MrBayes for Phylogenetic Inference Using Protein Data on a GPU Cluster

Shuai Pang, Rebecca J. Stones, Ming-ming Ren(✉), Gang Wang, and Xiaoguang Liu

College of Computer and Control Engineering, Nankai University, Tianjin, China
{pangshuai,rebecca.stones82,renmingming,wgzwp,liuxg}@nbjl.nankai.edu.cn

Abstract. MrBayes is a widely used software for Bayesian phylogenetic inference: we input biological sequence data from various taxonomic groups, and MrBayes returns its estimate of the phylogenetic tree which gave rise to those taxa. This paper presents ta(MC)^3, based on its predecessor a(MC)^3, which, for protein datasets, improves computational efficiency and overcomes major obstacles in analyzing larger datasets on HPCs with multiple Graphics Processing Units (GPUs). The major improvements are (a) a new task mapping strategy, (b) the use of Kahan summation to resolve non-convergence issues, and (c) the introduction of 64-bit variables. We evaluate ta(MC)^3 on real-world protein datasets both on a desktop server and the Tianhe-1A supercomputer. With a single GPU, ta(MC)^3 is nearly 90 times faster compared with the serial version of MrBayes, up to around 9 times faster than MrBayes utilizing a GPU via the BEAGLE library, and up to 2.5 times faster than a(MC)^3. On larger datasets with 64 nodes (GPUs) on Tianhe-1A, ta(MC)^3 is capable of obtaining 1000+ speedup vs. serial MrBayes.

Keywords: MrBayes · GPU · Protein · Task mapping strategy

1 Introduction

In biology, "phylogeny" refers to the evolutionary relationships between species or taxonomic groups, and can be inferred from the pattern of states at homologous characters in biological sequences; it is typically formulated as a phylogenetic tree. A number of numerical methods have been presented for using DNA or protein sequence data to infer phylogenetic trees, including parsimony methods [17], distance matrix methods [13], maximum probability methods [14], and the Bayesian method [7,8,12]. The Bayesian method outperforms other methods in terms of easy interpretation of results, the capability to incorporate prior information, and several computational advantages [6,16]. MrBayes is a popular program that implements the Metropolis Coupled Markov Chain Monte Carlo ((MC)^3 for short) sampling method for Bayesian phylogenetic inference.

MrBayes typically runs multiple Markov chains simultaneously, and since each chain runs almost independently, MrBayes is well-suited to parallel implementation on multi-core systems. To speed up Bayesian phylogenetic inference,

G. Wang et al. (Eds.): ICA3PP 2015, Part III, LNCS 9530, pp. 267–280, 2015.
DOI: 10.1007/978-3-319-27137-8_21

some parallel algorithms have been presented [1,15]. To our knowledge, g(MC)^3, by Pratas et al. [11], was the first attempt at parallelizing MrBayes on a GPU. The disadvantage of g(MC)^3 is that the transfer of transition probability matrices between the CPU and GPU is so frequent that it results in a large transfer overhead. Zhou et al. [18] subsequently presented n(MC)^3, an improved parallel version of MrBayes for the GPU. The authors decreased the frequency of transition probability matrix uploads and made both the CPU and GPU perform computations in parallel. This improvement results in overlap in CPU-GPU data communication and computation. A modified version of this GPU parallel algorithm, a(MC)^3 by Bao et al. [2] is a CPU-GPU cooperative algorithm which improves on n(MC)^3 such as by using dynamic task decomposition and mapping and a node-by-node scheduling model. Furthermore, BEAGLE, an open API library, has been designed to speed up probability calculations using the GPU. The BEAGLE library is now supported in MrBayes 3.2.1.

A GPU is a programmable many-core co-processor and has strong computational power and high memory bandwidth. Aside from graphics processing, many applications have been accelerated by using GPU, e.g. general signal processing, physics simulation, computational finance, and computational biology [3]. NVIDIA GPUs, for example, have a large number of cores grouped into *stream multiprocessors* (SMs, for short) which can run thousands of threads concurrently. CUDA, introduced by NVIDIA, is a general purpose parallel computing architecture and a parallel programming model. It allows developers to program NVIDIA GPUs using a minimally-extended version of the C language in a fashion very similar to CPU programming. The high-level unit of computation running on the GPU is called *kernel*, and is executed by potentially thousands of threads organized into *thread-blocks*, with all blocks able to be executed concurrently. A stream multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*; the threads in a block are divided into warps. A warp is the basic scheduling unit on GPU, and the threads in a warp must run synchronously on a SM. An important consideration of CUDA programming is memory hierarchy, there are three kinds of memory in NVIDIA GPUs: *global memory* is the largest memory space but the slowest; *shared memory* is much faster but much smaller; *registers* are the fastest, but smallest memory. Consequently, loading repeatedly accessed data into shared memory or registers is a common strategy to improve performance of GPU-based software.

To our knowledge, to date, a(MC)^3 obtains the greatest speedup among all GPU versions for MrBayes. Although originally focused on analyzing DNA data, a(MC)^3 has subsequently been extended to be capable of analyzing protein data. This paper presents an improvement upon a(MC)^3, which we call ta(MC)^3, which overcomes several obstacles to analyzing large protein datasets, particularly on high-performance computuational platforms:

– To improve computational efficiency when analyzing large protein datasets, ta(MC)^3 adopts an efficient task mapping strategy which makes better use of GPU cores and GPU memory and reduces redundant operations.

– When analyzing large datasets, due to the long inference process, MrBayes (and its parallel variants) might encounter non-convergence problems caused by large accumulating errors. To address this issue, we reformulate MrBayes's core computing flow by adopting the Kahan summation algorithm.
– We modify some data structures and processing logic to overcome inherent memory allocation limitations in MrBayes. With this improvement, ta(MC)^3 is capable of analyzing larger datasets than its predecessors.

2 Methods and Implementation

2.1 Optimizable Part of (MC)^3

The (MC)^3 procedure is outlined below (the details can be found in [1]). MrBayes runs H Markov chain simultaneously, and we let ψ_i denote the current tree for Markov chain i. Each is initialized randomly. For practical reasons, MrBayes also approximates the gamma distribution into several *categories*.

1. For each chain $i \in \{1, 2, \ldots, H\}$, we propose a new tree ψ_i' by randomly perturbing ψ_i, and replace ψ_i by ψ_i' with probability

$$R_i := \min\left[1, \left(\frac{f(X|\psi_i')}{f(X|\psi_i)} \times \frac{f(\psi_i')}{f(\psi_i)}\right)^{\beta_i} \times \frac{q(\psi_i)}{q(\psi_i')}\right].$$

   Here: (a) X denotes the input protein data, (b) $f(X|\psi_i)$ is the likelihood of data X occurring given tree ψ_i, (c) $f(\psi_i)$ is the probability of tree ψ_i, (d) $q(\psi_i')$ is the probability of proposing the new state, (e) the probability of proposing the old state starting from the new state is $q(\psi_i)$, and (f) β_i is the *heat* of chain i; for the cold chain, $\beta_i = 1$.
2. After all chains have advanced a given number of iterations, we randomly choose two chains (j and k) to swap states. We swap these states with probability

$$\min\left[1, \frac{f(\psi_k|X)^{\beta_j} f(\psi_j|X)^{\beta_k}}{f(\psi_j|X)^{\beta_j} f(\psi_k|X)^{\beta_k}}\right].$$

3. Go to Step 1.

A *transition probability matrix* gives the probabilities of transitioning from one amino acid to another. At a given site, the *conditional likelihood* is the likelihood of a node having a specific amino acid conditioned on its child nodes having their specific amino acids with their conditional likelihoods. Terminal-node conditional likelihoods are determined from the protein data. To calculate an acceptance probability R_i, we

1. calculate the transition probability matrices for every node according to the instantaneous transition probability matrix Q, then
2. traverse the tree in post order and compute the conditional likelihoods for each internal node according to its child nodes' conditional likelihoods.

The conditional likelihoods of the root node are used to calculate the local likelihoods of the tree (the likelihood at a given site), which combine to give the global likelihood. These computations are the most frequently called and time-consuming components of MrBayes.

The computation is performed as described in Algorithm 1, which implements Felsenstein's algorithm [4]. The 20×20 array TM_n denotes the transition probability matrix of an internal node n (in MrBayes, there are 20 possible amino acids). The vectors CL_n of length 20 are used to store the conditional likelihoods of node n.

Algorithm 1. Computing the conditional likelihoods in MrBayes.

```
1: for each internal node n, with child nodes l and r do
2:     for each site (or character) do
3:         for each Γ category do
4:             for each amino acid k = 0, 1, ..., 19 do
5:                 T_l ← 0
6:                 T_r ← 0
7:                 for m = 0, 1, ..., 19 do
8:                     T_l ← TM_l[k][m] × CL_l[m]
9:                     T_r ← TM_r[k][m] × CL_r[m]
10:                end for
11:                CL_n[k] ← T_l × T_r
12:            end for
13:        end for
14:    end for
15: end for
```

From a computational point of view, the main difference between DNA and protein data is the number of possible characters at a given site (4 nucleotides vs. 20 amino acids). Consequently, analyzing protein data on a GPU requires a different strategy than analyzing DNA data.

2.2 Task Mapping Strategy of a(MC)^3

In this section, we review a(MC)^3's method for computing conditional likelihoods on the GPU, described in Algorithm 2 and illustrated in Fig. 1. The data transfer method and chain scheduling method remain the same in ta(MC)^3, so we omit discussion of these components.

In a(MC)^3, each CUDA block is two dimensional with dimensions 4×20, and is responsible for computing the conditional likelihoods for an individual site in a non-terminal node. The threads (x, y), where threadIdx.$x \in \{0, 1, 2, 3\}$, together compute the conditional likelihood that the node has amino acid $k =$ threadIdx.y at that site. An individual thread (x, y) is responsible for the computation of 5 elements in this conditional likelihood for amino acid k. In a(MC)^3, a grid is

Algorithm 2. The contribution of thread (x, y) to the computation of the conditional likelihood $CL_n[k]$ for non-terminal node n and amino acid k in a(MC)^3; the node has child nodes l and r

$t_x \leftarrow$ threadIdx.x // we will have $t_x \in \{0, 1, 2, 3\}$
$t_y \leftarrow$ threadIdx.y // by design, $t_y = k$
Load TM_l, TM_r, CL_l, and CL_r into shared memory
Synchronize all threads
$T_l^{(x)}[t_y] \leftarrow 0$
$T_r^{(x)}[t_y] \leftarrow 0$
for $i = 0, 1, \ldots, 4$ **do**
    $T_l^{(x)}[t_y] \leftarrow T_l^{(x)}[t_y] + TM_l[t_y][5t_x + i] \times CL_l[5t_x + i]$
    $T_r^{(x)}[t_y] \leftarrow T_r^{(x)}[t_y] + TM_r[t_y][5t_x + i] \times CL_r[5t_x + i]$
end for
$T_l[t_y] \leftarrow \sum_{i=0}^{3} T_l^{(i)}[t_y]$
$T_r[t_y] \leftarrow \sum_{i=0}^{3} T_r^{(i)}[t_y]$
Synchronize all threads
if $t_x = 0$ **then**
    $CL_n[t_y] \leftarrow T_l[t_y] \times T_r[t_y]$
end if

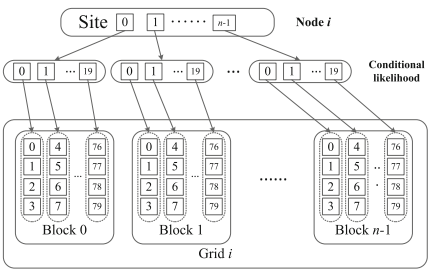

Fig. 1. Task mapping strategies for a(MC)^3.

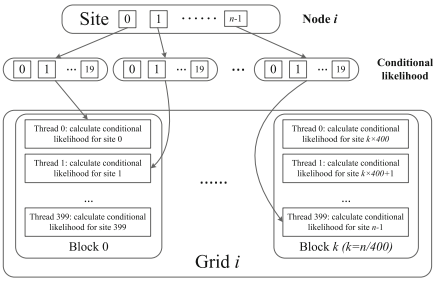

Fig. 2. Task mapping strategies for ta(MC)^3.

responsible for the computation of the conditional likelihoods over all sites of each non-terminal node, as indicated in Fig. 1.

We highlight two drawbacks in this algorithm:

- Since groups of four consecutive threads cooperatively compute a conditional probability, we require intermediate storage and calculation. In turn, this requires thread synchronization to ensure correct results are calculated.
- For large datasets, fine-grained task mapping results in a large number of small tasks. Therefore, the number of blocks may exceed the limits of NVIDIA Fermi architecture (i.e., at most $2^{16} - 1 = 65535$ blocks in a grid).
- Assigning small tasks results in each thread having low *arithmetic intensity*, defined as the ratio of the number of arithmetic instructions executed to the

number of memory accesses. Having low arithmetic intensity results in sub-optimal GPU utilization.

2.3 Proposed Task Mapping Strategy: ta(MC)^3

To address the aforementioned drawbacks of a(MC)^3, we design a coarse-grained task mapping strategy for ta(MC)^3, given in Algorithm 3 and illustrated in Fig. 2. In Algorithm 3, conditional likelihood vectors are stored in a matrix called the conditional likelihood table. Algorithm 4 gives the kernel function.

Algorithm 3. Outline of ta(MC)^3

```
1: Stage 1
2: for all Markov chains i do
3:     propose a new tree ψ'_i by randomly perturbing ψ_i
4:     for non-root nodes n do
5:         Calculate transition probability matrix TM_n
6:     end for
7: end for
8: Stage 2
9: Transpose conditional likelihood table
10: Transfer conditional likelihood table and TM_n from host memory to GPU memory
11: Stage 3
12: for all Markov chains i do
13:     for all nodes n do
14:         Call kernel function to calculate conditional likelihood of node n in
           chain i in parallel
15:     end for
16: end for
17: Stage 4
18: for all Markov chains i do
19:     Call kernel function to calculate the local likelihoods L_u for ψ'_i in parallel
20: end for
21: Stage 5
22: Transpose conditional likelihood table
23: Transfer conditional likelihood table from GPU memory to host memory
24: Transfer local likelihoods from GPU to CPU
25: Stage 6
26: for all Markov chains i do
27:     Synchronize corresponding GPU stream
28:     Calculate acceptance probability for ψ'_i using global likelihood
       ∏_{site u} L_u
29: end for
```

The proposed ta(MC)^3 utilizes the GPU memory hierarchy in two significant ways:

– We first make each block transfer the transition probability matrices TM_l and TM_r from global memory to shared memory. At a given node, the TMs are repeatedly used over all sites when calculating conditional likelihoods. Consequently, since each thread in ta(MC)^3 computes the conditional likelihoods for an individual site, TM is repeatedly accessed from shared memory by all threads in the same block.
– We use registers to store the conditional likelihoods of the current node's children and for the intermediate calculation as indicated in Algorithm 4.

In the end, the final result is written back to global memory.

Algorithm 4. The kernel function of ta(MC)^3 for computing conditional likelihoods

```
1: load TM_l and TM_r into shared memory
2: Synchronize all threads
3: for amino acid k = 0, 1, ..., 19 do
4:     register_var c_l ← CL_l[k]
5:     register_var c_r ← CL_r[k]
6:     register_var t_l ← 0
7:     register_var t_r ← 0
8:     for amino acid m = 0, 1, ..., 19 do
9:         t_l ← t_l + TM_l[m][k] × c_l
10:        t_r ← t_r + TM_r[m][k] × c_r
11:    end for
12:    CL_n[k] ← t_l × t_r
13: end for
```

We also coalesce global memory accesses by transposing the conditional likelihood table, as indicated in lines 9 and 22 in Algorithm 3. This has the benefit of aligning 32 consecutive 4-byte words into 128 bytes, which can be fetched from global memory in a single transaction. Without transposing this matrix, the threads in a warp load multiple discontinuous words from global memory.

Additionally, GPU utilization is linked to the number of simultaneously active warps. The time a warp needs to reach the next ready state is the *latency. Full utilization* occurs when latency is "hidden" by other warps. From the algorithm specifications, we can compute that about 40 warps are required to hide the latency under ta(MC)^3, whereas 240 warps are required under a(MC)^3. However, CUDA devices of compute capability 3.× allow no more than 64 simultaneous warps on a multiprocessor [9]. This implies that ta(MC)^3 is capable of achieving full utilization, whereas a(MC)^3 is incapable.

We set the block size in ta(MC)^3 to 400, which has the following benefits:

– The number of registers used by a kernel has a significant impact on the number of resident warps. In a CUDA device of compute capability 3.×, each multiprocessor has 65,536 registers. Since each thread uses up to 36 registers (using the `-maxrregcount` compiler option to control register usage)

in ta(MC)^3 and only very little shared memory are needed, the number of actual resident warps is determined by the number of registers used by kernel. When we set the block size to 400, each block consequently occupies $400 \times 36 = 14,400$ registers. Consequently, each multiprocessor can accommodate at most 4 blocks and there are 13 warps in a block, that is, 52 warps totally which is more than 40 warps needed to hide latency. This satisfies the requirement of full utilization.
– A block size of 400 matches the size of transition probability matrix, i.e., $20 \times 20 = 400$ cells, resulting in coalesced memory accesses and avoiding idle threads.

Experimental results agree with 400 being a good choice of block size; see Fig. 5.

2.4 Precision Optimization

Non-convergence problems can occur in MrBayes when we analyze protein data with too many taxa or amino acid sites (characters). The root cause of non-convergence is large truncation errors accumulating when analyzing larger datasets. Summation of a sequence of floating-point values is required in MrBayes. If we perform such summations naively, eventually non-negligible truncation errors will accumulate. In ta(MC)^3, we adopt the Kahan summation algorithm [5] to improve the precision. This is implemented on the GPU side when computing conditional likelihoods. Specifically, in Algorithm 4, we (a) declare new registers y, e_l, e_r, and t after Line 7, initializing e_l and e_r to zero, and (b) replace Line 9 with:

1: $y \leftarrow TM_l[m][k] \times c_l - e_l$
2: $t \leftarrow t_l + y$
3: $e_l \leftarrow (t - t_l) - y$
4: $t_l \leftarrow t$

A similar modification is made for Line 10.

Kahan summation minimizes truncation error by keeping a variable to account for truncation errors, which are corrected in the next iteration. Summing a sequence of n numbers naively has a worst-case error that grows proportional to n, whereas, with Kahan summation, the worst-case error is independent of n, depending on the floating-point precision of the machine.

2.5 A Memory Allocation Limitation

The length of real-world amino acid sequences can have tens of thousands or even hundreds of thousands of amino acids, which, along with the number of taxa studied, results in increased system memory requirements. Since the serial

version of MrBayes and its parallel versions are still use 32-bit variables, they are incapable of analyzing datasets of this size even if there is enough memory in the GPU devices. More specifically, if the dataset requires more than 4 GB memory per GPU to be analyzed, these versions of MrBayes will crash. To solve this problem, we use 64-bit variables in ta(MC)^3. The program component responsible for memory management is redesigned to deal with 64-bit addresses and manage large memory space. The specific variables changed are listed in Table 1.

Table 1. Variables changed to 64-bit.

Variable	Description
numCompressedChars	number of sites
condLikeRowSize	row size of CPT
globaloneMatSize	size of CPT
offsetclP, offsetclL, offsetclR, offsetclA	offset in CPT

As a specific example of this limitation, we include a real-world dataset in our experiments (dataset 7), which has 31 taxa and 360031 sites. At least 12.8 GB is required to store its conditional likelihood table, which is far beyond what a 32-bit variable can represent.

3 Experimental Results and Discussion

We evaluate the performance of ta(MC)^3 with eight real-world datasets. These datasets are used in phylogenetics research by Prof. Qiang Xie's research group and can be found in TreeBASE repository (both the datasets and source code are available from http://sourceforge.net/projects/mrbayes-gpu/). The dataset statistics are listed in Table 2.

The larger datasets (7 and 8) require substantially more memory and processing time. We use eight NVIDIA Titan cards in a GPU cluster to test the performance of ta(MC)^3 on these larger datasets. We test ta(MC)^3 both on a desktop server and the Tianhe-1A heterogeneous multi-core supercomputer. Each execution uses the same substitution model and lasts 100000 generations for datasets 1 to 6 and 10000 generations for datasets 7 and 8.

Table 2. Real-world protein datasets used in experiments

Dataset	1	2	3	4	5	6	7	8
No. taxa	32	85	8	48	59	39	31	14
No. characters	9377	13087	10088	11949	12428	11445	360031	407604

We compare run-times vs. MrBayes running in serial, and this is how we define "speedup" throughout. The most appropriate choice of baseline is not obvious: a(MC)^3 (which is the fastest known software prior to ta(MC)^3) was also developed by our research laboratory, and consequently cannot provide an independent baseline. Also, a(MC)^3 does not have the popularity of official versions of MrBayes.

In several experiments, the numerical value of the speedup is not important (e.g., to test scalability, or for block size optimization), and the speedup could, in principle, be normalized to the interval $(0, 1]$ without affecting the conclusions. Where the numerical value of the speedup is relevant, if the reader's preferred baseline differs from that used, they can readily divide through by its corresponding speedups (be that of a(MC)^3 or MrBayes + BEAGLE).

Desktop Server. We first test ta(MC)^3 on a desktop server with the following specifications: CentOS 6.2; 1× Intel Xeon E5645 (6 cores; 2.4 GHz); 6× 4 GB DDR3 1333 RAM; 8× NVIDIA GeForce GTX Titan. We compare its performance against a(MC)^3 and MrBayes 3.2.1 (the GPU parallel version of MrBayes which utilizes the BEAGLE library), and the serial version of MrBayes 3.1.2 is chosen as the baseline serial algorithm. Table 3 shows the experimental run-times on a desktop server on a single NVIDIA GeForce GTX Titan graphics card.

Table 3. Run-times of MrBayes 3.1.2, MrBayes 3.2.1, a(MC)^3, and ta(MC)^3 on datasets 1–6 in Table 2

Dataset	Execution time (sec.)				Speedup		
	MrBayes 3.1.2	MrBayes 3.2.1	a(MC)^3	ta(MC)^3	MrBayes 3.2.1	a(MC)^3	ta(MC)^3
1	40216	3212	1183	535	12	34	75
2	253953	17663	6194	2919	14	41	87
3	3744	401	267	117	9	14	32
4	57025	4320	1840	738	13	31	77
5	123656	8721	3171	1508	14	39	82
6	52650	4113	1423	675	13	37	78

Multi-GPU Hardware. Figure 3 shows the speedups on artificial datasets of ta(MC)^3 and a(MC)^3 on multiple GPUs on the desktop server. In this experiment, both algorithms distribute the eight Markov chains evenly between the GPUs as eight almost-independent processes.

Multi-GPU Hardware: Larger Datasets. Due to the use of 64-bit variables, we can analyze larger datasets using ta(MC)^3 than its 32-bit predecessors. The original MrBayes and its parallel variants will crash if set to analyze one of these larger datasets. To test ta(MC)^3 on these datasets, we compare its performance against modified versions of MrBayes 3.1.2 and a(MC)^3 in which 64-bit variables have

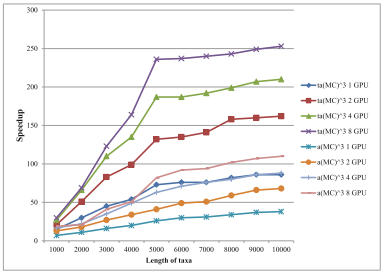

Fig. 3. Speedup on the multi-GPU desktop server.

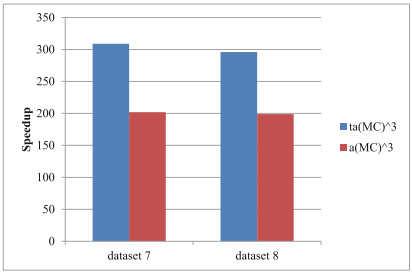

Fig. 4. Speedup of larger datasets on multi-GPUs.

been introduced. We use double precision floating point data type to reduce errors and improve precision. We also use eight NVIDIA GeForce GTX Titan graphics cards to analyze these datasets, since these datasets require more than 30 GB memory under double-precision. The results are plotted in Fig. 4.

Tianhe-1A Supercomputer. The Tianhe-1A (TH-1A) supercomputer was chosen as our experimental platform for testing ta(MC)^3 on a GPU cluster. TH-1A is one of the few petascale supercomputers in the world, located at the National Supercomputing Center in Tianjin, China. It once became the world's fastest supercomputer[1] with a peak performance of 2.507 petaflops in October 2010. TH-1A is now equipped with 2,048 NUDT FT1000 heterogeneous processors, 14,336 Xeon X5670 processors and 7,168 NVIDIA Tesla M2050 GPU cards. We conduct our experiments on 64 nodes of TH-1A, each of which is equipped with one NVIDIA Tesla M2050 GPU card. Here we use NVIDIA CUDA Toolkit Version 4.0 and GCC version 4.12.

Table 4. Speedup of ta(MC)^3 vs. MrBayes 3.1.2 on the Tianhe-1A supercomputer on datasets 1–8 in Table 2

Dataset	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1	117	165	264	361	424
2	139	186	290	381	451
3	50	69	108	188	288
4	123	168	269	333	425
5	128	177	283	359	451
6	122	169	271	334	425
7	–	201	378	672	1067
8	–	191	366	657	1039

[1] http://www.top500.org/lists/2010/11/.

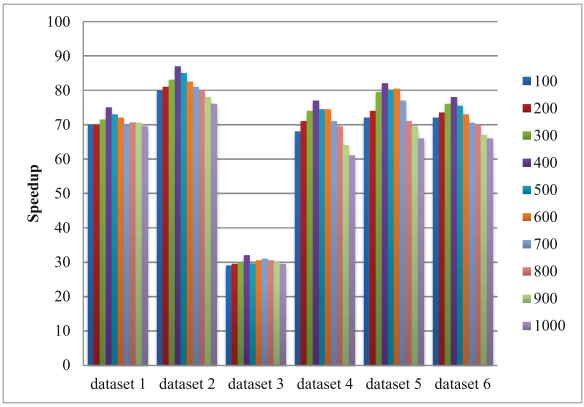

Fig. 5. Performance with various block sizes.

Table 5. Convergence speed (no. generations and run-time) with and without Kahan algorithm; the datasets are ordered as in the main text.

Dataset	With Kahan summation		Without Kahan summation	
	No. generations	Run-time (sec.)	No. generations	Run-time (sec.)
1	9156000	45000	9866000	50000
2	14581000	390000	15730000	410000
3	1997000	1900	2486000	2500
4	12235000	84000	14009000	100000
5	12142000	180000	12975000	190000
6	17067000	120000	18424000	130000

Experimental results on TH-1A are shown in Table 4 under single precision. The baselines are the runtimes of the modified version of MrBayes 3.1.2 with 64-bit variables on a single CPU of TH-1A. Since datasets 7 and 8 require more than 15GB memory under single precision and each GPU on TH-1A has 2G memory, we cannot analyze them with 4 nodes.

Block Size. We evaluate the impact of block size on performance of ta(MC)^3. The speedup vs. MrBayes 3.1.2 under varying block sizes is given in Fig. 5.

Kahan Summation. To evaluate the effectiveness of Kahan summation algorithm, we compare the convergence speed of ta(MC)^3 with and without Kahan summation algorithm. The convergence speed is measured in the generations and time spent in making the average standard deviation of split frequencies below 0.01. The results on datasets 1 to 6 are shown in Table 5.

4 Discussion

It is obvious from these experiments that GPU parallel algorithms significantly accelerate inference MrBayes. We further observe from Table 3 that ta(MC)^3 is around twice as fast as a(MC)^3 on single-GPU hardware. On multi-GPU hardware, we can see that ta(MC)^3 continues to outperform a(MC)^3, achieving, for example, up to 253 speedup vs. the serial algorithm on a quad-GPU configuration compared with 110 speedup of a(MC)^3. For the larger datasets, 7 and 8, ta(MC)^3 achieves around 300 speedup, while it would not be possible to analyze these data without modifying MrBayes or a(MC)^3.

On the Tianhe-1A supercomputer, Table 4 shows ta(MC)^3 is capable of achieving over 1000 speedup with 64 nodes on the larger datasets. The speedup of datasets 7 and 8 continues to increase as the number of GPU nodes grows to 64, whereas for the smaller datasets (datasets 1–6) the speedup tended to converge. For datasets 7 and 8, we can see that the efficiency (the ratio of speedup to the number of processors) is a relatively stable value (around 20). This result indicates that ta(MC)^3 is scalable and is capable of efficiently analyzing larger protein datasets on larger parallel systems.

In Fig. 5, we see that when ta(MC)^3 is assigned small increasing block sizes, the speedup increases, which we attribute to the arithmetic intensity increasing. However, once block size exceeds a certain value, the speedup curve goes down, which we attribute to the concurrency then dominating the performance and decreasing gradually. We can see that the peak speedup happens around when the block size is set to 400, which agrees with the theoretical analysis.

The introduction of Kahan summation had a surprisingly large impact (see Table 5); this results in the same quality of results in far fewer iterations compared with ta(MC)^3 without Kahan summation.

5 Conclusion

In this paper, we present a modified parallel version of MrBayes, called ta(MC)^3, designed to analyze large protein datasets efficiently on GPUs. The major improvements are: (a) a new task mapping strategy that can reduce intermediate computations and storage, (b) the introduction of Kahan summation to eliminate accumulating roundoff error, and (c) the introduction of 64-bit variables to make analyzing larger datasets possible.

In comparison to other GPU-accelerated versions of MrBayes, we see a runtime improvement of around 9 times faster than MrBayes + BEAGLE, and 2.5 times faster than a(MC)^3 (the predecessor of ta(MC)^3). With eight GTX Titan cards, ta(MC)^3 is nearly 300 times faster than serial MrBayes, and on a 64-node GPU cluster, it achieves over 1000 speedup vs. serial MrBayes.

Acknowledgements. A biology-focused version of this paper has been published [10]. This work is partially supported by NSF of China (grant numbers: 61373018, 11301288), Program for New Century Excellent Talents in University (grant number:

NCET130301) and the Fundamental Research Funds for the Central Universities (grant number: 65141021). Stones was supported by her NSF China Research Fellowship for International Young Scientists (grant number: 11450110409). We would also like to thank Hongju Xia, Jianfu Zhou, Jie Bao and Prof. Qiang Xie for their valuable input.

References

1. Altekar, G., Dwarkadas, S., Huelsenbeck, F., Ronquist, J.P.: Parallel metropolis coupled markov chain monte carlo for bayesian phylogenetic inference. Bioinformatics **20**, 407–415 (2004)
2. Bao, J., Xia, J., Zhou, J., Liu, X.G., Wang, G.: Efficient implementation of MrBayes on multi-GPU. Mol. Biol. Evol. **30**, 1471–1479 (2013)
3. Farber, R.: CUDA Application Design and Development. Morgan Kaufmann, San Francisco (2011)
4. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol. **17**, 368–376 (1981)
5. Kahan, W.: Pracniques: further remarks on reducing truncation errors. Commun. ACM **8**(1), 40 (1965). http://doi.acm.org/10.1145/363707.363723
6. Larget, B., Simon, D.L.: Markov chain monte carlo algorithms for the bayesian analysis of phylogenetic trees. Mol. Biol. Evol. **16**, 750–759 (1999)
7. Li, S., Pearl, D.K., Doss, H.: Phylogenetic tree construction using markov chain monte carlo. J. Am. Statist. Assoc. **95**, 493–508 (2000)
8. Mau, B., Newton, M.A.: Phylogenetic inference for binary data on dendrograms using markov chain monte carlo. J. Comp. Graph. Stat. **6**, 122–131 (1997)
9. NVIDIA: CUDA C Programming Guide (2013)
10. Pang, S., Stones, R.J., Ren, M.M., Liu, X.G., Wang, G., Xia, H., Wu, H.Y., Liu, Y., Xie, Q.: GPU MrBayes v3.1: GPU MrBayes on graphics processing units for protein sequence data. Mol. Biol. Evol. **32**(9), 2496–2497 (2015)
11. 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: 42nd International Conference on Parallel Processing, pp. 9–17 (2009)
12. Rannala, B., Yang, Z.: Probability distribution of molecular evolutionary trees: a new method of phylogenetic inference. J. Mol. Evol. **43**, 304–311 (1996)
13. Saitou, N., Nei, M.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol. **4**, 406–425 (1987)
14. Schmidt, H., Strimmer, K., Vingron, M., Haeseler, A.: Tree-puzzle: maximum likelihood phylogenetic analysis using quartets and parallel computing. Bioinformatics **18**, 502–504 (2002)
15. Thuiller, W., Lavergne, S., Roquet, C., Boulangeat, I., Lafourcade, B., Araujo, M.B.: Parallel algorithms for bayesian phylogenetic inference. J. Parallel Distrib. Comput. **63**, 707–718 (2003)
16. Xie, Q., Bu, W., Zheng, L.: The bayesian phylogenetic analysis of the 18s RNA sequences from the main lineages of trichophora (insecta: Heteroptera:pentatomomorpha). Mol. Biol. Evol. **34**, 448–451 (2005)
17. Yang, Z.: Phylogenetic analysis using parsimony and likelihood methods. J. Mol. Evol. **42**(2), 294–307 (1996)
18. Zhou, J., Liu, X.G., Stones, D.S., Xie, Q., Wang, G.: MrBayes on a graphics processing unit. Bioinformatics **27**, 1255–1261 (2011)