Load balancing and Parallel Multiple Sequence Alignment with Tree Accumulation

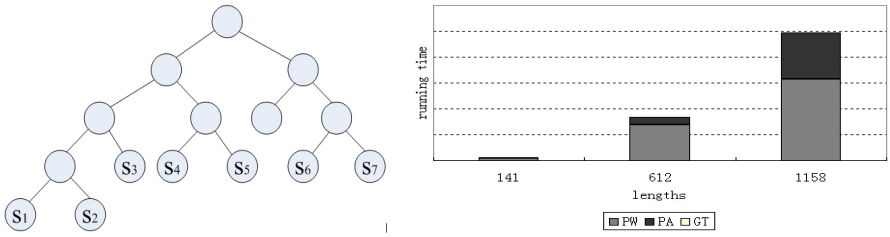
Guangming Tan[1,2], Liu Peng[1,2], Shengzhong Feng[1], Ninghui Sun[1]

[1] Institute of Computing Technology, Chinese Academy of Sciences
[2] Graduate School of Chinese Academy of Sciences
{tgm,pengliupl,fsz,snh}@ncic.ac.cn

Abstract. Multiple sequence alignment program, ClustalW, is time consuming, however, commonly used to compare the protein sequences. ClustalW includes two main time consuming parts: pairwise alignment and progressive alignment. Due to the irregular computation based on tree in progressive alignment, available parallel programs can not achieve reasonable speedups for large scale number of sequences. In this paper, progressive alignment is reduced to tree accumulation problem. Load balancing is ignored in previous efficient parallel tree accumulations. We proposed a load balancing strategy for parallelizing tree accumulation in progressive alignment. The new parallel progressive alignment algorithm reducing to tree accumulation with load balancing reduced the overall running time greatly and achieved reasonable speedups.

1 Introduction

Algorithms for multiple sequence alignment [1] are routinely used to find conserved regions in biomolecular sequences, to construct family and superfamily representations of sequences, and to reveal evolutionary histories of species. Conserved subregions in DNA/protein sequences may represent important functions or regulatory elements. The profile or consensus sequences obtained from a multiple alignment can be used to characterize a family or superfamily of species. Multiple sequences alignment is also closely related to phylogenetic analysis. From a mathematical point of view, the multiple sequences alignment is a more complex combinatorial problem which is NP hard. There has been a lot of interest in finding efficient approximation algorithms (PTAS)[2] for these problems. However, the PTAS algorithms have high time complexity so that they become impractical for many long sequences. Some popular heuristic approaches such as progressive alignment [1] that work reasonably well in practice have been proposed. The most widely used algorithm is the progressive alignment algorithms and its typical implementations are ClustalW [1] and DFALIGN [3]. Although the running time has been reduced, the time complexity of the progressive alignment algorithms is $O(n^2m^2)$, where n is the number of sequences and m is the maximum length of all sequences. Since the best known progressive alignment programs is ClustalW, we focus on the parallelization of ClustalW. The basic algorithm behind ClustalW proceeds in three steps.



(a) (b)

Fig. 1. a). A guide tree. Each leaf represents a sequences, the internal nodes represent the partial alignment from their children. b). The running time distribution of three parts in ClustalW. The number of protein sequences is 781, 1158 and 1770. In most cases the CPU times spent on building the guide tree is less than 1 percent (almost cannot be seen in this figure). The pairwise alignment occupies the most of the overall running time, however, the running time of the progressive alignment significantly increases with the larger number of sequences.

1. **Pairwise alignment(PW):** Compute the optimal alignment cost for each pair of sequences using standard dynamic programming. This results in a distance matrix whose entries indicates the degree of divergence of each pair of sequences in evolution. In fact, this step can be very time consuming and become the bottleneck of the whole process because it has to align *n(n-1)/2* pairs, where n is the number of sequences. Since each alignment is independent of the rest, the parallelization is a problem of allocating time-independent tasks to parallel processors and can achieve linear speedups [5][6][7][8].

2. **Guide tree(GT):** Compute an evolutionary tree from the distance matrix using some phylogeny reconstruction method. This tree will be used as the guide tree (See Figure 1(a)) which guides the final multiple alignment process are computed from the distance matrix by first using a popular distance based phylogeny reconstruction method, the Neighbor-Joining method [4]. In general, this step can be completed very fast.

3. **Progressive Alignment(PA):** The basic procedure of progressive alignment is to use a series of pairwise alignments to merge larger and larger groups of sequences, following the branching order in the guide tree. Each merger involves aligning two multiple alignments using a dynamic programming algorithm similar to that for the alignment of a pair of sequences. It contains a profile-profile/sequences alignment implemented by dynamic programming algorithm with linear space. In this way, sequences that are highly divergent from the rest of sequences are given due consideration in the alignment process.

Because ClustalW program is widely used and time consuming, there exist some contributions to parallelizing ClustalW algorithm. Mikhailov et al. [5] designed a parallel ClustalW for shared-memory multiprocessor machines. It runs only on SGI computers with OpenMP and achieves a maximum speedup of 10 for the whole alignment process on 16 processors machine for some protein sequences. Duzlevski [6] used Posix threads and its implementation can be run symmetric multiprocessor computers. Jamse et al. [7] and K. Li [8] implemented a parallel ClustalW for PC cluster using MPI, respectively. They report a fine linear speedup only for pairwise alignment, but the speedup and scalability for the whole alignment are poor because those parallel programs ignore the significant to parallelize progressive alignment.

For the small number of sequences, the efficient parallelization of the step 1 is enough because the running time of progressive alignment is not significant(See figure 1(b)). However, when the number of sequences becomes larger, the poor performance of parallelization in progressive alignment becomes a bottleneck because of the linear speedup in pairwise alignment. Because of the irregular structure based on tree in progressive alignment, it is difficult to efficiently parallelize step 3. The previous parallel programs focuses on the small scale problem, thus the performance of parallel progressive alignment is not important to the overall parallel program for the small number of sequences (less than few hundreds of sequences). However, when aligning the larger number of sequences, the progressive alignment becomes a bottleneck because of the linear speedup in step 1 and the poor performance for parallel progressive alignment. In this paper, we proposed a fast parallel algorithm for multiple sequences alignment program (ClustalW) using load balancing strategy.

2 Parallel Progressive Alignment

2.1 Reducing to Tree upward Accumulation

There are generally two kinds of accumulations on trees with bounded maximum degree: upward accumulations and downward accumulations[9]. Consider a tree of n nodes, each containing an operation drawn from a set S, and a binary associative operation $\bigotimes : S \times S \rightarrow S$. Let s_v denote the operation at node v, and u_1, u_2, ..., u_k be an ordered list children of v. Without loss of generality, the upward accumulation problem is to compute $A(v)$ for each node v in the tree where

$$A(v) = \begin{cases} s_v & \text{if v is a leaf} \\ A(u_1) \bigotimes A(u_2) \bigotimes ... \bigotimes A(u_k) & \text{otherwise} \end{cases} \quad (1)$$

If the binary operator is commutative, we can simply write the upward accumulation as:

$$A(v) = \bigotimes_{u \in subtree(v)} s_u \quad (2)$$

Progressive alignment is a profile/sequences alignment progress basing on the guide tree that is a complete binary tree. The leaves are sequences and the internal nodes are profiles. Basing on the guide tree, progressive alignment performs

the profile/sequences alignment from leaves to root. Thus, progressive alignment is reduced to the tree accumulation problem. If the binary operator $\bigotimes$ represents profile/sequences alignment, progressive alignment is reduced to tree accumulation naturally.

```
contractl(u):
push(u.right.stack, u, u.operator);
u.right.operator = u.operator
(u.left.opertor, u.right.opertor);
u.right.parent = u.parent;
if u.left != NULL
    u.parent.left = u.right;
else
    u.parent.right = u.right;
u.right.left = u.left;
if root = = u
    root = u.right;
```

(a)

```
distribution:
for each node u do in parallel
    wait until u.val is computed;
    while u.stack != NULL do
        (v, operator) = pop(u.stack);
        while dependency in operator do
            block;
        end
        v.val = operator(u.val);
        end
    end
```

(b)

Fig. 2. Pseudocode procedure for contractl and distribution

2.2 Parallel Tree Accumulation

To get round this problem, the PRAM tree accumulation algorithm operates in two phases [9]: a *contraction* phase in which the tree is reduced to a single leaf and some nodes are put aside on stacks, and a *distribution* phase in which the stacked nodes receive their final values. Each contraction operation removes a leaf node v and its parent (an internal node) by connecting v*'s* sibling directly to its grandparent. Although the final value to be assigned to the internal node is still unknown, yet it is the certain known function (binary operator) of the final value that is to be assigned to the siblings. The deleted internal node and its binary operator are put aside on a stack belonging to all the siblings. When the final value to be assigned to the sibling is computed, the value for the deleted parent can be computed in turn.

Contraction The contraction operations each remove two nodes, at least one of which is a leaf (See Figure 3). Assume that all leaf nodes of tree are numbered from left to right. Mark all even/odd numbered leaves. For every marked leaf that is left child of their parent u, perform the contraction operation: contractl(u), and then for every marked leaf that is right of their parent u, perform the contraction operation: contractr(u). This guarantees that parents of the leaves contracted are not adjacent [9]. The primitive operation is contraction, which is only called so when u is the internal node and its one of its children is a marked

leaf. contractl(u) and contractr(u) a pair of symmetric operation, contractl(u) is defined as fig. 2(a)

Distribution The contracted nodes are expanded and accumulations at all nodes are accomplished during the distribution phase. The premise is that before a leaf node is expanded, its siblings have correct accumulation. The information stored in each leaf node and the accumulations in its siblings are used to compute the final values of the node. Each node u has stack $u.stack$ with the data structure of (node, function). If *(v, h)* is in $u.stack$, then operator *(u.val)* should be assigned to $v.val$, once $u.val$ is computed (See fig. 2(b)).

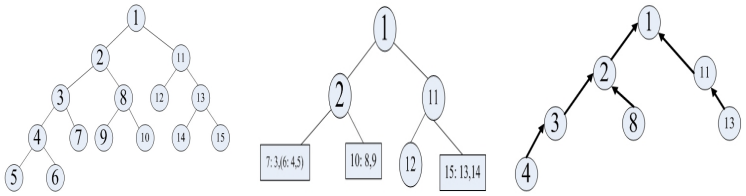


Fig. 3. An illustration of the contraction phase and mapping tree accumulation to task graph. (a). the original tree. The number in each node is number by preorder traversal. (b). the partial contracted tree. The internal node 3, 4, 8 and 11 are removed. The information are stored in their right child. c). The DAG task graph. The direction implicit the order of task dispatched.

2.3 Load balancing strategy

For a tree accumulation problem, tree contraction has been proven to be efficient if the operations associated with the internal nodes require $O(1)$ time[9]. The available parallel algorithms for tree accumulation, which rest on a common assumption that all binary operators are equal, that is, the computing time of all binary operators is the same, the order has no bearing on the runtime. On the other hand, if the each binary operator on nodes consumes different time, or at least two operations require different time for executing, different orders of operations might well lead to variety, even to the extent of great difference in runtime. Have a deeper analysis of this issue. In the parallel upward accumulation, $A(v)$ in the same level of the tree can be computed in parallel. However, if the binary operators need different running time, then the processor which has completed its computational task will have to wait until all the computational tasks of its brother nodes have finished, which causes poor load balancing, and consequently results in low processor utilization. Moreover, the topology of the tree also has influence on the performance of the parallel algorithm in that the critical path of accumulating from the leaves to the root determines the running

time. Unfortunately, previous parallel algorithms hardly focus on the effect of tree topology and almost all of them start accumulating from all leaves, let alone contrive efficient policy to map accumulation to proper processor, which gravely diminish the efficiency of tree accumulation because the processors which compute the shorter branches are left idle most of the time if the tree is unbalanced which is just the case in most practical applications.

Progressive alignment reducing to tree upward accumulation is an exact example for the shortcomings described above. The time of each pairwise alignment is proportional to the product of the length of two sequences. The length of sequences is different, thus each pairwise alignment has different running time. Because of the divergent of all sequences, the guide tree based on the distance matrix may not be a balancing tree. So a naive implementation of previous parallel tree accumulation algorithm can not promise good load balancing and high processor utilization.

Many scheduling algorithms have been proposed and two good surveys on static and dynamic scheduling algorithms can be found in [10], where a parallel program can be described as a directed acyclic graph (DAG). A weighted DAG task graph can be used to represent the problem of tree accumulation on the basis that a task is defined as an operator at a node. The task graph can be constructed in the contraction phase. Each stacked node corresponds to a certain node in the task graph. The weight of a node is an estimated running time of the operator while the weight of an edge is an estimated size of messages from the child task to the parent task. And how to calculate the two weights is determined according to real applications. An compelling example of mapping from the original tree to task tree is shown in Figure 3(c).

Define the b-level of a task as the length of the longest path from the task to the root task, where the length of a path is the sum of all the node and edge weights along the path. Further, the number of internal nodes from each node to root is added to calculate the b-level in order to consider the factor of tree topology. The b-level of a node is bounded from above by the length of a critical path, which is the longest path from the temporal node to the root node in the DAG. The *b-level* of a node is assigned to the node on the stacks in the contraction phase. In the distribution phase, a dynamic priority queue is maintained in order to schedule the tasks for processors. When any popped operator can be computed, it is inserted into the priority queue according to its *b-level*. And if there is any idle processor, remove the task at the head of the queue and schedule it to the processor. The modified distribution algorithm employs a coordinator-worker model. The coordinator maintains the priority queue and schedules and dispatches tasks to workers who execute the real operators. The algorithm of coordinator in the distribution phase are as follow:

Distribution with load balancing:
for each node u **do**
  wait until u.val is computed;
  while u.stack != **NULL do**

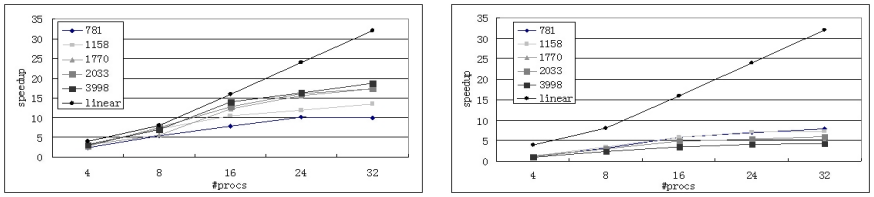
```
      (v, operator) = pop(u.stack);
      while dependency in operator do
        block;
      end
      insert(u, v, operator, queue);
      while (queue != NULL && idle_procs != 0) do
        dispatch(queue, processor);
      end
    end
end
```

The coordinator also maintains an idle processors pool. In the beginning, all processors are idle and the pool is full. When one task is assigned to one idle processor, the processor is deleted from the pool. And after one processor has completed its task, it sends a message to coordinator and coordinator adds this processor to the pool.

3 Performance Evaluation

Because the previous parallel programs can not get speedups for progressive alignment, we only evaluated the performance of load balancing. The experiment implemented the load balancing parallel algorithm in cluster systems—distributed memory parallel computers connected by networks. Each node of the cluster system is composed of Xeon 2.8Ghz SMP processors, 4GB memory, while all the nodes are connected via gigabit Ethernet switch. And the parallel program is written using C with MPI library. Moreover, the test data sets are downloaded from PDB bank [11], and they are five different protein family or domain(TROW, WOLPM, WIGBR, ZYMMO, YERPS). For simplicity, some notations are used in the evaluation: *lb* denotes the parallel algorithm with load balancing and *na* denotes the naive parallel algorithm.

Speedup: The performance of a parallel algorithm is measured by speedup or efficiency. The speedup of a parallel algorithm using p processors is defined as $Speedup = \frac{T_{serial}}{T_{parallel}(p)}$ and the efficiency is $Efficiency = \frac{Speedup(p)}{p}$. Strictly speaking, T_{serial} is the running time of the fastest known serial algorithm on one processor for the same problem. Figure 4(a) and 4(b) show the speedups as the number of processors and the size of problem size are increased for algorithms with load balancing strategy and without load balancing strategy, respectively. The speedups of *lb* are much higher than that of *na*. When the number of processors is less than 16, the parallel program with load balancing strategy can achieve approximate linear speedup. While the number of processors is larger, the speedup of both algorithms increases slowly. The highest speedup 18 of *lb* occurred when the number of sequences is 3998 and the number of processors is 32, while the highest speedup 8 of *na* occurred when the number of sequences is 781 and the number of processors is 32.



(a) (b)

Fig. 4. Speedup for naive parallel algorithm with/without load balancing

Time: The most important contribution of load balancing strategy is the reduction of overall running time. The tree accumulation process in progressive alignment presented above comprises computation, communication and other overhead such as scheduling and idle. Table 1 demonstrates the overall running time for two parallel algorithms with the different number of processors and different size of problems. The overall running time of parallel algorithm *lb* are reduced mostly 3 times as that of *na*. Because the relative time distributions

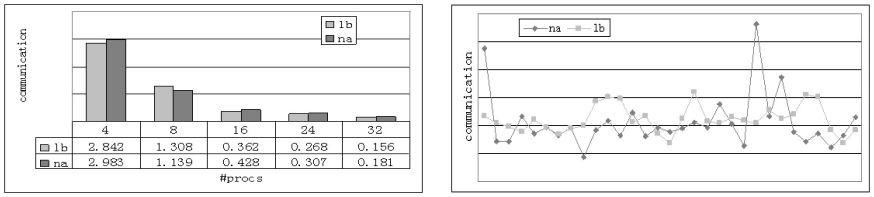
Table 1. The overall running time of two parallel algorithms. The number of sequences of 5 data sets are 781, 1158, 1770, 2033 and 3998, the number for processors are 4, 8, 16, 24 and 32. Time: second

		4	8	16	24	32
781	lb	163	73	51	40	40
	na	377	127	68	58	51
1158	lb	199	91	62	55	48
	na	502	189	113	93	91
1770	lb	311	173	79	63	56
	na	765	321	203	183	183
2033	lb	393	146	87	69	64
	na	998	373	226	206	185
3998	lb	448	200	100	86	74
	na	1394	609	406	345	327

of computation, communication and overhead for the different problem size are almost the same, we only analysis the experiment results of overhead in the case of 1158 sequences alignment.

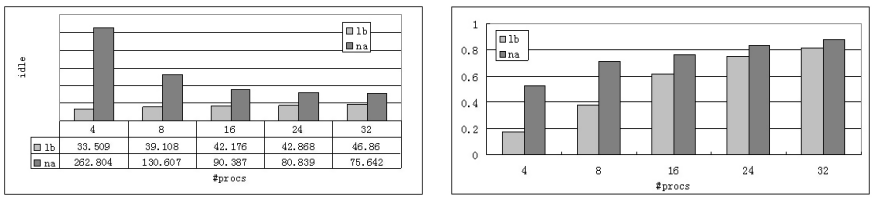
The parallel algorithms are implemented using coordinator-worker model, while the coordinator only performs scheduling and communication. It is the workers who execute the real computation and send/receive message from coordinator. Figure 5(a) shows that there is minor difference of the communication times between two algorithms. However, the communication distribution among all slave processors for *lb* is even more than the distribution for *na*. Figure 5(b)

demonstrates communication time distribution among all workers in 32 processors. In fact, the unbalanced communication is relative the reflection of the unbalance computation in each worker. In the presented algorithms, there only exist communications between the coordinator and workers, so the more computation load in one worker, the more communications are needed in the worker.



(a) (b)

Fig. 5. a). The maximum communication time in seconds with the different number of processors b). The communication time distribution in 32 processors for two parallel algorithms



(a) (b)

Fig. 6. a). The maximum idle time in seconds for the different number for processors. b).The time proportions of idle to the overall running time

Due to the different computing load and unbalanced binary tree, some slave processors may be idly waiting for another computation task that depends on some other computation tasks running on slower processors. Although the communication and computation load is unbalanced for the parallel algorithm *na*, neither of the time cost are higher than the cost of the parallel algorithm *lb* as shown in above analysis. Thereby the communication and computation time pale in terns of their influence on the time reduction. Measure the idle time in each worker processor. Figure 6(a) shows the maximum idle time for different number of processors. The parallel algorithm *lb* mainly focuses on the factors of computation weight and branch length in the task tree to schedule the computation task, and it proved to have reduced the idle time in each processor greatly. With the number of processors increasing, the computation loads in each processor

become less, that is, the overall computation is decreasing, so the proportions of idle time to the overall running time become higher (See Figure 6(b)). However, the larger the number of processors, the more the overhead of scheduling task among more processors is, and correspondingly the more the idle time is for the parallel algorithm *lb*.

4 Conclusions

In this paper, a new parallel implementation of progressive alignment through tree accumulation with load balancing is presented. And in the proposed implementations, the load balancing strategy is used in order to take advantage of both weighted tree contraction and tree topology. Moreover, a test for the performance of the algorithm and a comparison with the naive PRAM implementation on a 32-processors Linux cluster system is shown and analyzed in the context, which shows that the parallel tree accumulation algorithm achieves not only reasonable speedups for the data sets used in the evaluation, but also higher speedups than the naive parallel algorithm using load balancing.

This work is supported by National Natural and Science Foundation (90412010) and Youth Foundation of ICT.

References

1. D. T. Julie, G. H. Desmond and J. G. Toby, Clustal W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, Nucleic Acids Research, 1994, Vol. 22, No. 22, pp. 4673-4680.
2. D. Henikoff, Approximation Algorithms for NP-hard Problems, PWS publishers, 1996.
3. D. Feng and R. F. Doolittle, Progressive sequence alignment as prerequisite to correct phylogenetic trees, Journal of Molecular Evolution, 1987, Vol. 25, pp. 351-360.
4. N. Saitou and M. Nei, The neighbor-joining method: A new method for reconstructing phylogenetic trees. Molecular Biology and Evolutoin, 1987, Vol. 4, pp. 406-425.
5. D. Mikhailov, H. Cofer and R. Gomperts, Performance optimization of ClustalW: Parallel ClustalW, HT Clustal and MULTICLUSTAL. White papers, SGI, 2001.
6. O. Duzlevski, SMP version of ClustalW 1.82, http://bioinfor.pbi.nrc.ca/clustalw-smp.
7. J. J. Cheetham, F. Dehne, S. Pitre, A. R. Chaplin and P. J. Tailon, Parallel CLUSTALW for PC Clusters. Proceedings of International Conference on Computational Science and its Applications, Montreal, Canada , May 18-21, 2003.
8. K. Li, ClustalW-MPI: Clustalw analysis using distributed and parallel computing, Bioinformatics, 2003, Vol. 19, no. 12, pp: 1585-1586
9. J. Gibbons, W. Cai, D. Skillicorn, Efficient parallel algorithms for tree accumulations, Sci. Comput. Programming, 1994, 23, pp. 1-18
10. Y. K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys, 1999, Vol. 31, No. 4, pp.406-471
11. http://www.rcsb.org/pdb/