Bioinformatics, 35(20), 2019, 3961–3969
doi: 10.1093/bioinformatics/btz211
Advance Access Publication Date: 23 March 2019
Original Paper

Phylogenetics

ASTRAL-MP: scaling ASTRAL to very large datasets using randomization and parallelization

John Yin[1], Chao Zhang[2] and Siavash Mirarab[3,*]

[1]Department of Mathematics, [2]Bioinformatics and Systems Biology and [3]Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093, USA

*To whom correspondence should be addressed.

Associate Editor: Russell Schwartz

Received on December 1, 2018; revised on March 12, 2019; editorial decision on March 18, 2019; accepted on March 21, 2019

Abstract

Motivation: Evolutionary histories can change from one part of the genome to another. The potential for discordance between the gene trees has motivated the development of summary methods that reconstruct a species tree from an input collection of gene trees. ASTRAL is a widely used summary method and has been able to scale to relatively large datasets. However, the size of genomic datasets is quickly growing. Despite its relative efficiency, the current single-threaded implementation of ASTRAL is falling behind the data growth trends is not able to analyze the largest available datasets in a reasonable time.

Results: ASTRAL uses dynamic programing and is not trivially parallel. In this paper, we introduce ASTRAL-MP, the first version of ASTRAL that can exploit parallelism and also uses randomization techniques to speed up some of its steps. Importantly, ASTRAL-MP can take advantage of not just multiple CPU cores but also one or several graphics processing units (GPUs). The ASTRAL-MP code scales very well with increasing CPU cores, and its GPU version, implemented in OpenCL, can have up to 158$\times$ speedups compared to ASTRAL-III. Using GPUs and multiple cores, ASTRAL-MP is able to analyze datasets with 10 000 species or datasets with more than 100 000 genes in $<$2 days.

Availability and implementation: ASTRAL-MP is available at https://github.com/smirarab/ASTRAL/tree/MP.

Contact: smirarab@ucsd.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Genome-wide discordance is a defining feature of modern phylogenetic studies (e.g. Jarvis *et al.*, 2014; Pollard *et al.*, 2006; Wickett *et al.*, 2014; Zwickl *et al.*, 2014). A species tree represents the overall evolutionary history of species while gene trees show the phylogeny specific to regions of the genome (called loci or genes) (Maddison, 1997). Gene trees can be discordant with each other and with the species tree (Degnan and Rosenberg, 2009). Recent phylogenomic datasets include hundreds to thousands of loci. Ignoring discordance and concatenating all loci can be misleading (Kubatko and Degnan, 2007; Roch and Steel, 2015), and this observation has motivated the search for efficient and statistically rigorous methods of species tree estimation without concatenation. A ubiquitous cause of discordance is incomplete lineage sorting (ILS), a process related to coalescence histories of genealogies (Degnan and Rosenberg, 2009). ILS is often modeled by the multi-species coalescent model (Pamilo and Nei, 1988; Rannala and Yang, 2003).

Several types of statistically consistent methods have been developed to infer species trees in the presence of ILS (e.g. Bryant *et al.*, 2012; Chifman and Kubatko, 2014; Heled and Drummond, 2010; Liu, 2008). The most scalable family of methods are summary methods, which first estimate gene trees from the individual loci and then combine the gene trees to get the species tree. Researchers have developed many statistically consistent summary methods, including, STAR (Liu *et al.*, 2009), GLASS (Mossel and Roch, 2010), MP-EST (Liu *et al.*, 2010), STELLS (Wu, 2012), Bucky-quartets (Larget *et al.*, 2010), DISTIQUE (Sayyari and Mirarab, 2016a), NJst (Liu and Yu, 2011) and ASTRID (Vachaspati and Warnow, 2015).

ASTRAL (Mirarab *et al.*, 2014b) is a widely used summary method. It has been used for many groups of species (e.g. Arcila *et al.*, 2017; Blom *et al.*, 2016; Hosner *et al.*, 2015; Laumer *et al.*, 2015; Mitchell *et al.*, 2017; Rouse *et al.*, 2016; Tarver *et al.*, 2016; Wickett *et al.*, 2014). ASTRAL seeks to find the tree that shares the maximum number of induced quartet trees with the input set. This problem can be solved using a dynamic programing approach, an observation first made by Bryant and Steel (2001) and derived independently by Mirarab *et al.* (2014b). The dynamic programing solves an NP-hard problem (Lafond and Scornavacca, 2018) but uses heuristics to constrain the search space. The original ASTRAL code had $O(n^4k^3)$ running time for n species and k gene trees. Newer versions ASTRAL-II (Mirarab and Warnow, 2015), ASTRAL-III (Zhang *et al.*, 2018) and ASTRAL-multi-individual (Rabiee *et al.*, 2019) have changed the search space and the details of the dynamic programing. The current version, ASTRAL-III runs in $O((nk)^{1.73}D)$ where $D = O(nk)$ is the sum of degrees of all *unique* nodes in input trees. In practice, the running time of ASTRAL-III tends to grow quadratically with n and k, though the amount of gene tree discordance also matters (Zhang *et al.*, 2018). ASTRAL-III has been able to handle datasets with $k = 1000$ genes and $n = 5000$ species given at most three days of running time.

Despite the progress on scalability, large datasets that push ASTRAL-III to its limits are now available, and as new genomes get sequenced, more datasets will test limits of ASTRAL-III. Moreover, the number of input trees can also quickly grow. Even with only 48 bird species, ASTRAL-III took 32 h to analyze a dataset of 14 446 gene trees (Zhang *et al.*, 2018). Interestingly, several relationships remained unresolved even with 14 446 gene trees, an issue that may be solved by mining even more loci from the genomes. Increasing the number of loci to 30 000 could take about 5 days of running time, starting to make ASTRAL-III impractical. Moreover, several hundreds of avian genomes will be available in the near future. ASTRAL-III on 14 000 gene trees with 300 species will take close to 50 days of running time, again making it impractical. Moreover, the number of gene trees can increase many folds if we represent each locus not with a single point estimate of the tree but with a distribution of trees.

In this paper, we introduce ASTRAL-MP, a new version of ASTRAL that uses shared-memory CPU parallelism, graphics processing unit (GPU) parallelism, AVX2 vectorization and a new randomized algorithm to dramatically scale ASTRAL-III. ASTRAL-MP has great speedups compared to ASTRAL-III and shows perfect scaling (tested for up to 24 cores) on some datasets. With GPUs, the speedups can be up to 158× compared to ASTRAL-III.

2 Materials and methods

2.1 Background on ASTRAL

The input is a set $\mathcal{G}$ of unrooted gene trees labeled by the leaf-set $\mathcal{S}$ with $|\mathcal{S}| = n$ and $|\mathcal{G}| = k$. Each of the $\binom{n}{4}$ selections of four leaves induces an unrooted quartet tree in each of the k trees in $\mathcal{G}$. The weighted quartet score of a given tree T with respect to $\mathcal{G}$ is defined as the number of the $k\binom{n}{4}$ quartet trees induced by $\mathcal{G}$ that are identical topologically to the quartet tree induced by T on the same quartet. The definition can be easily extended to gene trees with missing data and multifurcations by simply counting fully resolved quartet trees present in a gene tree. ASTRAL finds the species tree that maximizes this score using dynamic programing.

A tripartition of $\mathcal{S}$ corresponds to a node in a binary unrooted species tree T. ASTRAL uses the fact that tripartitions can be scored with respect to the $\mathcal{G}$ in isolation from the rest of the tripartitions in T. Let $P = A_1|A_2|A_3$ be a tripartition and $M = M_1|\ldots|M_d$ be a d-partition of $\mathcal{S}$, and let $I_{j,i} = |A_j \cap M_i|$. A species tree that includes P will have to share some quartet trees with a gene tree that includes M; call this quantity $QI(P, M)$. Mirarab and Warnow (2015) showed how to compute QI in $\Theta(d^3)$ time and Zhang *et al.* (2018) derived a new formula for QI:

$$\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{3}\binom{I_{j,i}}{2}((S_{a_j} - I_{a_j,i})(S_{b_j} - I_{b_j,i}) - S_{a_j,b_j} + I_{a_j,i}I_{b_j,i}) \qquad (1)$$

where $a = [2\ 1\ 1]$, $b = [3\ 3\ 2]$, $S_j = \sum_1^d I_{j,i}$ and $S_{j,k} = \sum_1^d I_{j,i}I_{k,i}$. With this, ASTRAL-III can compute QI in $\Theta(d)$. We further define the weight of a tripartition, $w(P)$, as the total score of P:

$$w(P) = \frac{1}{2}\sum_{g\in\mathcal{G}}\sum_{P'\in\mathcal{N}(g)} QI(P, P') \qquad (2)$$

where $\mathcal{N}(g)$ is the set of internal nodes in g, represented as d-partitions.

The ASTRAL dynamic programing starts from the set $\mathcal{S}$ and recursively divides into two subsets that maximize the sum of the weights below the current subset. If we consider all ways of partitioning a 'cluster' $A \subset \mathcal{S}$ into A' and $A \setminus A'$, the problem is solved exactly in exponential time. To obtain a polynomial time algorithm, we constrain the search. Let X' be a set of bipartitions, and define $X = \{A : A|\mathcal{S}\setminus A \in X'\}$ and $Y = \{(C, D) : C \in X, D \in X, C \cap D = \varnothing, C \cup D \in X\}$. The dynamic programing only considers $(A', A \setminus A') \in Y$. Defining $V(A)$ as the score for an optimal subtree on the cluster A and setting $V(A) = 0$ for $|A| = 1$, the dynamic programing recursion is given by:

$$V(A) = \max_{(A', A\setminus A')\in Y} V(A') + V(A\setminus A') + w(A'|A\setminus A'|\mathcal{S}\setminus A) \qquad (3)$$

The definition of X has changed from ASTRAL-I to ASTRAL-III. All three versions include the set of bipartitions observed in (completed) input gene trees. ASTRAL-II and ASTRAL-III further add to that set using heuristics and ASTRAL-III ensures $|X| = O(nk)$ (Zhang *et al.*, 2018).

Depending on how gene trees are represented in memory, computing tripartition weights using Equation (2) can be done in either $\Theta(nk)$ or $\Theta(D) = O(nk)$ where D is the sum of degrees of all *unique* nodes in $\mathcal{G}$. Gene trees can be represented in memory condensely as an array of Θ(nk) numbers representing the post-order traversal of gene trees. Computing $w(P)$ for a tripartition $P = A_1|A_2|A_3$ will simplify to a post-order traversal of all gene trees by iterating through the array representation of $\mathcal{G}$ and keeping a stack for $I_{j,i}$ values, which can be each computed in $\Theta(d)$ (Supplementary Algorithm S1). An alternative introduced in ASTRAL-III is to use a polytree to represent $\mathcal{G}$. The polytree overlays all gene trees on top of each other into a DAG. In this polytree, nodes that appear identically in

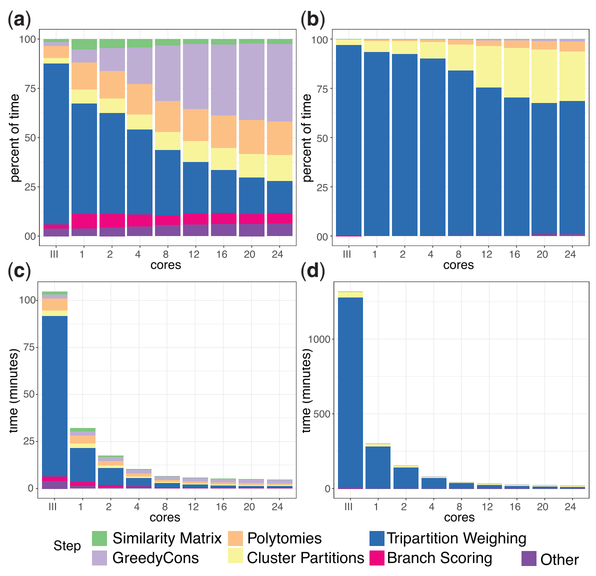

Fig. 1. Profiling ASTRAL running time. The running time of ASTRAL is profiled and divided into several steps (described in the text) shown both as percent of total time (**a, b**) and the actual running time (**c, d**). We profile SV-1000 (a, c) and avian (b, d) datasets which have high n and k, respectively (see Table 1). The first bar shows ASTRAL-III, followed by ASTRAL-MP given one CPU core, and then ASTRAL-MP with 2–24 cores. Weight calculation is the dominant step with no parallelization both for ASTRAL-III and ASTRAL-MP. For the avian dataset, even with 24 cores, weight calculation remains dominant. On the SV dataset, after parallelization, the computation of the Greedy Consensus, which is the only step that we have not effectively parallelized, takes the majority of the time. With >6 cores, processing polytomies and cluster partitioning steps take an increased portion of the time, indicating that their parallelization is not as effective as weight calculations

multiple gene trees are represented only once and thus, the time to traverse all nodes is reduced to $O(D)$. However, the polytree is a more complicated data-structure than the array and will take up more memory.

2.2 ASTRAL-MP: overview

To motivate steps taken to parallelize ASTRAL, we first profile ASTRAL-III. We divide the total running time of ASTRAL into seven categories: (i) compute the similarity matrix between pairs of species. (ii) Compute the greedy (i.e. extended majority) consensus of input trees. (iii) Process plytomies of greedy consensus trees. (iv) During dynamic programing, compute for every cluster A all ways of partitioning it (i.e. building the set Y from X). (v) Weight tripartitions using Equation (2) (the theoretical bottleneck of the dynamic programing). (vi) Once the tree is computed, compute branch support and branch length (Sayyari and Mirarab, 2016b). (vii) All other smaller steps are simply categorized together.

Consistent with theoretical expectations, in ASTRAL-III, depending on the choice of the data, the bottleneck is computing weights, taking 75–97% of the time (Fig. 1). However, the remaining part is significant, and to avoid the curse of Amdahl's law, we need to optimize these other parts as well. For optimizing the cluster partitioning step (iv), we developed a new randomized algorithm that improves its speed and we also parallelize it using CPU multithreading. For the dominant step, weight calculation, we parallelize it using CPU vectorization, CPU multithreading and GPU multithreading. Weight calculation is the step that best avails itself to limited memory and Single Instruction Multiple Data (SIMD) execution; thus, we parallelize only the weight calculation in GPU. The remaining steps are all parallelized in CPU, with varying levels of effectiveness. We start by describing the new randomized algorithm for cluster partitioning then describe our parallelization strategy.

2.3 Randomized cluster partitioning

2.3.1 Current method

The dynamic programing needs to implicitly build the set Y, which is equivalent to finding all possible ways of partitioning a *cluster* $A \in X$ to $A' \in X$ and $A \setminus A' \in X$. ASTRAL-III represent X as an array of sets where each set contains all clusters of the same cardinality, and each cluster is stored as a bitset. As the dynamic programing recursions progress, for any cluster A, ASTRAL-III recursively builds a subset of X with clusters contained in A: $X{\restriction_A} = \{B : B \in X, B \subset A\}$. As the cardinality of A reduces, so does the size of the $X{\restriction_A}$. Once $X{\restriction_A}$ is computed, to find all ways of partitioning A, we need to consider only $A', A \setminus A' \in X{\restriction_A}$; for $1 \le i \le |A|/2$, for all pairs of clusters $(C, D) \in X{\restriction_A}$ with $|C| = i$ and $|D| = |A| - i$, we test if the two clusters are disjoint; if they are, they constitute a $A'|A \setminus A'$ partition of A.

To summarize, for each cluster A, we need to compute the intersections of all pairs of clusters that are subsets of A and have cardinalities that add up to $|A|$. If for any cluster $A' \in X$ and $A' \subset A$, we had a quick way to test if $A - A' \in X$, we could avoid these costly pairwise computations. In ASTRAL-MP, we devise an algorithm to achieve just that by using a randomized representation of sets as hashed tuples. We next describe the randomized algorithm and why it has an astronomically low probability of collisions. At the end of an ASTRAL run, we check for the collisions, and in the unlikely event that one has occurred, we simply rerun ASTRAL with new randomization (this has never happened in our tests).

2.3.2 Cluster partitions using abelian groups

Let each cluster $A \subset \mathcal{S}$ be represented as a vector $\mathbf{a}$ of length n (number of species) of integers 0 and 1 where $\mathbf{a}_i = 1$ if and only if cluster A contains the i-th taxon. For a pair of clusters B and C represented as vector $\mathbf{b}$ and $\mathbf{c}$, $B|C$ is a partition of A if and only if $\mathbf{b} + \mathbf{c} = \mathbf{a}$. Defining an abelian group $Z = (\mathbf{Z^n}, +)$ with elements being vectors of integers of length n and with group operation $+$ as vector addition operation. Let G be a finite abelian group with group operation $+$. In choosing G, we need to be careful to keep the probability of collisions small; more specifically, we need G to satisfy two conditions: $\log|G| = O(\log|X|)$ and $\epsilon|H| \ge |X|^3$ for a small ϵ where $H = \{2g | g \in G\}$. To achieve these, we let G be the direct (Cartesian) product of p cyclic groups of size 2^{64} with an appropriate choice of p (discussed later). Also, let $g_1, \ldots, g_n$ be elements uniformly and independently chosen from G (with replacement). Finally, we define the function $\phi : Z \to G$ as $\phi(\mathbf{a}) = \sum_{i=1}^{n} \mathbf{a}_i g_i$ where $\mathbf{a}_i$ is an integer. It can be easily checked that ϕ is a homomorphism from Z to G. For clusters A, B, and C in X represented by $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \{0,1\}^n$, if $B|C$ is a partitioning of A, then $\mathbf{c} = \mathbf{a} - \mathbf{b}$ and by homomorphism of ϕ, it follows that $\phi(\mathbf{c}) = \phi(\mathbf{a}) - \phi(\mathbf{b})$. Thus, $B|C$ is a partitioning of A only if $\phi(\mathbf{c}) = \phi(\mathbf{a}) - \phi(\mathbf{b})$.

To find *all* partitions of A, we can go through all $B \in X$ and check if $(\phi(\mathbf{a}) - \phi(\mathbf{b})) \in \{\phi(\mathbf{x}) : \mathbf{x} \in X\}$. If so, we declare $B|A - B$ to be a valid partition of A that is fully present in X. This procedure is guaranteed to find all valid partitions of A. However, it can, in principle, also find invalid partitions of A; as we will show, the probability of such events can be made arbitrarily low with appropriate definitions of G (here, a large enough p, as we will see).

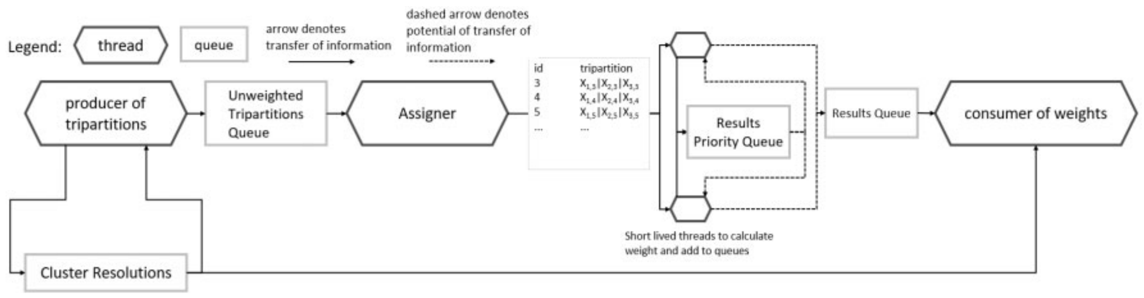
Legend:
thread
queue
arrow denotes transfer of information
dashed arrow denotes potential of transfer of information
producer of tripartitions
Unweighted Tripartitions Queue
Assigner
id tripartition
3 $X_{1,3}|X_{2,3}|X_{3,3}$
4 $X_{1,4}|X_{2,4}|X_{3,4}$
5 $X_{1,5}|X_{2,5}|X_{3,5}$
Results Priority Queue
Results Queue
consumer of weights
Short lived threads to calculate weight and add to queues
Cluster Resolutions

Fig. 2. ASTRAL-MP program architecture. Producer and consumer threads both run the dynamic programing recursions. The producer, instead of computing weights, simply adds the tripartitions to the 'unweighted Tripartitions Queue', whereas, the consumer, instead of computing weights, simply reads them from the 'Results Queue'. The assigner thread reads tripartitions, batches them up and spawns short-lived threads. Each short-lived thread uses Equations (1) and (2) to compute weights and uses Algorithm 1 to add weights first to a priority queue and then to the final queue, in a synchronized manner that guarantees that tripartitions are read by the consumer thread in the same order as produced by the producer thread. The short-lived threads may compute weights using CPU (with vectorization) or by delegating computations to a GPU (or other devices that support OpenCL)

Moreover, using our definition of G, this check can be performed efficiently using simple additions on 64-bit integers and the use of a hashed set to represent X.

For clusters $A, B, C \in X$ represented by $\mathbf{a}, \mathbf{b}, \mathbf{c}$, if $B|C$ is not a partition of A, then $\mathbf{c} \neq \mathbf{a} - \mathbf{b}$; equivalently, for $\mathbf{d} = \mathbf{b} + \mathbf{c} - \mathbf{a}$, we have $\mathbf{d} \neq 0$. Note that $\mathbf{d} \in \{-1, 0, 1, 2\}^n$. We now consider three possible cases and compute the probability of the event $\phi(\mathbf{c}) = \phi(\mathbf{a}) - \phi(\mathbf{b})$ when $\mathbf{c} \neq \mathbf{a} - \mathbf{b}$, or equivalently $\phi(\mathbf{d}) = 0$ when $\mathbf{d} \neq 0$.

Case 1: $\mathbf{d}_i = 1$ for some i. Then,

$$\begin{aligned} P(\phi(\mathbf{d}) = 0) &= P\left(\left(\sum_{j=1}^{i-1} \mathbf{d}_j g_j\right) + g_i + \left(\sum_{j=i+1}^{n} \mathbf{d}_j g_j\right) = 0\right) \\ &= P\left(g_i = -\sum_{j=1}^{i-1} \mathbf{d}_j g_j - \sum_{j=i+1}^{n} \mathbf{d}_j g_j\right) = \frac{1}{|G|} \leq \frac{1}{|H|} \end{aligned}.$$

The last equality follows from the fact that g_i is chosen uniformly at random from G and is chosen independently from all $g_j, j \neq i$ values. The inequality follows from the construction of H.

Case 2: $\mathbf{d}_i = -1$ for some i. By similar logic, $P(\phi(\mathbf{d}) = 0) \leq \frac{1}{|H|}$.

Case 3: $\mathbf{d} \in \{0, 2\}^n$ and thus, $\mathbf{d}_i = 2$ for some i. Then:

$$P(\phi(\mathbf{d}) = 0) \quad = P\left(2g_i = -\sum_{j=1}^{i-1} \mathbf{d}_j g_j - \sum_{j=i+1}^{n} \mathbf{d}_j g_j\right) \leq \frac{1}{|H|}.$$

Therefore, by a union bound, the probability that there exist clusters $A, B, C \in X$ such that $B|C$ is not a way of partitioning A and $\phi(\mathbf{c}) = \phi(\mathbf{a}) - \phi(\mathbf{b})$ is no greater than $|X|^3 \frac{1}{|H|}$. Finally, recalling that $\epsilon|H| \geq |X|^3$, the probability that the algorithm fails is no greater than:

$$|X|^3 \frac{1}{|H|} \leq |X|^3 \frac{\epsilon}{|X|^3} = \epsilon\,.$$

Choice of p. We implement G as an array of p unsigned 64-bit numbers. The choice of p controls $|G| = 2^{64p}$ and $|H| = 2^{63p}$. Therefore, for a given $|X|$ and a desired upperbound for failure rate ϵ, we need to ensure that $2^{63p} \geq \frac{|X|^3}{\epsilon}$, equivalently, $p \geq \frac{1}{21} \log_2 |X| + \frac{1}{63} \log_2 \left(\frac{1}{\epsilon}\right)$.

The algorithm first stores $\phi(\mathbf{c})$ for each $C \in X$ in a hash table and then for each pair of $A, B \in X$ checks whether there exists a C in the hash table such that $\phi(\mathbf{c}) = \phi(\mathbf{a}) - \phi(\mathbf{b})$. Thus, we access the hash table $O(|X|^2)$ times, and each requires $O(p)$ 64-bit operations. The total running time is $O(|X|^2 p) = O(|X|^2 (\log |X| + \log\left(\frac{1}{\epsilon}\right)))$.

In ASTRAL-III, $|X|$ has been in the order of 10^3–10^5 even for the large datasets we have tested (Zhang *et al.*, 2018) and has never exceeded 5×10^6 even for $n = 10^4$. When $|X| \gg 10^6$, ASTRAL-MP will likely become too slow to be useful. Thus, in practice, it is safe to assume $|X| \leq 10^9$ (note that we are allowing three orders of magnitude above what we have seen in practice). Setting the probability of error $\epsilon = 10^{-10}$, it can be checked that $p = 2$ is enough (Supplementary Fig. S1). Thus, by default, we use $p = 2$ integers (each 64-bit) to build G.

2.4 Parallelization

2.4.1 Weight calculation

We parallelize weight calculation using vectorization, multi-threaded CPU and GPU. The dynamic programing shown in Equation (3) needs to compute the weight for every tripartition corresponding to each element of Y. The weights of different tripartitions are independent, and therefore, we can compute them separately and in parallel. Parallelizing the calculation of weights over different tripartitions (as opposed to parallelizing the calculation of a single weight over different gene trees) avails itself to SIMD execution, which makes it suitable for vectorization and GPU.

CPU. We assign each thread a chunk of tripartitions to score. To have enough weights to score in parallel, we need to change the code structure. ASTRAL-III implements the dynamic programing recursively [as in Equation (3)] and computes weights only when it encounters new tripartitions. As a result, only a few tripartitions are examined in *a single* call of $V(A)$, resulting in only limited parallelism. To increase parallelism, we changed the architecture so that the discovery of new tripartitions that need to be weighted happens simultaneously with weight calculation. We achieve this using several threads, including a producer of tripartitions, an 'assigner', and a consumer of tripartitions (Fig. 2).

The producer and the consumer threads both run recursions of Equation (3) in an identical order. However, the producer does not compute weights; it simply computes cluster partitions and adds the resulting tripartitions to an 'unweighted tripartitions' queue. The 'assigner' thread reads from the unweighted tripartitions queue, builds batches of tripartitions (default batch size: 10) and creates a new short-lived thread for that batch of tripartitions. This short-lived thread computes the weights using Equations (1) and (2), and adds $w(P)$ to a 'results' queue (in two steps, as we will see). The consumer also runs through the recursive Equation (3) but each time it needs a $w(P)$, it simply reads the value off of the results queue, waiting on the queue if needed.

Note that producer and consumer threads need to run in exactly the same order. Thus, calculated weights should be put in the final results queue in the same order in which they are put in the 'unweighted tripartitions' queue. The assigner thread achieves this by attaching an ascending id to each tripartition it reads from the unweighted tripartitions. The short-lived threads send their results to a shared 'results priority queue' which sorts the computed tripartition weights by their ids and keeps track of the highest id that has been scored. When the top of the priority queue matches the next required id, results are moved from the priority queue to the final results queue. Accesses to the result queues and the highest id are synchronized using a lock. Algorithm 1 gives exact steps run by each short-lived thread.

Vectorization. When the processor supports AVX2, we further parallelize by computing several weights simultaneously using vectorization. AVX2 allows four 64-bit integer operations per instruction cycle and sixteen 16-bit integer operations per instruction cycle. Weight calculation consists of two parts: computing intersection sizes ($I_{j,i}$) and computing the final score using Equation (1). Intersections sizes are never more than n and thus comfortably fit a 16-bit integer; however, weight scores are of the order kn^4 and require 64-bit. Since each 512-bit cache line can store 32 16-bit integers, we process weights in batches of size 32 to fit one line. We use 16-bit integers to represent and compute intersection size and get 16-way parallelism using AVX2. We then convert 16-bit integers to 64-bit integers and compute tripartition scores using 64-bit integer operation and get 4-way parallelism. We implemented the weight calculation using C, which is invoked as a kernel from our Java code.

GPU parallelization. Parallelizing for GPU requires care because the memory is limited, data transfer between CPU and GPU can be costly, and GPU threads execute instructions in SIMD. The decoupled architecture of ASTRAL-MP (Fig. 2) enables us to delegate the weight calculation step to external devices such as GPU. To deal with the limited memory available to each GPU thread, we opted to use the simple and compact array representation of $\mathcal{G}$ instead of the polytree in the GPU kernel.

Due to the SIMD nature of GPU threads, the efficiency is higher if threads avoid taking different paths in the control flow of the code. For example, if there is a `if, then, else` condition, ideally, either all threads should go through `if` or all threads should go through `else`. Violating this will result in stalls and therefore delays. Our existing architecture accounts for this difficulty. As mentioned earlier, we parallelize weight calculation by simultaneously computing $w(P)$ functions for a collection of P tripartitions. The process for calculating weights of several tripartitions follows an identical control path, only with different data. Thus, our choice to parallelize by weights readily provides us with data parallelism, appropriate for SIMD architectures.

We implemented an OpenCL kernel for calculating a set of $w(P)$ values using Equation (2). We used the JOCL package to connect the GPU and CPU and to compile the OpenCL code for the given machine dynamically (to ensure portability). Each GPU device has an associated short-lived CPU thread, used to handle communication with the GPU, and these threads correspond to the short-lived threads in Figure 2. The assigner thread reserves some space on the main memory to keep (i) gene trees (represented as an array of numbers), (ii) a queue per GPU device to contain batches (default size $2^{13} = 8192$) of tripartitions to be sent to GPU for scoring and (iii) a queue per GPU device to be filled with computed scores. The CPU communicates with the GPU through buffers corresponding to these arrays (buffers are managed by OpenCL). The first two buffers are read-only, while the last one is write-only. On the GPU, all the read-only data reside in the global memory because due to their large size, they could not fit other types of memory on the device, such as local or constant memory. The short-lived thread, after invoking the GPU kernel, stalls and waits for the results. The GPU executes the kernel on the entire batch of tripartitions sent to it. When the GPU finishes, the CPU thread wakes up and uses the third buffer to read the results into the main memory. This whole process is in lieu of lines 2–4 of Algorithm 1; once the short-lived thread wakes up, it runs through the rest of Algorithm 1, starting line 5.

Algorithm 1 The procedure run by each short-lived thread to calculate weights and to propagate results first to the priority queue and then to the final queue. A is a list that contains r (tripartitions, id) pairs.

```
Global Variables:
resultsPQ: a priority queue sorted by id (ascending)
resultsQ, a queue of scores, read by the consumer thread
currentID, id of the last score added to resultsQ
globalLock, a lock for the previous three global variables
1: procedure ProcessTripartitions($A$)
2:     $R \leftarrow$ array of size $r$
3:     for $0 \leq i < r$ do
4:         $R[i] \leftarrow w(A[i].tripartition)$ computed by
           Equation (2)
5:     $lock(globalLock)$
6:     for $0 \leq i < r$ do
7:         $resultsPQ.add((w : R[i], id : A[i].id))$
8:     while $currentID = resultsPQ.top().id$ do
9:         $resultsQ.add(resultsPQ.top().w)$
10:        $resultsPQ.pop()$
11:        $currentID \leftarrow currentID + 1$
12:    $unlock(globalLock)$
```

2.4.2 Parallelizing other steps

Several other steps of ASTRAL are also parallelized for better scaling. Beyond the following, other smaller steps are not parallelized.

Similarity matrix. ASTRAL computes a similarity matrix by traversing all gene trees, measuring the number of quartets that put each pair of taxa together in that gene tree and adding up these numbers across gene trees. This part of the code is simply parallelized by having each thread work on a different gene tree, allocating a different matrix to each thread; in the end, all matrices are combined.

Building set X. ASTRAL needs to iterate over all gene trees and add their bipartitions to the set X. This step is trivially parallelized by letting each thread work on a different gene tree. Then, ASTRAL-III heuristics augment X using distance matrices and a series of greedy trees computed from gene trees. The computation of the greedy consensus trees does not avail itself to much parallelism. The processing of polytomies in the greedy consensus is parallelized by assigning each polytomy to a different thread.

Cluster partitioning. Computing partitions of a cluster A (as described in Section 2.3) is parallelized such that each thread computes all possible partitions with a certain cardinality for the two subsets of the partition.

Table 1. Datasets used to study ASTRAL-MP

Name	Original publication	# Species (n)	# Genes (k)	Type	# Generations	Contraction threshold	# Reps.
SV	Mirarab and Warnow (2015)	100, 200, 500, 1000	1000	Simulated	2×10^6	Fully resolved	10
Avian	Mirarab *et al.* (2014a)	48	14 446, 1000	Real	Unknown (order: 10^7)	Full, 0, 33, 50, 75%	1, 10
Insects	Sayyari *et al.* (2017)	144	1478	Real	Unknown	Fully resolved	1

Note: For SV, some outlier replicates have fewer than 1000 genes because poorly resolved gene trees are removed. For avian, the full dataset is subsampled randomly to create 10 inputs with 1000 gene trees.

Scoring branches. Once the final tree is computed, we need to compute localPP (Sayyari and Mirarab, 2016b) and branch length for all branches. Luckily, localPP and branch length both can be computed independently for all branches, allowing a trivial parallelization.

3 Results

3.1 Experimental setup

We compare the running time of ASTRAL-III (version 5.5.4) to ASTRAL-MP (version 5.12.4) run with a varying number of CPU cores and GPUs. Our results are expressed in terms of speedup compared to the runtime of ASTRAL-III, but we also show speedup versus ASTRAL-MP with one core in the supplement. On each replicate of each model condition of each dataset, we run ASTRAL-MP only once. However, for datasets that have many replicates, we report average speedups over all replicates. We also profile ASTRAL-MP and compare its quartet score to ASTRAL-III.

In testing the efficiency of ASTRAL-MP, we use several simulated and real datasets (Table 1). The datasets range in the number of species (n) between 48 and 1000 and have between 1000 and 14 446 gene trees (k). The datasets also have varying levels of gene tree discordance (controlled by the number of generations among other factors) though the amount of discordance for real data is not known. The SV dataset is used because it has four values of n going all the way to $n = 1000$ and the avian data are chosen because it has a large number of gene trees. Also, on the avian data, we run ASTRAL on fully resolved gene trees (denoted *full*) or gene trees that have branches with support below a threshold (0–75%) contracted.

In all the results reported, we use the San Diego Supercomputer Center's CPU and GPU nodes. The CPUs are Intel Xeon E5, which have 24 cores clocked at 2.5 GHz with 1.866 PFlops/s and 128 GB DRAM with memory bandwidth of 120 GB/s and 256-wide AVX2. The GPUs are Nvidia K80, which consists of two Tesla GK210 GPUs, each with 2496 processor cores (560 MHz) and 12 GB of memory. The GPU nodes have similar Intel CPUs with 24 cores, but a reduced 0.208 PFlop/s.

3.2 Quartet score

ASTRAL-MP is heavily re-engineered compared to ASTRAL-III. Also, the current ASTRAL-MP code is not deterministic because building set X involves random choices that can change based on thread execution orders. Changes, however, are expected to be small. To ensure integrity of results of ASTRAL-MP, we compared its quartet score to ASTRAL-III. On our data, the two versions agreed in 90% of runs. In remaining cases, the score improved or decreased by very small margins (Supplementary Table S1).

3.3 Profiling parallelization

We profiled ASTRAL-MP with up to 24 cores (Fig. 1) on the SV dataset with $n = 1000, k = 1000$ and avian dataset with $n = 48, k = 14\,446$. Comparing ASTRAL-MP given one CPU core with ASTRAL-III, we observe that ASTRAL-MP is overall 3.32 and 4.41 times faster respectively for SV (high-n) and avian (high-k) datasets. While in ASTRAL-III weight calculation is always the obvious bottleneck, in ASTRAL-MP, due to the use of the AVX2 vectorization, the time spent on weight calculation dramatically decreases. In particular, on the SV dataset, weight calculation goes from 97% of the total time in ASTRAL-III to 67% in ASTRAL-MP. Due to the randomized cluster partitioning algorithm, the total time spent in this step also reduces in ASTRAL-MP even though it takes an increased *share* of the time. As we provide ASTRAL-MP with more cores, the most successfully parallelized section, weight calculation, quickly shrinks in the proportion of the total running time it takes (down to 16% for SV and 67% for avian given 24 cores). On the SV dataset, the computation of greedy consensus trees grows to take up to 39% of the total, followed by additions to X and cluster partitioning, each of which takes around 15%. On the avian dataset, the only step that grows dramatically in its share of running time is cluster resolution (which enjoys limited parallelism).

3.4 Speedups

3.4.1 CPU parallelization

Compared to ASTRAL-III, the new ASTRAL-MP is faster even when a single CPU core and no GPUs are used. On the SV dataset, the speedup was on average around 3.25× irrespective of n. On the avian dataset, ASTRAL-MP was 4× faster when gene trees were fully resolved but introducing polytomies reduced the speedup to 2× (for 75% contraction). These speedups are mostly attributable to reductions in weighting time due to AVX2 vectorization and in cluster partitioning (Fig. 1).

The running time of ASTRAL-MP scales well with increasing numbers of CPU cores without a GPU but scaling depends on the dataset (Fig. 3). On the SV datasets, we find a speed increase of up to 23 times (compared to ASTRAL-III) with 24 threads. The larger datasets with 200 species or more benefit from parallelism more than the smallest dataset ($n = 100$). This observation is consistent with the idea that large datasets provide increased parallelism. For 100 taxa, no visible benefit is gained from using more than 12 cores. In contrast, with 1000 species, we continue to see speedups up to 24 cores, albeit at a slower rate beyond 16 cores. On the avian datasets, we see a near-perfect increase in speed with increased numbers of cores, even for up to 24 cores (Fig. 3 and Supplementary Fig. S2). Moreover, the speedups remain near-perfect even with multifurcating gene trees where low support branches are contracted (0–75%). When we subsample genes to $k = 10^3$, the scaling deteriorates; on this lower-k dataset, we see little gains beyond eight cores. We come back to this observation in the discussions.

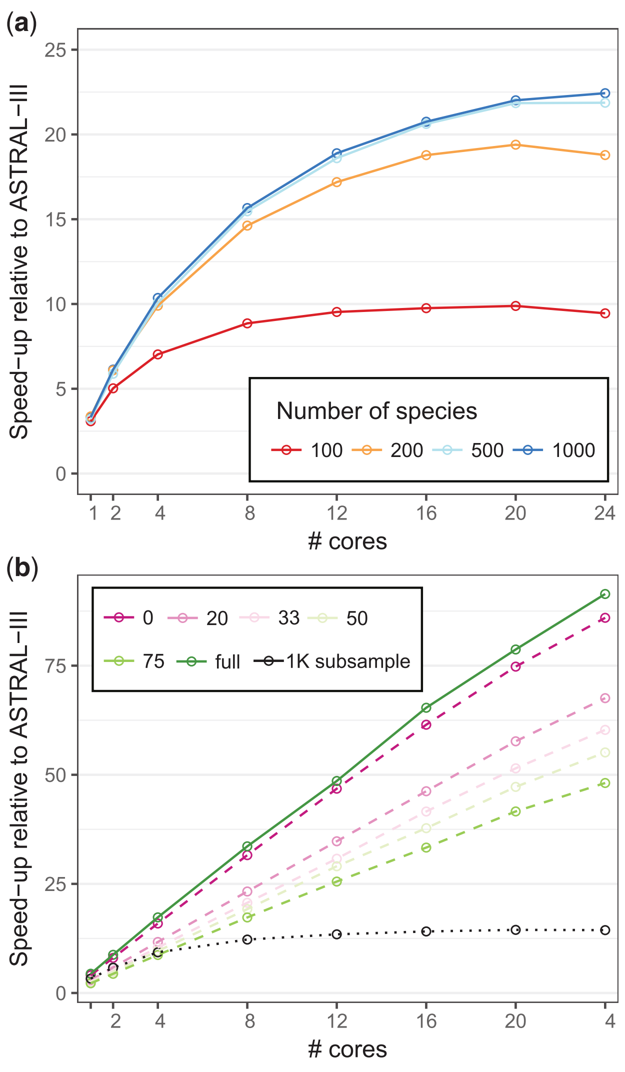

Fig. 3. Speedup of ASTRAL-MP with CPU parallelization with respect to ASTRAL-III. Dots show the average speedups over all replicates. (**a**) The SV dataset with 100–1000 species. ASTRAL-III took on average 1.5, 7, 40 min and 4 h, respectively for 100, 200, 500 and 1000 species (10 replicates each). (**b**) The avian dataset with various levels of contraction. ASTRAL-III took on average 22.5, 34, 40, 17.5, 7 and 2.5 h, with contraction levels respectively set to full, 0, 20, 33, 50 and 75% (one replicate each). The '1K subsample' corresponds to datasets (10 replicates) with random samples of 10^3 gene trees with no contraction. See Supplementary Figure S2 for speedups compared to ASTRAL-MP-1 core

3.4.2 GPU parallelization

We next test ASTRAL-MP using GPUs on our datasets with the largest n and k. On the avian dataset with large k, a single GPU with one CPU core results in more than 11× speedups compared to ASTRAL-MP without GPU and a single CPU core (Table 2). On the SV dataset with $n = 1000$ species, GPU improves the running time, but less substantially; here, using a GPU and a single CPU core decreases the running time by 2×.

We also test the combination of multiple CPU cores and GPUs. Using six CPU cores in addition to a single GPU is 122% faster than one CPU core and a single GPU on the SV dataset. Further increasing the number of CPU cores to 24 continues to improve the running time. On SV data, we observe 20.9× speedup compared to ASTRAL-III with 24 cores and a GPU (Table 2). On the avian dataset, adding CPU cores to GPUs reduces running time. For example, going from one core to six improves the speedups by 66% on this dataset. Overall, on avian, with 24 cores and a single GPU, we obtain 113× improvement compared to ASTRAL-III.

When we allow ASTRAL to use all four GPUs on each node, we see only minimal increases in the speed (Table 2) in the SV dataset with 1000 species. However, there is about a 1.4× increase in speed going from one to four GPUs for the avian dataset. These results are consistent with the observation that after GPU parallelization, weight calculation is no longer the bottleneck on the SV datasets. Since GPUs only parallelize weight calculation, by Amdahl's law, adding GPUs does not help. As shown in Table 2, ASTRAL-MP with GPU parallelization does come close to limits of speedup that can be achieved according to the Amdahl's law. On the avian dataset, the weight calculation takes a larger fraction of the time, thus adding more GPUs is effective in increasing the speed.

Finally, to see how GPU speedups vary across datasets, we analyze a large number of datasets, with varying n and gene tree resolution. Across all the datasets, the speedups with respect to ASTRAL-III obtained with a single GPU and 24 CPU cores range between 7× and 116× (Supplementary Table S2), but in all but one dataset, the speedup is at least 18×. Thus, while the exact speedup depends on the dataset, having a GPU and several CPU cores speeds ASTRAL up by one to two orders of magnitude.

Beyond scaling, ASTRAL-MP substantially reduced the running time compared to ASTRAL-III. On the largest SV dataset with 1000 species, the running time of ASTRAL-III is 2–10 h. However, on the same dataset and using 24 CPU cores, ASTRAL-MP analyzed the dataset in 10 min to 1.5 h. Using 24 cores and a GPU, we never needed more than 1 h for this large dataset ($n = k = 1000$).

3.5 Limits of scalability

On our two main datasets, ASTRAL-MP with 24 cores and one or more GPUs ran in under half an hour in every case. To further test the limits of scalability of ASTRAL-MP, we also tested two additional datasets with even higher n and k than our main datasets.

To test limits of n, we used an existing simulated dataset (20 replicates) with 10^4 species and 1000 gene trees, build using a similar procedure to SV1000 dataset (Zhang *et al.*, 2018). We used ASTRAL-MP with a single GPU and 24 cores to analyze this dataset. Of the 20 replicates, 19 of them finished within 48 h that we allotted them. The running times ranged between 5 and 32 h with a mean running time of 11.2 h.

To test limits of k, we used an insect transcriptomic dataset (Misof *et al.*, 2014; Sayyari *et al.*, 2017) with 144 taxa and 1478 genes, each with 100 bootstrapped gene trees (thus, a total $k = 1478 \times 100 = 147800$). One may want to run ASTRAL on the collection of bootstrapped gene trees; see prior work on pros and cons of this approach (Mirarab *et al.*, 2014b, 2016). For this dataset, we estimate that ASTRAL-III would need ≈ 180 days of running time. In contrast, ASTRAL-MP with four GPUs and 24 cores was able to analyze all 147 800 gene trees in 35 h. Interestingly, the resulting tree differed from the tree on 1478 best ML gene trees previously reported (Sayyari *et al.*, 2017) in only nine branches, of which eight had low localPP support in the original tree (<0.85).

The results on both of these test cases demonstrate that analyses that were not possible before are now possible with ASTRAL-MP. Both also took close to two days of running time, and thus, give us ballpark estimates of the limits of ASTRAL-MP.

Table 2. Speedup of ASTRAL-MP with GPU compared to ASTRAL-III and ASTRAL-MP run with one or the same number of CPU cores and no GPU

	Over ASTRAL-III				Over ASTRAL-MP with 1 core no GPU				Over ASTRAL-MP with the same number of cores no GPU			
	1&1	6&1	24&1	24&4	1&1	6&1	24&1	24&4	1&1	6&1	24&1	24&4
SV, 1000 species	7.0	15.5	20.9	21.7	2.1	4.7	6.3	6.5	2.1 (2.3)	1.3 (1.6)	0.93 (1.2)	0.97 (1.2)
Avian, fully resolved	50.3	83.7	113	158	11.4	19.0	25.7	35.9	11.4 (14.8)	3.29 (7.6)	1.2 (3.07)	1.7 (3.07)

Note: x&y in the header denotes running with x cores and y GPUs. Maximum possible speedup according to Amdahl's law is given parenthetically.

4 Discussion

ASTRAL-MP had perfect speedups up to 24 cores on datasets with a large number of gene trees. With GPUs, speedups ranged between 7× and 158× depending on the dataset and the number of CPU cores paired with GPU. Using several cores in addition to the GPU was very helpful as GPU parallelization targets only the weight calculation kernel, which is the majority of the running time without any parallelization but not after GPU parallelization. Using several GPUs on datasets with a low number of gene trees, however, does not confer additional benefits in our tests.

The scaling of ASTRAL-MP varied across datasets. The avian dataset, which has fewer species than the SV datasets, had a better speedup. But recall that the avian dataset has >14 times as many genes as SV. As our profiling results indicate (Fig. 1), the parallelization is the most efficient for the weight calculation step and less so for other steps. The reason is that the parallelization of weight calculation is synchronization-free (except for the final ordering of results) and is parallelized both for CPU and GPU (in addition to AVX2). In contrast, computing the similarity matrix, set X, or the cluster partitioning has less parallelism and more opportunities for conflict, and also these steps are parallelized only for CPU. Thus, these steps do not scale as well in our parallelization (Fig. 1). The most damaging step is the greedy consensus computation, which is not parallelized in our current implementation. However, Aberer *et al.* (2010) have developed methods for building the greedy consensus in parallel, and similar methods can be incorporated in future versions of ASTRAL-MP.

We expect the best speedups when the weight calculation step is dominant. The total weight calculation time scales as $O(k^{2.75})$ in the worst case and roughly with k^2 empirically (e.g. for typical datasets) as shown by Zhang *et al.* (2018). Steps other than weight calculation, in contrast, scale as $O(k^{1.75})$ in the worst case and close to $O(k)$ in practice. Thus, other things equal, increasing k leads to weight calculation becoming more dominant. In contrast, as n increases, the total weight calculation time scales with $O(n^{2.75})$ in the worst case and by roughly $O(n^2)$ empirically. However, other steps, too, scale as $O(n^2)$. Thus, the relative time of weight calculation slowly increases as n increases. To summarize, weigh calculations become more dominant as k and n increase, but this happens a lot faster with k. Thus, in general, we expect to see the best scaling of ASTRAL-MP with datasets that have large k. We observed this precise pattern on the avian dataset where reducing k deteriorated scaling.

A main limitation is that the GPU kernel only implements the weight calculation step. The other steps are not implemented in OpenCL mostly because of their relatively complex code and high memory bandwidth. Nevertheless, some of these other steps could perhaps be parallelized in future for GPU. For example, the computation of all possible cluster partitions may be doable in GPU. The challenge will be in doing so in a fashion that is compatible with SIMD parallelism and with minimal GPU–CPU communication. Moreover, the smaller steps that we have parallelized for CPU could perhaps be parallelized with more efficiency. For example, using blocking in the computation of the similarity matrix can reduce cache misses. Finally, better algorithms (perhaps using randomization) for computing greedy consensus trees could be developed.

We note that we were able to test ASTRAL-MP on K80 NVIDIA GPU nodes but no other GPU architectures. Whether similar patterns of scaling will be observed with other GPU machines will need further tests. Also, our OpenCL code can be used with devices other than GPUs, such as the Intel MIC architecture, which we have not tested here.

5 Conclusion

We improved the scalability ASTRAL, a method for reconstructing species trees from gene trees, using parallelization, randomization and vectorization. ASTRAL-MP is able to speed up computations for 100 times given a GPU and 24 cores. ASTRAL-MP was able to handle datasets with 10 000 species and 1000 gene trees within 48 h. Thus, ASTRAL-MP can analyze datasets that previous versions could not.

Funding

This work was supported by the National Science Foundation grant [IIS-1565862 to S.M. and C.Z.]; and a Frontiers of Innovation Scholars Program award from the University of California, San Diego [to J.Y.]. Computations were performed on the San Diego Supercomputer Center through XSEDE allocations, which is supported by the National Science Foundation grant [ACI-1053575].

Conflict of Interest: none declared.

References

Aberer,A.J. *et al.* (2010) Parallelized phylogenetic post-analysis on multi-core architectures. *J. Comput. Sci.*, **1**, 107–114.

Arcila,D. *et al.* (2017) Genome-wide interrogation advances resolution of recalcitrant groups in the tree of life. *Nat. Ecol. Evol.*, **1**, 20.

Blom,M.P.K. *et al.* (2016) Accounting for Uncertainty in Gene Tree Estimation: summary-Coalescent Species Tree Inference in a Challenging Radiation of Australian Lizards. *Syst. Biol.*, **66**, 352–366.

Bryant,D. *et al.* (2012) Inferring species trees directly from biallelic genetic markers: bypassing gene trees in a full coalescent analysis. *Mol. Biol. Evol.*, **29**, 1917–1932.

Bryant,D. and Steel,M. (2001) Constructing Optimal Trees from Quartets. *J. Algorithms*, **38**, 237–259.

Chifman,J. and Kubatko,L.S. (2014) Quartet inference from SNP data under the coalescent model. *Bioinformatics*, **30**, 3317–3324.

Degnan,J.H. and Rosenberg,N.A. (2009) Gene tree discordance, phylogenetic inference and the multispecies coalescent. *Trends Ecol. Evol.*, **24**, 332–340.

Heled,J. and Drummond,A.J. (2010) Bayesian inference of species trees from multilocus data. *Mol. Biol. Evol.*, **27**, 570–580.

Hosner,P.A. *et al.* (2015) Land connectivity changes and global cooling shaped the colonization history and diversification of New World quail (Aves: Galliformes: Odontophoridae). *J. Biogeogr.*, **42**, 1883–1895.

Jarvis,E.D. *et al.* (2014) Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science*, **346**, 1320–1331.

Kubatko,L.S. and Degnan,J.H. (2007) Inconsistency of phylogenetic estimates from concatenated data under coalescence. *Syst. Biol.*, **56**, 17–24.

Lafond,M. and Scornavacca,C. (2018) On the Weighted Quartet Consensus problem. *Theor. Comput. Sci.*, **769**, 1–17.

Larget,B.R. *et al.* (2010) BUCKy: gene tree/species tree reconciliation with Bayesian concordance analysis. *Bioinformatics*, **26**, 2910–2911.

Laumer,C.E. *et al.* (2015) Nuclear genomic signals of the 'microturbellarian' roots of platyhelminth evolutionary innovation. *ELife*, **4**, e05503.

Liu,L. (2008) BEST: Bayesian estimation of species trees under the coalescent model. *Bioinformatics*, **24**, 2542–2543.

Liu,L. and Yu,L. (2011) Estimating species trees from unrooted gene trees. *Syst. Biol.*, **60**, 661–667.

Liu,L. *et al.* (2010) A maximum pseudo-likelihood approach for estimating species trees under the coalescent model. *BMC Evol. Biol.*, **10**, 302.

Liu,L. *et al.* (2009) Estimating species phylogenies using coalescence times among sequences. *Syst. Biol.*, **58**, 468–477.

Maddison,W.P. (1997) Gene Trees in Species Trees. *Syst. Biol.*, **46**, 523–536.

Mirarab,S. *et al.* (2014a) Statistical binning enables an accurate coalescent-based estimation of the avian tree. *Science*, **346**, 1250463–1250463.

Mirarab,S. *et al.* (2016) Evaluating Summary Methods for Multilocus Species Tree Estimation in the Presence of Incomplete Lineage Sorting. *Syst. Biol.*, **65**, 366–380.

Mirarab,S. *et al.* (2014b) ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, **30**, i541–i548.

Mirarab,S. and Warnow,T. (2015) ASTRAL-II: coalescent-based species tree estimation with many hundreds of taxa and thousands of genes. *Bioinformatics*, **31**, i44–i52.

Misof,B. *et al.* (2014) Phylogenomics resolves the timing and pattern of insect evolution. *Science*, **346**, 763–767.

Mitchell,N. *et al.* (2017) Anchored phylogenomics improves the resolution of evolutionary relationships in the rapid radiation of *Protea* L. *Am. J. Bot.*, **104**, 102–115.

Mossel,E. and Roch,S. (2010) Incomplete lineage sorting: consistent phylogeny estimation from multiple loci. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, **7**, 166–171.

Pamilo,P. and Nei,M. (1988) Relationships between gene trees and species trees. *Mol. Biol. Evol.*, **5**, 568–583.

Pollard,D.A. *et al.* (2006) Widespread discordance of gene trees with species tree in drosophila: evidence for incomplete lineage sorting. *PLoS Genet.*, **2**, 1634–1647.

Rabiee,M. *et al.* (2019) Multi-allele species reconstruction using ASTRAL. *Mol. Phylogenet. Evol.*, **130**, 286–296.

Rannala,B. and Yang,Z. (2003) Bayes estimation of species divergence times and ancestral population sizes using DNA sequences from multiple loci. *Genetics*, **164**, 1645–1656.

Roch,S. and Steel,M. (2015) Likelihood-based tree reconstruction on a concatenation of aligned sequence data sets can be statistically inconsistent. *Theor. Popul. Biol.*, **100**, 56–62.

Rouse,G.W. *et al.* (2016) New deep-sea species of *Xenoturbella* and the position of *Xenacoelomorpha*. *Nature*, **530**, 94–97.

Sayyari,E. and Mirarab,S. (2016a) Anchoring quartet-based phylogenetic distances and applications to species tree reconstruction. *BMC Genomics*, **17**, 101–113.

Sayyari,E. and Mirarab,S. (2016b) Fast Coalescent-Based Computation of Local Branch Support from Quartet Frequencies. *Mol. Biol. Evol.*, **33**, 1654–1668.

Sayyari,E. *et al.* (2017) Fragmentary Gene Sequences Negatively Impact Gene Tree and Species Tree Reconstruction. *Mol. Biol. Evol.*, **34**, 3279–3291.

Tarver,J.E. *et al.* (2016) The Interrelationships of Placental Mammals and the Limits of Phylogenetic Inference. *Genome Biol. Evol.*, **8**, 330–344.

Vachaspati,P. and Warnow,T. (2015) ASTRID: accurate Species TRees from Internode Distances. *BMC Genomics*, **16** (Suppl. 10), S3.

Wickett,N.J. *et al.* (2014) Phylotranscriptomic analysis of the origin and early diversification of land plants. *Proc. Natl. Acad. Sci. USA*, **111**, 4859–4868.

Wu,Y. (2012) Coalescent-based species tree inference from gene tree topologies under incomplete lineage sorting by maximum likelihood. *Evolution*, **66**, 763–775.

Zhang,C. *et al.* (2018) ASTRAL-III: polynomial time species tree reconstruction from partially resolved gene trees. *BMC Bioinformatics*, **19**, 153.

Zwickl,D.J. *et al.* (2014) Disentangling methodological and biological sources of gene tree discordance on *Oryza* (Poaceae) chromosome 3. *Syst. Biol.*, **63**, 645–659.