For PDPTA '97

Conference Proceedings

Parallel Implementation of a Matrix Decomposition Method for Evaluating Phylogenetic Trees

Sanzheng Qiao
Department of Computer Science and Systems
McMaster University
Hamilton, Ontario, Canada, L8S 4K1

Sheldon Siu-tung Tam
Department of Electrical and Computer Engineering
McMaster University
Hamilton, Ontario, Canada, L8S 4K1

Abstract - Given a set of N objects and the distances between all pairs of objects, the goal is to construct the optimal phylogenetic tree. In this paper, a parallel implementation of a matrix decomposition method for evaluating phylogenetic trees [5] is presented. The techniques used in the implementation for load balancing and fault tolerance are described. These techniques can be applied to distributed computing in a heterogeneous cluster of workstations.

Keywords: phylogenetic trees, unrooted binary trees, Parallel Virtual Machine, load balancing, fault tolerance

1. Introduction

Phylogenetic trees are unrooted binary trees. They can be used to study the evolution of DNA, languages, human races, etc. Suppose we are given N objects and the distances between all pairs of these objects. For example, in linguistics, the objects are languages. The distances between any pair of languages may be measured in terms of some similarity index, based, for example, on the amount of shared basic vocabulary. Our aim is to construct an unrooted binary tree with tips of the tree representing the N objects. The distance between any pair of tips is the sum of the lengths of the branches which connect them. The tree should be optimal in the sense that the difference between the distances measured in the tree and the corresponding given distances is minimal.

Day [1] has shown that in general the problem is NP-hard. Several heuristic algorithms, for example [2,3,6], have been developed. In [5], Qiao, Minett, and Wang proposed a matrix decomposition method for evaluating phylogenetic trees. An unrooted binary tree can be represented by a matrix. The optimal branch lengths for a given tree topology can be found by the standard linear least squares algorithm. Also, matrices can be generated for all possible tree topologies. By inspecting all possible trees, we can find the optimal tree which fits the given distances best. We refer the details to [5].

The matrix decomposition method is exhaustive. We note that the number of all possible trees grows very fast as the number of objects increases. In general, there are $1\times3\times5\times...\times(2N-5)$ trees of N tips. An exhaustive method is infeasible in applications where the number of objects under study is large. In our application, linguistics, the number of objects (languages) is usually small. An exhaustive method allows us to list the ten best trees (or any number for that matter). This gives the investigator a rich body of data upon which to speculate the phylogenetic complexity in the evolution of the objects. However, even a moderate number of objects gives a large number of trees. For example, there are more than 650 million trees of 12 tips. Apparently, it is difficult to inspect large number of trees sequentially. To overcome this difficulty, parallel computing is necessary. Since the matrix decomposition method [5] provides an easy division of the problem into independent tasks, it is suitable for parallel computing. In this paper, we present a parallel implementation of the matrix decomposition method.

The paper is organized as follows. In Section 2, we introduce our data parallelism

approach to our implementation by showing how to use branch addition sequences to partition the data and distribute the computation to machines. In Sections 3 and 4, we discuss the two major issues, load balancing and fault tolerance, in distributed computing and describe the two techniques, dynamic job distribution and redundant computation, used in our implementation. Section 5 gives an example. Finally, Section 6 draws the conclusion.

2. Branch Addition Sequence

The branch addition sequence representation is the key to our data parallelism approach to the parallel implementation. It is essential in partitioning the data, reducing the communication cost, distributing the computation, and balancing the system load.

To illustrate the branch addition sequence representation, we start with the simplest unrooted binary tree: a tree with three tips, three branches, and one interior node called the center. We label the branches clockwise as shown in Figure 1 below.

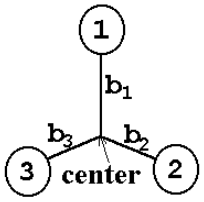

Figure 1. The initial tree of 3 tips

A tree with four tips can be obtained by adding a new branch to any of the three branches. After adding a new branch, we relabel the branches as follows. There are three subtrees rooted at the center, we visit these three subtrees clockwise and in each subtree we visit the branches inorder. Figure 2 shows a tree of four tips by adding a new branch to the branch b_1 of the trivial tree of three tips in Figure 1. Since this tree is obtained by adding the new branch to the branch b_1 of the initial tree, we use the sequence (1) to represent the tree in Figure 2. We call it branch addition sequence.

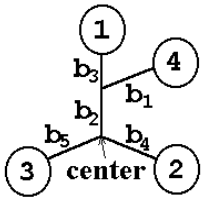

Figure 2. A tree of 4 tips

Similarly, we can get a tree of five tips by adding a new branch to any of the five branches in the tree of four tips. Thus, we can use the sequence (1, 5) to represent a tree of five tips by first adding a new branch to the branch b_1 of the initial tree of three tips and then, after relabeling the branches, adding another new branch to the branch b_5 of the tree with four tips in Figure 2. The resulting tree of 5 tips is shown in Figure 3.

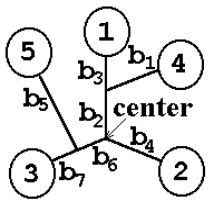

Figure 3. A tree of 5 tips

In general, we can represent a tree of N tips using a sequence of N-3 numbers: $(k_1, k_2, \ldots, k_{n-3})$. Note that k_1 ranges from 1 to 3 since the tree of three tips has three branches, k_2 ranges from 1 to 5 since a tree of four tips has five branches. In general, k_i ranges from 1 to $2i + 1$.

The branch addition sequence representation has several advantages in our parallel implementation. Since the sequences can be lexically ordered, we can easily partition the problem of inspecting trees. For example, in the PVM [4] environment, the master process merely assigns a sequence (1, 1, x, x, x) to a slave process. That means the slave process is responsible for inspecting the trees ranging from (1, 1, 1, 1, 1) to (1, 1, 7, 9, 11), a total of $7 \times 9 \times 11 = 693$ trees of eight tips. The master can assign another sequence (1, 2, x, x, x) to another slave process responsible for the trees ranging from (1, 2, 1, 1, 1) to (1, 2, 7, 9, 11). After receiving a sequence, a slave process generates

matrices representing the trees, performs the least squares algorithm [5], and sends the sequences representing the local best trees to the master, which selects the global best trees among the local best trees. This approach significantly reduces the communication. Each message consists of only a few integers. Moreover, this approach allows us to easily control the size of the task assigned to a slave. The master process controls the work load of every slave. This is important in balancing the system load.

3. Load Balancing

In distributed computing, load balancing is critical. In our implementation, load balancing is achieved by two techniques: dynamic job distribution and redundant computation.

3.1 Dynamic Job Distribution

A typical PVM environment includes a heterogeneous cluster of workstations. The characteristics of such environment are: the speed of machines varies widely; the load of system changes dynamically; machines may go down unpredictably.

Obviously, static job allocation, i.e., all tasks are allocated to slaves before execution, is inefficient. Thus we use dynamic job distribution in our implementation. Specifically, the master process spawns one slave process for each machine at a time and distributes one task for each slave process at a time. When a slave process finishes its task, it sends the results along with its process id to the master. Upon receiving the results and process id, the master process picks an unfinished task and sends it to the slave process which just sent the results.

Dynamic job distribution effectively "steals" CPU time from machines, since idle machines get more tasks than heavily loaded ones. Consequently, the system load is balanced. Moreover, since each machine runs only one slave process at a time, the degradation of the performance of the slave machines with time-sharing systems, such as UNIX, is minimized.

3.2 Redundant Computation

Another technique used for load balancing is called redundant computation. It is used to deal with the case when some machines on the network are much slower than others. In our case, the fastest machine is 200 times faster than the slowest one. Thus, the slowest machine may finish much later than others.

The idea of redundant computation is as follows. If the master receives the results from a slave after it sends out the last task, instead of freeing up this slave and waiting for other slaves to finish their tasks, it finds a task still running on another slave process and reassigns the task to the slave process just finished its task. Thus we have two slave processes running the same task. We call this technique redundant computation.

While the idea of redundant computation seems simple, the implementation is nontrivial. First, since there may be a group of slave processes running the same task, the master needs to free up all the processes in the same group when one of them finishes the task. Our solution is to keep a process-task table consisting of slave process id's along with their corresponding tasks represented by branch addition sequences. Second, it is possible that two or more slave processes in the group running the same task may finish the task at almost the same time. So, the master may receive duplicated results of the same task. This problem can be solved by the process-task table described above. When the master receives the results of a task, it immediately removes all slave processes associated with this task from the table. Thus, if duplicated results arrive, the master will fail to find the process associated with the task in the table and it will simply discard the results.

In addition to load balancing, redundant computation improves the scalability. We can always improve the performance by adding more machines, since there is no need to wait for the slowest machine to finish.

4. Fault Tolerance

In distributed computing, it is not unusual that some machines go down during the computation. Therefore, fault tolerance is necessary. In our

implementation, fault tolerance is ensured by the combination of dynamic job distribution and redundant computation.

The implementation is as follows. The master process first spawns one slave process for each available machine and sends one task to each slave process. Then it waits for the first slave process to finish its task. When the master process receives the results, it sends another task to this slave process. This task can be either a new task or a task which has been sent to another slave but not finished. Fault tolerance is ensured because if a machine goes down, the slave process on that machine will never finish its task. Consequently, the master will not receive results from that slave process and it will eventually send this unfinished task to another slave process.

Finally, for the efficiency on the machine running the master process, we block the master process when it waits for the results from a slave process. The master process is awoke by arriving results from a slave process. So the master process passively monitors the slave processes and takes very little CPU time. That enables us to spawn a slave process on the master machine to execute tasks.

5. Example

We implemented the parallel matrix decomposition algorithm in PVM [4] using a heterogeneous cluster of 18 workstations: six Sun/4 machines, one Sun/3, one Ultra SPARC, one SPARC with 2 processors, five PCs, and four Silicon Graphics. The Ultra SPARC was the fastest with low load. The Silicon Graphics machines were also fast but with medium load. The Sun/3 machine was the slowest but with low load. All the rest were of medium speed and load.

Our problem was to construct the 12 best trees of 12 Chinese languages. The whole computation took about a week using the cluster of workstations described above. During the computation, 5 machines went down. In our case, each task consists of the inspection of $15 \times 17 \times 19 = 4845$ trees. The fastest Ultra SPARC computed 39791 tasks while the slowest Sun/3 computed only 186 tasks. The master process sent out 135163 tasks of which 135135 tasks were finished by slave processes. The difference was due to the redundant computation.

6. Conclusions

In this paper, we present a parallel implementation of the matrix decomposition method for evaluating phylogenetic trees in [5]. We describe the techniques, namely dynamic job distribution and redundant computation, used in our implementation. These two techniques provide load balancing, improve scalability, and ensure fault tolerance. Although we use PVM in our implementation, these techniques can be applied to other distributed computing environment, especially in a heterogeneous cluster of workstations.

Acknowledgement

We would like to thank Chris Lesner for valuable discussions.

References

[1] Day, W. H. E. 1987. Computational Complexity of Inferring Phylogenies from Dissimilarity Matrices. *Bulletin of Mathematical Biology*, 49, pp. 46-467.

[2] Felsenstein, J. An Alternating Least Squares Approach to Inferring Phylogenies from Pairwise Distances. Manuscript.

[3] Fitch, W. and E. Margoliash. 1967. Construction of Phylogenetic Trees. *Science*, Vol. 155, No. 3760, pp. 279-284.

[4] Geist, Al, et al. 1994. PVM: Parallel Virtual Machine. The MIT Press, Cambridge, MA.

[5] Qiao, Sanzheng, J. W. Minett, and W. S-Y. Wang. 1996. Evaluating Phylogenetic Trees by Matrix Decomposition. Manuscript. Available at http://www.dcss.mcmaster.ca/~qiao/

[6] Saitou, N. and M. Nei. 1987. The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees. *Molecular Biology and Evolution*, 4(4), pp. 406-425.