AMO - Advanced Modeling and Optimization, Volume 12*, Number* 1*,* 2010

Parallel alignment of coding DNA

S.H. Alavi-Soltani, H. Ahrabian[1], A. Nowzari-Dalini
Center of Excellence in Biomathematics,
School of Mathematics, Statistics, and Computer Science,
University of Tehran, Tehran, Iran.
Email: {*alavi,ahrabian,nowzari*}*@ut.ac.ir.*

Abstract

We present a new parallel algorithm that computes an optimal alignment of the coding DNA sequences based on DNA/protein model proposed by Hein for the evaluating distance between two coding DNA sequence. The algorithm is proved to be adaptive and cost optimal with respect to the sequential algorithm. The parallel algorithm is implemented and experimental results show the efficiency of algorithm.

Keywords*:* Bioinformatics, Parallel algorithms, Sequence alignments.

1 Introduction

Sequence alignment is an important tool in biology for relating the molecular structure and function to the underlying sequences. In this problem, biological sequences such as DNA and protein sequences are considered as strings over a fixed alphabet of characters [Jones et al., 2004]. Sequence alignment on the biological sequences are known as an alignment. Dynamic programming has become the method of choice for "regions" aligned of DNA and protein sequences. For a number of useful alignment-scoring schemes, this method is guaranteed to produce an alignment of two given sequences with the highest possible score. Scoring is altered by considering the distances between two sequences. The scoring mechanism for two sequences can be designed in three different models: DNA model, protein model, DNA/protein model. In this paper we deal with DNA/protein model. Now we give a brief details about these three models.

A straight forward model of evolutionary distance between two coding DNA sequences is to ignore the encoding protein and compute the distance in some evolutionary model of DNA. The evolutionary distance between two sequences in a DNA level model can most often be formulated as a classical alignment problem and be efficiently computed by Dynamic programming [Jones et al., 2004; Needleman et al., 1970; Waterman, 1989].

It is usually more reliable to describe the evolutionary distance based on a alignment of the encoded proteins rather on a alignment of the coding DNA itself [Pearson, 1996].

[1]Corresponding author.

Hence, most often the evolutionary distance between two coding DNA sequences is modeled in terms of amino acid events, such as substitutions of a single amino acid and insertion-deletion of consecutive amino acids. These events cause transforming the one encoded protein into the other encoded protein. Such a model is called a protein model. The evolutionary distance between two coding DNA sequence in a protein level model can often be formulated as classical alignment problem of two encoded proteins [Pearson, 1996]. Even though a protein level model is usually more reliable than a DNA level model, it falls short because it postulates that all insertions and deletions on the underlying DNA occur of codon boundaries and it ignores similarities on the DNA level.

Hein [1994] presented a non classical model of the evolutionary distance between two coding DNA sequences in which a nucleotide event is penalized by the change it induces on the DNA as well as on the encoded protein. Their model is a natural combination of a DNA level model and a protein model and is called DNA/protein model. This model is a biological reasonable instance of the general model in which the evolution of coding DNA is idealized to involve only substitution of a single nucleotide and insertion-deletion of a multiple of three nucleotides.

Simple sequence alignment algorithms for two sequences of length n and m using DNA model or protein model, have been presented in [Jones et al., 2004; Needleman et al., 1970; Waterman, 1989], with $O(nm)$ time and space complexity. Using Hirchberg's technique [Hirchberg, 1975] developed in the context of the longest common sequence problem, Mayers and Miller [1988] presented a technique to reduce the space requirement of the sequence alignment to optimal $O(m + n)$ while retaining a time complexity of $O(nm)$. These algorithms are very important because the lengths of biological sequences can be large enough to render algorithms that use quadratic space infeasible. Hein [1994] presented an $O(n^2m^2)$ time complexity algorithm for computing the evolutionary distance in the DNA/protein model between two sequences of length n and m. Later Pedersen et al. [1998] presented an $O(nm)$ time algorithm that solves the same problem under the assumption of an affine gap cost. While space-optimal algorithms make large sequence alignment feasible, the quadratic time requirement still make it a time-consuming process. A natural approach is to reduce the time requirement with the use of parallel computers. Edmiston et al. [1988] presented parallel algorithms for sequence and subsequence alignment that achieved linear speed up and can use up to $O(min(m, n))$ processors with speed up 52 on Intel iPSC/1 hypercube which had up to 128 processors. Rajko et al. [2004] presented parallel algorithm with speed up 21 on 60-node IBM xSeries cluster for aligning two DNA sequences of length 80K. Chen et al. [2005] presented a parallel algorithm with speed up 11 on 18-node grid system for aligning two DNA sequences of length 100K. Driga et al. [2006] also presented a parallel algorithm with speed up 17.3 on 32-node SGI Origin 2400 for aligning two DNA sequences of length $319,030$ and $305,636$.

In this paper we present an efficient parallel implementation of the sequential alignment of coding DNA sequences which is proposed by Pedersen et al. [1998]. The parallel algorithm evaluates the distance between two coding DNA sequences and is based on DNA/protein model . The algorithm is both adaptive and cost optimal. It is implemented on a cluster of workstations and the experimental results show the efficiency of algorithm.

Also we archived a linear speed up with regard to the number of nodes in the cluster. We show that this approach generates high quality alignments and leads to significant runtime savings on a PC cluster.

The remaining of the paper is organized as follows. A brief discussion of the Pedersen's sequential alignment algorithm [Pedersen et al., 1998] is given in Section 2. The parallel version of this algorithm is presented in Section 3. In Section 4, the time and space complexity of the algorithm are discussed. Experimental results are given in Section 5. In Section 6, conclusion is given.

2 Sequential alignment algorithm

In this section we review the sequential alignment algorithm in DNA/protein model given by Pedersen et al. [1998]. Let $a = a_1a_2a_3 \ldots a_{3n-2}a_{3n-1}a_{3n}$ be a coding DNA sequence of length $3n$ with a reading frame starting at a_1. Let the notation $a_1^i a_2^i a_3^i$ denote the ith codon $a_{3i-2}a_{3i-1}a_{3i}$ and the notation A_i describes the amino acid coded by the ith codon. The amino acid sequence $A = A_1A_2A_3 \ldots A_n$ describes the protein coded by a.

Let $a = a_1a_2a_3 \ldots a_{3n}$ and $b = b_1b_2b_3 \ldots b_{3m}$ be two coding DNA sequences. It is desired to compute an optimal alignment of a and b in DNA/protein model. An alignment of two sequences describes a set of substitution or insertion-deletion (gap) events necessary to transform one sequence into the other sequence. These events are usually described by a matrix or a path in a alignment graph as illustrated in Figure 1. The cost of an alignment ($cost()$) is the optimal cost of any order of the events described by the alignment. Hence, the evolutionary distance in the DNA/protein model between two coding DNA sequences is the cost of an optimal alignment in the model. If the cost of any consecutive number of events is independent of the order but only depends on the set of events, then an optimal alignment can be computed efficiently using dynamic programming [Jones et al., 2004; Hirschberg, 1975; Myers et al., 1988].

The DNA level cost in the DNA/protein model is defined in the classical way by specifying a substitution cost and a gap cost in the alignment and is easy to determine as it is independent of the order of the events. The protein level cost of a nucleotide event which changes the encoded protein from A to A' should somehow reflect the difference between protein A and protein A'. Hence, the substitution cost of A by A' is defined as the minimum cost of a distance alignment of A and A' where we allow substitution of a single amino acid and insertion-deletion of consecutive amino acids. The protein level

$$\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{T} & - & - & - & \mathrm{C} \\ \mathrm{T} & - & - & - & \mathrm{C} & \mathrm{A} & \mathrm{T} & \mathrm{G} & \mathrm{C} \end{bmatrix}$$

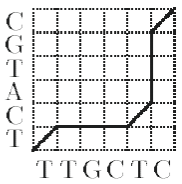

Figure 1: An alignment can be described by a matrix or path in the alignment graph.

cost in the DNA/protein model however depends on the order of events, so we cannot use a classical alignment algorithm to compute an optimal alignment in the DNA/protein model. Here it is assumed that the protein level cost of a nucleotide event only depends on the affected codons. So we decompose the alignment into codon alignments. A codon alignment is a minimal part of the alignment that corresponds to a path connecting two nodes $(3i', 3j')$ and $(3i, 3j)$ in the alignment graph. Recall from [Hein, 1994], we have eleven distinct types of codon alignments, with the assumption that an insertion or deletion has length a multiple of three as illustrated in Figure 2. Let $D^t(i,j)$, $1 \leq t \leq 11$, be the cost of an optimal alignment of $a_1a_2a_3 \ldots a_{3i}$ and $b_1b_2b_3 \ldots b_{3j}$ under assumption that the last codon alignment is of type t. Also let $D(i,j)$ be the cost of final optimal alignment of $a_1a_2a_3 \ldots a_{3i}$ and $b_1b_2b_3 \ldots b_{3j}$ and is defined as follows: If $i \leq 0$ or $j \leq 0$, then $D(i,j)$ is assumed to be infinity, otherwise $D(i,j)$, defined as:

$$D(i,j) = \min\{D^t(i,j) | t = 1, 2, \ldots, 11\}.$$

In the rest of this paper it is assumed that the amino acid substitution cost is h_q and amino acid gap cost of length k is $g_q(k)$ and nucleotide substitution cost is h_d and cost of inserting or deleting $3k$ consecutive nucleotides is $g_d(3k)$. The combined gap cost function $g(k) = g_d(3k) + g_q(k)$ is briefed as $\alpha + \beta k$ for some $\alpha, \beta \geq 0$, where starting a gap costs α and each additional symbol in the gap costs β. The cost $h_q^*(\sigma_1\sigma_2\sigma_3, \tau_1\tau_2\tau_3)$ where

$$h_q^* : \{A, C, G, T\}^3 \times \{A, C, G, T\}^3 \to R$$

is the minimum DNA level cost(nucleotide substitution cost) plus protein level cost of three substitutions $\sigma_1 \to \tau_1$, $\sigma_2 \to \tau_2$ and $\sigma_3 \to \tau_3$. These assumptions make it possible to compute $D^t(i,j)$ in constant time if $D(k,\ell)$ has been computed for all $(k,\ell) < (i,j)$ (we say that $(i',j') < (i,j)$ iff $i' \leq i \wedge j' \leq j \wedge (i' \neq i \vee j' \neq j)$). This implies that we can compute $D(n,m)$ in $O(nm)$.

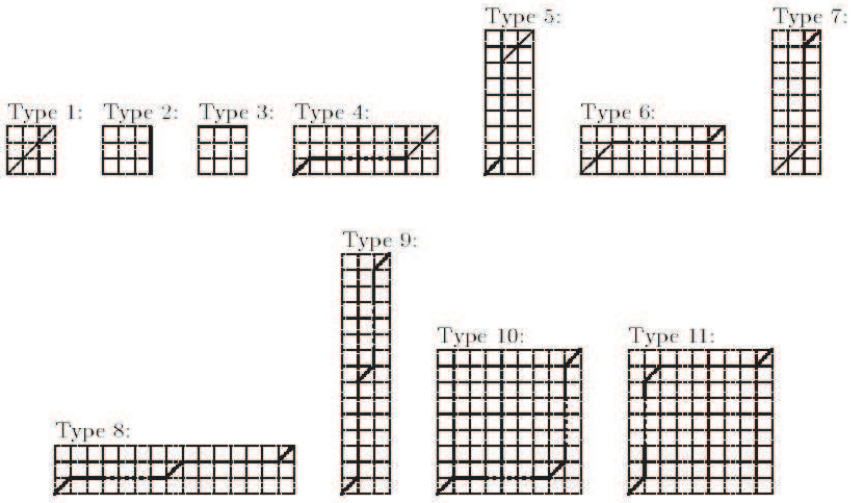

Figure 2: The eleven types of codon alignments.

For example the cost of $D^6(i,j)$ is computed by minimizing the cost of the last codon alignment plus the cost of the remaining alignment over all possible last codon alignment of type 6. This is done by minimizing the sum $cost(subs) + cost(del) + D(i-k-1, j-1)$ over all possible combinations of deletions of length k and remaining codons $x_1x_2x_3$. A combination of deletion length k and remaining codon $x_1x_2x_3$ is possible if $x_1 \in \{a_1^{i'}, b_1^j\}$, $x_2 \in \{a_2^{i'}, b_2^j\}$ and $x_3 \in \{a_3^i, b_3^j\}$ where $i' = i - k$. So we compute $D^6(i,j)$ as:

$$
\begin{aligned}
len^6_{x_1x_2}(i,j,k) &= D(i-k-1, j-1) + \\
&\qquad h_q^*(a_{3(i-k)-2}a_{3(i-k)-1}a_{3(i-k)}, x_1x_2a_{3(i-k)}) + \alpha + \beta k, \\
L^6_{x_1x_2}(i,j) &= \min\{len^6_{x_1x_2}(i,j,1), L^6_{x_1x_2}(i-1,j) + \beta\}, \\
F^6_{x_1x_2x_3}(i,j) &= \min\{L^6_{x_1x_2}(i-1,j) + \beta + \alpha q + \\
&\qquad h_q(a_{3(i-1)-2}a_{3(i-1)-1}a_{3(i-1)}, x_1x_2x_3), len^6_{x_1x_2}(i,j,1) + \\
&\qquad h_q(x_1x_2a_{3(i-1)}, x_1x_2x_3), F^6_{x_1x_2x_3}(i-1,j) + \beta\}, \\
D^6_{x_1x_2x_3}(i,j) &= \min\{L^6_{x_1x_2}(i,j) + h_q(a_{3i-2}a_{3i-1}x_3, x_1x_2x_3), F^6_{x_1x_2x_3}(i,j)\}, \\
D^6(i,j) &= \min_{x_1x_2x_3}\{h_q^*(a_{3i-2}a_{3i-1}a_{3i}, a_{3i-2}a_{3i-1}x_3) + \\
&\qquad h_q^*(x_1x_2x_3, b_{3i-2}b_{3i-1}b_{3i}) + D^6_{x_1x_2x_3}(i,j)\}.
\end{aligned}
$$

Where tables $len^6_{x_1x_2}(i,j,k)$ are corresponding to 16 combinations of x_1x_2 such that $len^6_{x_1x_2}(i,j,k)$ are the cost of remaining alignment plus the part of the cost of the last codon alignment that does not depend on the codon $a_1^ia_2^ia_3^i$ and the witness. The witness encodes the amino acid aligned with the remaining amino acid. Tables $L^6_{x_1x_2}(i,j)$ are equal to $\min\{len^6_{x_1x_2}(i,j,k) \mid 0 < k < i\}$. $D^6_{x_1x_2x_3}(i,j)$ is the minimum cost of the terms that depends on both the deletion length and the remaining codon under the assumption that the remaining codon is $x_1x_2x_3$. Tables $F^6_{x_1x_2x_3}(i,j)$ are corresponding to 64 combinations of $x_1x_2x_3$ that if $x_1x_2x_3$ is possible remaining codon and the end-codon $a_1^ia_2^ix_3$ is not the witness of $D^6_{x_1x_2x_3}(i,j)$, then $F^6_{x_1x_2x_3}(i,j)$ is equal to $D^6_{x_1x_2x_3}(i,j)$.

This algorithm is implemented and named *Combat* in [Pedersen et al., 1998]. The algorithm uses 400 table entries per step, each table of size $O(mn)$ (for more details see [Pedersen et al., 1998]). The tables can be filled together row by row, column by column, or diagonal by diagonal (where a diagonal represents all entries (i,j) of the table such that $i+j$ is a constant). Either way, when computing an entry of the table, the entries required in computing are already known. The tables are typically filled using a row by row scan, required $O(mn)$ time. So algorithm has $O(mn)$ time and space complexity.

3 The parallel alignment of coding DNA

The sequential *Combat* has quadratic time complexity and real time increases dramatically with the increase in size of the sequences. In order to alleviate this problem, we have developed a parallel version of the *Combat* algorithm in a cluster of workstations, subsequently referred to as Parallel *Combat* algorithm. Our algorithm employs the previous techniques given in the parallelization of dynamic programming algorithms [Driga et al., 2006; Rajko et al., 2004; Chen et al., 2005].

Parallel *Combat* algorithm consists of two subtasks:

1. The first task is parallelization of the computation of tables entries in order to calculate best distance.

2. Second task is the calculation of the actual alignment.

For the first task, we define a *phase* as the computational step during which nodes of the cluster calculate the sub-matrices in parallel at the same time. Also we define a *block* as a sub-matrices which is calculated by a node of the cluster during one phase. Let p be the number of processors with id's ranging from 1 to p, and for simplicity, we assume m and n are multiples of p. The processor i is responsible for computing the rows $(i-1)\frac{n}{p}+1$ through $i\frac{n}{p}$ of the tables. Distribution of the sequence a is trivial because a_i is needed only in computing row i. Therefore, the subsequence $a_{(i-1)\frac{n}{p}+1} \ldots a_{i\frac{n}{p}}$ is assigned to processor i. Each b_j is employed by all the processors at the same time when the column j is being computed. So the tables are logically partitioned in $p \times c$ equally sized blocks, with $c \geq 1$. For example, in Figure 3, the steps of the computation are illustrated for $p = 8$ and $c = 15$.

The parallelization of the calculation of tables entries is based on the wavefront communication pattern [Driga et al., 2006]. Most of the parallel algorithms for sequence alignment are designed so far to fill the dynamic programming table diagonal by diagonal that is called wavefront. This is because all the entries required for computing a diagonal depends on only the previous two diagonals, facilitating concurrent computation. Such an algorithm results in optimal time complexity $O(\frac{mn}{p})$.

In Figure 3, each diagonal of blocks labeled with the same number forms a wavefront line. At any moment during the parallel processing of the tables, a processor is either idle or is working on only one block. Furthermore, only one processor can work on a block. Once the processing of a block ends, no processor will work on that block again.

Figure 4 displays the dependency relationship: each block (i, j) of the matrices is computed from the blocks $(i-1, j)$, $(i, j-1)$, $(i-1, j-1)$. The wavefront moves in anti-diagonals as depicted in Figure 4a, i.e. the wavefront is from north-west to south-east.

	C = 15														
P1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P3	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
P4	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P5	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P6	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P7	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
P8	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Figure 3: Computational steps.

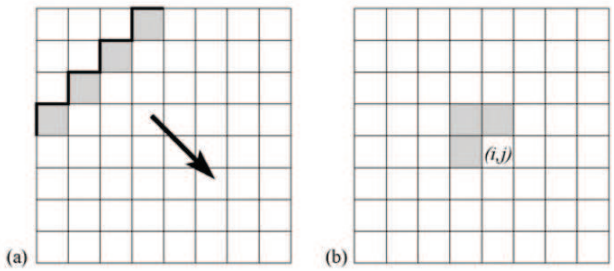

Figure 4: Wavefront computation of the similarity matrix: (a) shift direction,(b) dependency relationship.

From step p to step c, all the p processors can work in parallel because the wavefront line consists of exactly p blocks. The parallel computation ends when all the $p \times c$ blocks have been computed. When the parallel phase ends, all the tables entries are computed and are available in memory of processors.

For second task, calculation of the actual alignment as in the sequential version of the *Combat* algorithm, trace back procedure follows an optimal path which extends from the bottom-right corner of the tables to its left or upper boundary. This can be accommodated without significantly affecting the parallel runtime provided.

The algorithm shown in Figure 5, illustrates all the above discussion. This algorithm is executed independently by each processor. In this parallel algorithm, p is the number of available processors and *rank* is the processor's id ranging from 0 to $p-1$. a and b are two given sequence objects of length n and m respectively. The block size is demonstrated by *blocksize*, and the end of current block is dented by *endBlock*. The sequence a is divided among each processes, such that the length of the subsequence in each process is equal to $\frac{n}{p}$ and is stored in *lenSubSeq*.

In the *ParallelCombat* shown in Figure 5, *Alignment* is a class composed of the matrices employed in our algorithm. The *align* is an object of *Alignment* class in which the result of alignment is stored. The variable i shows the current step in the parallel wavefront. The function *init_align* initializes the matrices in the object *align*. We have two types of communications in MPI, blocking and non blocking [Snir, 1995]. The functions *sendtoN* and *receive* are used for blocking communications and $Ireceive$ a non blocking reception. MPI_Wait stops processes until the non blocking receive, receives needed information and *isRecv* stores status of current non blocking receive that demonstrated by the *level*. The function *Compute_Sequential_D* uses sequential *Combat* for computing sub matrix (block) D and *ComputeTraceBack* is the function that computes trace back on all nodes for obtaining the final result.

4 Time and space complexity

As shown in Figure 3, the computation of the blocks advances following a diagonal wavefront pattern. In Figure 3 each diagonal of blocks labeled with the same number forms

```
ParallelCombat( a , b )
//    input: sequence a and b
//    output: optimal distance alignment and the score of alignment in align
{
    int i, endBlock, lenSubSeq, level
    int isRecv[2]
    Alignment align

    init_align( a, b, align )
    isRecv[0] = 0
    isRecv[1] = 0
    level = 1
    i = 0
    lenSubSeq = n / P
    if ( rank == P - 1 )
        lenSubSeq = n - lenSubSeq * ( P - 1 )

    if(rank > 0){
        receive()
        if( m > (2 * blocksize) - 1){
            Ireceive()
            isRecv[0] = 1
        }
    }
    while( i <= m ){
        if(rank > 0)
            if( m >= i + (3 * blocksize) - 1){
                Ireceive()
                isRecv[ level ] = 1
            }
        endBlock = i + blocksize - 1
        if((endBlock + blocksize) > m){
            endBlock = m
            i = endBlock + 1
        }
        align = Compute_Sequential_D( a, b, lenSubSeq, endBlock, i )
        if(rank < P - 1)
            sendtoN()
        if(rank > 0 &&  isRecv[ 1 - level ] > 0)
            MPI_Waitall()
        isRecv[ 1 - level ] = 0
        level = 1 - level
        i = i + blocksize
    }
    align = ComputeTraceBack( align )
    return align
}
```

Figure 5: *ParallelCombat* algorithm.

a wavefront line. A wavefront line is important because the blocks that form it are independent and can be computed in parallel.

Theorem 1 *The* ParallelCombat *algorithm is cost optimal and adaptive.*

Proof. The computation time of the blocks can be divided into three distinct phases. Figure 6 shows these three phases corresponding to a computing subproblem which is solved on $p = 8$ processors. If T_b is sum of the calculation time per block plus the cost of communicating one processor to next processor then we have:

$$T_b = O(\frac{m \times n}{p \times c}) + (s + (e \times \ell)),$$

where the communication time per block is $(s + (e \times \ell))$, and s is the time required to initiate the communication, and the transfer time per (typically four-byte) word is e,

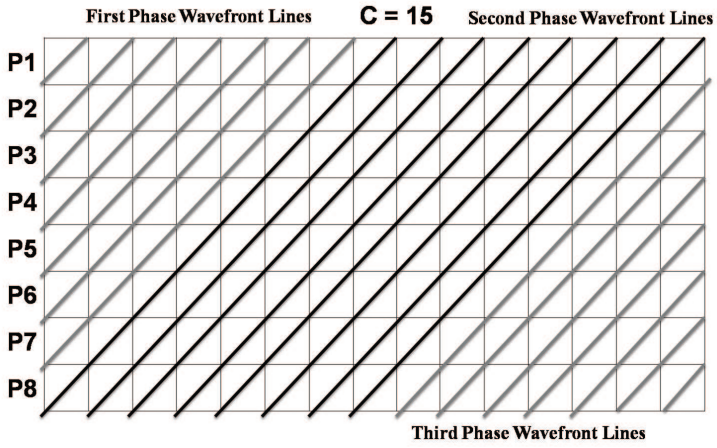

Figure 6: The three phases of a parallel computing subproblem.

which is determined by the physical bandwidth of the communication channel linking the source and destination processors and ℓ is the length of the largest message. This message is equal to the information required for computing the next block in the next processor that is computed by current processor and it sends it to the next processor. In the first phase the number of blocks in each wavefront line increases from 1 to $p-1$. In this phase a total of $\frac{p(p-1)}{2}$ blocks are computed. In the worst-case scenario each wavefront line is solved in a parallel stage that lasts a time of T_b; thus, the time spent on the first phase is at most $(p-1)T_b$. The third phase consists of the wavefront lines that are formed from less than p blocks and that are not computed in the first phase. An example of wavefront lines forming a third phase is depicted in Figure 6. The third phase has at most the same number of wavefront lines as the first phase, i.e., $p-1$. Because each wavefront line can be solved in a parallel stage of time T_b , the third phase cannot last longer than $(p-1)T_b$. The second phase is the true parallel phase. Enough blocks are available so that all the processors can work in parallel. An upper bound for the number of blocks computed in this phase is the total number of blocks, minus the lower bound for the number of blocks computed in the first phase and the lower bound for the number of blocks computed in the third phase, i.e.,

$$(p \times c) - \frac{p(p-1)}{2} - \frac{p(p-1)}{2} = p \times c - p^2 + p.$$

Because these blocks are computed in parallel, the time spent in the second phase is

$$\frac{(p \times c - p^2 + p)}{p} \times T_b.$$

So the upper bound of parallel computing tables is:

$$\begin{aligned} T_p &= (p-1)T_b + (c-p+1)T_b + (p-1)T_b \\ &= (c+p-1)T_b = (c+p-1)(\tfrac{m \times n}{p \times c} + (s + (e \times \ell))), \\ &= \tfrac{m \times n}{p}(1 + \tfrac{p-1}{c}) + (s + (e \times \ell))(c+p-1). \end{aligned}$$

If c was big enough, then evaluation of the tables requires $O(\frac{m \times n}{p})$ time. Trace back runs in $O(m + n)$, therefore the total time is $O(\frac{m \times n}{p})$. Therefore the cost is equal to $O(m \times n)$, which comparing to the complexity of the sequential algorithm, this cost is optimal. Obviously the space required on each processor is $O(\frac{m \times n}{p})$. Since the number of processors is independent of n and m, therefore the algorithm is adaptive. ■

5 Experimental results for ParallelCombat

For investigating the efficiency of our parallel algorithm the speed up of the algorithm is evaluated, which is defined as follows:

$$S_p = \frac{t_s}{t_p},$$

where t_s is the execution time of the sequential algorithm and t_p is the execution time of the parallel algorithm with p processors. Obviously, algorithms with S_p close to p are more efficient algorithms.

Our algorithm *ParallelCombat* is implemented on the cluster of University of Tehran, department of computer science (eight nodes: four Dual-Core AMD Opteron(tm) Processor 2 GHz). The *ParallelCombat* is coded in *C* using LAM/MPI.

For testing the algorithm, we implement our algorithm on two different data sets. These data sets are extracted from NCBI [Pruitt et al., 2000]. The first data set is composed of two sequences: NM_009791 that is 9363 bp long and XM_422197 that is 9078 bp long. The results on these sequences are reported in Tables 1 and Figure 7. These results show the speed up of *ParallelCombat* for $c = 3$ and $0 \leq p \leq 8$. As we can see the speed up increases in a linear manner, and its value is close to the number of the processors.

The second data set is built from CDS of DNA sequences obtained from NCBI and named by their ACCESSION which is refereed in [Pruitt et al., 2000]. Table 2 shows the speed up of *ParallelCombat* for $c = 3$ and $p = 1, p = 4$ and $p = 8$ where S_4 and S_8 denotes speed up obtained by 4 and 8 processors respectively. As we see in Table 2 and Figure 8 when the data size is larger, the increase in speed up appears to be better.

Table 1: Execution time and speed up of *ParallelCombat*.

#Processor	Execution time	Speed up
1	2644.118	1
2	1335.176	1.98
3	892.80	2.96
4	668.83	3.95
5	548.02	4.82
6	459.19	5.76
7	395.60	6.68
8	348.02	7.60

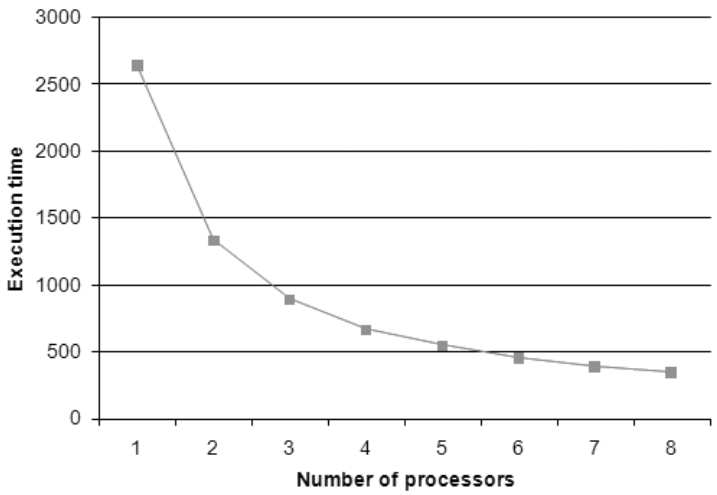

Figure 7: *ParallelCombat* Execution time.

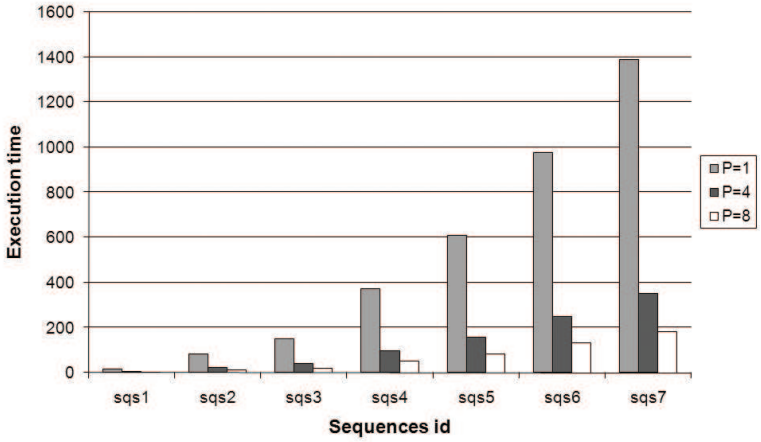

Figure 8: Comparison of the overall execution time.

Table 2: Comparison of the overall execution time.

Id.	Sequences	length (bp)	p=1	p=4	S_4	p=8	S_8
sqs_1	AY370879, XM_87157	735, 648	14.7	4.1	3.6	3.1	4.7
sqs_2	XM_547400, XM_001161722	1701, 1557	82.1	21.7	3.8	13.3	6.2
sqs_3	XM_421412, XM_538386	2424, 2112	147.8	38.2	3.9	22.4	6.2
sqs_4	XM_516767, HUMPP2A130	3474, 3453	369.7	95.5	3.9	52.9	7
sqs_5	XM_001139169, NM_000014	4431, 4425	609.1	156.3	3.9	84.4	7.2
sqs_6	XM_687036, AB028985	5901, 5316	975.9	248.2	3.9	132.5	7.4
sqs_7	XM_537788, XM_534893	7188, 6111	1389.0	348.7	4	184.5	7.5

6 Conclusion

The exponential growth of genomic databases demands even more parallel and distributed solutions for different biological problems in the future. In this paper we have presented a parallel algorithm for an existing sequential algorithm for aligned of the coding DNA sequences. The result of this algorithm on two different biological data sets extracted from NCBI, shows the efficiency of our algorithm. The measure for efficiency of our algorithm is justified by evaluating the speed up of our parallel algorithm. It is shows that the value of the speed up for this algorithm is nearly close to the number of processors. As an alternative to special-purpose systems, and expensive supercomputers, we advocate the use of cluster architecture.

Acknowledgements

This research was partially supported by University of Tehran.

REFERENCES

Chen, C., and Schmidt, B. (2005) An adaptive grid implementation of DNA sequence alignment. *Future Generation Computer Systems,* Vol. **21**, pp.988–1003.

Driga, A., Lu, P., Schaeffer, J., Szafron, D., Charter, K., and Parsons, I. (2006) FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. *Algorithmica,* Vol. **45**, pp.337-375.

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

Hein, J. (1994) An algorithm combining DNA and protein alignment. *Journal of Theoretical Biology,* Vol. **167**, pp.169–174.

Hirschberg, D.S. (1975) A linear space algorithm for computing longest common subsequences. *Communication of the ACM,* Vol. **18**, pp.341–343.

Jones, N.C., and Pevzner, P.A. (2004) *An Introduction to Bioinformatics Algorithms.* MIT Press, Cambridge.

Myers, E., and Miller, W. (1988) Optimal alignments in linear space. *Computer Applications in the Biosciences,* Vol. **4**, pp.11–17.

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

Pearson, W.R. (1996) Effective protein sequence comparison. *Methods in Enzymology,* Vol. **266**, pp.227–258.

Pedersen, C., Lyngso, R., and Hein, J. (1998) Comparision of coding DNA. *Lecture Notes in Computer Science,* Vol. **1448**, pp.153–173.

Pruitt, K.D., Katz, K.S., Sicotte, H., and Maglott, D.R. (2000) Introducing RefSeq and LocusLink: curated human genome resources at the NCBI. *Trends in genetics,* Vol. **16**, pp.44–47.

Rajko, S., and Aluru, S. (2004) Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems,* Vol. **15**, pp.1070–1081.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1995) *MPI: The Complete Reference.* MIT Press, Cambridge.

Waterman, M.S. (1989) *Mahematical methods for DNA sequences.* CRC Press, Boca Raton.