Configuring Concurrent Computation of Phylogenetic Partial Likelihoods: Accelerating Analyses Using the BEAGLE Library

Daniel L. Ayres(✉) and Michael P. Cummings(✉)

Center for Bioinformatics and Computational Biology,
University of Maryland, College Park, MD 20742, USA
{ayres,mike}@umiacs.umd.edu

Abstract. We describe our approach in augmenting the BEAGLE library for high-performance statistical phylogenetic inference to support concurrent computation of independent partial likelihoods arrays. Our solution involves identifying independent likelihood estimates in analyses of partitioned datasets and in proposed tree topologies, and configuring concurrent computation of these likelihoods via CUDA and OpenCL frameworks. We evaluate the effect of each increase in concurrency on throughput performance for our partial likelihoods kernel for a four-state nucleotide substitution model on a variety of parallel computing hardware, such as NVIDIA and AMD GPUs, and Intel multicore CPUs, observing up to 16-fold speedups over our previous implementation. Finally, we evaluate the effect of these gains on an domain application program, MrBayes. For a partitioned nucleotide-model analysis we observe an average speedup for the overall run time of 2.1-fold over our previous parallel implementation, and 10-fold over the native MrBayes with SSE.

Keywords: Bayes methods · Biology computing · Evolution (biology) · Phylogeny · Maximum likelihood estimation · Multicore processing · Parallel programming · High performance computing

1 Introduction

The most effective methods for inferring phylogenetic trees are based on either maximum likelihood estimation or Bayesian analysis, which share the same computational bottleneck: calculation of the likelihood of trees [7]. When profiling GARLI [11], a leading phylogenetic inference program, we have observed that, for nucleotide models, likelihood related calculations typically constitute over 94% of the overall run time. For more complex models (e.g., amino-acid or codon-based), likelihood calculation will typically incur an even greater proportion of the analysis time. Speeding the calculation of the likelihood function is key to increasing the performance of statistical inference-based phylogenetic analyses.

The core likelihood calculations apply to a subtree comprising a parent node, k, two child nodes, ℓ and m, and connecting branches of length, t_ℓ and t_m, and

S. Ibrahim et al. (Eds.): ICA3PP 2017, LNCS 10393, pp. 533–547, 2017.
DOI: 10.1007/978-3-319-65482-9_39

is repeated for all such subtrees within the larger tree being considered. This partial likelihood function is as follows [7]:

$$L_k^{(i)}(z) = \left(\sum_x \Pr(x|z, t_\ell) L_\ell^{(i)}(x) \right) \times \left(\sum_y \Pr(y|z, t_m) L_m^{(i)}(y) \right) \quad (1)$$

This calculation is repeated for each character i in the data (i.e., sequence site pattern), for each state z that a character can assume, and for each internal node in the proposed tree. The computational complexity of the likelihood calculation for a given tree is $O(p \times s^2 \times n)$, where p is the number of patterns in the sequence (typically on the order of 10^2 to 10^6), s is the number of states each character in the sequence can assume (typically 4 for a nucleotide model, 20 for an amino-acid model, or 61 for a codon model), and n is the number of operational taxonomic units (e.g., species, alleles). Additionally the tree search space is very large; the number of unrooted topologies possible for n operational taxonomic units is given by the double factorial function $(2n-5)!!$ [6]. Thus, to explore even a fraction of the total search space, a very large number of topologies are evaluated, and hence a very great number of likelihood calculations have to be performed. This leads to analyses that can take days, weeks or even months to run. Further compounding the issue, rapid advances in the collection of DNA sequence data have made the limitation for biological understanding of these data an increasingly computational problem.

1.1 The BEAGLE Library and API

The BEAGLE library and API [2] is a high-performance likelihood-calculation platform for evolutionary models. It defines a uniform application programming interface (API) and includes a collection of efficient implementations for calculating a variety of likelihood-based models on different hardware devices, such as graphics processing units (GPUs) and multicore central processing units (CPUs).

The BEAGLE library was designed to support a variety of hardware-specific implementations, each optimized for a different processor type. The library includes a set of parallel computing implementations that use the CUDA and OpenCL external computing frameworks.

The BEAGLE library has been very successful in accelerating evolutionary analyses. The library has been integrated into the most recent versions of popular phylogenetics software including BEAST [5], MrBayes [10], and PhyML [8], and has been widely used across a diverse range of evolutionary studies.

Previously, given the fine-scale parallelization of the phylogenetic likelihood function in the BEAGLE library, the problem with few sequence patterns, or one broken into small data subsets, was always *small*, and thus generally not amenable to speedups, as patterns (for a given model type and category rate count, e.g., nucleotide with four distinct rates) were the only dimension being parallelized.

In this paper we describe our recent work to configure concurrent computation of phylogenetic likelihoods by exploiting additional independent calculation

opportunities. The result is that a wider variety of analyses benefit from parallel computing performance gains.

1.2 Concurrent Computation: Independent Likelihood Estimates

We have focused on the following opportunities for concurrent computation of phylogenetic likelihoods that were previously unrealized in BEAGLE.

Pattern Partitions. Evolutionary analyses benefit from increases in modeling flexibility. One clear way of improving model flexibility is to allow independent estimation of model parameters for different character data subsets (e.g., genes, codon positions). This is typically referred to as a partitioned model and is a technique available in all phylogenetic software packages that support BEAGLE. Until now partitioned analyses with BEAGLE have required the client program to create multiple instances of the library, one for each data subset defined by the partitioning scheme. When BEAGLE instances share a hardware resource they are executed in sequence, thus incurring significant performance and memory inefficiencies, specially for problems with a large number of small data subsets.

Independent Subtrees. The number of subtrees requiring calculation for any full tree is $n - 1$, where n is the number of operational taxonomic units (e.g., species, alleles), which is the number of tips (leaves) on the tree. Phylogenetic algorithms typically use a post-order traversal when calculating tree likelihood, calculating each of the $n-1$ subtrees in series. In the case of a fully pectinate tree no subtrees are independent (Fig. 1, left). However, in the case of more balanced topologies there are independent subtrees (Fig. 1, middle). The likelihoods for sets of these independent subtrees can be calculated concurrently. In order to more easily realize potential concurrency related to independent subtrees present in a given topology, partial likelihood arrays need to be processed according to a reverse level-order, or breadth-first, traversal of the tree being evaluated. In the case of a fully balanced tree the number of independent subtrees is maximized, and partial likelihood calculations can be done in sets of concurrent operations corresponding to the number of levels in the tree, $\lceil log_2\, n \rceil$ (Fig. 1, right). This exploit of tree level-group concurrency is somewhat similar to a classic parallel reduction scheme.

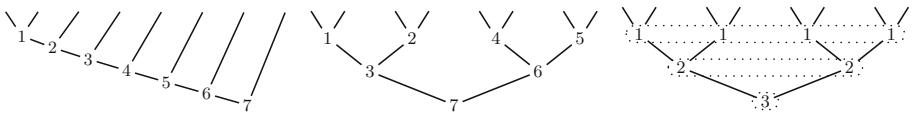

Fig. 1. Example pectinate tree (*left*), and example of a fully balanced tree (*middle*); with sequential calculation both trees require $n - 1 = 7$ partial likelihood operations in series, corresponding to the order of the node numbers. Balanced tree (*right*) with concurrent computation requiring $\lceil log_2\, n \rceil = 3$ sets of independent partial likelihood operations in the order of the shared node numbers.

2 Methods

2.1 Benchmarking and Testing

Our approach to increase concurrency in BEAGLE has been focused on the partial likelihoods kernel that is the computational bottleneck for phylogenetic analyses. To evaluate the performance of this function we used our test program (*genomictest*), which generates random synthetic datasets of arbitrary sizes. This test program is included with the BEAGLE source code and the results shown throughout this paper can be reproduced by using the default random seed, 1.

Table 1. System specifications

	System 1	*System 2*
CPU(s)	Intel Core i7-930	Dual Intel Xeon E5-2680v4
GPU(s)	AMD Radeon R9 Nano	AMD FirePro S9170
	NVIDIA Quadro P5000	
Linux kernel	4.8.13	3.10.0
GCC version	6.2.1	6.2.0
CUDA release	8.0	—
OpenCL drivers	AMD 1912.5	AMD 1800.8
	NVIDIA 375.26	Intel 1.2.0

We report a measure of throughput in terms of the effective number of floating point operations per second (GFLOPS) for computation of the partial likelihoods function (see Eq. 1). In contrast to a direct timing benchmark, throughput allows us to more easily compare performance across different problem sizes. We report benchmark results for two system configurations (Table 1). For conciseness, many results are shown only for the two best performing platforms we had available, the NVIDIA Quadro P5000 GPU under CUDA and the AMD Radeon R9 Nano GPU under OpenCL. Further comparisons across hardware platforms and frameworks are reported elsewhere [1].

2.2 Pattern Partition Concurrency

Multiple versus Single Library Instances. An initial design goal for the BEAGLE library was to make a library instance relatively light-weight, and to leave it up to the client program to manage these instances. This design objective was fitting for processors at the time, because it was easier to achieve good saturation as the number of cores and supported threads for CPUs and GPUs were modest compared to recent processors. However, we have found that this light-weight model is limited, as the client program does not have direct access to

the parallel devices and cannot configure concurrent communication efficiently. Furthermore, this model of separate instances also limits us to the concurrency afforded to asynchronous kernel executions by the parallel computing framework used (i.e., CUDA or OpenCL).

Given our desire to improve concurrency for partitioned analyses, our first decision was to move away from one library instance per data subset. This gave us greater potential for concurrency, such as via single kernel launches, and more control over how computation is combined into concurrent executions. Using a single library instance also results in significant memory savings given many overhead costs become shared for all partitions.

API Changes. In order to support partitioning in a single library instance we have modified the BEAGLE API to support data subset assignment and per-subset operations. Partition assignment can be done via a pattern-count length array of integers, with support for noncontiguous assignments. These changes were done as additions to the existing BEAGLE v1 API, and the interface remains backwards compatible.

CUDA First. Our work to increase concurrency, and thus efficiency, for partitioned analyses initially focused on our parallel implementation for the CUDA framework. We have found this framework to be generally more mature than OpenCL, and to support more features. We identified two solutions to allow independent data subsets to be concurrently computed: (a) using CUDA *streams*, which would allow separate likelihood kernel launches to run concurrently; and (b) developing a *multi-operation* likelihood kernel, which would compute multiple likelihood arrays within a single kernel launch. Below we describe each approach.

Streams. This feature of the CUDA framework is described by NVIDIA as follows:

> "The CUDA programming model provides streams as a mechanism for programs to indicate dependence and independence among kernel launches. Kernels launched into the same stream are guaranteed to execute consecutively, while kernels launched into different streams are permitted to execute concurrently. Streams describe independence between work items and hence allow potentially greater efficiency through concurrency."

To achieve partition concurrency we launch our likelihood kernels on separate streams according to the data subset of the likelihood array operation. We do so in a breadth-first manner, that is, the kernel launch for the first partial likelihood array operation for data subset 1 is followed by the launch for the first operation for subset 2, and so on. This is to compensate for signal delay in each stream. We use this multi-stream approach for both partial likelihood and likelihood integration kernels. For all other kernel launches in BEAGLE we use the `null` stream which synchronizes with all streams.

Multi-operation Kernel. Our second solution for data subset concurrency involved modifying our partial likelihood CUDA kernel to compute multiple likelihood arrays in a single execution launch. We used pointer arithmetic to allow different input and output arrays for different execution blocks.

Figure 2 contrasts available data arrays (`nodes`, `branches`) and likelihood array index (`pattern`) for our single and multi-operation partial likelihood kernels. The first implementation is restricted to a single set of input likelihood arrays (for nodes c_1 and c_2), input branch length arrays (t_1 and t_2), and output array (d_0), for all execution blocks. Additionally the pattern computed by each execution thread is directly determined by block index n, block size $blockSize$, and thread index $threadId$.

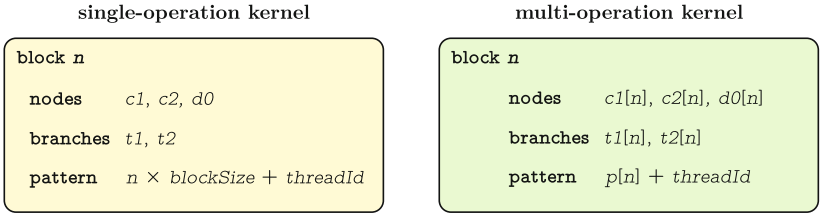

Fig. 2. Organization of data arrays and indexing for single and multi-operation kernel execution blocks for partial likelihoods computation in BEAGLE.

With the *multi-operation* approach, input and output arrays are determined based on the block index. Further, the pattern computed by each thread is only indirectly determined by n, which allows padding of data subsets when these do not fall along block-sized boundaries.

Additionally, to maximize device global memory throughput we rearrange site patterns on device memory so that data subsets are contiguous. This is done when sequence partition assignment is made by the client program and enables each execution block to operate on a single data subset more efficiently.

2.3 Independent Subtree Concurrency

As we developed the above approaches to partition concurrency, we noted we could also leverage those methods to concurrently compute partial likelihood arrays for independent subtrees. This would be specially beneficial for large trees with short sequences when running on manycore processors such as GPUs. This combination of problem size and hardware resource previously left many processing cores underutilized. Below we describe implementation details for independent subtree operations via both our *streams* and *multi-operation* solutions.

Algorithm 1. Streams and partial likelihood array operations

```
Data: a sequence of likelihood operations in reverse level-order traversal
Result: computation of partial likelihood arrays in concurrent streams
streamIndex ← 0
foreach operation in the operations sequence do
    node ← operation.parent
    if node.child1.streamIndex is not null then
        node.streamIndex ← node.child1.streamIndex
        node.waitIndex ← node.child2.streamIndex
    else if node.child2.streamIndex is not null then
        node.streamIndex ← node.child2.streamIndex
        node.waitIndex ← node.child1.streamIndex
    else
        node.streamIndex ← streamIndex + 1
        streamIndex ← streamIndex + 1
    end
    if node.waitIndex is not null then
        cudaStreamWaitEvent(event node.waitIndex, stream node.streamIndex)
    end
    cudaLaunchKernel(kernel pLikelihoods, stream node.streamIndex)
    cudaEventRecord(event node.streamIndex, stream node.streamIndex)
end
```

Streams. We further leveraged the use of CUDA streams to concurrently compute partial likelihood arrays of independent subtrees by assigning them as described by Algorithm 1. This algorithm shows how we assign a likelihood array kernel launch (`pLikelihoods`) to a stream based on an inherited index from either of the child nodes (`child1` or `child2`). Additionally, we may wait on a CUDA event that has been recorded for the other child node before launching the kernel.

Multi-operation Kernel. To implement subtree concurrency with this kernel, we process partial likelihood subtree operations according to a reverse level-order traversal of the proposed tree. We add each consecutive operation to a set until we find an operation that is dependent on the result of a previous operation in the set. We then start a new operation set, repeating the same process. Once we have processed all operations in this manner, we successively launch each operation set for concurrent computation using our *multi-operation* partial likelihoods kernel.

2.4 Extending Concurrency Gains to OpenCL

Our next step was to extend the above work, using the CUDA framework, to our OpenCL implementation.

Queues. The OpenCL equivalent to CUDA streams are concurrent execution queues. We implemented our approach in an analogous manner but found the use of concurrent queues only offered at best minimal gains in performance for the OpenCL devices we had access to (AMD Radeon R9 Nano and FirePro S9170 GPUs, and Intel Xeon E5-2680v4 CPU).

Multi-operation Kernel. For this approach, in a comparable manner to CUDA blocks, we launch OpenCL work-groups such that multiple partial likelihood operations can be performed concurrently. In contrast to CUDA, we found that the OpenCL solution was generally more performance sensitive to implementation details such as operation order and synchronization points. This was ultimately beneficial, as we iteratively refined of our likelihood kernel to optimize performance, and could then translate back some of the gains to the CUDA solution.

2.5 Memory Transfer Optimizations

For the *multi-operation* approach under either CUDA or OpenCL, we necessitate an explicit memory transfer from host to device for each tree likelihood estimation. Such memory transfers can be costly for GPU devices as they may have to go over the PCI bus. BEAGLE was designed to minimize this type of transfer and previously explicit host to device transfers only occurred at the initialization phase of an inference run.

This additional memory transfer for our *multi-operation* kernel is used to copy the address offsets for the input and output arrays each block in device memory will operate on. In order to minimize costs for this additional memory transfer, we process all subtree operations in a partial likelihoods call to the library, and perform a single transfer for multiple launches of our *multi-operation* kernel.

Table 2. GPU memory transfer optimizations; throughput in GFLOPS

Framework	GPU	Solution	*tree* A	*tree* B
CUDA	NVIDIA P5000	*write*	328.27	188.76
		pinned	**328.57**	**203.47**
OpenCL	NVIDIA P5000	*write*	320.10	183.78
		map/unmap	**321.24**	**199.58**
	AMD R9 Nano	*write*	397.92	178.04
		map/unmap	**403.72**	**210.30**

Further, we use faster methods than we had done before for host to device transfer: *pinned* host memory allocations under CUDA; and *map* and *unmap* approach with OpenCL. Table 2 shows kernel throughput performance with these approaches when compared to the performance when using the regular memory *write* transfer method under each framework. This comparison was done for two

tree sizes: *tree* A has 16 tips and 100,032 sequence patterns; and *tree* B has 256 tips and 1024 patterns. We observe that the *pinned* and *map/unmap* approaches have a positive impact on overall throughput, especially for *tree* B, which has many more tips, and thus more partial likelihood operations with an ensuing larger data transfer size.

2.6 Combining Pattern Partition and Independent Subtree Concurrency

We have found that the most efficient approach (i.e., *stream/queues*, or *multi-operation*) to concurrent partial likelihood array operations depends on the number of patterns being processed per operation. In order to determine which approach to use for different problem sizes, we have benchmarked the throughput for our partial likelihood kernel when evaluating a tree with 16 tips and 100,032 patterns for an increasing number of equal-sized data subsets (Table 3) across our different parallel solutions. The CUDA implementation was tested on an NVIDIA Quadro P5000 GPU, the OpenCL-GPU implementation on an AMD Radeon R9 Nano, and the OpenCL-x86 implementation on dual Intel Xeon E5-2680v4 CPUs. Systems were as specified in Table 1.

Table 3. Concurrency solutions and partition sizes; throughput in GFLOPS with bold text indicating which concurrency approach within a parallel solution offers best performance at each problem size.

Partition		CUDA		OpenCL–GPU		OpenCL–x86	
count	size	*streams*	*multi-op*	*queues*	*multi-op*	*queues*	*multi-op*
1	100,032	**321.82**	272.61	**346.26**	335.62	**79.97**	79.43
2	50,016	**330.08**	228.21	**354.79**	341.02	**79.85**	77.85
16	6,252	**316.72**	225.64	226.77	**330.68**	70.60	**76.10**
24	4,168	**227.63**	223.40	182.71	**318.97**	65.92	**75.21**
32	3,126	164.06	**217.59**	141.50	**317.28**	54.65	**73.00**
64	1,563	87.75	**212.71**	87.98	**326.49**	24.92	**73.61**

With the CUDA implementation, we observe that for larger numbers of patterns (above 4,168) the Quadro P5000 GPU is near saturation, and the one-time overhead of the *multi-operation* approach makes it relatively inefficient (Table 3). However, for smaller problem sizes there is less work per stream, and the overhead cost for each stream makes that approach the less efficient alternative. For the OpenCL implementations we observe that the *multi-operation* approach is the most efficient or close to most efficient for any partitioned problem.

Based on these findings, and on further intermediate analyses not shown in Table 3, we set a fixed crossover point for each solution which determines which approach is used. For the CUDA implementation we have set this at 4,168

patterns, for the OpenCL-GPU it is set at 8,192 patterns, and for the OpenCL-x86 implementation the *multi-operation* approach is always used. Additionally, client programs can also explicitly request either the *streams* or *multi-operation* implementation via the library API.

2.7 Other Aspects

Although BEAGLE supports inferences with models of arbitrary state counts, the work described here has thus far only been implemented for nucleotide model inferences.

It is also worth mentioning that our implementation allows partitions to be reassigned at any point. With each new partition assignment we rearrange patterns in device memory to maintain efficient throughput. This functionality may be used by client programs in the future to enable efficient inference of partition assignments in conjunction with currently inferred parameters.

Finally, we use the `--default-stream per-thread` NVIDIA CUDA compiler (NVCC) option so that each BEAGLE instance runs on a separate default stream. This allows further concurrency gains for other independent work in addition to partitioning, such as Metropolis-coupled, Markov chain Monte Carlo chains or run replicates.

2.8 Modifications to MrBayes

In order to fully evaluate the efficacy of the concurrency improvements to the library, we have adapted MrBayes version 3.2.6 to use the new BEAGLE API partitioning extensions. This enabled MrBayes to use a single BEAGLE library instance for computing the likelihood of multiple data subsets. This modified version of MrBbayes is open-source under GPL version 3.0, and is available at https://github.com/ayresdl/mrbayes-beagle3.

2.9 Library Availability

The BEAGLE project is open source under the GPL v3.0 license. The work described here will be part of an upcoming release, and is available under a development branch of the library located at https://github.com/beagle-dev/beagle-lib/tree/kernel-concurrency.

3 Results

Here we explore the performance effect of the concurrency gains on various parallel hardware resources. System specifications are as shown in Table 1.

3.1 Pattern Partition Concurrency Gains

We observe that for both the Quadro P5000 and Radeon R9 Nano GPUs the previous approach of sequential computation of data subsets produces a sharp drop-off in throughput as we increase the number of subsets (Fig. 3). This is because as we increase the partition count the data subsets have decreasing numbers of patterns, resulting in increasingly underutilized GPU capacity.

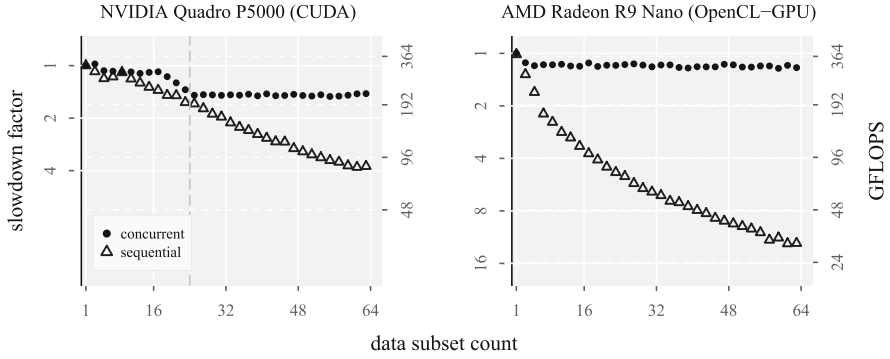

Fig. 3. Plots showing throughput for the partial likelihood kernel with data subset concurrency (black dots) and with no data subset concurrency (open triangles) for a problem with 100,032 total sequence patterns and increasing number of equal-sized data subsets for two GPU device/framework pairs. Left-axis *slowdown factor* indicates performance loss relative to the unpartitioned case. Slowdown factors and throughput in GFLOPS are on a log-scale.

For concurrent computation with the CUDA device, throughput is higher than with the sequential approach at all subset sizes. When there are fewer than 24 subsets we use the *streams* approach. Throughput with this approach starts to drop quickly after 17 subsets (corresponding to a subset size of approximately 6,000 patterns). We then note the crossover point at 24 subsets (subset size of 4,168 patterns, and indicated by a dark grey dashed line) where we switch to our *multi-operation* kernel approach. This approach exhibits consistent throughput independent of subset size.

With the OpenCL solution we use the *multi-operation* approach for all partitioned cases and note consistent and near best-case throughput, independent of the number of data subsets.

3.2 Independent Subtree Concurrency Gains

Figure 4 shows the performance improvement associated with concurrent computation of independent subtrees for a problem with 512 patterns. The pectinate case (open triangle) also represents performance for any tree topology with our previous solution of serial computation of subtree partial likelihood arrays.

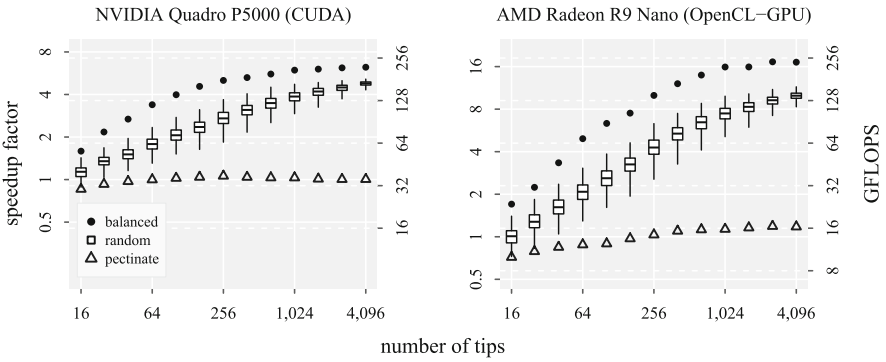

Fig. 4. Plots showing throughput for the partial likelihood kernel with subtree concurrency for fully balanced trees (black dots), for 1,000 random topology trees (distribution characterized by box plot), and for pectinate trees (open triangles) for a problem with 512 site patterns and increasing number of tips for two GPU device/framework pairs. Left-axis *speedup factor* indicates performance gain relative to the average pectinate tree throughput. Speedup factors, throughput, and number of tips are on a log-scale.

For both GPUs, we observe increasing speedups with tree size for the average random tree or for fully balanced trees. We also note that for larger trees the throughput distribution for a random tree is skewed towards the fully balanced case, which is associated with GPU saturation at these problem sizes. Finally, we note that pectinate-case performance is approximately twice as fast with the P5000 GPU under CUDA as compared to the R9 Nano GPU using our OpenCL implementation. Effective performance towards the pectinate end of the tree symmetry scale remains highly relevant as phylogenetic inference programs are optimized such that only a subtree representing the modified portion of the overall tree is recomputed for each topology change. These subtrees are often much less balanced than the full tree.

3.3 Application-Level Results

We used our adapted version of MrBayes 3.2.6 to assess application-level performance gains for our concurrency work across a variety of parallel computing devices. For these benchmarks we used a dataset with 500 taxa and 759 unique site patterns of *rbc*L, the chloroplast gene encoding the large subunit of ribulose-1,5-bisphosphate carboxylase/oxygenase, which is derived from a study of angiosperm relationships [4]. We partitioned the sequence data based on codon position, resulting in 3 subsets with 253 unique site patterns each, and inferences were run using the MrBayes default single-precision floating point format.

We chose a dataset with a high number of sequences and with few patterns, further broken into independent subsets, to best showcase the gains in concurrency described in this paper. Previously problems with these characteristics have been the most challenging for effective parallelization. BEAGLE-enabled

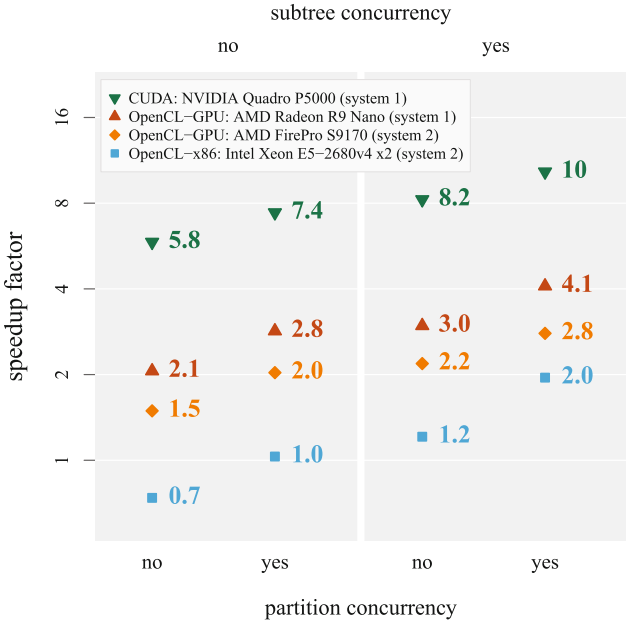

Fig. 5. Performance gains for a MrBayes nucleotide-model analysis for various hardware platforms when using the BEAGLE library, with and without partition and subtree concurrency. Speedup factors are relative to the total run time when using the standard MrBayes SSE likelihood calculator and are shown on a log-scale.

MrBayes peak performance for datasets with many more patterns and using higher state-count models are reported elsewhere [1,2].

Speedups for this challenging MrBayes analysis improve as we enable partition and subtree concurrency, across all hardware resources and corresponding frameworks (Fig. 5). We observe an average speedup gain of 1.5-fold for subtree concurrency and 1.4-fold for partition concurrency across all hardware devices. For the best performing resource (NVIDIA Quadro P5000 GPU with CUDA) we observe a 1.7-fold gain in speedup when using both concurrency improvements, ultimately resulting in a 10-fold speedup over the native MrBayes SSE run time.

We have attempted but were unable to compare our work to the most recent proposals from other authors for parallel MrBayes acceleration. For aMC^3 [3], which proposes an adaptive multi-GPU approach, we were unable to perform any analyses with the publicly available code due to execution errors. Additionally, aMC^3 is based on MrBayes 3.1.2 which lacks several features and converges more slowly than version 3.2 [10], making it unsuitable for a direct comparison to our work. For sMC^3 [9], which proposes more efficient CPU + GPU parallelism and reports speedups over previous versions of BEAGLE, neither the source code nor a binary file appear to be readily available.

4 Conclusion

Enabling further concurrency of computation in BEAGLE as described here allows a wider range of phylogenetic inferences to benefit from parallel computing hardware. Analyses with many small data subsets or with large trees but few site patterns, now benefit from increased throughput on multi and manycore resources. This work represents an important step in combining the capabilities of increasingly parallel hardware, and the demands of progressively more sophisticated phylogenetic inference analyses.

Acknowledgments. We thank Marc Suchard, University of California, Los Angeles, and Andrew Rambaut, University of Edinburgh; Mark Berger, NVIDIA; and Greg Stoner and Ben Sander, AMD. This work was supported by the National Science Foundation grant numbers DBI-0755048 and DBI-1356562.

References

1. Ayres, D.L., Cummings, M.P.: Heterogeneous hardware support in BEAGLE, a high-performance computing library for statistical phylogenetics. In: 2017 46th International Conference on Parallel Processing Workshops (ICPPW), Bristol, UK (2017, in press)
2. Ayres, D.L., Darling, A., Zwickl, D.J., Beerli, P., Holder, M.T., Lewis, P.O., Huelsenbeck, J.P., Ronquist, F., Swofford, D.L., Cummings, M.P., Rambaut, A., Suchard, M.A.: BEAGLE: an application programming interface and high-performance computing library for statistical phylogenetics. Syst. Biol. **61**(1), 170–173 (2012). doi:10.1093/sysbio/syr100
3. Bao, J., Xia, H., Zhou, J., Liu, X., Wang, G.: Efficient implementation of MrBayes on multi-GPU. Mol. Biol. Evol. **30**(6), 1471 (2013). doi:10.1093/molbev/mst043
4. Chase, M.W., Soltis, D.E., Olmstead, R.G., Morgan, D., Les, D.H., Mishler, B.D., Duvall, M.R., Price, R.A., Hills, H.G., Qiu, Y.L., Plunkett, G.M., Soltis, P.S., Swensen, S.M., Williams, S.E., Gadek, P.A., Quinn, C.J., Eguiarte, L.E., Golenberg, E., Learn Jr., G.H., Graham, S.W., Barrett, S.C.H., Dayanandan, S., Albert, V.A.: Phylogenetics of seed plants: an analysis of nucleotide sequences from the plastid gene *rbc*L. Ann. Mo. Bot. Gard. **80**(3), 528–580 (1993). doi:10.2307/2399846
5. Drummond, A.J., Suchard, M.A., Xie, D., Rambaut, A.: Bayesian phylogenetics with BEAUti and the BEAST 1.7. Mol. Biol. Evol. **29**, 1969–1973 (2012). doi:10.1093/molbev/mss075
6. Felsenstein, J.: The number of evolutionary trees. Syst. Biol. **27**(1), 27–33 (1978). doi:10.2307/2412810
7. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol. **17**(6), 368–76 (1981). doi:10.1007/BF01734359
8. Guindon, S., Dufayard, J.F., Lefort, V., Anisimova, M., Hordijk, W., Gascuel, O.: New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0. Syst. Biol. **59**(3), 307–321 (2010). doi:10.1093/sysbio/syq010
9. Kuan, L., Pratas, F., Sousa, L., Toms, P.: MrBayes sMC3: accelerating Bayesian inference of phylogenetic trees. Int. J. High. Perform. C. (2016). doi:10.1177/1094342016652461

10. Ronquist, F., Teslenko, M., van der Mark, P., Ayres, D.L., Darling, A., Hohna, S., Larget, B., Liu, L., Suchard, M.A., Huelsenbeck, J.P.: MrBayes 3.2: efficient Bayesian phylogenetic inference and model choice across a large model space. Syst. Biol. **61**(3), 539–542 (2012). doi:10.1093/sysbio/sys029
11. Zwickl, D.J.: Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion. Ph.D. thesis, University of Texas, Austin, TX (2006)