An Efficient Parallel Algorithm for Constraint Multiple Sequence Alignment

Hau-Jui Tsai, Chun-Yuan Lin, Yeh-Ching Chung, and Chuan Yi Tang*
Department of Computer Science, National Tsing Hua University, Hsin chu, Taiwan 300, ROC
{g934390@oz, cyulin@mx, ychung@cs, *cytang@cs}.nthu.edu.tw

ABSTRACT

The concept of constrained sequence alignment is proposed to incorporate the biologist's knowledge into sequence alignment such that the user-specified residues/nucleotides are aligned together in the computed alignment. Tang et al. were first to investigate the constrained multiple sequence alignment problem. Their algorithm for two sequences alignment with constraints runs in $O(\alpha n^4)$ *time and needs* $O(n^4)$ *space. Later, this result was improved by two groups of researchers independently to* $O(rn^2)$ *time and space using the same approach of dynamic programming. Recently, Lu and Huang designed a memory-efficient algorithm to improve the two sequence alignment with constraints (CPSA) by adopting the divide-and-conquer approach, this algorithm for solving CPSA problem can run in* $O(rn^2)$ *time and only need* $O(un)$ *space, where u is the sum of the length of constraints and usually* $u \ll n$ *in the practical applications. In this paper, we design an efficient parallel algorithm for the constrained multiple sequence alignment based on the memory-efficient algorithm designed by Lu and Huang and the progressive strategy.*

1: INTRODUCTIONS

Generally speaking, biologists have the knowledge of their datasets of the structures/functionalities/consensuses. The constrained sequence alignment is trying to include the biologist's knowledge into sequence alignment to increase the correctness of the alignment results. For example, many ribonucleases (RNases) including bovine and human pancreatic RNaseAs have been isolated and characterized in terms of their amino acid sequences, coding genes, three-dimensional structures and biological functions. The major structural features of all RNases contain three conserved His12, Lys41 and His119 active site residues and four disulfide bonds as compared to bovine pancreatic RNaseA. Since the RNases with solved three-dimensional structures all show very high homology among the catalytic domains and disulfide linkages, we would expect that their alignment should place His12 (Lys41 and His119, respectively) of bovine pancreatic RNase and other His (Lys and His, respectively) residues in the same column. So that we should treat His12 (Lys41 and His119, respectively) as the constraints when aligning these RNases sequences and we expect that the alignment result will like we discuss above. Tang *et al.* [10] were first to investigate the constrained multiple sequence alignment problem (CMSA). Their algorithm for two sequences alignment with α constraints runs in $O(\alpha n^4)$ time and need $O(n^4)$ space. The complexity of CMSA for K sequences alignment is $O(\alpha K n^4)$. Later, this result was improved by two groups of researchers independently to $O(\alpha n^2)$ time and space using the same approach of dynamic programming [4, 15]. Furthermore, each constraint expected to be aligned together can be seen as a conserved site of a protein/DNA/RNA family. And each conserved site may not only consist of a single residue/nucleotide, but a short segment of residues/nucleotides. It means that the constraint specified by the biologists can be a segment of residues/nucleotides with size of r. In some applications, biologists may further allow some mismatches among the residues/nucleotides of the columns requested to be aligned. So that Tsai *et al.* [10] studied such a kind of the constrained sequence alignment and designed an algorithm of $O(rn^2)$ ($O(rK^2n^2)$) time and $O(rn^2)$ space for two (K) sequences.

Although the improvement discussed above greatly increase the efficiency and practical usage of the CMSA algorithm designed by using the progressive strategy, the space complexity $O(rn^2)$ still limits the CMSA algorithm to just align a set of short sequences, at most several hundreds of the sequence length. Hence, Lu and Huang [8] designed a memory-efficient algorithm to improve the two sequence alignment with constraints (CPSA) by adopting the divide-and-conquer approach, this algorithm for solving CPSA problem can run in $O(rn^2)$ ($O(rK^2n^2)$ for progressive CMSA) time and only need $O(un)$ space, where u is the sum of the length of constraints. In this paper, we design an efficient parallel algorithm for the constrained multiple sequence alignment based on the memory- efficient algorithm [8] and the progressive strategy.

This paper is organized as follows. In Section 2, we first introduce the progressive strategy of the CMSA problem. In Section 3, we describe the detail of our method improved from progressive strategy to parallelize the CMSA algorithm. In Section 4, we analyze the performance of our parallel algorithm and show the experimental results.

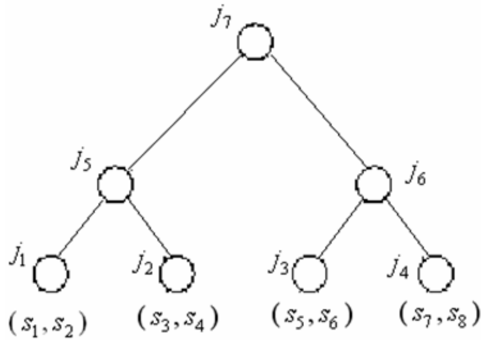

Figure 1: One example of the guide tree created.

2: PROGRESSIVE STRATEGY OF CMSA

The CMSA algorithm we introduced here is based on the progressive strategy by using CPSA as the kernel and the guide tree constructed from the Kruskal MST, called Kruskal merging order tree. The detailed algorithm of the CMSA is described in the following:

Algorithm CMSA

Input: K sequences $S = \{S_1, S_2, \ldots, S_K\}$ and α constraints $C = \{C_1, C_2, \ldots, C_\alpha\}$.

Output: The constrained multiple sequence alignment of sequences $S = \{S_1, S_2, \ldots, S_K\}$ with constraints $C = \{C_1, C_2, \ldots, C_\alpha\}$.

Step1. Compute the score of the global sequence alignment without any constraint using the Needleman-Wunsch algorithm [9] between all pairs of the K sequences.

Step2. Create a complete graph $G = (V, E)$ of K sequences in a way that each vertex $V_i \in V$ represents a sequence $S_i \in S$ and each edge e of E between V_i and V_j is associated with a weight $d(e)$ to represent the score of sequence alignment that is computed in **Step1** between S_i and S_j.

Step3. Using the complete graph G created in **Step2** to construct a Kruskal merging order tree T_K. The Kruskal merging order tree is constructed in the following way.

1. Sorting all edges of E in non-decreasing order according to their weights.
2. Build a Kruskal MST T. Initially, T is empty. Then we repeatedly add the edges of E in non-decreasing order to T in a way that if the currently adding edge e to T does not create a cycle in T, then we add e to T; otherwise, we discard e. Let $V = \{V_1, V_2, \ldots, V_K\}$ be the vertexes of T and $e_1, e_2, \ldots, e_{K-1}$ be the edges of T with $d(e_1) \le d(e_2), \le \ldots \le d(e_{K-1})$.
3. For each $V_i \in V$, we create a tree T_i such that T_i contains only a node v_i and T_i's root is v_i. Define the merge process of two trees T_i and T_j respectively rooted at v_i and v_j to be a new tree rooted at a new vertex u such that v_i and v_j become the children of u.
4. For each $e_k = (v_i, v_j)$, where k increases from 1 to K-1, we merge the tree T_i and T_j containing v_i and v_j respectively into a new tree. This process is continued until the remaining is only one tree.
5. Drop the leaf nodes of the tree build in 4. Then this final tree is the Kruskal merging order tree T_K. We use this Kruskal merging order tree T_K as the guide tree.

Step4. Use the Kruskal merging order tree T_K created in **Step3** as the guide tree. Progressively align the sequences according to the branching order of the guide tree T_K in a way that the currently two closest pre-aligned groups of sequences are joined by applying Algorithm CPSA to the pair-wised represented sequences of these two groups.

3: A PARALLEL ALGORITHM FOR CMSA

The main idea of our parallel algorithm is based on the observation of the guide tree T_K created in the Algorithm CMSA. In Algorithm CMSA Step 4, sometimes there are more than one pair of pre-aligned groups of sequences can be aligned in parallel, but they can just be aligned one after another in the Algorithm CMSA. Using a guide tree created like Figure 1 as example, in this case, job $j_1 \sim j_4$ can be done in parallel, but they will just be done one after another. In fact, we can treat each node in guide tree T_K as a job. Then, each job in guide tree T_K can be departed into three states: locked, free and assigned. Locked state means there is at least one child of this job is not assigned to any processor, and it can not be assigned now. Free-state means that all of this job's child have been assigned or this job does not have any child (leaf node in T_K), and this job can be assigned now, but not be assigned. Assigned state means that this job has been assigned to one of the processors. While deciding each job's assigned processor, we should consider the balance of the communication overhead and each processor's loading. If a job j_a is assigned to processor P_s, but it's child j_a is assigned to P_t, then before P_s start to run job j_a, P_s has to receive the aligned result from P_t, and there is one more communication between P_s and P_t. For the purpose of decreasing the total communication as much as possible, we may assign j_a to processor P_t, so that there is no communication before P_t start to run job j_a. But we should still consider one more point, if there is too much job assigned to P_t, and P_s is always in idle state, then the processor utilization will be too low. For

example, if we assign j_a to P_t, and P_t is always busy until time t_x, but P_s is idle between time t_y to t_x ($t_y < t_x$), then assigning j_a to P_s and increase one more communication may be worthy because job j_a can be done before time t_x and communication overhead will cost no more than t_x - t_y.

In order to decreasing the total communication time and increase the processor utilization, we use the concept of priority to decide each job's assigned processor. Initially, each job has its own priority table to record every processor's priority respect to this job, and these values are all set to zero. In each turn, we scan the guide tree T_K to find which jobs are now in free-state. Suppose job j_a is in free-state in this turn, and job j_b and j_c both are child of job j_a, j_b and j_c are all in assigned state. Define pri(j_a, P_s) as the priority of processor P_s respect to job j_a. We discuss the following two possible cases. In case I, if j_b and j_c are both assigned to the same processor, say P_s, then pri(j_b, P_s) and pri(j_c, P_s) will both increase one unit. Before we decide which processor gets job j_a, j_a will update its priority table as pri(j_a, P_s) = pri(j_b, P_s) + pri(j_c, P_s). Similarly, in case II, if j_b and j_c are assigned to different processors, say P_s and P_t, respectively. Then pri(j_b, P_s) and pri(j_c, P_t) will both increase one unit, and before assigning job j_a, j_a will update its priority table as pri(j_a, P_s) = pri(j_b, P_s) and pri(j_a, P_t) = pri(j_b, P_t). After j_a updates its priority table, we will choose one processor which has highest priority and is not be assigned any job in this turn from j_a's priority table, and assign j_a to this chosen processor. By the way, if job j_a has only one child, say j_b, and j_b is assigned to P_s, then j_a only need to update its priority table as pri(j_a, P_s) = pri(j_b, P_s).

Now we discuss the way we use to assign jobs to processors. After guide tree T_K is built, we create a state table which has $K - 1$ element, and each element state(a) records job j_a's state. Initially, all elements in state table are set as locked state except the jobs that are leaf nodes in guide tree. In each turn, we scan the state table to find which jobs are in free-state. Each time we find a job is in free-state, we will check if this job has a brother node in guide tree, if it has a brother and this brother job is also in free-state, then we put these two job into a queue list we call it as free-two, notes that these two jobs in free-two queue are treated as one element but not two elements. If this job does not have any brother job in guide tree or its brother job is in assigned state or locked state, then we just put this job into a queue list we call as free-one. We use Figure 2 to explain more clearly. In Figure 2, j_1, j_2 and j_3 are all leaf nodes and they are in free-state, and the others are in locked state, initially. Job j_2 and j_3 are brother job, and job j_1 does not have any brother job. In the first turn to scan the guide tree, j_1 will be put into free-one queue and j_2 and j_3 will be put into free-two queue. After scanning from the guide tree to find which jobs are in free-state, we start to decide the job assignment. In this time, we depart the job assignment in several cases according to the number of elements in free-one queue and free-two queue. Let free(1) and free(2) denote the number of elements in free-one queue and free-two queue, respectively. And let p denotes the total number of processors. We list the conditions of each case in Table 1. In case 1, the number of elements in free-two queue are more than the number of total processors, so we assigned each processor one element in the free-two queue, each processor will be assigned two jobs in this case, and these two jobs are brother job, respectively. The chosen processor is the one which has highest priority in these two job's priority table and has not been assigned any job in this turn. For example, if an element in free-two queue stores job j_a and j_b. Suppose the processor which has highest priority respect to j_a is P_s and P_s is not assigned any job yet, and the processor which has highest and second highest priority respect to j_b are P_t and P_u, respectively. But P_t has been assigned jobs in this turn and P_u is not yet. Then we will choose a processor such that has higher value between pri(j_a, P_s) and pri(j_b, P_u). That means if pri(j_a, P_s) is higher than pri(j_b, P_u), we choose P_s, otherwise, we choose P_u, and the chosen processor will be assign these two jobs. In case 2, the assignment policy is similar with case 1, each element in free-two are assigned to one processor, and after all elements in free-two are assigned, we select two elements in free-one each time, and assigned them to a processor which has highest priority between these two job and has not been assigned any jobs in this turn yet, until all processors have been assigned jobs. In case 3, after assigned each element in free-two to a processor, we then assign each element in free-one to a processor, and the chosen processor for each job is the one who has highest priority respect to this job and has not been assigned any job yet. Noting that in this case, the processor chosen by the element in free-two will be assigned two jobs, and the processor chosen by the element in free-one will only be assigned one job. In case 4, each element in free-two queue will be assigned to two different processors, it means that two jobs in each element will select its own highest priority processor and assigned to its chosen processor, respectively. If all elements in free-two queue are assigned and there are still some processors not assigned any job yet, then assign element in free-one to these processors, until all processors are assigned job. In this case, each processor will only be assigned one job. In case 5, the assignment policy is similar with case 4, the only different is that there will be some processors not assigned any job. In each turn, as the assignment is finish, the state of each job that is assigned in this turn will be set to assigned state, and check its parent job, if all child jobs of its parent job are in assigned state, then change the state of its parent job from locked state to free-state, otherwise, let it still be in locked state. Finally, delete all remain element in queue free-two and free-one. This assignment policy will recursively run until all jobs are in assigned state. Using this assignment policy, each processor's job loading can be balanced and each job will have higher possibility to be assigned to the same processor that its child job assigned to.

After finishing job assignment, each processor has to know when should it send its aligned result to other processor or receive them from other processor by building its own sending and receiving list. We define a time-stamp as the time each job uses to finish one job. When job assignment is finish, processor can know which job it should run in each time-stamp by the order of each job adding into its own job list. Suppose job j_a is the first job that is assigned to processor P_s, then processor P_s will run j_a in the first time-stamp, after j_a is finish, P_s will run the second job assigned to itself in the second time-stamp. For building sending and receiving list, each processor check the parent-job's assigned processor and child-job's assigned processor of every job that is assigned to it. If the assigned processor of child-job is different with itself, then before processor start to run this job, it has to receive the aligned result of this job's child-job from child-job's assigned processor. For example, if job j_a's assigned processor is P_s, and j_a's child-job j_b is assigned to P_t. Then as P_s check j_a and find its child-job j_b is assigned to P_t, P_s will check job assignment table to know j_b will be run by P_t in which time-stamp, say t_x, and add one record into its receiving list to denote that it should receive j_b's aligned result from P_t in the end of time-stamp t_x. Similarly, if the assigned processor of parent-job is not itself, then after processor finish this job, it has to send the aligned result of this job to the assigned processor of this job's parent-job, so processor has add one record into its sending list that records the information of time-stamp and processor's id.

As processor's sending and receiving list is created, each processor start to run their job according to their job list in each time-stamp, and as finishing this job, check its sending and receiving list and send aligned result to other processor or receive from them if necessary. The detailed algorithm of the PCMSA is described in the following:

Algorithm PCMSA
Environment: p processors.

Input: K sequences $S= \{S_1, S_2, \ldots, S_K\}$ and α constraints $C= \{C_1, C_2, \ldots, C_\alpha\}$.

Output: The constrained multiple sequence alignment of sequences $S= \{S_1, S_2, \ldots, S_K\}$ with constraints $C= \{C_1, C_2, \ldots, C_\alpha\}$.

Step1. Compute the score of the global sequence alignment without any constraint using the Needleman-Wunsch algorithm between all pairs of the K sequences. And assign these $K\times(K-1)$ pairs to p processors such that each processor get $K\times(K-1)/p$ pairs. After each processor finish its own $K\times(K-1)/p$ pairs, send these scores to the first processor. After the first processor get all $K\times(K-1)$ scores, it broadcast them to the other p-1 processors.

Table 1: Condition of each case

Case id	Condition
Case 1	$free(2) \geq p$
Case 2	$free(2) + free(1)/2 \geq p$
Case 3	$free(2) + free(1) \geq p$
Case 4	$free(2) \times 2 + free(1) \geq p$
Case 5	others

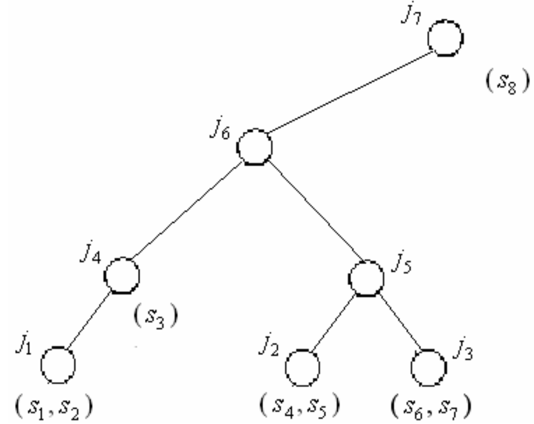

Figure 2: Initially, $j1$ will be treated as free-one and ($j2$, $j3$) will be treated as free-two.

Step2. Each processor create a complete graph $G = (V, E)$ of K sequences in a way that each vertex $V_i \in V$ represents a sequence $S_i \in S$ and each edge e of E between V_i and V_j is associated with a weight $d(e)$ to represent the score of sequence alignment that is computed in **Step1** between S_i and S_j.

Step3. Each processor using the complete graph G created in **Step2** to construct a Kruskal merging order tree T_K. Using the method introduced in **Algorithm CMSA Step 3** to construct the Kruskal merging order tree.

Step4. Each processor runs the job assignment policy.

Step5. Each processor checks the parent-job and child-job of the job that is in its job list. If the assigned processor of job's parent-job is not itself, then check this parent-job will be run in which time-stamp, and add one data that records parent-job's assigned processor and running time-stamp into its own receiving list. If the assigned processor of job's child-job is not itself, then add one data that records child-job's assigned processor and the time-stamp that it will run this job into its own sending list.

Step 6. Each processor run the job that according to its own job list in each time-stamp, after this job is finished, check its sending and receiving list to send to or receive from other processor if necessary.

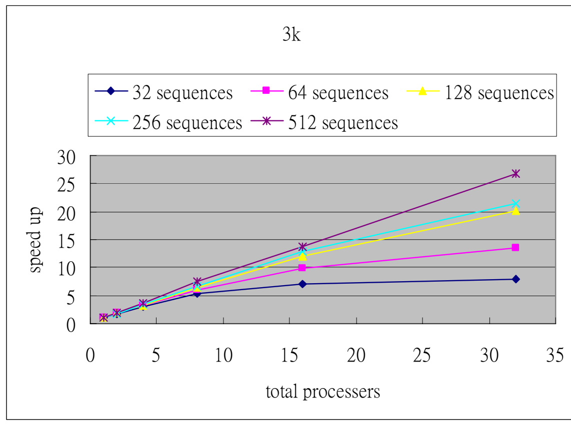

Figure 3: Speed up of sequence length 3k.

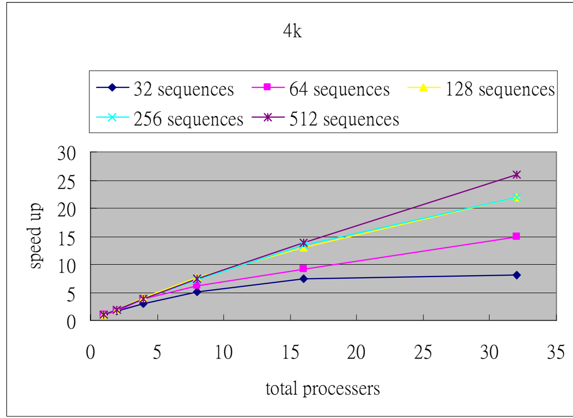

Figure 4: Speed up of sequence length 4k.

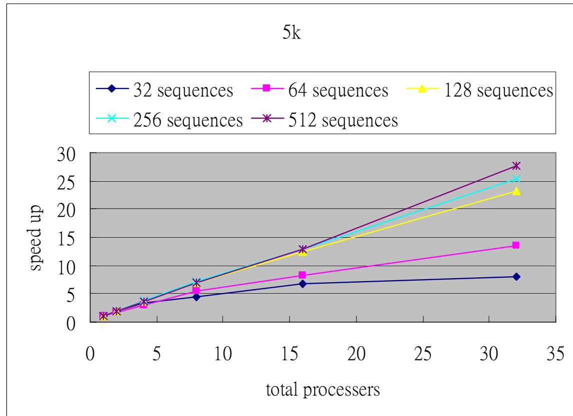

Figure 5: Speed up of sequence length 5k.

Table 2: Hardware and software list of NCHC Formosa PC Cluster

CPU	Intel Xeon 2.8GHz
Memory	2GB
Network Switch	1Gbps
Complier	GNU

4: EXPERIMENT RESULT AND ANALYSIS

We used MPI + C to implement our parallel method, and tested it on NCHC Formosa PC Cluster. The hardware and software information of NCHC Formosa PC Cluster is shown in Table 2, and Figure 3 ~ Figure 5 are performance result. Notice that we use the best case – complete binary tree as the guide tree to test our parallel program.

The best case for our parallel algorithm is when the guide tree T_K is a complete binary tree like Figure 2. In our parallel algorithm, Step 1 and Step 6 are the parallelized parts, and Step 2 to Step 5 can not be parallelized. Analysis our implement for the best case, we can estimate the running time using the following formula:

Serial: $\frac{k(k-1)n^2}{2}\times t_1^1 + \frac{k^4-2k^3-5k^2+6k}{8}\times t_1^2 + (k-1)rn^2\times t_1^3$

Parallel:

Step 1:

$$\left(\frac{1}{p}\times\frac{k(k-1)n^2}{2} + (p-1)\times\frac{k(k-1)}{2} + \log_2(k)\times\frac{k(k-1)}{2}\right)\times t_1^1$$

$$+\ (p-1+\log_2(k))\times t_2 + \frac{k(k-1)}{2}\left(1-\frac{1}{p}+\log_2(k)\right)\times t_3$$

Step 2 ~ Step 5: $\frac{k^4-2k^3-5k^2+6k}{8}\times t_1^2$

Step 6:

If $\log_2(k)-1 \geq \log_2(p)$

Then

$$\left(\log_2(p)+\frac{k}{p}-1\right)\times rn^2\times t_1^3 + 3\log_2(p)\times t_2$$

$$+\left(3\log_2(p)\times size(\text{int}) + \frac{k}{p}(p-1)n\times size(char)\right)\times t_3$$

If $\log_2(k)-1 < \log_2(p)$

Then

$$\log_2(k)\times rn^2\times t_1^3 + 3(\log_2(k)-1)\times t_2$$

$$+(3(\log_2(k)-1)\times size(\text{int}) + (\log_2(k)-2)n\times size(char))\times t_3$$

Parameters t_1^1, t_1^2, t_1^3, t_2 and t_3 are computed on NCHC Formosa PC Cluster, t_1^1 , t_1^2 , t_1^3 are computation time, t2 is communication start up time, and t3 is communication transmission time. The values of t_1^1, t_1^2, t_1^3, t_2 and t_3 are 64.5ns, 14.7ns, 55.1ns, 56us and 50ns, respectively. Using the formulas shown above, we can estimate the running time and the speed up, and know how many processors will be needed to achieve our speed up goal. For example, suppose we have 512 sequences with length 5k, and 5 constraints. If we want the speed up can be at least 15, then we will have the inequality *Serial/Parallel* >= 15, and set k=32, n=5000 and r=5. After computing this inequality, we will have the result p>= 16. So we will need at least 16 processors to running this testing to have speed up 16.

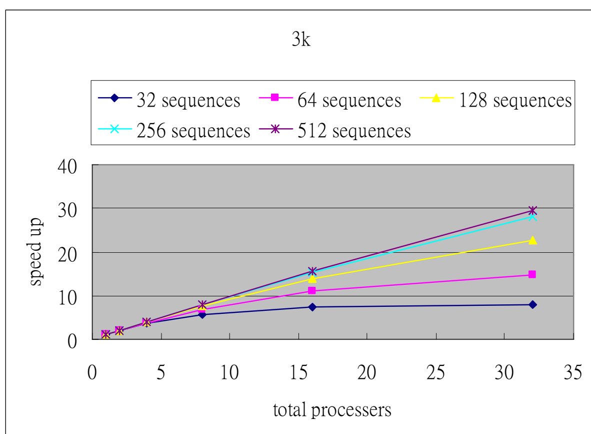

Figure 6: Theoretical Speed up of sequence length 3k.

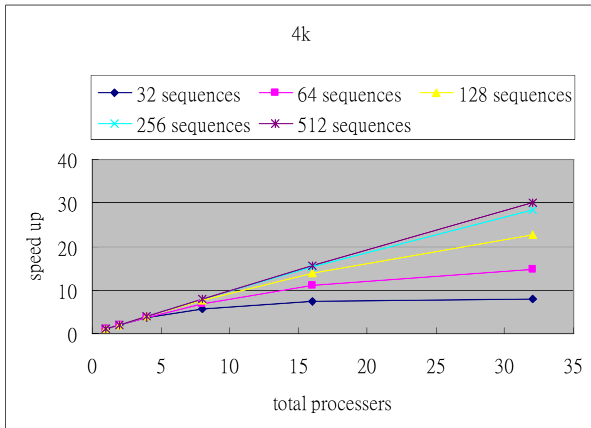

Figure 7: Theoretical Speed up of sequence length 4k.

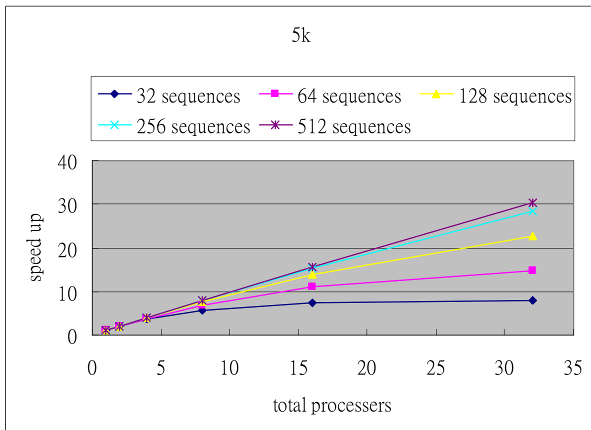

Figure 8: Theoretical Speed up of sequence length 5k.

Figure 6 to Figure 8 are the theoretical speed up computed from Serial/Parallel. Comparing them with the experimental result, it shows that our tool can achieve 80% of the theoretical speed up.

5: CONCLUSIONS

In this paper, we have proposed a parallel algorithm for the constraint multiple sequences alignment with time complexity $O((k^2+kr)n^2/p+k^4)$ where k is the number of sequences, r is the number of constraints, n is the length of sequence and p is the number of processors. By the experimental results, we can show that our algorithm is more time-efficient than the serial algorithm, and can achieve a good speed up.

6: REFERENCES

[1]. P. Bonizzoni and G.D.V edova, "The complexity of multiple sequence alignment with SP-score that is a metric," *Theoretical Computer Science*, vol. 259, 2001, pp.63-79, 2001.

[2]. H. Carrillo and D. Lipman, "The multiple sequence alignment problem in biology," *SIAM J. Applied Mathematics,* vol. 48, 1988, pp.1073-1082.

[3]. S.C. Chan, A.K.C. Wong, and D.K.Y. Chiu, "A survey of multiple sequence comparison methods," *Bulletin of Mathematical Biology,* vol. 54, 1992, pp.563-598.

[4]. F.Y.L. Chin, N.L. Ho, T.W. Lamy, P.W.H. Wong and M.Y. Chan. "Efficient constrained multiple sequence alignment with performance guarantee," *Proc. IEEE Computer Society Bioinformatics Conference*, 2003, pp. 337-346.

[5]. F. Corpet, "Multiple sequence alignment with hierarchical clustering," *Nucleic Acids Research*, vol. 16, 1988, pp.10881-10890.

[6]. D.F. Feng and R.F. Doolittle, "Progressive sequence alignment as a prerequisite to correct phylogenetic trees," *J. Molecular Evolution*, vol. 25, 1987, pp. 351-360.

[7]. D. Higgins and P.S harpe, "CLUSTAL: a package for performing multiple sequence alignment on a microcomputer," *Gene*, vol. 73, 1988, pp.237-244.

[8]. C.L. Lu and Y.P. Huang, "A memory-efficient algorithm for multiple sequence alignment with constraints," *Bioinformatics*, vol. 21, 2005, pp.20-30.

[9]. S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Molecular Evolution*, vol. 48, 1970, pp.443-453.

[10]. 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 application to RNase family alignment," *J. Bioinform. Comput. Biol.*, vol. 1, pp.267–287.

[11]. W.R. Taylor, "Multiple sequence alignment by a pairwise algorithm," *CABIOS*, vol. 3, 1987, pp.81-87.

[12]. J.D. Thompson, D.G. Higgs, and T.J. Gibson, "CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties, and weight matrix choice," *Nucleic Acids Research*, vol. 22, 1994, pp.4673-4680.

[13]. Y.T. Tsai, Y.P. Huang, C.T. Yu, and C.L. Lu, "MuSiC: a tool for multiple sequence alignment with constraints," *Bioinformatics*, vol. 20, 2004, pp. 2309–2310.

[14]. L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *J. Computational Biology*, vol. 1, 1994, pp.337-348.

[15]. C.T. Yu, "Efficient algorithms for constrained sequence alignment problems," *Master's Thesis*, Department of Computer Science and Information Management, Providence University, 2003.