

FastPCMSA: An Improved Parallel Algorithm for the Constrained Multiple Sequence Alignment Problem

Dan He and Abdullah N. Arslan
Department of Computer Science
University of Vermont
Burlington, VT 05405, USA
e-mail: {dhe, arslan}@cs.uvm.edu

Abstract

The *constrained multiple sequence alignment (CMSA)* problem is to align given sequences S_1, S_2, \dots, S_n to maximize a similarity score with the constraint that P is “contained” in the resulting alignment. The *CMSA* problem can be considered as a constrained path search problem in the dynamic programming matrix. The problem has a dynamic programming solution that requires $O(2^n |S_1| |S_2| \dots |S_n| |P|)$ time where we denote by $|X|$ the length of any string X . There is a parallel *CMSA* algorithm that uses $|P| + 1$ processors. We propose a more general parallel algorithm which further improves the time requirement of the problem in practical settings.

Keywords: constrained sequence alignment, multiple alignment, dynamic programming, parallel algorithm.

1 Introduction

The *constrained multiple sequence alignment (CMSA)* problem was introduced by Tang et al. [12]. The problem aims to incorporate the biologically meaningful prior knowledge of the structure or pattern of the input sequences into the alignment process. The problem is to find an optimal multiple alignment of given n strings S_1, S_2, \dots, S_n such that the alignment contains a given pattern string P , i.e. in the alignment matrix there exists a sequence c of columns each entirely composed of symbol $P[k]$ for every k where $P[k]$ is the k th symbol in P , $1 \leq k \leq |P|$, and in the sequence c , a column containing $P[i]$ appears before column containing $P[j]$ for all i, j , $i < j$. An application of the problem is the alignment of RNase sequences. Such sequences are all known to contain three active residues

His(H), Lys(K), His(H) that are essential for *RNA* degrading. Therefore, it is natural to expect that in an alignment of *RNA* sequences, each of these residues should be aligned in the same column. The *CMSA* problem when $k = 2$ is called the *constrained pairwise sequence alignment (CPSA) problem*.

There are many dynamic programming algorithms for the *CMSA* and *CPSA* problems, and their variations [12, 3, 13, 14, 1, 4, 7, 8].

In this paper, we propose a more general parallel algorithm that further parallelizes the parallel *CMSA* algorithm *PCMSA* of He and Arslan [8]. Experimental evidence shows that our algorithm improves the results obtained by Algorithm *PCMSA*.

The outline of this paper is as follows: In Section 2 we summarize the parallel *CMSA* algorithm *PCMSA* of He and Arslan [8]. We present our parallel algorithm *FastPCMSA* for the problem in Section 2.1 and show the results of our experiments in Section 3. We include our final remarks in Section 4.

2 Parallel Computation of CMSA

Given n sequences S_1, S_2, \dots, S_n with lengths, respectively, s_1, s_2, \dots, s_n , the *multiple sequence alignment (MSA)* problem can be considered as a path problem in a graph whose vertices are placed at entries in an n -dimensional matrix of size $s_1 \times s_2 \times \dots \times s_n$, and to each vertex v there are vertices from each of its neighbors, i.e. vertices whose coordinates are either the same or one less than that of v in each dimension. In this graph the shortest path (or the longest path depending on the similarity model) between vertices $(0, 0, \dots, 0)$ and (s_1, s_2, \dots, s_n) is the optimal solution of the *MSA* problem. Clearly the problem has a simple dynamic programming solution

that involves an n -dimensional matrix [15].

For the *CMSA* problem Chin et. al [3] presents a dynamic programming formulation obtained by modifying the solution for the *MSA* problem.

The *CMSA* problem can also be considered as a problem of finding a shortest path in the dynamic programming matrix which we can visualize in layers indexed by its last dimension (positions in the pattern string P) where each layer is an n -dimensional matrix. We can see that a shortest path goes through each layer of the dynamic programming matrix beginning at layer 0 and ending at layer r , where r is the length of the pattern string. We call an optimal solution of the *CMSA* problem as a *global shortest path*. A global shortest path enters each layer at a vertex (we call it an *entry vertex*), after traversing a number of vertices in each layer exits the layer at a vertex (we call it an *exit vertex*) never to come back to this layer again. An exit vertex of layer k is also the entry vertex of layer $k + 1$ for $0 \leq k < r$. The length of a global shortest path is the sum of the length of the sub-paths on each layer, and each sub-path on layer k in a global shortest path is a shortest path between the entry and exit vertices on layer k .

He and Arslan [8] present the following parallel *CMSA* algorithm, *PCMSA*:

1. Find all entry and exit vertices for each layer k , $0 \leq k \leq r$.
2. Compute shortest paths for all entry-exit vertex-pairs on each layer in parallel.
3. Compute a global shortest path between vertices $(0, 0, \dots, 0, 0)$ and $(s_1, s_2, \dots, s_n, r)$.

In Step 1 of Algorithm *PCMSA* we use part of the *CMSA* algorithm of He and Arslan [7] to find rectangular boundary necessary to consider for each layer k in the dynamic programming matrix (see Figure 1). The execution of these steps determines at each layer k a rectangular region whose two diagonal corners respectively, are $(S_{1begin}[k], S_{2begin}[k], \dots, S_{nbegin}[k])$, and $(S_{1last}[k], S_{2last}[k], \dots, S_{nlast}[k])$. Then we find entry and exit vertices on each layer. The entry vertices on layer k are all the vertices where the symbol is $P[k]$ for all sequences at these coordinates. These vertices are in the overlapping region of layer $k - 1$ (for $k \geq 1$) and k . Similarly, the exit vertices on layer k (for $k < r$) are all the vertices where the symbol is $P[k + 1]$ for all sequences at these coordinates. These vertices are in the overlapping region of layer k and $k + 1$. Note that the entry vertices on layer k are also the exit vertices on layer $k - 1$, and the exit vertices on layer k are also the entry vertices on layer $k + 1$.

Steps from Algorithm *FastCMSA* [7]

```

2. For each  $k$ , find every pair of
   first and last possible positions
   that match  $P[k]$  in each of  $S_1, S_2, \dots, S_n$ 
   in a constrained alignment:

   for  $t = 1$  to  $n$  do
     for  $k = 0$  to  $r - 1$  do
       set  $S_{first}[t][k]$  = the first position  $f$ 
         in  $S_t$  such that  $P[1..(k + 1)]$  is
         a subsequence of  $S_t[1..f]$ 
       set  $S_{last}[t][k]$  = the last position  $l$ 
         in  $S_t$  such that  $P[(k + 1)..r]$  is
         a subsequence of  $S_t[l..r]$ 

3. For each  $k$ , find start-end point-pairs
    $(S_{1begin}[k], S_{1last}[k]), (S_{2begin}[k], S_{2last}[k]),$ 
    $\dots, (S_{nbegin}[k], S_{nlast}[k])$ :

   for  $k=0$  to  $r$  do
     if  $(k == 0)$ {
        $S_{1begin}[0] = 0;$ 
        $S_{2begin}[0] = 0;$ 
       .....

        $S_{nbegin}[0] = 0;$ 
     } else {
        $S_{1begin}[k] = S_{first}[1][k - 1] + 1;$ 
        $S_{2begin}[k] = S_{first}[2][k - 1] + 1;$ 
       .....

        $S_{nbegin}[k] = S_{first}[n][k - 1] + 1;$ 
     }
     if  $(k == r)$ {
        $S_{1last}[k] = r;$ 
        $S_{2last}[k] = r;$ 
       .....

        $S_{nlast}[k] = r;$ 
     } else{
        $S_{1last}[k] = S_{last}[1][k] + 1;$ 
        $S_{2last}[k] = S_{last}[2][k] + 1;$ 
       .....

        $S_{nlast}[k] = S_{last}[n][k] + 1;$ 
     }

```

Figure 1: Steps of Algorithm *FastCMSA* of He and Arslan [7] that compute a boundary at each layer where an optimal path can possibly pass through.

Layer 0 has only one entry vertex, $(0, 0, \dots, 0, 0)$, and layer r has only one exit vertex $(s_1, s_2, \dots, s_n, r)$.

In Step 2 of *PCMSA*, for each layer except for the first and the last, we use the *bidirectional-method-based A** algorithm to compute all pairs shortest paths among the entry vertices and exit vertices, on each layer in parallel.

The *A** algorithm [6] is a very popular heuristic search algorithm which is the extension of Dijkstra's single source shortest path algorithm [5]. It uses a heuristic estimator for the distance from each vertex in the graph to the destination. The score for each vertex is the sum of the heuristic value and the actual distance from the source to the vertex. The algorithm always expands the vertex with the minimum score. In most practical cases, the *A** algorithm is very efficient.

The *bidirectional algorithm* [2] applies the Dijkstra algorithm simultaneously from both the source s and the destination e . The search of the Dijkstra algorithm terminates if the forward and backward explorations meet. Then the shortest distance is obtained by picking a point f in the forward exploration points set p_s and a point b in the backward exploration points set p_e such that f and b have direct link and $e(f, b) + p_s(f) + p_e(b)$ is minimized, where $e(f, b)$ is the distance of the edge between f and b , $p_s(f)$ is the shortest distance from s to f , and $p_e(b)$ is the shortest distance from e to b . A shortest path can be obtained by combining the $s - f$ shortest path, edge (f, b) and the $e - b$ shortest path.

Ordinarily, the bidirectional method and *A** algorithm for the shortest path problem are used for the case when there is a single source and a single destination. It has been believed that both algorithms are only suitable for this case of the problem. Shibuya [11] was the first to apply the *A** algorithm to the $n \times m$ shortest paths problem. Shibuya [11] presents the *bidirectional-method-based A** algorithm.

Shibuya's algorithm uses the following heuristic function:

$$h(v) = \min_i h^*(v, t_i)$$

where $h^*(v, t_i)$ is the heuristic estimator from vertex v to destination t_i , and $h(v)$ is the smallest estimator to the set of destinations.

We can compute this heuristic function by the *backward Dijkstra algorithm* which is a variation of the ordinary Dijkstra algorithm, and it has the same time complexity. The backward Dijkstra algorithm is similar to the ordinary Dijkstra algorithm except that the scores $p(v)$ for all vertices v in the destination set T are initialized to 0. The order of the points to be expanded is decided by the distances of

those points to the destination set T , while in the ordinary Dijkstra algorithm the order is decided by the distance to the single destination. In the backward algorithm the heuristic estimator $h(v)$ is the actual distance from the vertex v to the destination set T , which is the distance from v to its closest destination.

After we compute the heuristic estimators for all source vertices, we can use the *A** algorithm directly to compute shortest paths to each destination vertex. Since the heuristic estimator for each vertex is the same in the search of every pair of source and destination, we do not need to recompute these estimators thus the heuristic search would be very efficient. Shibuya [11] reports that the bidirectional-method-based *A** algorithm is much faster than the Dijkstra algorithm for $n \times m$ shortest paths problem on real data.

When computing the heuristic estimators for each vertex, instead of the backward Dijkstra algorithm, we use a backward dynamic programming algorithm on the alignment matrix such that the heuristic value for each vertex is still the shortest distance from this vertex to its closest exit vertex. We initialize the scores for all the exit vertices on each layer in the dynamic programming matrix to be 0. We use the dynamic programming algorithm to compute the score of the dynamic programming matrix for each layer.

In general the numbers of entry and exit vertices on each layer are much smaller than the total number of vertices on that layer. Chin et al. [3] show that the average number of occurrences of pattern string is small. For the *RNase* sequences that are used in the experiments by He and Arslan [7], the sum of the numbers of the entry and exit vertices on each layer is always between 1/10,000th and 1/100,000th of the total number of vertices on that layer.

In addition, for each entry vertex we only need to find the shortest paths to the exit vertices whose coordinates on all dimensions are larger than those of the entry vertex since we picture the dynamic programming matrix as an acyclic directed graph. Because of the structure of this graph, on each layer the shortest paths problem for multiple sources and destinations is a much simpler problem compared to all pairs shortest paths problem on general graphs. Therefore, we can expect that this problem can be solved more efficiently.

In Step 3 of *PCMSA*, after we compute all possible shortest paths on each layer, we find a global shortest path between vertices $(0, 0, \dots, 0, 0)$ and $(s_1, s_2, \dots, s_n, r)$ by selecting one shortest path connecting an entry and an exit vertex on each layer such that the sum of the shortest paths from all layers is minimized. A global shortest path is the combina-

tion of these shortest paths on each layer. In this final step of the algorithm we can use a single-source shortest paths algorithm.

He and Arslan [8] present experimental evidence suggesting that *PCMSA* takes time $O(2^n s_1 s_2 \dots s_n)$ in practice indicating a factor of $\Omega(r)$ improvement over a naive sequential *CMSA* algorithm implementing the dynamic programming solution of Chin et. al [3].

2.1 FastPCMSA Algorithm

In Algorithm *PCMSA* of He and Arslan [8], although we can do the computations on different layers independently and in parallel, the computation time on each layer is still high ($O(2^n s_1 s_2 \dots s_n)$). In Algorithm *PCMSA* the number of processors we can use is dependent on the number of layers, namely the lengths of the pattern string, and this can only eliminate the factor r in the total time complexity $O(2^n s_1 s_2 \dots s_n r)$ if we use $r + 1$ processors. If we can parallelize the computation at each layer, further speed-up is possible.

The computation for each layer consists of two parts: the backward dynamic programming computation and the bidirectional-method-based A^* search. We parallelize both. We develop a new parallel algorithm *FastPCMSA* by parallelizing steps of *PCMSA* as we describe below.

The backward dynamic programming algorithm may take a long time if the required part of the dynamic programming matrix does not fit in the cache of the processor and the computation often needs to switch between the cache and the main memory. We can use the method Martins et. al [10] proposed. The method proposes a multithreaded parallel approach for the dynamic programming algorithm on pairwise sequence alignment problem. First, the dynamic programming matrix is divided into rectangular blocks and the computation is done in an anti-diagonal way such that the computation for each block along the anti-diagonal, namely each block in the wavefront, can be done in parallel. This is shown in Figure 2. The same idea can be extended to the *MSA* problem, only with the difference that we are considering a hyper-rectangular region in multiple dimensions instead of a rectangle in two dimensions. Therefore, we can begin to compute a hyper-rectangular block only if all the neighbor blocks have been computed and all the necessary information, namely the values of the vertices on the boundaries between this block and each of its neighbors, is sent to this processor.

After the processor receives all necessary information, it can begin the computation of the cor-

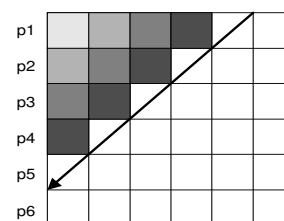


Figure 2: Parallelizing the dynamic programming computation of pairwise alignment. Computation for each block along the anti-diagonal can be done in parallel. Here each row of blocks is assigned to one processor P_i . The computation begins with the start block (the top left block).

responding block of dynamic programming matrix. The way we divide the dynamic programming matrix into blocks is decided by the size of the cache for each processor. The reason is that if all the computations can be done in the cache and the computation does not need to switch between the cache and the main memory then the dynamic programming can be performed very fast (In this case, the matrix size in each layer is bounded from above by the cache-size. This will not be the case for large instances of the problem).

After we finish the computation for one block, we can send that block to the main memory such that this computed block of dynamic programming matrix can be used in the bidirectional-method-based A^* search. The details of this parallelization are shown in Figure 3. For the bidirectional-method-based A^* algorithm, all the shortest paths can be computed independently using the dynamic programming matrix of the current layer. We can pre-calculate the positions for all the candidate entry vertices and the candidate exit vertices and the exit vertex set for each entry vertex (i.e. set of exit vertices reachable from each entry vertex) in the dynamic programming matrix. Then we can parallelize the bidirectional-method-based A^* search by assigning these entry vertices to different processors and let each processor compute the shortest paths from each entry vertex assigned to it to all the exit vertices in their corresponding exit vertex set.

In order to balance the computation time for each processor, we try to assign the entry vertex with large exit vertex set, under which case it is often time consuming to compute the shortest paths, to different processors. Since the computation time for the bidirectional-method-based A^* search mainly depends on those entry vertices with large exit vertex set, and if there are many such entry vertices, the

search can be very time consuming. By parallelizing the computations for these vertices we can achieve a significant speed-up. As we show in the algorithm in Figure 4 we sort the size of exit vertex-sets into descending order (we precompute these sets) and do the processor assignments in a round-robin manner. This is how we solve the load-balancing problem here but we note that this is an interesting optimization problem on its own. If we use a shared-memory parallel machine, the computations on each processor are similar to the computations on a sequential machine. The details of this parallelization are shown in Figure 4.

```

Parallelization of the backward dynamic
programming algorithm
Let  $P_1 < P_2 < \dots < P_N$  be a sequence of  $N$ 
available processors ordered by their processor
numbers
Let each processor has a cache of size CACHE
  If  $CACHE > \text{size}(\text{dynamic programming matrix})$  {
    assign the whole dynamic programming matrix
    to processor  $P_1$ 
  }else{
    Divide the dynamic programming matrix into
    blocks such that  $\text{size}(\text{block}) = \text{CACHE}$ ;
    Assign the start block (i.e the block with
    lowest indices in all dimensions) to  $P_1$ ;
    Initialize finished block set  $F = \text{NULL}$ ;
    After  $P_1$  finishes computation, send start
    block to main memory; add start block to  $F$ ,
    While not finished all blocks {
      Find an adjacent block set  $A$  such that
      each block in  $A$  is adjacent to at least
      one block in  $F$ ;
      For  $block_i \in A$  {
        If all neighbor blocks of  $block_i$  are
        finished {
          Assign  $block_i$  to  $P_j$ , where  $P_j$  is
          the smallest numbered free processor;
          After  $P_j$  finishes computation,
          remove  $block_i$  from  $A$ ,
          send  $block_i$  to the main memory,
          add  $block_i$  to  $F$ .
        }
      }
    }
  }

```

Figure 3: Parallelization of the backward dynamic programming algorithm

3 Experiments

We have implemented and run our algorithm *FastPCMSA* on a sequential Intel Xeon 2.4GHz machine with 2GB memory. By running our algorithm on this sequential machine we aim to collect experimental evidence to help us estimate the performance of our algorithm *FastPCMSA* on an SGI Origin 2400 parallel computer which uses shared-memory architecture.

```

Parallelization for the
bidirectional-method-based  $A^*$  algorithm
Let  $P_1 < P_2 < \dots < P_N$  be a sequence of  $N$ 
available processors ordered by their processor
numbers
  For all layers  $k$ ,  $1 \leq k \leq r$ ,
  find all candidate entry vertices
   $en_i = (t_1, t_2, \dots, t_n, k)$ 
  such that  $S_1[t_1] = S_2[t_2] = \dots = S_n[t_n] = P[k]$ 
  in the overlapping region of layers  $k-1$ 
  and  $k$ 
  For all layers  $k$ ,  $0 \leq k \leq r-1$ ,
  find all candidate exit vertices
   $ex_i = (m_1, m_2, \dots, m_n, k)$ 
  such that  $S_1[m_1] = S_2[m_2] = \dots = S_n[m_n] = P[k+1]$ 
  in the overlapping region of layers  $k$ 
  and  $k+1$ 
  For each  $en_i$  {
    For each  $ex_j$  {
      If  $(\text{test}(en_i, ex_j) == \text{true})$  {
        Add  $ex_j$  to  $E_i$ , which is the exit
        vertex set for  $en_i$ ;
      }
    }
  }
  sort  $E_i$  in descending order of their sizes,
  let the resulting ordering be
   $E'_1 > E'_2 > \dots > E'_h$ 
  For  $i = 1$  to  $h$  do {
    Assign  $E'_i$  and  $en_i$  to  $P_j$ , where  $P_j$  is
    the smallest numbered free processor,
    wait if there is no free processor;
  }
boolean function  $\text{test}((x_1, x_2, \dots, x_n), (y_1, y_2, \dots, y_n))$ 
{
  if for all  $i$ ,  $y_i \geq x_i$  then return TRUE
  else return FALSE
}

```

Figure 4: Parallelization of the bidirectional-method-based A^* algorithm

In our estimates we ignore the communication cost

which is insignificant when we run our algorithm on a shared-memory architecture. We consider the longest of the steps executed in parallel whenever there is parallelism, and find a total execution time for our algorithm by adding the execution times of these steps, which is pessimistic. We strongly believe that if our algorithm is run on a shared-memory parallel architecture, the inter-processor communication costs will be insignificant because each processor will access a separate block in memory, and they do not communicate with each other directly. Each processor should be informed about the termination of neighboring processors, and if all the neighbors finish then the processor fetches the boundary information, which is small in size, into its cache from main memory, and this can be done very fast. If we assume a shared memory architecture then in the bidirectional-method-based A^* algorithm there will not be any inter-processor communication because each processor will fetch the required data directly from the shared memory.

In our tests we use the first 4 *RNase* sequences used by Chin et al. [3], and *HKSTH* as the pattern string. The 4 sequences are:

Seq1 : *gi*|119124|*sp*|p12724|*ecp_human*

Seq2 : *gi*|2500564|*sp*|p70709|*ecp_rat*

Seq3 : *gi*|13400006|*pdb*|*ldyt*

Seq4 : *gi*|20930966|*ref*|*xp_142859.1*

The Origin 2400 has 64 processors (400MHz R12000 MIPS CPUS), each with 32Kb L1 cache, 8Mb L2 cache. The main memory is 16GB. We first show the size of each layer and the numbers of entry vertices and exit vertices for each layer in Figure 5. In Algorithm *PCMSA*, layer 4 is the most time consuming layer. For layer 4, the size of the dynamic programming matrix is $62 \times 61 \times 62 \times 41 = 9,613,844$. Each vertex in the dynamic programming matrix is about 10bytes, therefore the memory requirement for the complete dynamic programming matrix is $62 \times 61 \times 62 \times 41 \times 10bytes \approx 9.61 \times 10^7b$. Then if we divide the dynamic programming matrix into blocks with size $31 \times 30 \times 31 \times 21$, the memory requirement for each block is $31 \times 30 \times 31 \times 21 \times 10bytes \approx 6.26 \times 10^6b$, which fits into the 8Mb L2 cache of those parallel processors. Then the original dynamic programming matrix for layer 4 is divided into 16 blocks. Since the number of blocks on the wavefront is at most 6, we need at most 6 processors to parallelize the backward dynamic programming computation. The computation time for one block on each processor is about 1.855 seconds, so the total time for the backward dynamic programming computation is $1.855 * 5 = 9.275$ seconds, where 5 is the number of wavefronts from the start vertex to the end vertex of

<i>layer</i>	entry vertices	exit vertices	total vertices
0	1	30	3,659,085
1	30	5	1,584,000
2	5	25	1,177,088
3	25	54	222,507
4	54	54	9,613,844
5	54	1	9,613,844

Figure 5: Numbers of entry, exit and total vertices on each layer for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of [3]. We considered the first 4 sequences and used *HKSTH* as the pattern string

the original dynamic programming matrix.

For the bidirectional-method-based A^* search, there are 54 entry vertices and 54 exit vertices. 9 entry vertices have exit vertex sets with size 54, 16 entry vertices have exit vertex sets with size 9, 9 entry vertices have exit vertex sets with size 6, 20 entry vertices have exit vertex sets with size 1. The computation for the shortest paths among those 9 entry vertices and their corresponding exit vertex sets with size 54 in Algorithm *PCMSA* takes nearly 90% of the total computation time. If we use 9 processors to parallelize the bidirectional-method-based A^* search on layer 4, the execution time can be reduced to only one third of the execution time of *PCMSA*. Therefore, if we use 9 processors to parallelize the computations on layer 4, the total execution time for this layer is only about 20.885 seconds, which is about one third of the total execution time of *PCMSA* on the same layer. Since the computation time for layer 4 is the longest among all 6 layers, the complete execution time for Step 2 is 20.885 seconds. The total execution time is the combination of the execution times for all three steps. The comparison of the time requirement of the algorithms *FCMSA*, *PCMSA*, and *FastPCMSA* are shown in Figure 6. Although we did not consider the communication cost among processors, we expect that this cost is insignificant on a shared-memory parallel machine compared with the computation cost on the same machine.

4 Conclusion

We propose a new parallel algorithm for the constrained multiple sequence alignment problem. Our algorithm further parallelizes the main step of the parallel algorithm proposed by He and Arslan [8]. We present experimental evidence on real data suggesting that our algorithm improves the time require-

FCMSA	PCMSA	FastPCMSA
79.199	56.124	21.509

Figure 6: Estimated execution times in seconds of algorithms *FCMSA*, *PCMSA* and *FastPCMSA* for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of [3]. We considered the first 4 sequences and took pattern string as *HKSTH*

ment of solving the constrained multiple sequence alignment in practical settings.

References

- [1] A. N. Arslan and Ö. Egecioglu. Algorithms for the constrained common sequence problem. *Proc. Prague Stringology Conference 2004*, (Eds. M. Simanek and J. Holub), pp. 24-32, Prague, August 2004.
- [2] D. Champeaus. Bidirectional Heuristic Search Again. *J. ACM*, vol. 30, pp.22-32, 1983.
- [3] F. Y. L. Chin, N. L. Ho, T. W. Lam, P. W. H. Wong, M. Y. Chan. A. Efficient constrained multiple sequence alignment with performance guarantee. *Proc. IEEE Computational Systems Bioinformatics (CSB 2003)*, pp. 337-346, 2003.
- [4] F. Y.L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, S. K. Kim. A simple algorithm for the constrained sequence problems. *IPL* Vol. 90, pp. 175-179, 2004.
- [5] Dijkstra, E. A note on two problems in connection with graphs. *Numerical Mathematics*, 1, 395-142, 1959.
- [6] P.Hart, N.Nilsson,and B.Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100-107, 1968.
- [7] D. He and A. N. Arslan. A fast algorithm for the Constrained Multiple Sequence Alignment problem. *Proc. 11th International Conference on Automata and Formal Languages* pp. 131-143, May 2005.
- [8] D. He and A. N. Arslan. A parallel algorithm for the Constrained Multiple Sequence Alignment problem. *Proc. Fifth IEEE Bioinformatics and Bioengineering Conference*, pp. 258-262, 2005.
- [9] Ikeda,T., and Imai, H. Fast A* algorithm for multiple sequence alignment. *Genome Informatics Workshop 94*, 90-99, 94.
- [10] W. S. Martins, Juan del Cuwillo, F. J. Useche, Kevin B. Theobald, Guang R. Gao. A Multi-threaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison. *Pacific Symposium on Biocomputing*, 311-322, 2001.
- [11] T. Shibuya. Computing the $n \times m$ Shortest Paths Efficiently. *the ACM Journal of Experimental Algorithmics*, ISSN 1084-6654, Vol. 5, No. 9, 2000.
- [12] C. Y. Tang, C. L. Lu, M. D.-T. Chang, Y.-T. Tsai, Y.-J. Sun, K.-M. Chao, J.-M. Chang, Y.-H. Chiou, C.-M. Wu, H.-T. Chang, and W.-I. Chou. Constrained multiple sequence alignment tool development and its applications to rnaase family alignment. *Proceeding of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002)*, pp. 127-137, 2002.
- [13] Y.-T. Tsai. The constrained common sequence problem. *IPL*, 88:173–176, 2003.
- [14] Y.-T. Tsai, C. L. Lu, C. T. Yu, and Y. P. Huang. MuSiC: A tool for multiple sequence alignment with constraint. *Bioinformatics*, 20(14):2309-2311, 2004.
- [15] M. S. Waterman. Introduction to computational biology. *Chapman & Hall*, 1995.