Application of Parallelized DP and A* Algorithm to Multiple Sequence Alignment

Shiho ARAKI‡, Masahiro GOSHIMA†,
Shin-ichiro MORI†, Hiroshi NAKASHIMA†, Shinji TOMITA†
Yutaka AKIYAMA*, Minoru KANEHISA*

‡ : NTT Network Information System Laboratories
1-2356,Take,Yokosuka-shi, Kanagawa-ken, 238–03 Japan
E-mail: shiho@sucaba.ntt.jp

† : Department of Information Science, Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606–01 Japan
E-mail: {goshima, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

∗ : Institute for Chemical Research, Kyoto University
Gokasho, Uji, Kyoto 611 Japan

Abstract

This paper makes two proposals to speed up the Parallel Iterative Method, which is based on the iterative strategy of the Berger-Munson algorithm.

The first proposal is to exploit finer-grained parallelism in the DP(Dynamic Programming) procedure itself. This proposal makes the processing speed proportional to the number of processors.

The second proposal is to apply the A^* algorithm, a well known heuristic search algorithm, instead of DP. A^* reduces the search space using heuristics, while DP traverses the whole space blindly.

We have implemented these two proposals on a parallel computer, the AP1000. In a test of parallelizing DP, ten 1000-character sequences are aligned by using 10 processors per one DP procedure at a speed 8.11 times faster than sequential processing. By applying the A^* algorithm to 30 sets of test problems, we obtain optimal alignment by reducing the search space by 95%.

荒木 志帆　現 NTT 情報通信網研究所
〒 238-03 神奈川県横須賀市武 1-2356

五島 正裕，森 眞一郎，中島 浩，富田 眞治　京都大学工学部情報工学教室
〒 606-01 京都市左京区吉田本町

秋山 泰，金久 實　京都大学化学研究所
〒 611 宇治市五ヶ庄

1 Introduction

Many computer-based methods for aligning multiple protein sequences to show common relationships have been developed. One of them is the Parallel Iterative Method[1], which is based on the iterative strategy of the Berger-Munson algorithm[2].

Figure 1 shows the Parallel Iterative Method. The method aligns $n (n \geq 2)$ protein sequences as follows;

I. The unaligned sequences (the initial state) are divided into two subgroups.

II. From $2^{n-1} - 1$ possible partitions for n sequences, a small number of partitions are selected. Between every pair of subgroups the optimal alignment is computed by one processor in parallel using group-to-group DP-matching.

III. The alignment with the best score is selected as the initial state for the next iteration.

IV. This procedure is repeated until the score of the resulting alignment converges.

If step II is executed for all the possible partitions, the degree of parallelism is $2^{n-1} - 1$. Such an implementation is impractical and we are compelled to limit the number of the sequences we can align.

One solution to this problem is "the restricted partitioning technique"[1]. This method doesn't select all possible partitions but only 1:n-1 partitions; n sequences are divided into one subgroup with n-1 sequences and the other with 1 sequence. The degree of parallelism is n.

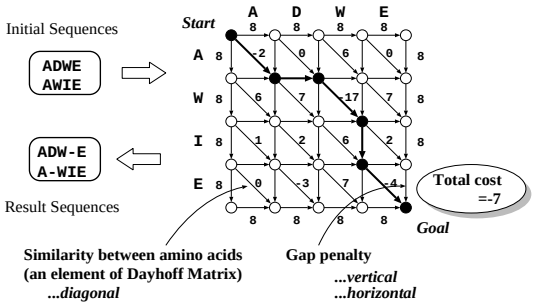

Figure 2: Alignment by DP-matching

Figure 1 describes Parallel Iterative Method with this technique. First, initial sequences are divided into 2 subgroups. There are 4 partitions such as (a)-(b)(c)(d). Secondly, the optimal alignment is computed in parallel. Of four resultant alignments, that of partition #3 is optimal and is selected as the initial state of next iteration.

2 Proposal 1: Parallelizing DP procedure

Many of the alignment algorithms are based on Dynamic Programming (DP) and usually need much execution time. We can reduce DP execution time by parallelizing it. We extract finer-grained parallelism in the DP-matching procedure itself and compute the optimal alignment between every pair of subgroup by plural processors in parallel in step II of figure 1.

2.1 DP procedure

Figure 2 shows that the alignment is obtained by finding the best path from the top left node to the bottom right node maximizing the similarity between horizontal sequences and vertical sequences.

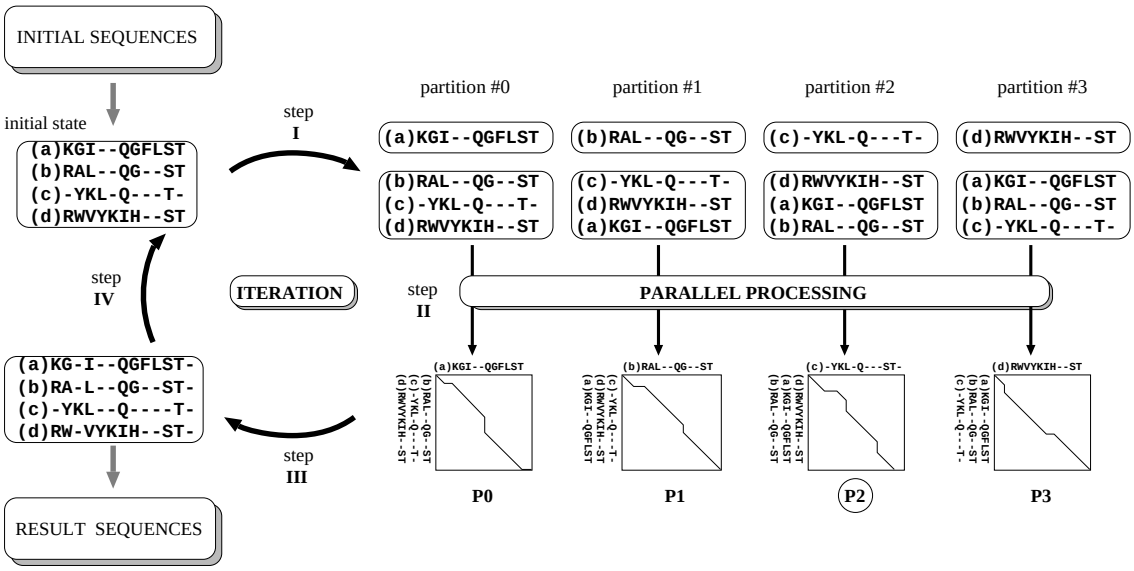

Figure 1: Parallel Iterative Method

Each diagonal arc cost reflects the similarities between the characters to be compared. We use the Dayhoff matrix [3], and reverse the sign of the elements. For example, we use '-17' for W-W while we see '17' in the Dayhoff matrix. The horizontal and vertical arc costs reflect the *gap penalty*. We use the smallest value(-8) in the Dayhoff matrix.

The sum of all costs of the arcs in the best path is the measure of the similarity of the sequences. We find the optimal alignment by finding the lowest-cost path.

To find the lowest-cost path using DP we proceed as follows; Each node has 3 nodes that reach the node by one arc. Add each of the 3 arcs (horizontal,vertical, and diagonal one) to the cost of the lowest-cost path from the start node to these nodes. Choose the lowest value as the cost of the lowest-cost path from the start node to the node.

We conduct this procedure from the top left node to the bottom right node. In sequential processing, the DP procedure creates a 2 level nested DO-ACROSS style loop.

2.2 How to parallelize DP procedure

The AP1000 is a distributed memory parallel computer. As for data partition, we adopted the *column oriented* scheme as described in figure 3 taking account of the items listed below.

target The goal is to simultaneously align ten 1000-character sequences at most in this system. But we can align theoretically about 4000-character sequences since each processor of the AP1000 has 16MB local memory[1].

overhead caused by communication Compared with the execution time

[1]For each processor l^2/p memory space is required when the sequence length is l and the number of processor is p.

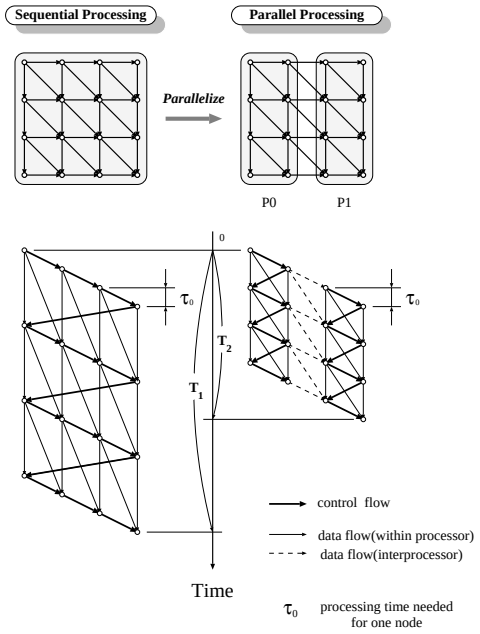

Figure 3: Parallelizing DP procedure

required at each node, we can't make light of the overhead caused by communication between processors. We should decide the data partition not to require too many communications between processors.

In parallel processing, after processor P_i finishes the computation of its first row and sends the cost of the lowest-cost path to the rightmost node to the right processor P_{i+1}, P_{i+1} can start its computation. The parallel execution is as shown in figure 3.

Implementation The AP1000 is a MIMD, distributed memory parallel computer with its processors connected by a 2-dimensional torus net[4]. We realized machine mapping as described in figure 4.

Let n be the number of sequences to be aligned and p be the number of processors per one partition and let i be integer; 0,1,2... .

X direction Computations for all partitions are executed in parallel. The degree of parallelism is n.

Y direction Processor P_{pi}, P_{pi+1}... and $P_{p(i+1)-1}$ compute the optimal alignment in parallel for $\#i$ partition.

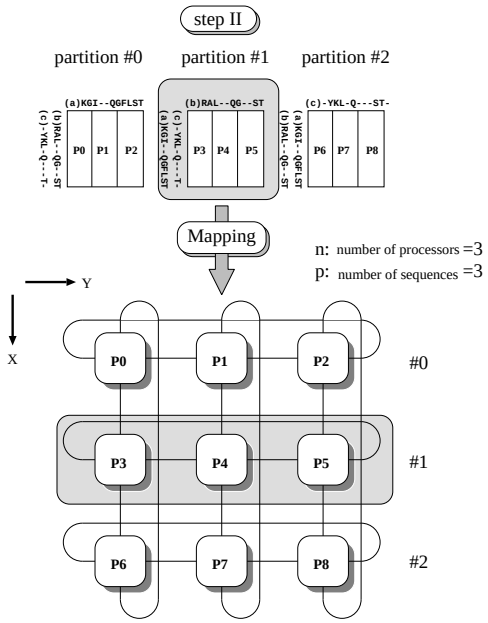

Figure 4: Implementation on AP1000

2.3 Experimental results

In order to evaluate the degree of speed enhancement of the parallelizing DP procedure, we investigated the following items in step II in figure 1.

T_p : *Execution time by parallel processing*
T_1 : *Execution time by sequential processing*
$S = T_1/T_p$: *Speed up rate*

Figure 6 indicates the speed increase achieved by parallelizing the DP procedure. It shows that we can speed up

	T_p $(sec.)$		$S = T_1/T_p$	
	$l = 1000$	$l = 200$	$l = 1000$	$l = 200$
$p = 1$	8.261	0.643	1.000	1.000
2	4.274	0.362	1.933	1.778
3	2.971	0.281	2.781	2.287
4	2.289	0.227	3.609	2.831
5	1.856	0.195	4.450	3.306
10	1.018	0.132	8.111	4.857
20	0.596	0.105	13.850	6.098
30	0.460	0.096	17.975	6.702

Figure 5: Performance of parallelizing DP

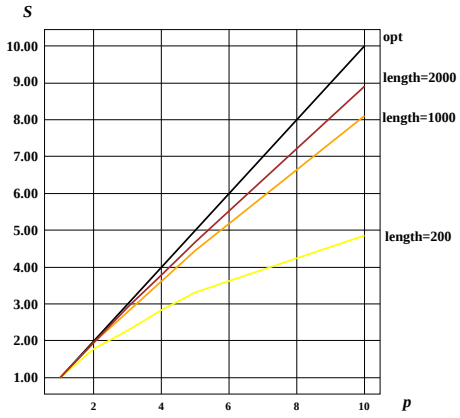

Figure 6: Speed-up

DP procedure, especially with longer sequences.

As the sequence length is longer, the overhead caused by communication between processors is relatively smaller because the granularity coarsens. We used random sequences in these experiments, since the computation cost needed for DP doesn't depend on the sequence data.

3 Proposal 2: Applying A^* algorithm instead of DP

The second proposal is to use the A^* algorithm[5] instead of the DP procedure. The A^* algorithm is a well known heuristic search algorithm. It reduces the search space using heuristics, while DP traverses the whole space blindly.

Many fast algorithms that attack the search space have been proposed. They can be classified into 2 types. One type are the algorithms which give fast, though not necessarily optimal, alignment. For example, cutting out the lower and upper triangular space can speed up the execution. The other type are the fast and optimal alignment algorithms such as the one proposed by Fickett[6].

The A^* algorithm uses a heuristic function to prune the search space and we can select a suitable heuristic function $\hat{h}$ by exploiting the characteristics of the sequences to be aligned.

The A^* algorithm is not directly applicable because both positive and negative costs exist in the search space as shown in figure 2. Our proposal is to translate the search space to make all costs positive which makes it possible to apply the A^* algorithm. Furthermore, we have devised a heuristics function which has small computation cost and high pruning efficiency. Our proposal exploits the characteristics of the general alignment problem.

3.1 A^* algorithm

The A^* algorithm adopts the evaluation function to allow the usage of heuristic information. The purpose of the evaluation function is to provide a means of ranking those nodes that are candidates for expansion and so permit us to determine which one is most likely to be on the best path to the goal. Let $\hat{f}$ be the evaluation function. Then by

$\hat{f}(n)$ we denote the value of this function at node n. We order nodes for expansion in increasing order of their $\hat{f}$ values. The node having the smallest $\hat{f}$ value is selected for expansion. The evaluation function of the A^* algorithm is given below.

$$\hat{f}(n) = g(n) + \hat{h}(n)$$

$g(n)$ represents the actual cost of the minimal cost path from the start node to node n. $h(n)$ gives the actual cost of the minimal cost path from node n to the goal and $\hat{h}(n)$ is an estimate of $h(n)$. Thus $\hat{f}(n)$ is an estimate of the cost of the minimal cost path from the start node to the goal node that is constrained to go through node n.

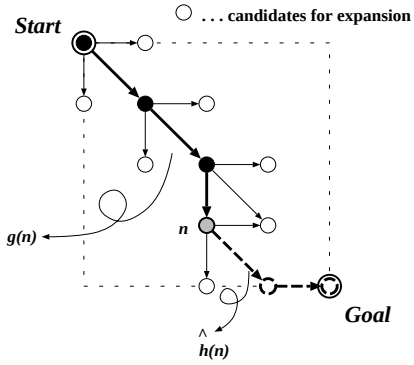

Figure 7: A^* search

Note that when $\hat{h} = 0$, algorithm A^* is identical to Dijkstra's[7]. Also note that the dynamic programming algorithms of Bellman are essentially breadth-first search methods and $\hat{h} = 0$ and $\hat{g}$ =*the number of the arcs from start point.*

We shows below the theorems known about the A^* algorithm. For further details of these, see [5].

Theorem 1 The Admissibility of A^*

A^* search algorithm is admissible if for any graph it always terminates in an optimal path to a goal whenever such a path exists.

If $\hat{h}(n) \leq h(n)$ for all n, and if all arc costs are positive, then algorithm A is admissible.*

Theorem 2 The Optimality of A*

If the A^* algorithm satisfies the conditions given above, then for any graph it never expands more nodes than any other admissible algorithm.

3.2 Transformation of the search space

In order to use the A^* algorithm instead of DP, we need to transform the search space; all arc costs should be positive to guarantee the admissibility and optimality of A*, as mentioned in Theorem 1 and 2.

Figure 2 shows the search space. The A* algorithm searches for the best path in the space from the top left node to the bottom right node minimizing the total cost of arcs.

If we add a constant "c" to them in order to make all the arc costs positive, we can't find the lowest-cost path. This is because the more arcs the path contains, the more "c"s are added to the total cost of the path. This problem can be solved as follows. *If we add a constant "2c" to every diagonal arc cost and "c" to every horizontal and vertical arc cost, we can raise the cost equally for any path.*

Brief Proof Define $n(i,j)$ as the node located at row i, column j, the start node as $n(0,0)$, and the goal node as $n(l,l)$. l is the sequence length. If we define $\mathcal{H}(i,j)$ as the number of hops

contained in any path from $n(0,0)$ to $n(i,j)$, $\mathcal{H}(i,j)$ is given below. Let d be the number of diagonal arcs contained in the path.

$$\mathcal{H}(i,j) = i + j - d$$

Assume that we count one diagonal arc as "2 hops", and *the number of hop* $\mathcal{H}(i,j)$ *is always* $i+j$ *for any path for any* i *and* j. As a result, if we add "c" per 1 hop, we can raise the cost equally for any path. Thus we have transformated the search space and made all arc costs positive. ■

3.3 Selection of the heuristic function

The selection of the heuristic function $\hat{h}$ is crucial in determining the heuristic power of the A^* algorithm. Setting $\hat{h}$ equal to the highest possible lower bound on h expands the fewest nodes consistent with maintaining admissibility.

We show below how to estimate the highest possible lower bound on h, taking account of the computation cost of $\hat{h}$. For all paths from the node $n(i,j)$ to the goal node $n(l,l)$, the paths which can give the lowest cost are those that go through $min(l-i, l-j)$ diagonal arcs and $|i-j|$ horizontal or vertical arcs showed in figure 8. That is because we set the diagonal arc cost higher than any horizontal or vertical arc cost; the total path cost is the lowest when the path goes through as many diagonal arcs as possible.

We can estimate $\hat{h}$ as follows; the path could be the lowest-cost path if it went through the lowest diagonal cost arc in every column, since we set the same cost (=gap penalty) to every horizontal and vertical arc[2].

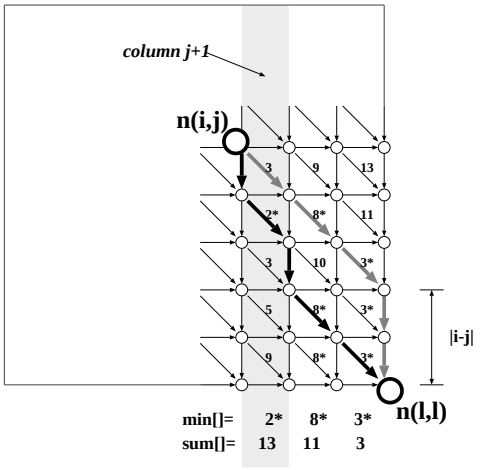

Figure 8: Estimate of h

Define $C_{i,j}$ as the cost of the diagonal arc from $n(i-1, j-1)$ to $n(i,j)$ and $min[k]$ as the lowest cost in column k ; i.e. $min[k] = \min_{m \in \{x | 1 \le x \le l\}} C_{m,k}$

Define p as the *gap penalty*, and $\hat{h}(n)$ is given below. if $i < j$,

$$\hat{h}(n) = \sum_{k=j+1}^{l} min[k] + |i-j| \times p$$

$$else\ \hat{h}(n) = \sum_{k=i+1}^{l} min[k] + |i-j| \times p$$

Computation of $\hat{\mathbf{h}}(\mathbf{n})$ Before starting the search by A^* algorithm, the computation of the *profile matrix (size: 22* $\times$ *l,* where "22" is the number of all characters each of which means an amino acid) has been finished with the computation of $22^2 \times l$ times multiplication. *Profile matrix* is needed for determining all diagonal costs in the search space. Now that *profile matrix* has been

[2]It is known that we can obtain biologically better alignment if we set the gap penalty according to the length of insertion or deletion [8]. It is possible to set the gap penalty in this way with our proposal.

computed, we can find the least cost of each column $min[k]$ by the computation $o(l)$. $Sum[j]$ is computed in advance as follows;

$$\begin{aligned} sum[l] &= min[l] \\ sum[l-1] &= min[l-1] + min[l] \\ &\vdots \\ sum[1] &= min[1] + min[2] + \cdots + min[l] \end{aligned}$$

While traversing the node $n(i,j)$ in the search space, we can compute $\hat{h}(n)$ by computing only $\hat{h} = sum[j] + |i-j| \times p$.

3.4 Experimental results

First we examined how much the search space is reduced by the A^* algorithm. Figure 9 shows that $75 \sim 95\%$ of the nodes in the search space are pruned.

We used as experimental data in these experiments a group of 6 serine protease (length=$276 \sim 613$) and a group of 14 endonuclease (length=$161 \sim 184$). Figure 9 shows the result of experiments on 30 sequence pairs selected randomly in the same group.

This figure also shows the relationship between the pruning efficiency and the similarity of the sequences. The horizontal axis of figure 9 is the average score per one column of the alignment. We obtained the average score by dividing the total score[3] by the length of the resultant alignment. The horizontal axis reflects the similarity of the alignment. We observed that the higher the similarity of the sequences is, the more nodes are pruned.

Second, we examined the execution time. Figure 10 shows the execution time required for step II (in figure 1) using sequential DP or A^*. We used as

[3] *Total score* is obtained by reversing the sign of *total cost* of resultant alignment

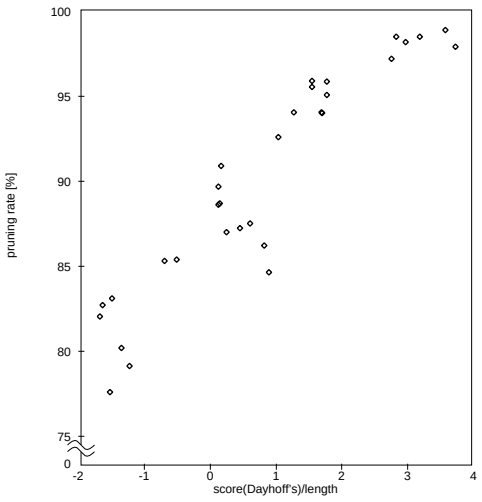

Figure 9: Pruning efficiency of A^*

experimental data in these experiments random sequences which are very similar. The average pruning rate of these data is 87.23% and the average score per one column is 0.3.

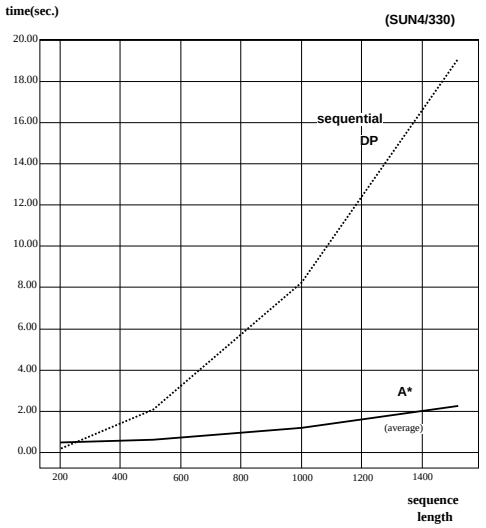

Figure 10: Execution time (DP vs. A^*)

While searching with the A^* algorithm, it is very important how to manage the *OPEN list* which is a list of candidate nodes for expansion. Whenever we add a node to the OPEN list, we

need to sort and insert in a effective manner. Therefore, we adopted *heap* as the data structure. The results given above were obtained by using *heap*.

4 Conclusion

We have made two proposals to speed up the Paralle Iterative Method. We have achieved speed increase by parallelizing the DP procedure for any sequence data. The pruning rate by A^* algorithm depends on the similarity of the sequences, but as figure 10 indicates, the heap can manage many nodes at a high speed. Actually our experiments showed that it takes only 2.28 seconds to traverse 57726 nodes by using heap.

Future Work We have applied the A^* algorithm to the Parallel Iterative Method; the 2-dimensional search space. In the same manner, we could apply it to the n-dimensional search space. We will attempt to apply the A^* algorithm to the n-dimensional search space.

5 Acknowledgement

We would like to express our sincere gratitude to Mr. Hoshida and Mr. Tanaka and other members of the ICOT genome application group for answering our questions and providing the compilations of protein sequences. This work was done under the auspices of the Fujitsu Laboratories Ltd. with access to an AP1000.

References

[1] M.Hoshida, M.Ishikawa, M.Hirosawa, T.Toya, and T.Totoki. Protein sequence alignment by parallel iterative method. Technical Report 92-FI-27-2, Information Processing Society of Japan, 1992.

[2] M. P. Berger and P. J. Munson. A novel randomized iterative strategy for aligning multiple protein sequences. *Computer Applications in the Biosciences*, 7:479–484, 1991.

[3] M.O.Dayhoff et al. A model of evolutionary change in proteins. In *Atlas of Protein Sequence and Structure 5;3*, pages 367–373. Nat. Biomed. Res. Found., Washington, D.C., 1987.

[4] T.Shimizu, T.Horie, and H.Ishihata. Low-latency message communication support for the ap1000. In *Joint Symposium on Parallel Processing JSPP'92*, pages 195–201, 1992.

[5] Nils. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*, chapter 3. McGRAW-HILL BOOK CO.,INC, 1971.

[6] James W.Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12, 1984.

[7] E.Dijkstra. A note on two problems in connection with graphs. *Numerische Math.*, 1:269–271, 1959.

[8] O.Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.