

Space-Efficient Parallel Algorithms for the Constrained Multiple Sequence Alignment Problem

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

Abstract

Given sequences S_1, S_2, \dots, S_n , and a pattern string P the constrained multiple sequence alignment problem (CMSA) is to align similar subsequences of these sequences with the constraint that the alignment “contains” P . The CMSA problem can be considered as an optimal 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 and $O(|S_1| |S_2| \dots |S_n| |P|)$ space where $|S_1|, |S_2|, \dots, |S_n|$ are the lengths of sequences S_1, S_2, \dots, S_n , and $|P|$ is the length of the pattern string, respectively. There is a parallel algorithm that uses $|P| + 1$ processors. The algorithm requires $O(|S_1| |S_2| \dots |S_n|)$ space for each processor. The memory requirement is a major bottleneck for the CMSA problem. In this paper, we propose two parallel algorithms which solve the CMSA problem and use less space on each processor than the ordinary dynamic programming algorithm and existing parallel algorithms for the problem.

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

1 Introduction

The Multiple Sequence Alignment (MSA) problem is one of the most important problems in computational biology. Detecting similarities in DNA sequences gives clues about the evolutionary relatedness of different species, and similarities in protein sequences point our similar functionality. The Constrained Multiple Sequence Alignment (CMSA) problem was introduced by Tang et al. [7]. The problem is motivated by the problem of comparing multiple sequences under the guidance of a given pattern (constraint). The problem aims to incorporate biologically meaningful prior knowledge of the

structure or pattern of the input sequences into the alignment process. These constraints can lead to more biologically meaningful alignments other than alignments with mathematically high scores. An example is the alignment of RNase sequences where each of the three residues His (H), Lys (K), His (H) should be aligned in the same column.

Given n strings S_1, S_2, \dots, S_n and pattern string P , the CMSA problem is to find an optimal multiple sequence alignment such that the alignment contains a given pattern string P . The CMSA problem when $n = 2$ is called the Constrained Pairwise Sequence Alignment (CPSA) problem.

Recently many dynamic programming algorithms for the CMSA and CPSA problems have been proposed [7, 5, 8, 9, 4, 6, 1]. The time and space complexities of these dynamic programming algorithms for the CMSA problem are $O(2^n s_1 s_2 \dots s_n r)$ and $O(s_1 s_2 \dots s_n r)$ respectively (see for example Chin et al. [5], or Tsai et al. [9]) where s_1, s_2, \dots, s_n are, respectively, the lengths of the strings S_1, S_2, \dots, S_n , and r is the length of the pattern P . Heuristic algorithms for the unconstrained multiple sequence alignment problem can also be used to approximate the solution of the CMSA problem. One idea is to progressively align the sequences into a multiple alignment by using a minimum spanning tree obtained from a pairwise distance matrix of the sequences [7, 5]. But these heuristic algorithms cannot guarantee an optimal solution for the CMSA problem, and sometimes they perform poorly.

He and Arslan [1] presented an algorithm (FCMSA) for the CMSA problem based on the dynamic programming solution devised by Chin et al. [5]. The algorithm avoids redundant computations in the original dynamic programming matrix by precomputing, using the pattern string P , regions (which are usually much smaller than the entire matrix) where an optimal alignment can possibly pass. He and Arslan [1] showed that this algorithm is much faster in practice than a naive implementation of

the algorithm of Chin et al. [5].

He and Arslan further presented a parallel algorithm (*PCMSA*) [2] for the *CMSA* problem with $r + 1$ processors such that each processor requires $O(s_1 s_2 \dots s_n)$ space. Algorithm *PCMSA* considers the *CMSA* computations in layers of an $(n + 1)$ -dimensional matrix (n sequences and the pattern) where layers are indexed by positions in P , and each layer k involves a multiple sequence alignment of S_1, S_2, \dots, S_n to contain in the alignment the symbol $P[k]$. These computations can be done independently and in parallel. It has been shown that Algorithm *PCMSA* is much faster than the sequential dynamic programming algorithm [2]. However, its $O(s_1 s_2 \dots s_n)$ space requirement is too high for comparatively long sequences or large number of sequences.

In this paper, we propose two parallel algorithms for the *CMSA* problem based on the *PCMSA* algorithm of He and Arslan [2]. Instead of the *bidirectional-method-based* A^* algorithm [3] by Shibuya used in the *PCMSA* algorithm, we use the *Euclidean-distance-based* A^* algorithm [3], which is also proposed by Shibuya and which has been shown to use less space to solve the multiple sources multiple destinations shortest paths problem. We further developed another A^* algorithm based on the features of the dynamic programming matrix. Our experiments on real *RNase* sequences suggests that our algorithms use less space than *FCMSA* algorithm and *PCMSA* algorithm with $r + 1$ processors.

The outline of this paper is as follows: In Section 2 we summarize the solutions of Chin et al. [5], and He and Arslan [1, 2] for the *CMSA* problem. In Section 3 we describe our space efficient parallel algorithms. This section is organized in three subsections. In Subsection 3.1 we describe the basic principle of our algorithms. In Subsections 3.2 and 3.3 we present our two space efficient parallel algorithms respectively. We show our experiment results in Section 4. We include our final remarks in Section 5.

2 Algorithms for the *CMSA* Problem

Given n sequences S_1, S_2, \dots, S_n with lengths, respectively, s_1, s_2, \dots, s_n , the *constrained multiple sequence alignment (CMSA)* problem seeks an alignment matrix in which 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 , $i < j$.

For the *CMSA* problem Chin et. al [5] presents a dynamic programming formulation that modifies the solution for the *MSA* problem to consider the additional string P . He and Arslan [2] visualize the dynamic pro-

gramming computations as $r + 1$ layers in a layer-by-layer manner where each layer is an n dimensional dynamic programming matrix. These layers are determined by matching positions of the symbols in pattern string P . The score for each node in the dynamic programming matrix comes only from all the neighbor nodes (nodes that have direct links to this node) on the same layer k , or on the preceding layer $k - 1$ (if it exists).

Based on the observation that if the symbol $P[k]$ is aligned with a symbol of S_i then the region before this symbol $P[k]$ in S_i can never be aligned with the region after $P[k]$ in $S_1, S_2, \dots, S_{i-1}, S_{i+1}, \dots, S_n$, He and Arslan [1] improved the naive dynamic programming algorithm of Chin et al. [5] by avoiding the redundant computations in the original dynamic programming matrix and by computing only the regions which need to be considered and which can be pre-calculated using the pattern string P . A significant speed-up is achieved by this way when the pattern string is long, or the number of sequences n is large. Although this algorithm does not improve the worst-case complexity of the *CMSA* problem (its time complexity is $O(2^n s_1 s_2 \dots s_n r)$ and its space complexity is $O(s_1 s_2 \dots s_n r)$) in practical applications this algorithm improves both the time and space requirements significantly. Experiments [1] show that for 5 *RNase* sequences, and a pattern of length 4 the algorithm is more than 60 times faster than a naive implementation of the dynamic programming solution by Chin et. al [5].

Based on the *FCMSA* algorithm, He and Arslan introduced a parallel algorithm *PCMSA* algorithm [2], which solves the *CMSA* problem with $r + 1$ processors. The *PCMSA* algorithm considers the *CMSA* computations in layers of an $(n + 1)$ -dimensional matrix (n sequences and the pattern) where layers are indexed by positions in P , and each layer k involves a multiple sequence alignment of S_1, S_2, \dots, S_n to contain in the alignment the symbol $P[k]$. The computations in each layer can be considered as finding candidate parts (sub-alignments each containing a symbol of P) of an optimal alignment that contains P when parts in all layers are combined. These computations can be done independently and in parallel. Then the problem that needs to be solved at each layer can be considered as an all pairs shortest paths problem with multiple sources and multiple destinations. Shibuya [3] introduced two algorithms for the $n \times m$ shortest paths problem. The *PCMSA* algorithm uses the *bidirectional-method-based* A^* algorithm [3] to solve the all pairs shortest paths problem. The *PCMSA* algorithm is shown to be faster than *FCMSA* algorithm [2]. But the computation for each layer still requires $O(s_1 s_2 \dots s_n)$ space, which is often too high for comparatively long sequences or large number of sequences.

3 Space Efficient Parallel Algorithms of *CM**SA*

3.1 The Basic Principle

The dynamic programming solution for the *CM**SA* problem can be considered as finding a shortest path in the dynamic programming matrix which we can visualize in layers indexed by its last dimension (the positions in the pattern string) where each layer is an n -dimensional matrix. We observe that a shortest path has to go 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 *CM**SA* 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. 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 the shortest path between the entry and exit vertices on layer k . We can actually precalculate all of the candidate entry and exit vertices on each layer [2]. Given all possible entry and exit vertices on each layer, we consider the computation of shortest paths between all entry-exit vertex-pairs.

The A^* algorithm [12] is a very popular heuristic search algorithm which is the extension of Dijkstra's single source shortest paths algorithm [10]. It in general uses much less space than ordinary search algorithms such as the dynamic programming algorithm and the *Dijkstra* algorithm.

Shibuya [3] applied the A^* algorithm to the $n \times m$ shortest paths problem. Two algorithms the *Euclidean-distance-based* A^* algorithm and the *bidirectional-method-based* A^* algorithm were proposed by Shibuya [3]. Both algorithms develop a new heuristic estimator which can be used in the search between each pair of source and destination. The Euclidean-distance-based A^* algorithm uses as the heuristic estimator of vertex u the smallest estimator of all estimators from u to each destination. It uses a k - d tree to find the smallest heuristic estimator quickly. The bidirectional-method-based A^* algorithm uses the real distance from vertex u to its closest destination t as the heuristic estimator. The heuristic estimator for each vertex is computed by one application of the backward Dijkstra search from the destination set. The latter algorithm has been used by the *PCMSA* algorithm of He and Arslan [2].

The A^* algorithm was first introduced to the multiple sequence alignment problem by Ikeda and Imai [13]. They successfully applied A^* algorithm on the dynamic programming matrix to improve the efficiency of the

shortest path search. The heuristic function used in their algorithm is the sum of the pairwise sequence alignment:

$$h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^*(v_{ij})$$

where $h_{ij}^*(v_{ij})$ represents the shortest path length from vertex v_{ij} to destination t_{ij} in the 2-dimensional matrix for S_i and S_j .

Our two space efficient parallel algorithms share the same steps as *PCMSA* algorithm by He and Aslan [2]:

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 $(s_1, s_2, \dots, s_n, r)$.

The major differences among our algorithms and Algorithm *PCMSA* are the strategies to compute shortest paths for all entry-exit vertex-pairs on each layer in Step 2, which is the most important step in all 3 steps. Step 2 dominates the time and space requirements of the whole algorithm. The Step 1 and Step 3 are exactly the same as those of Algorithm *PCMSA*.

In Step 1, we first find the boundary for each layer k in the dynamic programming matrix. This can be done by finding 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. The boundary of layer k is decided by the first possible position that matches $P[k]$ and the last possible position that matches $P[k+1]$. Then the entry vertices on layer k are all the vertices where the symbol is $P[k]$ for all sequences at these coordinates and these entry vertices are in the overlapping region of layer $k-1$ and k . The exit vertices on layer k are all the vertices where the symbol is $P[k+1]$ for all sequences at these coordinates and these exit vertices are in the overlapping region of layer k and $k+1$. 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 $k+1$. Layer 0 has only one entry vertex, $(0, 0, \dots, 0, 0)$, and layer r has only one exit vertex $(s_1, s_2, \dots, s_n, r)$. We illustrate this in Figure 1. The time and space complexities of this step are both linear.

In Step 3, after we compute all 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 combination of these shortest paths on each layer. In this final

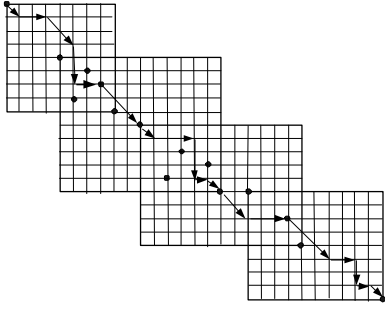


Figure 1: For the *CP*SA (*CMSA* with $n = 2$) with pattern string of length 3, a global shortest path passing through entry and exit vertices, and connecting sub-paths on each layer.

step of the algorithm we can use a single-source shortest path algorithm such as the Dijkstra algorithm. Since this global shortest path search problem is a very small-size problem in practice, ordinary search algorithm such as Dijkstra algorithm is very efficient and the execution of this step is very fast. We refer the reader to [1, 2] for details of Step 1 and Step 3.

In Step 2 of *PCMSA*, for each layer except the first and the last, the *bidirectional-method-based* A^* algorithm [3] is used to compute all pairs shortest paths among the entry vertices and exit vertices, simultaneously on each layer. Instead of backward Dijkstra algorithm, we can use the backward dynamic programming algorithm developed by He and Arslan [2] to compute the heuristic estimators of the bidirectional-method-based A^* algorithm. Therefore, this algorithm requires the same amount of space as that of the dynamic programming matrix on each layer (i.e. $O(s_1 s_2 \dots s_n)$). For the *MSA* and *CMSA* problems, space complexity is a major concern. In the *PCMSA* algorithm, even though for each layer we only need to compute the reduced regions, the space complexity $O(s_1 s_2 \dots s_n)$ may still be too high for each processor. We describe two strategies to improve the space efficiency for each processor in Subsections 3.2 and 3.3.

3.2 Algorithm *EDB-PCMSA*

The experiments of Shibuya [3] show that the *Euclidean-distance-based* A^* algorithm uses less space than the *bidirectional-method-based* A^* algorithm, although it takes longer time. Therefore, in Step 2 of *PCMSA* algorithm, we can use the *Euclidean-distance-based* A^* algorithm to compute the shortest paths for all entry-exit vertex-pairs on each layer in parallel so that the new algorithm uses less space on each processor in practical settings of the problem. We call the new algorithm *EDB-PCMSA*. The *Euclidean-distance-based* A^* algorithm uses as the heuristic estimator of vertex u (we call

the *global estimator*) the smallest estimator of all estimators from u to each destination. It uses a k-d tree to find the smallest heuristic estimator quickly. Although this algorithm requires less memory than the *bidirectional-method-based* A^* algorithm, it takes much longer time when the number of vertices in the graph is very large, as shown by the experiments of Shibuya [3]. This is because the efficiency of A^* algorithm depends on its heuristic estimator. The tighter the estimator gets, the more efficient the A^* algorithm becomes. For the *Euclidean-distance-based* A^* algorithm, when finding a shortest path to destination t , the global estimator of the vertex u , which we are going to expand next, is often smaller than the estimator from u to t since we pick only the smallest estimator of all estimators from u to each destination. Therefore, in this case, the heuristic estimator is not tight and the A^* algorithm often needs to explore much greater space. Although the algorithm saves time by avoiding the re-computation of the heuristic estimator for each vertex, it spends an impractically long time on computing global heuristic estimators for newly explored vertices and managing its open and close lists when it loads many vertices into memory, which is often the case for the *CMSA* problem. Therefore, this algorithm is only applicable to small *CMSA* problem.

3.3 Algorithm *SE-PCMSA*

We design a novel A^* algorithm to improve the space requirement on each layer and then combine this A^* algorithm with our previous parallel strategy to further improve its time complexity.

Compared with *EDB-PCMSA* algorithm, ordinary A^* algorithm from each source to each destination has a tighter heuristic estimator as introduced by Ikeda and Imai [13] and can thus improve the efficiency of search. We can use ordinary A^* algorithm directly from each exit vertex to each entry vertex. But this is very time consuming since the heuristic estimator is too expensive to compute. One strategy is to simply let all the searches from one exit vertex-set to its corresponding entry-vertex share the same heuristic estimator of each vertex. This would save time significantly since we avoid redundant computations of heuristic estimators. But the heuristic estimators cannot be shared by the computations from other exit vertex sets to their corresponding entry vertices. If the numbers of entry vertices and exit vertices are large, this strategy still takes too much time to compute the heuristic estimators. We do not only want to use the original sum of pairwise alignments as tighter heuristic estimators [13], but we also want to avoid re-computation of the heuristic estimators for the same vertex in searches between each entry and exit vertex.

Our second algorithm *SE-PCMSA* (*Space Efficient PCMSA*) is based on the observation that the heuris-

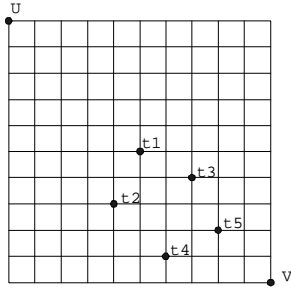


Figure 2: The pairwise alignment score between u and t_i can be obtained in the pairwise dynamic programming matrix from u to v .

tic estimator from vertex u to exit vertex v is the sum of the pairwise alignment scores. The pairwise dynamic programming matrix from u_{ij} to v_{ij} on the two dimensions indexed by S_i and S_j contains the pairwise dynamic programming matrix from u_{ij} to t_{ij} for $t_i \leq v_i$ and $t_j \leq v_j$. This means if there is an exit vertex t with $t_i \leq v_i$ and $t_j \leq v_j$, the pairwise alignment score from u_{ij} to t_{ij} can be obtained directly from the pairwise dynamic programming matrix from u_{ij} to v_{ij} , without re-computation, which is shown in Figure 2.

We do not need to recompute the pairwise alignment score from u_{ij} to t_{ij} . In the same manner, the heuristic estimators from one vertex to all the exit vertices in its exit vertex-set can be computed using the same pairwise alignment matrix which we compute only once. The time complexity of this pairwise alignment on dimensions indexed by S_i and S_j is $O(s_i s_j)$. In order to avoid the necessity for re-computing the pairwise dynamic programming matrix, this matrix beginning from vertex u should contain all exit vertices in the exit vertex set of u . For current layer k , we pick an exit vertex, as the end point of the pairwise dynamic programming matrix, which has the largest coordinates in all dimensions, namely the last possible vertex that matches $P[k+1]$ in all dimensions (we call it *local end vertex*). In Figure 1, the local end vertex on each layer is the right bottom vertex on each layer). Therefore, by computing the pairwise alignment from vertex u to the local end vertex only once, we can also obtain all the estimators from vertex u to each exit vertex in its exit vertex set. For each vertex, we then record these heuristic estimators. The corresponding heuristic estimator from vertex u to exit vertex t can be reused by all the searches to t via u . The algorithm is shown in Figure 3.

Step 2 is done in parallel on each layer. Assuming that we use $r+1$ processors, the time taken by this step is the longest time spent on any layer. The execution time of our algorithm SE-PCMSA is mainly determined by this step (more than 99% of the total time). The execution times of Step 1 and Step 3 are insignificant compared with that

Step 2 of Algorithm *SE-PCMSA* on layer k

1. For each entry vertex u on layer k , compute its exit vertex set T , namely the last possible vertex in the dynamic programming matrix that matches $P[k+1]$ in all dimensions.

2. For each exit vertex $v \in T$, use A^* algorithm to find a shortest path from u to v :

In the current search, for each vertex s the search explores

If s was not explored before {
 Compute its exit vertex set T_s . Find out the local end vertex t which has the greatest coordinates in all dimensions, namely the end vertex of the current layer.

For $i=1$ to n

For $j=i+1$ to n {

Compute the pairwise dynamic programming matrix from s_{ij} to t_{ij} in dimensions indexed by S_i and S_j using ordinary dynamic programming algorithm.

For each exit vertex $e \in T_s$, obtain the pairwise alignment score from s_{ij} to e_{ij} , which is the score of the vertex e_{ij} in the pairwise dynamic programming matrix from s_{ij} to t_{ij} .

}

Compute and record the sum of pairwise scores from s to each exit vertex $e \in T_s$.

Use the newly computed heuristic estimator from s to v in the current search.

} else

Use the heuristic estimator from s to v which was computed before in the current search.

3. Stop if found all-pair shortest paths on layer k

Figure 3: Redesign of Step 2 of Algorithm *PCMSA* [2] on layer k

Pattern string	FCMSA (sec.)	SE-PCMSA (sec.)
<i>HKSTH</i>	79.199	35.208
<i>HKASH</i>	108.717	114.399

Figure 6: The execution times in seconds of Algorithm *FCMSA* and Algorithm *SE-PCMSA* for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of [5]. We considered the first 4 sequences, and used *HKSTH* and *HKASH* as the pattern string, respectively.

Pattern string	FCMSA (MB)	SE-PCMSA (MB)
<i>HKSTH</i>	690	87
<i>HKASH</i>	820	340

Figure 7: The memory usages of Algorithm *FCMSA* (which is the same as that of Algorithm *PCMSA*), and Algorithm *SE-PCMSA* for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of [5]. We show only their greatest memory requirements on all layers. We considered the first 4 sequences, and used *HKSTH* and *HKASH* as the pattern string, respectively.

of Step 2. Our experiments show that the execution time of our algorithm is acceptable compared with the total time of sequential execution through all the layers, and it requires much less memory.

4 Experiments

In our experiments, we use the first 4 *RNase* sequences used by Chin, et al [5], and we use pattern string *HKSTH* and *HKASH*, respectively. These 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*

We compare the number of loaded vertices by Algorithm *FCMSA* [1] (the same as that of *PCMSA* algorithm [2]) with the number of vertices loaded by the Algorithm *SE-PCMSA*, on each layer except for the first and the last layer. These numbers indicate the memory usage of each algorithm. Since the problems on the first and the last layers are actually single source problems, Algorithm *SE-PCMSA* loads fewer vertices. We do not include the performance measurements for Algorithm *EDB-PCMSA* since it takes too long time in most experiments. We also show the number of entry vertices and number of exit vertices on each layer. The experiment results are shown in Figure 4 and Figure 5, respectively.

In the experiments, since the numbers of entry vertices and exit vertices are much smaller compared with the total number of vertices on each layer, Algorithm *SE-PCMSA* loads much less vertices on each layer than Algorithm *FCMSA*, and therefore Algorithm *PCMSA*.

Instead of running Algorithm *SE-PCMSA* on a parallel machine, we approximate the execution time of this algorithm by adding the execution time of each step on a sequential machine. If we use the share-memory parallel machine, where different processors share the same memory, their communications can be very efficient and the communication cost among these processors will be insignificant given that the numbers of entry and exit vertices on each layer are quite small. We calculate an estimate for the parallel execution using maximum time spent in each step. All experiments are done on a 2.4GHz Intel Xeon processor with 2GB memory and 512KB L2 cache. The experimental results are shown in Figure 6. We also show the memory usages of Algorithm *FCMSA* and Algorithm *SE-PCMSA* in Figure 7. We show only the largest memory requirement for each layer for algorithms *FCMSA* (the same as *PCMSA*), and *SE-PCMSA*.

When layer size is small or the number of entry vertices and exit vertices is comparatively small, the execution time for Algorithm *SE-PCMSA* can be much faster than Algorithm *FCMSA*. For example, in our experiments, the execution time of our algorithm *SE-PCMSA* is only one half of that of Algorithm *FCMSA*, when pattern string *HKSTH* is taken. This is because in such cases the memory usage of our algorithm *SE-PCMSA* is much smaller than that of Algorithm *FCMSA* as shown in Figure 7. Therefore, our algorithm runs very fast, while Algorithm *FCMSA* is slow since it needs to switch between the cache and the main memory quite frequently, which is very time consuming. When the pattern string is *HKASH*, both the sizes of the layers and the number of exit and entry vertices increase. Although the execution time of our *SE-PCMSA* increases sharply compared to the execution time by *FCMSA*, its memory usage is still much less than that of algorithms *FCMSA* and *PCMSA*. These experimental evidences clearly indicate that our algorithm *SE-PCMSA* improves the space requirement of the *CMMSA* problem in practical settings.

5 Concluding Remarks

We propose two space efficient parallel algorithms for the constrained multiple sequence alignment problem. Using the same idea of Algorithm *PCMSA* presented by He and Arslan [2], namely compute the layers of dynamic programming matrix simultaneously, we further develop two space efficient strategies for the computation of each

<i>layer</i>	FCMSA	SE-PCMSA	#entry vertices	#exit vertices
1	1,584,000	31,564	30	5
2	1,177,088	13,128	5	25
3	222,507	35,363	25	54
4	9,613,844	256,424	54	54

Figure 4: The numbers of vertices loaded by Algorithm *FCMSA* and Algorithm *SE-PCMSA* for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of [5]. We considered the first 4 sequences, and used *HKSTH* as the pattern.

<i>layer</i>	FCMSA	SE-PCMSA	#entry vertices	#exit vertices
1	1,584,000	31,564	30	5
2	11,414,718	349,353	5	64
3	5,913,093	760,650	64	160
4	8,677,900	617,239	160	32

Figure 5: The numbers of vertices loaded by Algorithm *FCMSA* and Algorithm *SE-PCMSA* for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of [5]. We considered the first 4 sequences, and used *HKASH* pattern string.

layer: Algorithm *EDB-PCMSA* and Algorithm *SE-PCMSA*. Our experiments show that Algorithm *SE-PCMSA* is much more space efficient than both algorithms *FCMSA* and *PCMSA* with acceptable execution time.

References

- [1] 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 (AFL 2005) (Eds. Zoltan Ésik, Zoltan Fülöp), Institute of Informatics, University of Szeged*, pp. 131-143, Dobogoko, Hungary.
- [2] D. He and A. N. Arslan. A parallel algorithm for the Constrained Multiple Sequence Alignment problem. *Proc. IEEE Bioinformatics and Bioengineering, BIBE2005*.
- [3] T. Shibuya. Computing the $n \times m$ Shortest Paths Efficiently. *the ACM Journal of Experimental Algorithmics*, ISSN 1084-6654, Vol. 5, No. 9, 2000.
- [4] 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.
- [5] 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.
- [6] F. Y.L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, S. K. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters* Vol. 90, pp. 175-179, 2004.
- [7] 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 rnase family alignment. *Proceeding of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002)*, pp. 127-137, 2002.
- [8] Y.-T. Tsai. The constrained common sequence problem. *Information Processing Letters*, 88:173-176, 2003.
- [9] 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.
- [10] E. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1, 1959, 395-142.
- [11] D. Champeaus. Bidirectional Heuristic Search Again. *J. ACM*, vol. 30, 1983, pp.22-32.
- [12] 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.
- [13] T. Ikeda, and H. Imai. Fast A* algorithm for multiple sequence alignment. *Genome Informatics Workshop 94*, 90-99.
- [14] M. S. Waterman. Introduction to computational biology. Chapman & Hall, 1995.