parallel execution is about $500 - 1000$ per processor.

The speedups as a function of the number of processors for a syntenic alignment of two sequences of length $30,000$ are shown in Figure 3. Notice that superlinear speed up is observed in several cases. Apart from the typical beneficial effect due to better caching, this is due to the fact that increasing the number of processors causes a proportionate increase in the number of special columns, which reduces total work. Based on an approximate

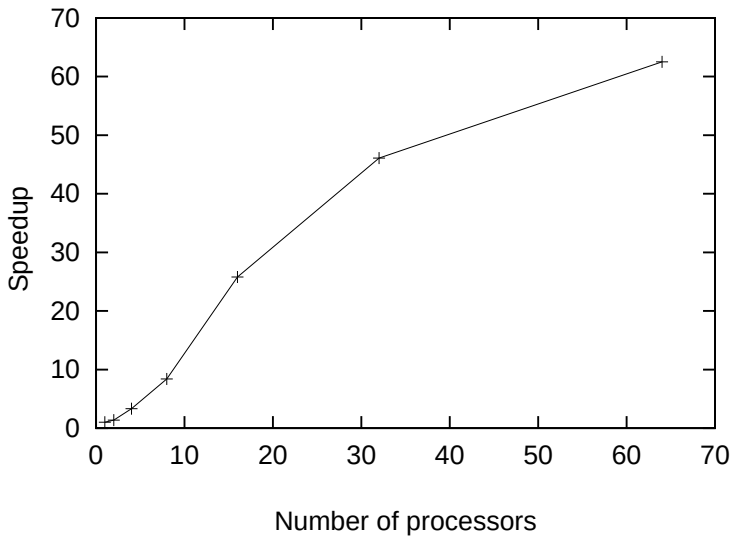

Figure 3: Speedup as a function of the number of processors for syntenic alignment of two sequences of length $30,000$.

calculation (ignoring lower order terms etc.), our parallel algorithm requires computing $6mn$ entries per table in the problem decomposition stage. Note that parallel prefix on a row of a table requires processing each table entry twice, and only two of the four tables need parallel prefix. If the subproblems are perfectly balanced, each processor computes $\frac{8mn}{p^2}$ entries in the local computation stage. Thus, the total work done by our parallel algorithm is $6mn + \frac{8mn}{p}$, including the idle time on processors (taking work to be the product of parallel run-time multiplied by the number of processors). This is a major reason for the superlinear speedup observed. Table 1 shows the total run-time and the time spent in the problem decomposition and local computation stages as the number of processors is varied. The local computation stage can scale anywhere between linearly and quadratically (run-time reduces by a factor of 4 for a twofold increase in the number of processors).

The effect of caching is the key factor in the superlinear speedup observed in the problem decomposition stage, when the number of processors is increased from 8 to 16. On 16 processors, each processor has an approximate row size of $2,000$ entries per table. We need to store 4 tables, 2 rows per table, and need 3 memory words (12 bytes) per entry. Thus, the memory required in the problem decomposition stage is 192KB per processor, which will nicely fit into the 256KB cache. On 8 processors, the rows will have to be continually swapped between cache and main memory, causing significant slowdown.

The program is used to compare two syntenic human and mouse sequences containing 17 genes [2]. The human sequence is of length $222,930$ bp (GenBank Accession U47924) and

Number of processors	Problem decomposition stage	local computation stage	Total time
1	1497	409	1906
2	1220	166	1386
4	507	67	574
8	200	28	228
16	61	13	74
32	35	6	41
64	27	3	30

Table 1: Run-time (in seconds) spent in the problem decomposition and local computation stages for $m = n = 30,000$, as the number of processors is varied.

the mouse sequence is of length 227,538 bp (GenBank Accession AC002397). The following parameters are used based on our prior experiences with standard alignment programs: match = 10; mismatch = -20; gap opening penalty, $h = 60$; gap continuation penalty, $g = 2$. A value for the parameter d was selected on the basis of internal exon lengths, often of length at least 50 bp. The score of 50 matches at 10 per match is 500. The value of 300 was used for the parameter d. The human and mouse sequences were screened for repeats with RepeatMasker [17]. The masked sequences are then used as input. The program produced a syntenic alignment of the two sequences in 23.32 minutes on 64 processors. The alignment consists of 154 ordered subsequence pairs separated by unmatched subsequences. The alignment fully displays the similar regions but omits most of the dissimilar regions. The 154 similar regions are mostly coding exon regions and untranslated regions. Gaps occur much more frequently in alignments of untranslated regions than in alignments of coding exon regions. The total length of the 154 similar regions is 43,445 bp and their average identity is 79%. The 154 similar regions constitute about 19% of each of the two sequences.

5 Conclusions

In this paper, we developed a parallel algorithm and its implementation for comparing sequences with intermittent similarities. The proposed method allows fast computation of syntenic alignments of long DNA sequences. It enables the comparison of long genomic regions with weak similarities.

References

[1] S. Aluru, N. Futamura and K. Mehrotra, Biological sequence comparison using prefix computations, *Proc. International Parallel Processing Symposium* (1999) 653-659.

[2] M.A. Ansari-Lari, J.C. Oeltjen, S. Schwartz, Z. Zhang, D.M. Muzny, J. Lu, J.H. Gorrell, A.C. Chinault, J.W. Belmont, W. Miller and R.A. Gibbs, Comparative sequence analysis of a gene-rich cluster at human chromosome 12p13 and its syntenic region in mouse chromosome 6, *Genome Research, 8* (1998) 29-40.

[3] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger and E.S. Lander, Human and mouse gene structure: comparative analysis and application to exon prediction, *Genome Research, 10* (2000) 950-958.

[4] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. While and S.L. Salzberg, Alignment of whole genomes. *Nucleic Acids Research, 27* (1999) 2369-2376.

[5] E.W. Edmiston and R.A. Wagner, Parallelization of the dynamic programming algorithm for comparison of sequences, *Proc. International Conference on Parallel Processing* (1987) 78-80.

[6] E.W. Edmiston, N.G. Core, J.H. Saltz and R.M. Smith, Parallel processing of biological sequence comparison algorithms, *International Journal of Parallel Programming, 17(3)* (1988) 259-275.

[7] O. Gotoh, An improved algorithm for matching biological sequences. *Journal of Molecular Biology, 162* (1982) 705-708.

[8] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Communications of the ACM, 18(6)* (1975) 341-343.

[9] X. Huang, A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor, *International Journal of Parallel Programming, 18(3)* (1989) 223-239.

[10] X. Huang, A space-efficient algorithm for local similarities, *Computer Applications in the Biosciences,6(4)* (1990) 373-381.

[11] X. Huang and K. Chao, A generalized global alignement algorithm, manuscript in preparation.

[12] N. Jareborg, E. Birney, and R. Durbin, Comparative analysis of noncoding regions of 77 orthologous mouse and human gene pairs, *Genome Research, 9*, (1999) 815-824.

[13] E. Lander, J.P. Mesirov and W. Taylor, Protein sequence comparison on a data parallel computer, *Proc. International Conference on Parallel Processing* (1988) 257-263.

[14] E.W. Mayers and W.Miller, Optimal alignments in linear space, *Computer Applications in the Biosciences, 4(1)* (1988) 11-17.

[15] S.B. Needleman and C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, *Journal of Molecular Biology, 48* (1970) 443-453.

[16] S. Schwartz, Z. Zhang, K. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller, PipMaker–A web server for aligning two genomic DNA sequences, *Genome Research, 10* (2000) 577-586.

[17] A. Smit and P. Green, http://ftp.genome.washington.edu/RM/RepeatMasker.html, 1999.

[18] T.F. Smith and M.S. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology, 147* (1981) 195-197.