A New Hybrid Parallel Algorithm for MrBayes

Jianfu Zhou, Gang Wang, and Xiaoguang Liu

Nankai-Baidu Joint Laboratory, Nankai University, Tianjin, China
jugeombu@gmail.com,wgzwp@163.com,liuxg74@yahoo.com.cn

Abstract. MrBayes, a popular program for Bayesian inference of phylogeny, has not been fast enough for Biologists when dealing with large real-world data sets. This paper presents a new parallel algorithm that combines the chain-partitioned parallel algorithm with the chain-parallel algorithm to obtain higher concurrency. We test the proposed hybrid algorithm with the two old algorithms on a heterogeneous cluster. The results show that, the hybrid algorithm actually converts more CPU cores into higher speedup compared with the two control algorithms for all of four real-world DNA data sets, therefore is more practical.

Key words: MrBayes; hybrid; parallel; algorithm

1 Introduction

MrBayes is a widely-used program for phylogenetic inference. It uses Bayes's theorem to estimate the posterior probability of a phylogenetic tree, which is called Bayesian inference of phylogeny [1, 2]. The posterior probability, although easy to formulate, involves a summation over all trees and, for each tree, integration over all possible combinations of branch lengths and substitution model parameter values. The explicit solution of such a problem is computationally intractable for trees of biologically significance, so heuristics must be used to simplify the problem. MrBayes uses Markov chain Monte Carlo (MCMC) method to approximate the posterior probability of a phylogenetic tree. Standard implementations of MCMC can be prone to entrapment in local optima, so a variant of MCMC, known as Metropolis-coupled MCMC [or $(MC)^3$ for short], is proposed. It allows peaks in the landscape of trees to be more readily explored. However, both the standard MCMC and the $(MC)^3$ methods are suffering from the unacceptable execution time when dealing with large data sets. Fortunately, some effective methods are available that can execute MrBayes in parallel. A chain-partitioned parallel algorithm for MrBayes [3] distributes Markov chains among processes. It is based on the fact that a relatively small amount of messages are needed to be exchanged among those chains during a run of $(MC)^3$. However, this algorithm has a fatal drawback. It can not break the upper bound of concurrency caused by the typically small number of chains during a run, which is enough for most of real-world applications. Another chain-parallel algorithm exclusively focuses on element-level parallelism in the Phylogenetic Likelihood Functions [5]. Although it solves the problem of the chain-partitioned algorithm, the chain-parallel algorithm has higher interaction overhead. This paper presents a new

hybrid parallel algorithm for MrBayes, which tries to combine the advantages of the chain-partitioned and chain-parallel algorithms. Then, this paper empirically compares the proposed algorithm with the two previous algorithms on four real-world DNA data sets.

2 The Hybrid Algorithm

2.1 The Chain-Partitioned Parallel Algorithm

Metropolis-coupled MCMC [or $(MC)^3$] algorithm runs one cold Markov chain with some heated Markov chains to sample the posterior probability distribution of a phylogenetic tree [2, 4]. Relatively small amount of information is exchanged among these chains during a run. So we can distribute these chains among processes, and run them in parallel efficiently [3]. A process just performs all computation associated with chain(s) assigned to it. This algorithm exchanges heat values instead of complete state information between cold chains and heated chains to reduce communication overhead. Another advantage of this algorithm is only local instead of global synchronization is needed.

Considering its data partition method and synchronization mechanism, the chain-partitioned parallel algorithm is very suitable for Message-Passing Interface (MPI) implementing. Of course it can also be implemented in shared memory platforms (multi-core systems) easily and efficiently.

2.2 The Chain-Parallel Algorithm

According to the profiling of MrBayes's execution, the functions *CondLikeDown*, *CondLikeRoot* and *CondLikeScaler* spend more than 85% of the total execution time. Fortunately, the main part of each function is independent vector/matrix operations. So a chain-parallel algorithm that executes these operations by multiple threads in parallel is presented [5]. In this algorithm, the likelihood vector elements (therefore the associated computational work) are distributed over threads evenly. During a run of MrBayes, these three functions are called iteratively. Since the results of *CondLikeDown* or *CondLikeRoot* are used by *CondLikeScaler* in each iteration, and the result of *CondLikeScaler* is used by the next iteration, a global synchronization must be done in each iteration which is the major overhead of this algorithm.

Apparently, the chain-parallel algorithm is more suitable for the shared-memory mode than the message-passing mode.

2.3 The Proposed Hybrid Parallel Algorithm

The proposed hybrid parallel algorithm combines the advantages of the chain-partitioned and the chain-parallel algorithms. First, the chain-partitioned strategy is used, that is, each chain is assigned to a unique process. Then each chain is

calculated by its owner process and auxiliary threads using chain-parallel strategy. For load balance, the number of chains (processes) assigned to a physical machine is approximately proportional to its relative computing power. Since cores in a machine are typically homogeneous, computational work in *CondLikeDown*, *CondLikeRoot* and *CondLikeScaler* of a chain is evenly distributed over its owner process and auxiliary threads.

In a word, the proposed algorithm breaks through the concurrency limit of the chain-partitioned algorithm caused by the relatively small number of Markov chains involved in MrBayes. It exploits two-level parallelism to make its concurrency greater than the total number of chains.

3 Experiments

This section compares the performance of the proposed hybrid parallel algorithm with the chain-partitioned and the chain-parallel algorithms on a heterogeneous cluster with a head node and five computation nodes. The head node has one Intel Core i7 920 processor and 2GB*3 of DDR3 1333 memory. The first computation node has one AMD Phenom II X4 945 processor and 2GB*2 of DDR3 1333 memory. The second one has one Intel Core 2 Quad Q6600 processor and 2GB*2 of DDR2 1066 memory. Both the third one and the fourth ones have one Intel Core 2 Duo E4400 processor and 1GB*2 of DDR2 800 memory. The fifth one has one Intel Core i7 920 and 2GB*3 of DDR3 1333 memory. These nodes are connected by gigabit Ethernet. The operating system is Red Hat Enterprise Linux 5. MPICH2 1.1 was used to compile and execute the hybrid algorithm and the chain-partitioned algorithm. For the chain-parallel algorithm program, GCC 4.3 was used. Table 1 shows the DNA data sets used in our experiments. These data sets come from real-world DNA data used in the research projects of the College of Life Sciences, Nankai University. Each data point is the average of 10 executions. Each execution computed 2 runs with 4 chains, which is a typical setting for MrBayes executions. All the executions used the 4by4 nucleotide substitution model and lasted 10000 generations.

Table 1. Data sets used in our experiments

	Number of taxa	Number of characters
Data set 1	26	1546
Data set 2	37	2238
Data set 3	100	1619
Data set 4	111	1506

We first performed a baseline test. As Fig. 1 on the next page shows, the original serial version of MrBayes runs fastest on the head node. In our experiments, the speedup of a parallel algorithm is defined as the result of the average running time of the original serial version of MrBayes on the head node divided by

the average running time of a parallel algorithm program on our heterogeneous cluster for the same problem.

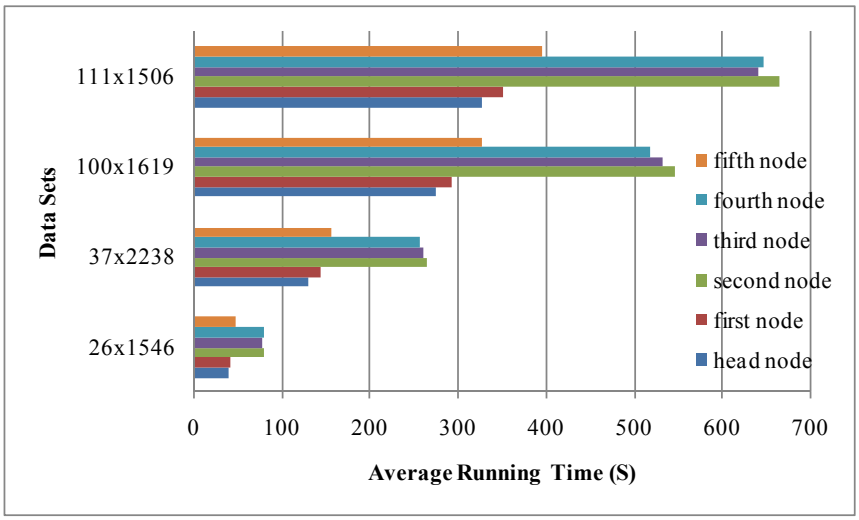

Fig. 1. Performance of different nodes.

3.1 Comparison on the same number of cores

Fig. 2 on the facing page shows the performance of the proposed hybrid parallel algorithm and the chain-parallel algorithm on the head node. For the chain-parallel algorithm, the likelihood vector elements were assigned among 8 threads (including the main thread). Therefore, 8 (logical) CPU cores were used. For the hybrid algorithm, the 8 Markov chains were distributed evenly among 4 processes. Then each process created a child thread. The likelihood vector elements were distributed to the process (namely the main thread) and its child thread. Hence the hybrid algorithm also used 8 cores.

Fig. 3 on the next page shows the performance of the proposed hybrid parallel algorithm and the chain-partitioned parallel algorithm. For the chain-partitioned algorithm, the 8 Markov chains were distributed evenly over 8 processes. 4 of these processes were assigned to the head node, and the remaining processes were distributed evenly between the first and the second nodes. For the hybrid algorithm, the 8 Markov chains were distributed evenly over 4 processes, of which 2 were assigned to the head node and the remaining were distributed evenly between the first and the second nodes. Then each process created a child thread and the likelihood vector elements were assigned between the process and its child thread evenly. Therefore, both algorithms used 8 cores.

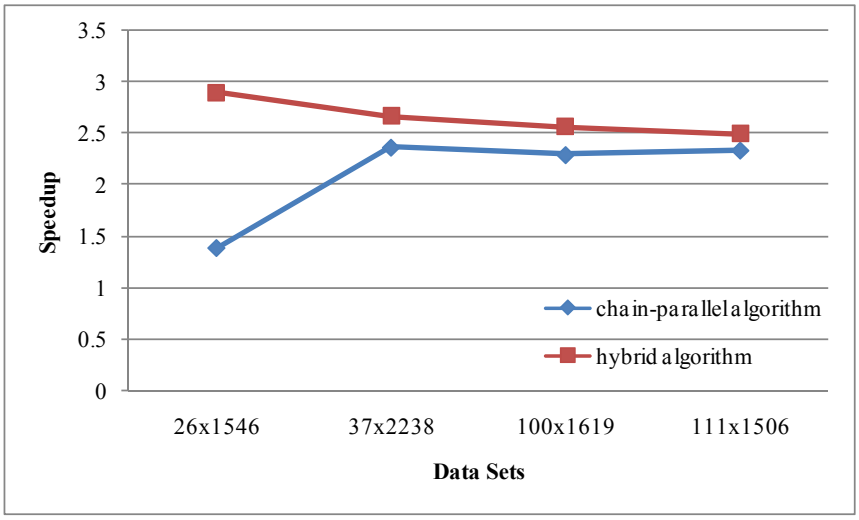

Fig. 2. The hybrid algorithm vs. the chain-parallel algorithm on a single node.

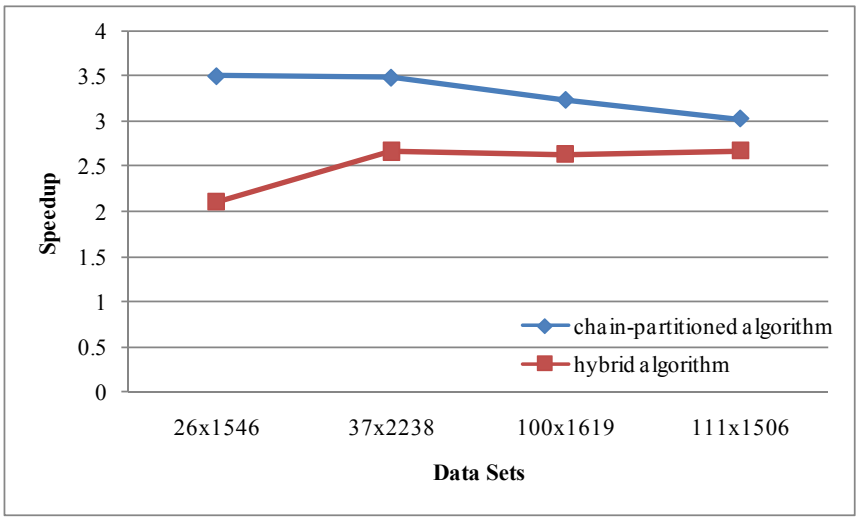

Fig. 3. The hybrid algorithm vs. the chain-partitioned algorithm on a cluster.

We can see that the performance of the hybrid algorithm is between the chain-parallel algorithm and the chain-partitioned algorithm. This result is expected. The global synchronization in element-level parallelism causes much higher overhead than the local synchronization in chain-level parallelism. The hybrid algorithm mixes the two parallel formulations, therefore has mixed synchronization, and the other two both have pure synchronization pattern.

3.2 The Hybrid Algorithm on more CPU cores

Although the chain-partitioned algorithm performs slightly better than the hybrid algorithm in the previous test, as mentioned above, it can not break through the concurrency limit caused by the total number of Markov chains involved. In our experiments, the total number of chains was 8. When the chain-partitioned algorithm assigned all the chains to 8 processes, it reached its maximum concurrency. There would be no more performance increase even if more processors are available. By contrast, the hybrid algorithm does not have this problem. Even if it has distributed the 8 chains among 8 processes, it can use the element-level parallelism to improve its concurrency. Moreover, the number of the likelihood vector elements is generally large. In our experiments, to achieve higher concurrency, we distributed all the chains among 8 processes, of which 4 were assigned to the head node and the remaining were distributed evenly between the first and the second nodes. Then each process created a child thread. The elements were distributed between the process and its child thread. The hybrid algorithm used 16 threads in total, thus reached the maximum concurrency (namely 16 cores) provided by these nodes. Fig. 4 on the facing page shows that, the hybrid algorithm indeed uses extra CPU cores effectively and gains higher speedup than the chain-partitioned algorithm.

3.3 Load Balance for the Hybrid Parallel Algorithm

As Fig. 1 on page 4 shows, inside our heterogeneous cluster, the computing power of each node is different with each other. Without considering the difference, it will result in load imbalance and therefore a poor performance.

The relative computing power of cores on nodes is about:

$$Head : First : Second : Third : Fourth : Fifth = 2 : 2 : 1 : 1 : 1 : 2 \ . \qquad (1)$$

Comparison on the same number of cores. Fig. 5 on page 8 shows the performance of the hybrid algorithm without or with load balance on the same number of cores. As a control group, the hybrid algorithm without load balance just assigned the 8 Markov chains evenly among 8 processes, which were distributed evenly to the head, the first, the second and the fifth nodes. Then each process created a child thread. The likelihood vector elements were distributed to the process and its child thread. Hence the computation associated with one chain was performed by two threads (namely a process and its child thread),

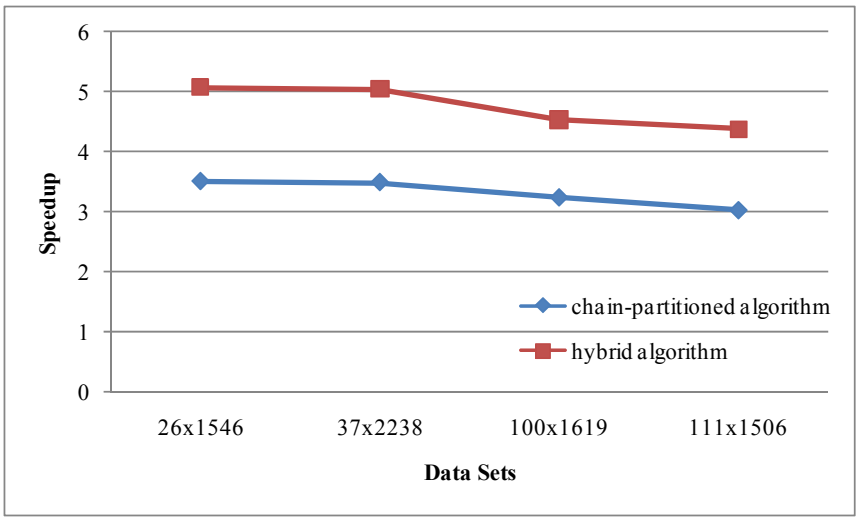

Fig. 4. High concurrency of the hybrid algorithm.

both of which used one core respectively. So, 16 cores were used. We denote the "relative load" (load divided by computing power - roughly the equivalent of the running time) of node X by R_X:

$$\begin{aligned} & R_{head} : R_{first} : R_{second} : R_{fifth} \\ = & \frac{2\ chain}{8\ power} : \frac{2\ chain}{8\ power} : \frac{2\ chain}{4\ power} : \frac{2\ chain}{8\ power} \\ = & 1 : 1 : 2 : 1\ , \end{aligned} \tag{2}$$

which implies serious load imbalance.

The hybrid algorithm with load balance also assigned the 8 Markov chains among 8 processes. Considering the computing power of one core on each node, 3 of the 8 processes were assigned to the head node, 1 to the second node, and the remaining were divided evenly between the first and the fifth nodes. Each of the processes assigned to the head node, the first node and the fifth node just created a child thread respectively. And the process assigned to the second node created 3 child threads. For all of the four nodes, the likelihood vector elements were distributed to the process and its child thread(s). Therefore, for the head node, the first node and the fifth node, the computational work associated with one chain was performed by two threads, both on one core respectively. For the second node, the computation associated with one chain was performed by four threads, each on one core respectively. Note that, although the head node has only 4 physical cores, it is arguably that each thread run on unique physical core since hyper-threading is supported by Intel i7 CPU. Therefore, this test also

used 16 cores, and:

$$\begin{aligned} & R_{head} : R_{first} : R_{second} : R_{fifth} \\ = & \frac{3\ chain}{12\ power} : \frac{2\ chain}{8\ power} : \frac{1\ chain}{4\ power} : \frac{2\ chain}{8\ power} \\ = & 1:1:1:1\ , \end{aligned} \tag{3}$$

which implies approximate load balance.

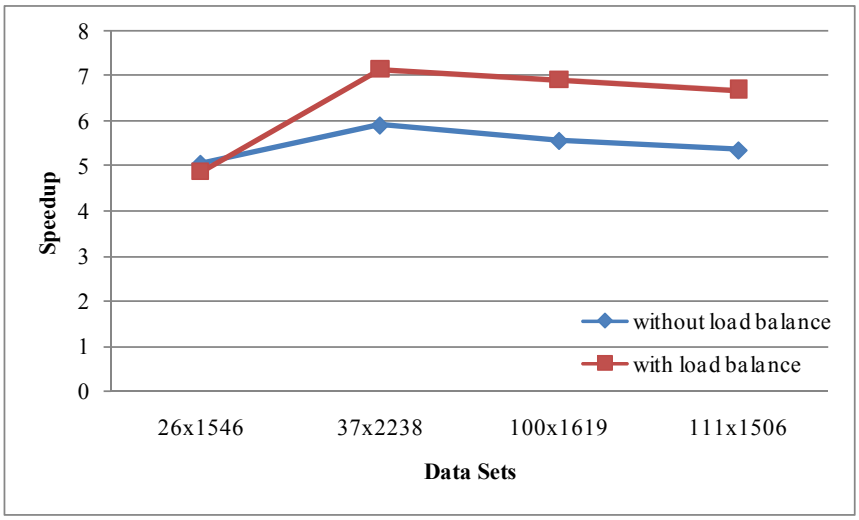

Fig. 5. Performance of the hybrid algorithm with or without load balance on the same number of cores.

The result shown in Fig. 5 verifies our analysis. Load balance strategy actually achieves a higher speedup. In particular, when dealing with the relatively large data sets (i.e. data set 3 and 4), load balance strategy increases performance by about 25%.

Comparison on different numbers of cores. Fig. 6 on the facing page shows the speedup of the hybrid algorithm with or without load balance on different numbers of cores. For the hybrid algorithm without load balance, the 8 Markov chains were assigned evenly among 8 processes, of which 4 were distributed to the head node, and the remaining were divided evenly between the first and the second nodes. Then each process created a child thread. And the likelihood vector elements were distributed to the process and its child thread. So the computational work associated with each chain was performed by two threads, both of which used one core respectively. Hence 16 cores were used, and:

$$R_{head} : R_{first} : R_{second} = \frac{4\ chain}{16\ power} : \frac{2\ chain}{8\ power} : \frac{2\ chain}{4\ power} = 1:1:2\ , \tag{4}$$

which implies serious load imbalance.

For the hybrid algorithm with load balance, the 8 Markov chains were divided into 3 groups. The first group included 4 chains, while the remaining two groups included 2 chains respectively. Then the first group was assigned to the head node, while one of the remaining two groups was distributed to the first node, and the other to the second node. On the head node, the 4 chains were assigned evenly between 2 processes; on the first node, the 2 chains were assigned to 1 process; on the second node, the 2 chains were assigned evenly between 2 processes. And all of these processes created a child thread respectively. The likelihood vector elements were distributed to the process and its child thread. Therefore, on the head node and the first node, each thread group (a process and its auxiliary thread) calculated 2 chains in sequential; on the second node, each thread group calculates only one chain. Each thread run on a unique real core. So the hybrid algorithm with load balance only used 10 cores in total, which was less than the number of cores used by the hybrid algorithm without load balance. And:

$$R_{head} : R_{first} : R_{second} = \frac{4\ chain}{8\ power} : \frac{2\ chain}{4\ power} : \frac{2\ chain}{4\ power} = 1 : 1 : 1 \ , \quad (5)$$

which implies rough load balance.

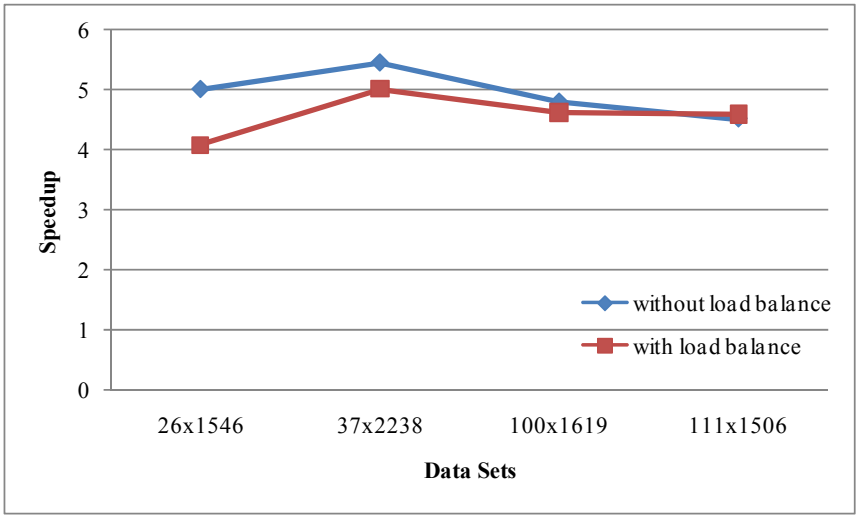

Fig. 6. Performance of the hybrid algorithm with or without load balance on different numbers of cores.

As Fig. 6 shows, although using less physical cores, the hybrid algorithm with load balance achieved approximately the same speedup as the hybrid algorithm without load balance. When dealing with the relatively large data set (i.e.

data set 4), the hybrid algorithm with load balance achieves even a little higher speedup than the hybrid algorithm without load balance.

All in all, with load balance, the proposed hybrid parallel algorithm can yield better performance.

4 Related Works

As far as the authors know, except PBPI [7], that conducts multigrain Bayesian inference on the BlueGene/L, no result has been published on hybrid parallelization for MrBayes. PBPI basically represents a proof-of-concept work rather than a production level parallelization. It is not qualified for real-world analyses required urgently by Biologists. However, some results are known for the chain-partitioned parallelization or the chain-parallel parallelization for MrBayes, which both accelerate the execution of MrBayes for large data sets.

Gautam Altekar et al. [3] presented a parallel algorithm for Metropolis-coupled MCMC. This algorithm keeps the advantage to explore multiple peaks in the posterior distribution of trees while getting a shorter running time. This algorithm was implemented using both message passing parallel programming model and shared memory parallel programming model. Experiment results showed speedups in both programming models for small and large data sets.

Frederico Pratas et al. [5] proposed a chain-parallel algorithm for MrBayes and its Phylogenetic Likelihood Functions using different architectures. The experiments compared the scalability and performance achieved using general-purpose multi-core processors, the Cell/BE, and Graphics Processor Units (GPU). The results showed that the general-purpose multi-core processors resulted in the best speedup, and yet GPU and Cell/BE processors both got poor performance because of data transfers and the execution of the serial portion of the code.

5 Conclusion and Future Work

A new hybrid parallel algorithm has been proposed for MrBayes. On our heterogeneous cluster, when using more processors (namely 16 processors), the proposed algorithm without load balance runs maximum 3.67 times faster than the chain-parallel algorithm and maximum 1.447 times faster than the chain-partitioned parallel algorithm on four read-world DNA data sets. With load balance strategy, a further up to 25% performance increment was achieved. Therefore, the proposed hybrid parallel algorithm is very practical for many real biological analyses.

In this paper, a static, manual load balance method was used. Dynamic and automatic load balance algorithms are worth studying in the future. GPU for general purpose computing has shown its great power in many areas. Accelerating MrBayes using GPU is also planned.

Acknowledgement. This work was supported in part by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), RFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000). In order to get the four real-world DNA data sets, in addition to the chance to do this research, we would like to thank Professor Xie Qiang with the College of Life Sciences in Nankai University.

References

1. Huelsenbeck, J.P., Ronquist, F., Nielsen, R., Bollback, J.P.: Bayesian inference of phylogeny and its impact on evolutionary biology. Science 294, 2310–2314 (2001)
2. Huelsenbeck, J.P., Ronquist, F.: MrBayes: Bayesian inference of phylogenetic trees. Bioinformatics 17, 754–755 (2001)
3. Altekar, G., Dwarkadas, S., Huelsenbeck, J.P., Ronquist, F.: Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. Bioinformatics 20, 407–415 (2004)
4. Ronquist, F., Huelsenbeck, J.: MrBayes 3: Bayesian Phylogenetic Inference under Mixed Models. Bioinformatics 19, 1572–1574 (2003)
5. Pratas, F., Trancoso, P., Stamatakis, A., Sousa, L.: Fine-grain Parallelism Using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function. In: The 38th International Conference on Parallel Processing, pp. 9–17. Vienna, Austria (2009)
6. Stamatakis, A., Ott, M.: Load Balance in the Phylogenetic Likelihood Kernel. In: The 38th International Conference on Parallel Processing, pp. 348–355. Vienna, Austria (2009)
7. Feng, X., Cameron, K.W., Buell, D.A.: PBPI: a High Performance Implementation of Bayesian Phylogenetic Inference. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, pp. 75. ACM Press, New York (2006)
8. Larget, B., Simon, D.: Markov chain Monte Carlo algorithms for the Bayesian analysis of phylogenetic trees. Molecular Biology and Evolution 16, 750–759 (1999)
9. Yang, Z., Rannala, B.: Bayesian phylogenetic inference using DNA sequences: a Markov chain Monte Carlo method. Molecular Biology and Evolution 14, 717–724 (1997)
10. Mau, B., Newton, M., Larget, B.: Bayesian phylogenetic inference via Markov chain Monte Carlo methods. Biometrics 55, 1–12 (1999)
11. Ott, M., Zola, J., Stamatakis, A., Aluru, S.: Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pp. 1–11. ACM Press, New York (2007)