Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning

Ivo Baar, Lukas Hübner, Peter Oettig, Adrian Zapletal, Sebastian Schlag, Alexandros Stamatakis
Institute for Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
{alexandros.stamatakis, sebastian.schlag}@kit.edu

Benoit Morel
Computational Molecular Evolution group
Heidelberg Institute for Theoretical Studies
Heidelberg, Germany
benoit.morel@h-its.org

***Abstract*—The so-called site repeats (SR) technique can be used to accelerate the widely-used phylogenetic likelihood function (PLF) by identifying identical patterns among multiple sequence alignment (MSA) sites, thereby omitting redundant calculations and saving memory. However, this complicates the optimal data distribution of MSA sites in parallel likelihood calculations, as the cost of computing the likelihood for individual sites strongly depends on the sites-to-cores assignment. We show that finding a 'good' sites-to-cores assignment can be modeled as a hypergraph partitioning problem, more specifically, a specific instance of the so-called judicious hypergraph partitioning problem. We initially develop, parallelize, and make available HyperPhylo, an efficient open-source implementation for this flavor of judicious partitioning where all vertices have the same degree. Using empirical MSA data, we then show that sites-to-core assignments computed via HyperPhylo are substantially better than those obtained via a previous naïve approach for phylogenetic data distribution under SRs.**

***Index Terms*—phylogenetic inference, data distribution, parallel efficiency, judicious hypergraph partitioning**

I. Introduction

Phylogenetic inference, that is, the reconstruction of evolutionary trees based on the molecular sequence data of the species under study, has numerous applications in medical and biological research. Thus, tools for phylogenetic inference such as RAxML [1] or MrBayes [2] are widely used and highly cited. With the advent of new molecular sequencing technologies, the field faces a substantial scalability challenge as phylogenetic analyses of empirical data sets under the standard likelihood models of sequence evolution can require thousands to millions of CPU hours. Most state-of-the art tools for phylogenetic inference have already been parallelized and are regularly deployed on large clusters for production runs. Therefore, an optimal data distribution is key for achieving 'good' parallel efficiency.

Here, we analyze specific properties of the, so far, most complex variant of the phylogenetic data distribution problem. We initially transform the problem into a hypergraph partitioning problem and show that it corresponds to a particular instance of the so-called judicious hypergraph partitioning problem [3] (henceforth, denoted as judicious partitioning). As no efficient implementation of a judicious partitioning algorithm was available, we develop, parallelize, and present HyperPhylo[1] an efficient open-source implementation of the specific judicious partitioning flavor where all vertices have the same degree. We then apply HyperPhylo to our phylogenetic data distribution problem and compare the data distribution computed via judicious partitioning to the data distribution obtained via a simple ad hoc heuristic [4]. We find that phylogenetic data distributions computed with HyperPhylo are substantially better than those obtained via the ad hoc heuristic.

The remainder of this paper is organized as follows. In Section II we introduce the phylogenetic preliminaries, motivate as well as state our problem, and discuss related work. In the subsequent Section III we introduce judicious partitioning and describe our efficient algorithm, implementation, and parallelization. In Section IV we present experimental results. We conclude in Section V.

II. Phylogenetic Preliminaries

The input to a phylogenetic inference is a *multiple sequence alignment* (MSA) which comprises the sequences of the species under study. The MSA columns are also called *sites*. In empirical phylogenetic analyses, the sites of the MSA are typically subdivided into p disjoint *partitions* (e.g., corresponding to individual genes) for which a separate set of likelihood model parameters is estimated. Given the MSA and a partitioning scheme, one can calculate the phylogenetic likelihood function (PLF) on a given candidate tree. PLF calculations typically account for 85-95% of total running time in current tools such as RAxML and MrBayes. Thus, the PLF

Part of this work was supported by the Klaus-Tschira foundation and DFG grant STA 860/6-1.

[1]available at https://github.com/lukashuebner/HyperPhylo

is *the* target function for optimization and parallelization. PLF calculations are typically parallelized over MSA sites as per-site likelihood calculations are independent from each other. Once all per-site likelihoods have been computed, only one collective communication is required to compute the product (or the sum over the logarithms) over all per-site likelihoods to obtain the overall likelihood score. For instance, if there is only one partition, the MSA sites can be distributed to the cores in a cyclic fashion. However, given a list of p partitions and c cores for a partitioned MSA, finding a 'good' (w.r.t. load balance) site-to-core assignment becomes more challenging [5]. As the per-site likelihood calculation cost is identical for all sites, each partition has a computation cost that is linear in the number of sites it comprises.

At an abstract level, the data distribution problem can be stated as follows: For a given number of c cores we need to balance the data (MSA sites) among the c cores such that the maximal per-core load (sites assigned to core) is minimized. The MSA partitions *are* divisible, since a partition consists of sites whose likelihoods can be computed independently. We can thus improve load balance by splitting partitions into disjoint sets (hereafter referred to as *blocks*) of sites that are allocated to distinct cores. However, splitting a partition does not come for free as the computational cost of a partition has two components: All partitions incur an identical constant base cost α at every core (see [5] for details). Thus, if we split a partition among cores, each block of that partition incurs this base cost α. For instance, if the sites of a partition are assigned to two distinct cores, α needs to be computed redundantly (i.e., once per core). The second component of the per-partition PLF calculation cost is the variable cost β which is linear in the number of sites per partition. To maximize parallel efficiency, we need to minimize redundant calculations of α by only splitting partitions when necessary, while distributing sites evenly among cores. In [5], we have shown that the problem is NP-hard and introduced an approximation algorithm which solves it in practice.

Thus far, we assumed that per-site calculations costs are identical for each site. That means, if we split a partition into two blocks of equal size among two cores, the respective computational cost is $\alpha + \beta/2$ per core. With the introduction of *site repeats* (SR [6]), an algorithmic optimization of the PLF, the above does not hold. The technique takes repeating (shared) patterns in distinct partial sites (subsets of characters of MSA sites) of a partition into account to reduce per-site calculation cost. The amount of savings depends on the current tree topology *and* the MSA. Site repeats reduce the cost β by detecting and re-using identical intermediate results among two or more sites belonging to the same partition. Thus, distinct sites of a single partition now exhibit *varying* computational cost. If we assign two sites that share a large fraction of common intermediate results to different cores, the accumulated computational cost and memory for those two sites will increase.

The above complicates data distribution of parallel SR-based PLF calculations. We can still split a single partition among several cores, but now have to sacrifice some SRs which induces additional redundant computations. We denote this phenomenon as *repeat loss*. Note that, computing a lower bound for the minimal repeat loss is straight-forward, as the minimum amount of SR-based calculations can easily be computed for a sequential execution of the PLF (see [7]). Thus, for SR-based parallel PLF computations we might have to also conduct redundant computations for component β. Hence, the cost β increases if sites that belong to the same partition and share repeats are allocated to distinct cores [7]. We illustrate the impact of SRs on load balance via a simple example (Figure 1). Given a tree, assume a single MSA partition comprising 5 sites that has to be split among 2 cores, and contains some SRs (highlighted by colored boxes). In Figure 1 we depict the required PLF calculations in terms of so-called Conditional Likelihood Vector (CLV) updates—involving a substantial amount of floating point operations—per core with and without SRs, respectively.

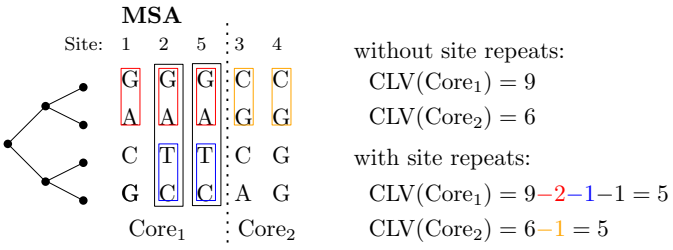

Fig. 1. One MSA partition split among two cores and its respective PLF computation costs with and without SRs.

Our goal is thus to devise a data distribution algorithm that yields a 'good' trade-off between minimizing repeat loss and load balance in terms of floating point operations per core to, in turn, reduce overall parallel PLF execution times.

We initially outline our current ad hoc randomized data distribution algorithm (RDDA [4]) and then formally state the problem. Given a MSA, and a partitioning scheme, RDDA initially computes the fraction of expected SR-induced computational savings for each partition over a set of random trees. We have empirically assessed that using random trees is sufficient to account for the variance of topology-dependent SR-induced savings. RDDA then simply distributes the partitions according to these fractions using the algorithm from [5]. As RDDA is unaware of repeat loss, that is, sites that share a large proportion of repeats *can* be assigned to different cores. Hence, the obtained data distribution can become sub-optimal. To the best of our knowledge, the RDDA algorithm and our previous exploratory work [7] constitutes the only related work on this problem.

For the remainder of this paper, we will assume that (i) we are given a fixed phylogenetic tree and (ii) that we only have *one* MSA partition (i.e., $p := 1$) whose sites we need divide into two or more blocks in order to assign them to two or more cores, since the coarse-grain distribution (i.e., assigning *entire* MSA partitions to cores) is already handled well by RDDA. To now formally state our problem, we introduce some

definitions.

Let S be a set of n MSA sites and c the given number of cores. By $\mathcal{A}_c(S)$ we denote the set of all possible sites-to-cores assignments to c cores (i.e., all possible partitions of S into c blocks). An element $Z \in \mathcal{A}_c(S)$ then represents a specific sites-to-cores assignment where $|Z| = c$.

Let $flops(\zeta)$ be the number of accumulated floating point operations required to compute the per-site likelihoods of all sites ζ assigned to a core for one specific sites-to-cores assignment Z in $\mathcal{A}_c(S)$. Note that it is straight-forward to exactly compute $flops(\zeta)$ by calculating the SRs of the sites in ζ, without having to carry out the actual compute-intensive likelihood calculations.

Given these definitions, we can now state the simple problem of splitting just *one* ($p := 1$) MSA partition among c cores. Find the sites-to-cores assignment $Z \in \mathcal{A}_c(S)$ that minimizes:

$$f(Z) = \max_{\zeta \in Z} flops(\zeta).$$

III. Judicious Hypergraph Partitioning

A. Preliminaries

An *unweighted, undirected hypergraph* $H = (V, E)$ is defined as a set of n vertices V and a set of m hyperedges/nets E, where each net e is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins*. A vertex v is *incident* to a net e if $v \in e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The *degree* of a vertex v is $d(v) := |\mathrm{I}(v)|$. With Δ, we denote the maximum degree of a hypergraph. The *size* $|e|$ of a net e is the number of its pins. A *k-way partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$, and $V_i \cap V_j = \emptyset$ for $i \neq j$. Given a k-way partition Π, the number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. Net e *is connected* to block V_i if $\Phi(e, V_i) > 0$.

Judicious hypergraph partitioning (JDP) strives to find a k-way partition Π of a hypergraph H that minimizes the *maximum* number of nets a block is connected to. In other words, judicious partitioning attempts to minimize $\max(L(V_1), \ldots, L(V_k))$, where $L(V_i) := |\{e \in E \mid \Phi(e, V_i) > 0\}|$ is the load of a block V_i. The problem is known to be NP-hard [8] and has mainly been studied in the context of extremal combinatorics [9]–[11]. To the best of our knowledge, Tan *et al.* [3] were the first to investigate algorithmic aspects of the problem.

JDP differs significantly from the classical k-way hypergraph partitioning problem (HGP). While JDP is *solely* concerned with minimizing the *maximum* number of nets a block is connected to, the goal of HGP is to partition the vertex set into k disjoint blocks of *bounded size* (at most $1 + \varepsilon$ times the average block size) while minimizing an objective function such as cut-net (i.e., the weight of all nets that connect more than two blocks) or connectivity (which additionally takes into account the actual number of blocks connected by a cut net). Because of this difference, we do not use existing HGP tools such as hMetis [12] and KaHyPar [13], which are geared towards computing vertex-balanced partitions with small cuts.

B. Connection between Judicious Partitioning and the Phylogenetic Data Distribution Problem

The input of the phylogenetic data distribution problem is one MSA partition with n sites s_i, a binary tree topology specifying the order of PLF calculations (and hence, the SRs), and a given number of cores c. Two MSA sites s_i and s_j are repeats of one another at an inner node q of the phylogenetic tree, if the partial sites $s_i|q$ and $s_j|q$ induced by the subtree rooted at q are *exactly* identical (e.g., the colored boxes in Figure 1 are such partial sites). By partial site, we denote those nucleotides of a MSA site that are contained in a subtree rooted at q. If two partial MSA sites $s_i|q$ and $s_j|q$ are repeats of one another we say that they belong to to the same *repeats class*. Thus, each partial site will be in exactly one repeats class at every inner node of the phylogenetic tree. Note that if a site does not repeat any other site at an inner node, it is assigned to a separate repeats class of size 1. At the root of our example phylogeny in Figure 1, core 1 has two repeat classes (sites $\{1\}, \{2, 5\}$) and core 2 also has two repeat classes (sites $\{3\}, \{4\}$). The number $flops(\zeta)$ of accumulated floating point operations required to compute the per-site likelihoods of all sites ζ assigned to a core is directly proportional to the accumulated number of repeats classes over all inner nodes of the given phylogenetic tree. For instance, the number of accumulated repeats classes for core 1 in our example is $1 + 2 + 2$.

We can now use a hypergraph $H = (V, E)$ to formulate our data distribution problem: Each site s_i corresponds to a vertex $v_i \in V$ and each repeats class $r = \{s., \ldots, s.\}$ induced by an inner node of the phylogenetic tree corresponds to a hyperedge $e \in E$ of size $|r|$ containing the corresponding vertices $e = \{v., \ldots, v.\}$. Thus, the number of vertices in the hypergraph is equal to the number of sites of the MSA and the number of hyperedges corresponds to the total number of repeats classes induced by all inner nodes of the entire tree topology. An example is shown in Figure 2. Since each inner node of a phylogenetic tree incurs computations for all MSA sites, each site is in *exactly one* repeat class for each inner node of the phylogeny. Therefore, each inner node incurs one hyperedge for each MSA site and the degree of each vertex v_i equals the number of repeats classes the corresponding site s_i belongs to. By construction, the degree of every vertex $v \in V$ is equal to the number of inner nodes of the phylogeny, that is, $d(v) = \Delta, \forall v \in V$.

The site-to-cores assignment $Z \in \mathcal{A}_c$ that minimizes $\max_{\zeta \in Z} flops(\zeta)$ for a given number of cores c therefore corresponds to a judicious k-way partition of the corresponding hypergraph H into $k = c$ blocks.

A note on terminology: In phylogenetic inference, a partition of a MSA corresponds to a subset of sites (e.g., corresponding to individual genes), whereas in graph and hypergraph partitioning literature the term partition is used in the mathematical sense, that is, a partition is a grouping of

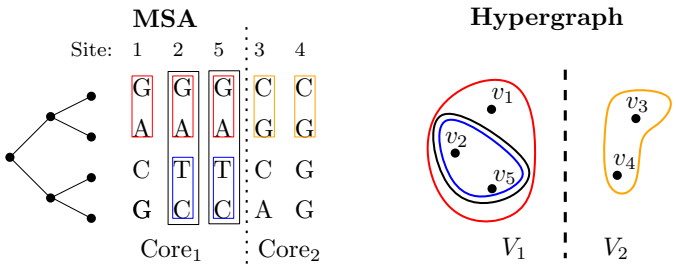

Fig. 2. Example of a partitioned MSA and the corresponding partition $\Pi = \{V_1, V_2\}$ of the hypergraph H. Nets of size 1 have been omitted for clarity.

the vertex set into non-empty, pairwise-disjoint subsets called blocks. In the following, we therefore explicitly use the term *MSA partition* to refer to a subset of sites of a MSA.

C. The Judicious Partitioning Algorithm of Tan et al. [3]

The algorithm is based on the insight that the objective score, that is, the cost of a k-way partition Π of a hypergraph H can be bounded both, from below, and from above. The lower bound is obtained by observing that the maximum load $L(V_i)$ of a block V_i cannot be smaller than the maximum degree Δ. This is because the maximum-degree vertex has to be assigned to one of the blocks. An upper bound for $L(V_i)$ is given by the number of hyperedges $|E|$.

The algorithm does not directly partition the hypergraph into k blocks while minimizing the maximum load. Instead, it repeatedly computes partitions of increasing cost $\Delta + d$ for $0 \leq d \leq |E| - \Delta$ without restricting the number of blocks a priori. Once the number of blocks k' of a computed partition is $\leq k$, the algorithm stops and returns the current partition. Otherwise (i.e., if $k' \gneq k$), the algorithm uses the current solution as input for the next round/iteration. Thus, it increases the cost in each round to find a partition into fewer blocks.

To compute a partition with a solution cost of $\Delta + d$ the algorithm relies on solving a minimum set cover problem. In the first pass of the algorithm (i.e., $d = 0$) the set cover problem is constructed as follows: The universe $\mathcal{U}$ is defined to be the set of vertices. Each vertex v is implicitly represented via its incident nets $\mathrm{I}(v)$, that is, vertex v_i is encoded as $\mathrm{I}(v_i)$. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_{\binom{m}{\Delta+d}}\}$ be the set that contains all combinations for choosing $\Delta + d$ out of m hyperedges. The collection $\mathcal{S}$ of sets that cover U is then defined as $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots\}$, where $\mathcal{S}_j = \{\mathrm{I}(v) \mid \mathrm{I}(v) \subseteq t_j, v \in V\}$. Then, the standard greedy algorithm (i.e., in each round, choose the set that contains the largest number of uncovered elements) is used to find a sub-collection $\mathcal{S}^*$ of $\mathcal{S}$ that covers $\mathcal{U}$. In later rounds of the algorithm, $\mathcal{U}$ is replaced by the current solution $\mathcal{S}^*$. In each iteration, $\mathcal{S}^*$ thus induces a partition of the elements in $\mathcal{U}$, which in turn corresponds to a partition of the vertex set of the hypergraph. We provide the corresponding pseudo code in Algorithm 1. An example is shown in Figure 3.

Algorithm 1: Judicious Partitioning Algorithm of Tan et al. [3]

Input: Hypergraph $H = (V, E)$ with n nodes, m hyperedges
Input: number of blocks k
Output: k-way partition Π of H, solution cost $\Delta + d$
$\mathcal{U} := \{\mathrm{I}(v_1), \ldots, \mathrm{I}(v_n)\}$ *// incidence encoding of vertices*
for $d := 0$; $d \leq m - \Delta$; $++d$ **do** *// increasing cost*
  $\mathcal{T} :=$ *all* $\binom{m}{\Delta+d}$ *comb. to select* $\Delta + d$ *out of* m *nets*
  $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots\}$, with $\mathcal{S}_j = \{\mathrm{I}(v) \mid \mathrm{I}(v) \subseteq t_j, v \in V\}$
  $\mathcal{S}^* :=$ `GreedySetCover`$(\mathcal{U}, \mathcal{S})$
  if $|\mathcal{S}^*| > k$ **then**
    $U := \mathcal{S}^*$
  else
    $\Pi :=$ partition induced by $\mathcal{S}^*$
    return $(\Pi, \Delta + d)$

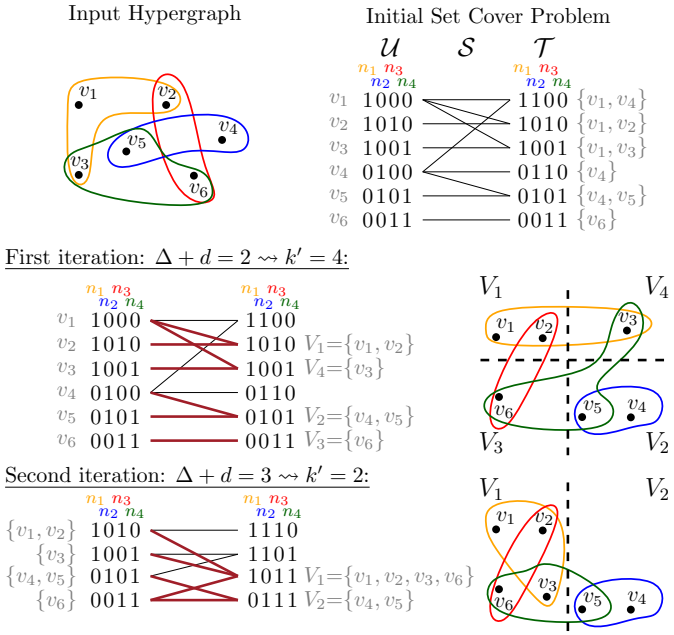

Fig. 3. Modified example from Tan *et al.* [3]: A hypergraph with 6 vertices and 4 nets is partitioned into $k = 2$ blocks using their algorithm. The final partition has a cost of $\Delta + d = 3$. Note that sets $\mathcal{S}_j$ are represented as *edges* in the bipartite graph.

D. Towards an Efficient Implementation

The algorithm might already require $\mathcal{O}(m^{\Delta})$ time for the first round, as $\mathcal{S}$ can contain *all* $\binom{m}{\Delta}$ combinations. This is computationally infeasible, even for small hypergraphs.

a) Exploiting Instance-specific Features.: However, we can exploit the fact that, by construction of our specific instance, *all* vertices have the same degree, that is, $\forall v \in V : d(v) = \Delta$. Thus, all elements in $\mathcal{U}$ are represented by exactly the same number of hyperedges. Therefore, it is possible to omit the first round ($d = 0$) of the algorithm as it will only group vertices u and v that have the exact same incidence structure (i.e., $\mathrm{I}(u) = \mathrm{I}(v)$). Instead, we perform one pass over the elements of $\mathcal{U}$ and remove all duplicate entries (i.e.,

vertices with the same incidence structure).

In all subsequent iterations, the representations of elements in $\mathcal{U}$ and $\mathcal{S}$ will only differ in *one* additional hyperedge. Thus, it is possible to generate all elements in $\mathcal{S}$ by combining all pairs of elements in $\mathcal{U}$. Hence, generating the initial collection of sets $\mathcal{S}$ only requires $\mathcal{O}(\Delta n^2)$ time, as we can compute the union of all pairs of elements in $\mathcal{U}$ and keep exactly those where $|\mathcal{S}_j| = |\mathcal{U}_j| + 1$. Since the greedy set cover algorithm always chooses the next set $\mathcal{S}_i$ such that it contains the largest number of *uncovered* elements, it is not necessary to compute all elements $\mathcal{S}_i$ with $|\mathcal{S}_i| = 1$ a priori. Instead, these elements can be generated on-the-fly when the greedy algorithm has no elements with $|\mathcal{S}_i| \geq 2$ left to process *and* has not found a solution that covers $\mathcal{U}$. This approach substantially reduces the overall memory consumption, since generating $\mathcal{S}$ naïvely would lead to $\mathcal{O}(nm)$ $\mathcal{S}_i$ elements with $|\mathcal{S}_i| = 1$, while our approach generates at most n such elements.

b) Improving Convergence Behavior.: Henceforth, we denote such elements in $\mathcal{S}$ that are generated on-the-fly and only cover one element $u \in \mathcal{U}$ as *filler* elements. When the greedy algorithm has no elements $|\mathcal{S}_i| \geq 2$ left and therefore adds a filler element to the solution $\mathcal{S}^*$, the block of the hypergraph partition corresponding to this filler element does not change. It will contain the same vertices as in the previous iteration, since filler elements only cover a single element of $\mathcal{U}$. While this is not a problem in theory, it can substantially affect the convergence speed of the algorithm and hence overall running time. This is because with a large fraction of filler elements, the current number of blocks k' converges only slowly to k, since many blocks will remain stable during subsequent iterations. Furthermore, this behavior can deteriorate solution quality, since more iterations will be required for merging two blocks and the solution cost (i.e., $\Delta + d$) increases with each iteration. By employing the greedy set cover algorithm, the judicious partitioning algorithm of Tan *et al.* [3] chooses an *arbitrary* filler element out of all combinations of $\Delta + d$ nets that constitute a superset of u. Since the chosen filler set $\mathcal{S}_j$ will be included in $\mathcal{S}^*$ and thus becomes a new element of $\mathcal{U}$ in the next iteration, some choices of $\mathcal{S}_j$ may be more difficult to cover using $\Delta + d + 1$ nets than others. In the worst case, the element will again only be covered by another filler element with $\Delta + d + 1$ nets.

To alleviate this issue, we deploy the following strategy when choosing a filler element for a yet uncovered element $u \in \mathcal{U}$: We first compute the symmetric set difference between u and all other elements $u' \in \mathcal{U}$. We then choose the filler element $\mathcal{S}_j$ such that the difference to the closest element u' decreases. This increases the chance that both, u, and u' can be covered using a non-filler $\mathcal{S}$ element in subsequent iterations. We will provide additional details on this in the paragraph after next.

c) Representing $\mathcal{U}$ and $\mathcal{S}$ Sets.: Both, the elements of $\mathcal{U}$, and the sets of $\mathcal{S}$ comprise elements that are represented by hyperedges of the hypergraph. While Tan *et al.* [3] do not explicitly discuss how set elements are represented, their example (an enhanced version of which is shown in Figure 3) already hints at a potential implementation: each element can be represented as a m-bit bitvector, where a set bit at the ith position denotes that hyperedge i is part of the set element. Thus, set unions and intersections become bit-wise *and* and *or* operations, respectively. However, if m is large and Δ is small, these bitvectors become large and sparse and waste space as well as time. We therefore use a dedicated sparse bitvector representation where each element is explicitly represented by a set of hyperedge IDs. Evidently, there is a time and memory trade-off between the sparse and the bit-wise set operations and representations.

The two alternative representations need to support the following operations: (i) Compute the symmetric set difference between two sets, (ii) given two sets a and b, add an element from the relative complement $a \setminus b$ to b (supporting our strategy for handling filler elements as discussed above), and (iii) merge two sets in order to create new $\mathcal{S}$ elements from two $\mathcal{U}$ elements. In the dense bitvector representation (i) can be implemented by computing a `XOR` on the two bitvectors. The size of the set difference then corresponds to the number of set bits in the result (which can be efficiently counted using `popcount` instructions), (ii) amounts to simple bit-wise operations, and (iii) corresponds to a bit-wise `OR`. In the sparse representation, (i) is implemented using a merge-based symmetric set difference, (ii) simply scans both sets and adds the corresponding element to the set, while (iii) amounts to a merge-based set-union operation. Note that for the sparse representations these operations require the sets to be sorted.

While the running times of all operations are constant for the dense representation, running times of the sparse representation slowly increase with growing set sizes in successive iterations of the algorithm. Given that we typically have to analyze large and comparatively sparse input instances, this performance degradation is offset by the acceleration of the initial iterations that are the most time consuming. In our experiments, the sparse representation performed better than the dense representation if set elements contained $< 0.1\%$ of the total elements. Also, the sparse representation outperformed the dense representation overall.

d) Putting Things together and Parallelization.: In Algorithm 2 we provide the pseudo code for HyperPhylo. We generate the set $\mathcal{S}$ by iterating over all pairwise combinations of elements in $\mathcal{U}$, checking whether the union of the two elements creates a valid $\mathcal{S}$ element (i.e., $|u_1 \cup u_2| = |u_1| + 1$), and adding the new element to $\mathcal{S}$ if the condition is fulfilled. Furthermore, we calculate the minimal distances between elements of $\mathcal{U}$ and store a potential filler element for each element $u \in \mathcal{U}$ in a map d_{min}. Since there are no data dependencies between these operations, they can be performed in parallel for all pairs $(u_1, u_2) \in \mathcal{U}$.

We then compute $\mathcal{S}^*$ by first executing the greedy set cover algorithm sequentially until no more elements of $\mathcal{S}$ can be used to cover $\mathcal{U}$. In this case, we have to resort to filler elements in order to cover the remaining $\mathcal{U}$ elements. Filler elements are created by adding an element x_i from the relative complement of $d\_min[u]$ and u to the element u. Again, this can be done

Algorithm 2: HyperPhylo Judicious Partitioning

Input: Hypergraph $H = (V, E)$ with n nodes, m hyperedges
Input: number of blocks k
Output: k-way partition Π of H, solution cost $\Delta + d$

$\mathcal{U} := \{\mathrm{I}(v_1), \ldots, \mathrm{I}(v_n)\}$ *// incidence encoding of vertices*
for $d := 0;\ d \leq m - \Delta;\ {+}{+}d$ **do** *// increasing cost*
  $\mathcal{S} := \emptyset$
  $d_{\min} = [\,]$ *// map $u_i \rightarrow u_j$*
  parallel for $\{\{u_1, u_2\} \mid u_1, u_2 \in \mathcal{U}\}$ **do** *// compute $\mathcal{S}$*
    $d := |u_1 \triangle u_2|$ *// symmetric set difference*
    if $d = 2$ **then**
      $\mathcal{S}_i := \{u_1 \cup u_2\}$ *// new $\mathcal{S}$ element*
      $\mathcal{S} := \mathcal{S} \cup \mathcal{S}_i$
    else *// maintain distances to determine filler elements*
      if $|d_{min}[u_1]| < d$ **then** $d_{\min}[u_1] := u_2$
      if $|d_{min}[u_2]| < d$ **then** $d_{\min}[u_2] := u_1$
  $\mathcal{S}^* := \emptyset$
  while $\mathcal{S}^* \neq U$ **do** *// sequential greedy set cover*
    // $\mathcal{S}'$ contains largest # of uncovered elements
    $\mathcal{S}' := \arg\max_{\mathcal{S}_i \in \mathcal{S}} (\mathcal{S}_i \setminus \mathcal{S}^*)$
    if $\mathcal{S}' \neq \emptyset$ **then** *// $\mathcal{S}' = \emptyset \rightsquigarrow$ filler elements needed*
      $\mathcal{S}^* := \mathcal{S}^* \cup \mathcal{S}'$
      $\mathcal{S} := \mathcal{S} \setminus \mathcal{S}'$
    else break
  parallel for $\{u \mid u \in \mathcal{U} \wedge u \notin \mathcal{S}^*\}$ **do** *// uncovered elems.*
    $x := \{d_{\min}[u] \setminus u\}$
    // adding one $x_i \in x$ to u creates filler element for u
    $u := u \ \cup \{x_i \mid x_i \in x\}$
    $\mathcal{S}^* = \mathcal{S}^* \cup u$
  if $|\mathcal{S}^*| > k$ **then** $\mathcal{U} := \mathcal{S}^*$
  else
    $\Pi :=$ partition induced by $\mathcal{S}^*$
    return $(\Pi, \Delta + d)$

in parallel for all yet uncovered elements of $\mathcal{U}$.

e) Implementation Details.: Our dense set representation exploits x86 vector instructions through appropriate bit-vector starting address alignments in the memory allocation. Therefore, all loops that iterate over consecutive 64-bit blocks are vectorized automatically by the compiler.

In the greedy set cover algorithm, we need to repeatedly check if the entire set $\mathcal{U}$ is already covered by $\mathcal{S}^*$. Instead of assessing if both sets, that is, the already covered elements of $\mathcal{U}$ and $\mathcal{U}$, are identical, it suffices to compare the corresponding set sizes.

The *parallel for* sections in Algorithm 2 are parallelized using OpenMP. As already mentioned, both parallel sections exhibit multiple possible execution paths resulting in a data-dependent running time performance. Therefore, we use the OpenMP dynamic workload scheduler to attain improved load balance.

The shared-memory data structures we use from the Intel® Threading Building Blocks library [14] are listed in Table I.

TABLE I
INTEL® TBB DATA STRUCTURES USED IN OUR IMPLEMENTATION.

Data Structure	TBB Implementation
$\mathcal{U}$	`tbb:concurrent_unordered_set`
$\mathcal{S}$	`tbb::concurrent_unordered_set`
$\mathcal{S}^*$	`tbb::concurrent_vector`
$d_{\min}$	`tbb::concurrent_unordered_map`

IV. EXPERIMENTAL RESULTS

HyperPhylo is written in `C++` and was compiled using `CLANG v.6.0.0` with the `-O3` flag. We also used OpenMP 3.1 [15] as well as the Intel® Threading Building Blocks library version 2017.0 [14] for shared-memory parallelism.

A. Hardware

We performed computational experiments on 3 distinct hardware platforms. Platform **A** has four 8-core Intel® Xeon® E5-4640 (Sandy Bridge) processors running at 2.4 GHz and 512 GB main memory. Platform **B** consists of two 16-core Intel® Xeon® E5-2683v4 (Broadwell) processors (2.1 GHz) and has 512 GB main memory. Platform **C** contains a 32-core AMD EPYC™ 7551P processor (2.0 GHz) and has 256 GB main memory. All platforms run Ubuntu 18.04.1 LTS.

B. Verification

To increase our confidence that the implementation is correct, we verified its results on a small test case for which we explicitly computed the expected result using pen and paper. We also integrated an, as large as possible as well as reasonable, number of assertions into the code. Finally, we verified that all results represented a valid sites-to-cores mapping, that is, that each site of the input MSA is indeed assigned to exactly one core.

C. Instances (MSAs)

For our experiments we use a total of four MSAs containing empirical sequence data from previous collaborative studies with biologists. The three smaller MSAs [16] comprise 59, 128, and 404 sequences, respectively. We will henceforth refer to these as *D59*, *D128*, and *D404*. *D59* has 7 MSA partitions and 6 951 sites in total, *D128* has 34 MSA partitions and 29 198 sites, and *D404* has 11 MSA partitions and 13 158 sites. From each of these four data sets, we extract both the smallest (referred to using suffix *-s* in Table II) and the largest (suffix *-l*) MSA partition. Furthermore, we use one substantially larger MSA, taken from the one thousand insect transcriptome evolution project [17], which we refer to as *supermatrix*. From this data set, which contains 50 MSA partitions and 413 459 sites in total, we selected four MSA partitions comprising 11 756, 20 753, 31 854, and 170 859 sites, respectively, with the last MSA partition being the largest in the entire *supermatrix* data set.

For each instance, we generated corresponding random trees and subsequently transformed them into hypergraphs as described in Section III-B. Note that each hypergraph corresponds to only *one* MSA partition of the respective

TABLE II
PROPERTIES OF THE HYPERGRAPHS GENERATED FROM OUR DATA SETS. D59-S REPRESENTS THE SMALLEST MSA PARTITION OF DATA SET *D59*, D59-L THE LARGEST ETC. SM-P1 IS MSA PARTITION 1 IN THE *supermatrix* DATA SET. THE NUMBER OF SITES OF AN INSTANCE IS EQUAL TO THE NUMBER OF VERTICES OF THE CORRESPONDING HYPERGRAPH.

Instance	Vertices	Hyperedges	Pins	Δ
D59-s	160	671	9 120	57
D404-s	588	2 525	236 376	402
D128-s	204	1 170	25 704	126
D59-l	2 183	10 205	124 431	57
D404-l	2 161	40 648	868 722	402
D128-l	2 933	23 618	369 558	126
sm-p24	11 756	99 713	1 669 352	142
sm-p12	20 753	163 514	2 946 926	142
sm-p3	31 854	185 662	4 523 268	142
sm-p1	170 859	196 836	24 261 978	142

TABLE III
PROPERTIES OF THE HYPERGRAPHS (DERIVED FROM THE *supermatrix* DATA SET) USED IN WEAK AND STRONG SCALING EXPERIMENTS.

# Sites	Scaling	Vertices	Hyperedges	Pins	Δ
5 000	weak	5 000	57 631	710 000	142
10 000	weak	10 000	100 256	1 420 000	142
20 000	weak	20 000	132 093	2 840 000	142
40 000	weak	40 000	153 489	5 680 000	142
45 000	weak	45 000	158 230	6 390 000	142
50 000	weak/strong	50 000	163 381	7 100 000	142
60 000	weak	60 000	173 496	8 520 000	142
80 000	weak	80 000	192 543	11 360 000	142
85 000	weak	85 000	197 064	12 070 000	142
90 000	weak	90 000	200 072	12 780 000	142
100 000	weak	100 000	208 215	14 200 000	142
120 000	weak	120 000	200 967	17 040 000	142
125 000	weak	125 000	204 300	17 750 000	142
130 000	weak	130 000	207 048	18 460 000	142
140 000	weak	140 000	213 763	19 880 000	142
160 000	weak	160 000	229 558	22 720 000	142

MSA. The basic properties of the hypergraphs are provided in Table II.

D. Parallel Scalability

To the best of our knowledge, there is no publicly available implementation of any judicious hypergraph partitioning algorithm. Therefore, we use the sequential implementation of HyperPhylo as a baseline for our speedup measurements. We demonstrate the scalability of our algorithm via the strong and weak scaling experiments summarized in Figures 4 and 5, respectively. Properties of the hypergraphs used in these experiments are summarized in Table III. We performed strong scaling experiments on all three hardware platforms. For strong scaling, the hypergraph is created by extracting a subset of 50 000 sites from the *supermatrix* data set. The corresponding hypergraph is then partitioned into $k = 50$ blocks (i.e., to distribute these sites onto $c = 50$ cores). The comparison of the dense and sparse representations in Figure 4 shows that the sparse representation performs better on all hardware platforms. We also observe that the sparse representation is more scalable than its dense counterpart as parallel efficiency of the dense representation drops noticeably when using more than one socket. Absolute running times for

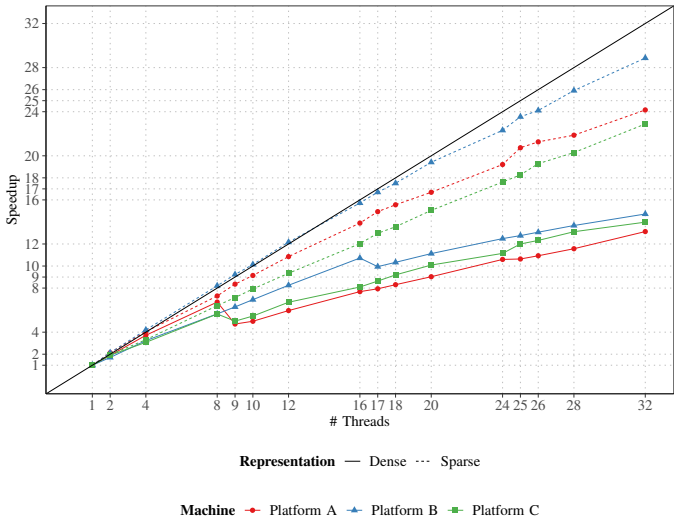

Fig. 4. Speedup of HyperPhylo with increasing number of cores. The MSA instance contains 50 000 sites and the corresponding hypergraph is always partitioned into $k = 50$ blocks.

the strong scaling experiments are shown in Table IV. Note that these times are small compared to the times required for conducting production level inferences of phylogenetic trees under maximum likelihood on large clusters or supercomputers using hundreds or thousands of cores. Overall, when using all 32 cores of our three hardware platforms, the running time for judicious partitioning decreases by more than an order of magnitude. We generated data sets such that the number of sites divided by the number of cores used for computing the judicious partitioning remained constant at 5000 sites per core (e.g., 160 000 sites for 32 cores). The corresponding hypergraphs are always partitioned into $k = 160$ blocks. We use this setting because for a parallel PLF computation a core conducting likelihood calculations needs to be assigned at least 1 000 alignment sites (i.e., 160 000/1 000 = 160) for the PLF parallelization to scale. Note that Δ remains the same for all hypergraphs, as the same tree topology is used for all instances. The results depicted in Figure 5 show that our algorithm scales well on both platforms with an increasing number of threads. For small input sizes, the dense representation outperforms the sparse representation because it uses highly efficient vectorized bit operations, while the sparse representation employs element-wise integer comparisons. However, as the input size increases, the bit-wise encoding becomes less efficient. Therefore, the sparse representation performs better in these cases. For 160 000 sites, for example, each $\mathcal{U}$ element and every element of a set $\mathcal{S}_j \in \mathcal{S}$ is encoded using 229 558 bits (all of which are potentially affected by the corresponding set operations), whereas the sparse representation only uses set operations on $\Delta+1$ integers (i.e., the corresponding hyperedge IDs) in the first iteration.

E. Solution Quality

To evaluate the quality of the phylogenetic data distribution of our HyperPhylo implementation compared to RDDA (the

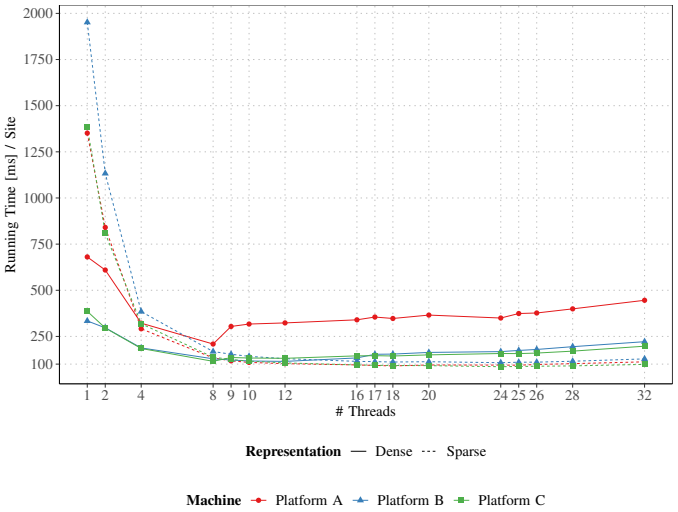

Fig. 5. Weak scaling experiment with 5 000 sites per thread. The corresponding hypergraphs are always partitioned into $k = 160$ blocks.

TABLE IV
ABSOLUTE RUNNING TIMES IN MINUTES FOR STRONG SCALING EXPERIMENTS ON PLATFORMS **A**, **B**, AND **C**.

Threads	Dense Representation			Sparse Representation		
#	A	B	C	A	B	C
1	1 368	789	629	746	1 112	825
2	704	434	321	379	560	418
4	359	229	199	199	284	246
8	198	131	108	111	146	127
9	281	118	122	97	130	114
10	267	107	112	88	118	103
12	223	90	91	74	98	87
16	174	70	76	58	76	68
17	168	75	71	54	72	63
18	161	71	67	52	68	60
20	148	67	61	48	62	54
24	126	59	55	42	54	46
25	125	58	51	39	51	45
26	122	56	50	38	50	42
28	115	54	47	37	46	40
32	102	50	44	33	42	36

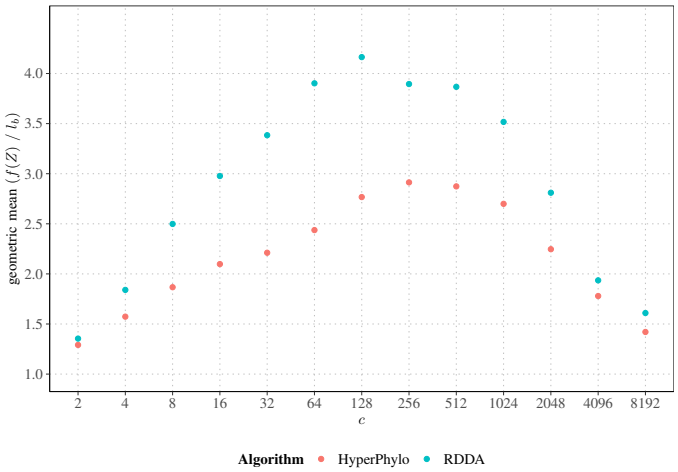

Fig. 6. Average solution quality $f(Z) = \max_{\zeta \in Z} flops(\zeta)$ of the RDDA and HyperPhylo algorithms relative to the lower bound l_b.

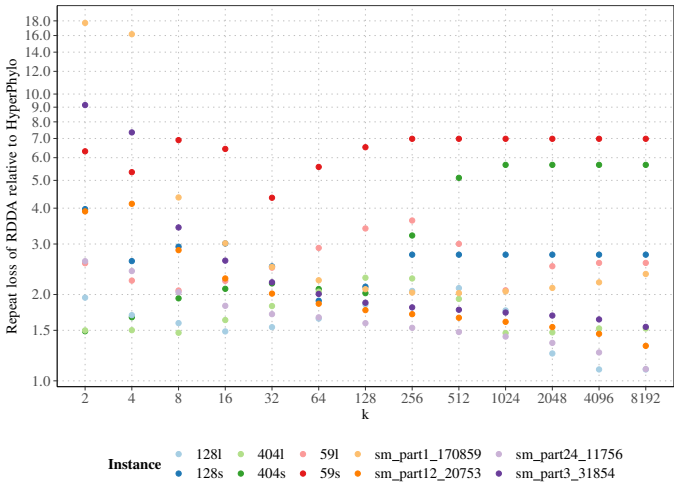

Fig. 7. Repeat loss of RDDA relative to HyperPhylo. Note the log scale on the y-axis.

previous ad hoc algorithm), we computed data distributions for the MSAs described in Section IV-C on random trees generated with RAxML using both approaches. As solution quality measure, we use the ratio between $f(Z) = \max_{\zeta \in Z} flops(\zeta)$ and the lower bound. The lower bound is the value of $flops()$ for a sequential calculation of the PLF, which is minimal as no repeats are lost, divided by the number of cores c. Thus, a quality measure value of 1 indicates that the solution is as good as the lower bound (i.e., optimal); a value of 2 means that the solution is two times worse than the lower bound etc. Figure 6 shows the average solution quality over all MSA partitions and over the number of cores c for RDDA and HyperPhylo. Since our quality measure is normalized by the lower bound, we use the geometric mean to appropriately average the ratios over all instances [18]. Detailed results for the distinct individual, single MSA partitions are shown in the Appendix.

We observe that on average, HyperPhylo outperforms RDDA for all core counts. This is because RDDA does not take SRs into account when assigning the sites of a single MSA partition to two or more cores. In other words, the RDDA data distribution is more coarse grained as it only accounts for SRs at the MSA partition level.

In Figure 7, we show that the repeat loss (i.e., the number of SRs that are lost by assigning sites containing those SRs to distinct cores and which therefore lead to redundant computations on multiple cores) can be substantially reduced for data distributions on $c = 2$ up 8 192 cores (k-way partitions of the hypergraph). This shows that HyperPhylo computes data distributions which are better than those of RDDA in terms of *both* load balance *and* repeat loss.

V. CONCLUSION

We have developed HyperPhylo, a highly efficient and scalable open-source implementation for a specific flavor of judicious hypergraph partitioning where every vertex has the same degree. The focus on this specific flavor is motivated by a real-world phylogenetics problem: optimal data distribution for

massively parallel likelihood calculations with site repeats. We describe an algorithm and its efficient implementation for this specific version of judicious partitioning which we also make available as open-source code. Then, via weak and strong scaling experiments, we show that our algorithm exhibits good parallel efficiency. We also demonstrate that HyperPhylo can improve data distribution quality by up to 50% compared to the naïve algorithm. In addition, the repeat loss can be reduced by up to a factor of 18 compared to the naïve RDDA algorithm that can not take repeat loss into account.

While we suspect that the optimal phylogenetic data distribution problem under site repeats is NP-hard, we have thus far, not been able to devise a proof, mainly because of the unclear restrictions that the structure of the phylogenetic tree induces on any mapping. Further investigating this represents an avenue of future work.

In addition, we plan on developing an enhanced version of HyperPhylo that can dynamically switch between sparse and dense set representations depending on the current iteration and data set at hand.

In terms of practical deployment for phylogenetics, we consider that a hybrid implementation, that is, initially using the RDDA algorithm for assigning MSA partitions to cores and subsequently using HyperPhylo to assign sites to cores only for those MSA partitions that need to be split up among several cores, might work best. We intend to integrate this into our RAxML tool for maxmimum likelihood based phylogenetic inference.

Acknowledgment

The authors wish to thank Matthias Wolf, Marcel Rademacher, and Sabine Kugler for discussions on the potential NP-hardness proof of the problem.

References

[1] A. Stamatakis, "Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies," *Bioinformatics*, vol. 30, no. 9, pp. 1312–1313, 2014. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btu033

[2] J. P. Huelsenbeck and F. Ronquist, "Mrbayes: Bayesian inference of phylogenetic trees," *Bioinformatics*, vol. 17, no. 8, pp. 754–755, 2001. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/17.8.754

[3] T. Tan, J. Gui, S. Wang, S. Gao, and W. Yang, "An efficient algorithm for judicious partition of hypergraphs," in *Combinatorial Optimization and Applications*, X. Gao, H. Du, and M. Han, Eds. Cham: Springer International Publishing, 2017, pp. 466–474.

[4] B. Morel, T. Flouri, and A. Stamatakis, "A novel heuristic for data distribution in massively parallel phylogenetic inference using site repeats," in *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on*. IEEE, 2017, pp. 81–88.

[5] K. Kobert, T. Flouri, A. Aberer, and A. Stamatakis, "The divisible load balance problem and its application to phylogenetic inference," in *Algorithms in Bioinformatics*, D. Brown and B. Morgenstern, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 204–216.

[6] K. Kobert, A. Stamatakis, and T. Flouri, "Efficient detection of repeating sites to accelerate phylogenetic likelihood calculations," *Systematic Biology*, vol. 66, no. 2, p. 205, 2017.

[7] C. Scholl, K. Kobert, T. Flouri, and A. Stamatakis, "The divisible load balance problem with shared cost and its application to phylogenetic inference," *bioRxiv*, 2016.

[8] Y. Zhang, Y. C. Tang, and G. Y. Yan, "On judicious partitions of hypergraphs with edges of size at most 3," *Eur. J. Comb.*, vol. 49, pp. 232–239, 2015. [Online]. Available: https://doi.org/10.1016/j.ejc.2015.03.015

[9] B. Bollobás and A. D. Scott, "Judicious partitions of hypergraphs," *J. Comb. Theory, Ser. A*, vol. 78, no. 1, pp. 15–31, 1997. [Online]. Available: https://doi.org/10.1006/jcta.1996.2744

[10] ——, "Problems and results on judicious partitions," *Random Struct. Algorithms*, vol. 21, no. 3-4, pp. 414–430, 2002. [Online]. Available: https://doi.org/10.1002/rsa.10062

[11] A. D. Scott, "Judicious partitions and related problems," in *Surveys in Combinatorics, 2005 [invited lectures from the Twentieth British Combinatorial Conference, Durham, UK, July 2005]*, 2005, pp. 95–117.

[12] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Domain," *IEEE Transactions on Very Large Scale Integration VLSI Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[13] T. Heuer, P. Sanders, and S. Schlag, "Network Flow-Based Refinement for Multilevel Hypergraph Partitioning," in *17th International Symposium on Experimental Algorithms (SEA 2018)*, 2018, pp. 1:1–1:19.

[14] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. [Online]. Available: http://www.oreilly.com/catalog/9780596514808/index.html

[15] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[16] A. Stamatakis and N. Alachiotis, "Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data," *Bioinformatics*, vol. 26, no. 12, pp. i132–i139, 06 2010. [Online]. Available: https://dx.doi.org/10.1093/bioinformatics/btq205

[17] Misof *et al.*, "Phylogenomics resolves the timing and pattern of insect evolution," *Science*, vol. 346, no. 6210, pp. 763–767, 2014.

[18] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, 1986. [Online]. Available: https://doi.org/10.1145/5666.5673

APPENDIX
DETAILED EXPERIMENTAL RESULTS

Note that for some instances, the number of cores becomes larger than the number of sites. In this case the assignment becomes trivial, which is why the quality measure is 1.

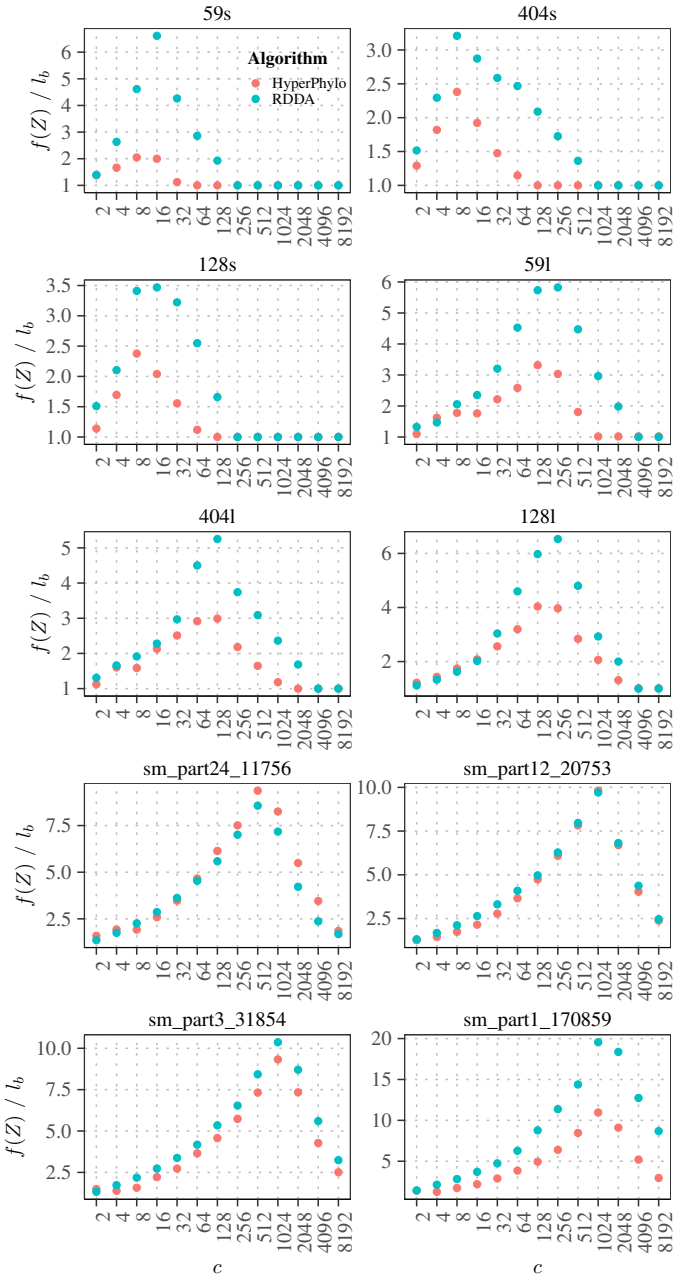

Fig. 8. Per-instance solution quality $f(Z) = \max_{\zeta \in Z} flops(\zeta)$ of the RDDA and HyperPhylo algorithms relative to the lower bound l_b.