The divisible load balance problem and its application to phylogenetic inference

Kassian Kobert[1], Tomáš Flouri[1], Andre Aberer[1], and Alexandros Stamatakis[1,2]

[1] Heidelberg Institute for Theoretical Studies, Germany
[2] Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Postfach 6980, 76128 Karlsruhe

Abstract. Motivated by load balance issues in parallel calculations of the phylogenetic likelihood function we address the problem of distributing divisible items to a given number of bins. The task is to balance the overall sum of (fractional) item sizes per bin, while keeping the maximum number of unique elements in any bin to a minimum. We show that this problem is NP-hard and give a polynomial time approximation algorithm that yields a solution where the sums of (possibly fractional) item sizes are balanced across bins. Moreover, the maximum number of unique elements in the bins is guaranteed to exceed the optimal solution by at most one element. We implement the algorithm in two production-level parallel codes for large-scale likelihood-based phylogenetic inference: ExaML and ExaBayes. For ExaML, we observe best-case runtime improvements of up to a factor of 5.9 compared to the previously implemented data distribution algorithms.

1 Introduction

Maximizing the efficiency of parallel codes by distributing the data in such a way as to optimize load balance is one of the major objectives in high performance computing.

Here, we address a specific case of job scheduling (data distribution) which, to the best of our knowledge, has not been addressed before. We have a list of N divisible jobs, each of which consists of s_i atomic tasks, where $1 \leq i \leq N$, and B processors (or bins). All jobs have an equal, constant startup latency α, and each task, regardless of the job it appears in, requires a constant amount of time β to be processed. Although these times are constant, they depend on the available hardware architecture, and hence are not known a priori. Moreover, the jobs are independent of one another. We also assume that processors are equally fast. Therefore, any task takes time β to execute, independently of the processor it is scheduled to run on. Any job can be partitioned (or decomposed) into disjoint sets of its original tasks, which can then be distributed to different processors. However, each such set incurs its own startup latency α on the processor on which it is scheduled to run. Thus, a job of k tasks takes time $k \cdot \beta + \alpha$ to execute on any processor. The tasks (even of the same job) are independent of each other, that is, they can be executed in any order, and the sole purpose of the job

configuration is to group together the tasks that require the same initialization step and hence minimize the overall startup latency.

Our work is motivated by parallel likelihood computations in phylogenetics (see [4, 9] for an overview). There, we are given a multiple sequence alignment that is typically subdivided into distinct partitions (e.g., gene partitions; jobs in our context). Given the alignment and a partition scheme, the likelihood on a given candidate tree can be calculated. To this end, transition probabilities for the statistical nucleotide substitution model need to be calculated (start-up cost α in our context) for each partition separately because they are typically considered to evolve under different models. Note that, all alignment sites (job size) that belong to the same partition have identical model parameters.

The partitions are the divisible jobs to be distributed among processors. Each partition has a fixed number of *sites* (columns from the alignment), which denote the size of the partition. The sites represent the independent tasks a job (partition) consists of. Since alignment sites are assumed to evolve independently in the likelihood model, the calculations on a single site can be performed independently of all other sites. Thus, a single partition can easily be split among multiple processors. Finally, note that, parallel implementations of the phylogenetic likelihood function now form part of several widely-used tools and the results presented in this paper are generally applicable to all tools.

Related work. A related problem is *bin-packing* with item fragmentation. Here, items may be fragmented, which can potentially reduce the total number of bins needed for packing the instance. However, since fragmentation incurs overhead, unnecessary fragmentations should be avoided. The goal is to pack all items in a minimum number of bins. For an overview of the fractional bin packing problem see [5, Chapter 33]. However, in contrast to our problem, the number of bins is not part of the input but is the objective function. The most closely related domain of research is *divisible load theory* (DLT). Here, the goal is to distribute optimal fractions of the total load among several processors such that the entire load is processed in a minimal amount of time. For a review on DLT, see [1]. However, in general DLT can accommodate more complex models, taking into account a number of factors, such as network parameters or processor speeds. Our problem falls into the category of scheduling divisible loads with start-up costs (see for instance [2, 8]). To our knowledge the problem we present has not been solved before. Finally, there exists previous work by our group on improving the load-balance in parallel phylogenetic likelihood calculations. There, we considered, mostly for the sake of code simplicity, that single partitions/jobs are indivisible. Thus, the scheduling problem we addressed in this work was equivalent to the 'classic' multi-processor scheduling problem. The paper also provides a detailed rationale as to why the calculation of transition probabilities (the overhead α) can become performance-critical [10].

Overview. In Section 2 we formally define two variations of the problem. We then prove that the problem is NP-hard (Section 3). The main contribution of

this paper can be found in Section 4, where we give a polynomial-time approximation algorithm which yields solutions that assign at most one element more to any processor (or bin) than the optimal solution. We analyze the algorithm complexity and prove the $OPT+1$ approximation in (Section 5). Unless $\mathsf{P} = \mathsf{NP}$ [3, 6], no polynomial time algorithm can guarantee a better worst case approximation. Finally, in Section 6, we present the performance gains we obtain, when employing our algorithm for distributing partitions in ExaML[3] [7]

2 Problem Definition

Assume we have N divisible items of sizes $s_1, s_2, \ldots, s_N$, and B available bins. Our task is to find an assignment of the N items to the B bins, by allowing an item to be partitioned into several sub-items whose total size is the size of the original item, in order to achieve the following two goals:

1. The sum of sizes of the (possibly partitioned) items assigned to each bin is well-balanced.
2. The maximum load over all bins is minimal with respect to the number of items added.

In the rest of the text we will use the term *solid* for the items that are not partitioned, and *fractional* for those that are partitioned.

We can now formally introduce two variations of the problem; one where we only allow items of integer sizes, and one where the sizes can be represented by real numbers. In the case of integers, the problem can be formulated as the following integer program.

Problem 1 (LB$\mathbb{N}$) *Given a sequence of positive integers $s_1, s_2, \ldots, s_N$ and a positive integer B,*

$$\textit{minimize} \qquad \max\{\textstyle\sum_{j=1}^{N} x_{i,j} \mid i = 1, 2, \ldots, B\}$$

subject to

$$\textstyle\sum_{i=1}^{B} q_{i,j} = s_j, \qquad 1 \le j \le N$$

$$\textstyle\sum_{j=1}^{N} q_{i,j} \ge \lfloor \sigma/B \rfloor, \qquad 1 \le i \le B$$

$$\textstyle\sum_{j=1}^{N} q_{i,j} \le \lceil \sigma/B \rceil, \qquad 1 \le i \le B$$

$$\sigma = \textstyle\sum_{i=1}^{N} s_i$$

$$0 \le q_{i,j} \le x_{i,j} \cdot s_j, \qquad 1 \le i \le B,\ 1 \le j \le N$$

$$q \in \mathbb{N}_{\ge 0}^{B \times N}$$

$$x \in \{0,1\}^{B \times N}$$

[3] Available at http://www.exelixis-lab.org/web/software/examl/index.html

Variable $x_{i,j}$ is a boolean value indicating whether bin i contains part of item j and if it does, $q_{i,j}$ denotes the amount. By removing the imposed restriction of integer sizes, and hence allowing for positive real values as the sizes of both solid and fractional items, we obtain the following mixed integer program.

Problem 2 (LB$\mathbb{R}$) *Given a sequence of positive real values $s_1, s_2, \ldots, s_N$ and a positive integer value B,*

$$\textit{minimize} \qquad \max\{\textstyle\sum_{j=1}^{N} x_{i,j} \mid i = 1, 2, \ldots, B\}$$

subject to

$$\textstyle\sum_{i=1}^{B} q_{i,j} = s_j, \qquad 1 \le j \le N$$

$$\textstyle\sum_{j=1}^{N} q_{i,j} = \sigma/B, \qquad 1 \le i \le B$$

$$\sigma = \textstyle\sum_{i=1}^{N} s_i$$

$$0 \le q_{i,j} \le x_{i,j} \cdot s_j, \qquad 1 \le i \le B,\ 1 \le j \le N$$

$$q \in \mathbb{R}^{B \times N}$$

$$x \in \{0,1\}^{B \times N}$$

If for some bin i and element j we get a solution with $q_{i,j} < s_j$, we say that element j is only assigned to bin i partially, or that only a fraction of element j is assigned to bin i. If $q_{i,j} = s_j$ we say that element j is fully assigned to bin i.

3 NP-hardness

We now show that problems LB$\mathbb{N}$ and LB$\mathbb{R}$ are NP-hard by reducing the well-known Partition [6] problem. We reduce it to another decision problem called *Equal Cardinality Partition* (ECP) that decides whether a set can be broken into disjoint sets of equal cardinality and equal sum of elements (see Def. 2), which can be solved by the two flavors of our problem.

Definition 1 (Partition). *Is it possible to partition a set S of positive integers into two disjoint subsets Q and R, such that $Q \cup R = S$ and $\sum_{q \in Q} q = \sum_{r \in R} r$?*

Definition 2 (ECP). *Let p, k be two positive integers and S a set of $p \cdot k$ positive integers. Can we partition S into p disjoint sets $S_1, S_2, \ldots, S_p$ of k elements each, such that $\bigcup_{i=1}^{p} S_i = S$ and $\sum_{s \in S_i} s = \sum_{s \in S_j} s$, for all $1 \le i, j \le p$?*

Clearly, if we can solve our original optimization problems LB$\mathbb{N}$ and LB$\mathbb{R}$ for any S exactly, we can also answer whether ECP returns *true* or *false* for the same set S. Thus, if we can show that ECP is NP-Complete we know that the original problems are NP-hard.
To show that ECP is NP-Complete, it is sufficient to show that ECP is in NP, that is the set of polynomial time verifiable problems, and some NP-Complete problem (here Partition) reduces to it.

Lemma 1. ECP *is* NP*-Complete.*

Proof. The first part, i.e., ECP $\in$ NP, is trivial. Given a solution (that is, the sets $S_1,\ldots,S_p$), we are able to verify, in polynomial time to p, that the conditions for problem ECP hold, by summing the elements of each set.

For the reduction of PARTITION to ECP consider the set S to be an instance of PARTITION. We derive an instance $\hat{S}$ of ECP from S, such that PARTITION(S) is true *iff* ECP($\hat{S}$) is true for 2 bins (that is $p = 2$). We define $\hat{S} = S \cup (a \cdot S)$ a set of integers, with $a = (1 + \sum_{s \in S} s)$ and $(a \cdot S) = \{\, a \cdot s \mid s \in S \,\}$. Clearly, if there is a solution for PARTITION given S, there must also be a solution for ECP given $\hat{S}$. If $Q, R \subset S$ is a solution for PARTITION, then $Q \cup (a \cdot R)$, $R \cup (a \cdot Q)$ is a solution for ECP.

Similarly, let $\hat{Q}, \hat{R}$ be a solution for ECP given $\hat{S}$. Let $Q = \hat{Q} \cap S$, $R = \hat{R} \cap S$, $(a \cdot Q) = \hat{Q} \cap (a \cdot S)$ and $(a \cdot R) = \hat{R} \cap (a \cdot S)$. Trivially, it holds that $Q = \{\, q \in \hat{Q} \mid q < a \,\}$, $R = \{r \in \hat{R} \mid r < a\}$ and $(a \cdot Q) = \hat{Q} \setminus Q$, $(a \cdot R) = \hat{R} \setminus R$. Thus, we obtain $Q \cup R = S$ and $(a \cdot Q) \cup (a \cdot R) = (a \cdot S)$. We also obtain that $\sum_{q \in Q} q = \sum_{r \in R} r$ (and $\sum_{q \in (a \cdot Q)} q = \sum_{r \in (a \cdot R)} r$). We prove that the equations hold by contradiction: suppose this was not the case for some solution of ECP, that is $\sum_{q \in Q} q \neq \sum_{r \in R} r$ and hence $\sum_{q \in (a \cdot Q)} q \neq \sum_{r \in (a \cdot R)} r$. By definition, $(a \cdot Q)$ and $(a \cdot R)$, q/a and r/a are integer values for any $q \in (a \cdot Q)$ and $r \in (a \cdot R)$, and therefore:

$$
\begin{aligned}
|\sum_{q \in (a \cdot Q)} q - \sum_{r \in (a \cdot R)} r| &= |\sum_{q \in (a \cdot Q)} a \cdot q/a - \sum_{r \in (a \cdot R)} a \cdot r/a| \\
&= a \cdot |\overbrace{\sum_{q \in (a \cdot Q)} q/a - \sum_{r \in (a \cdot R)} r/a}^{\geq 1}| \geq a
\end{aligned}
$$

However, $\sum_{s \in S} s < a$. Thus, $\sum_{q \in \hat{Q}} q \neq \sum_{r \in \hat{R}} r$ which contradicts the assumption of $\hat{Q}$, $\hat{R}$ being a solution for ECP($\hat{S}$,2). Therefore, PARTITION reduces to ECP, which means that ECP is NP-Complete. □

Corollary 1. *The optimization problems* LB$\mathbb{N}$ *and* LB$\mathbb{R}$ *are* NP*-hard.*

This follows directly from Lemma 1 and the fact that an answer for ECP can be obtained by solving the optimization problem.

4 Algorithm

As seen in Section 3, finding an optimal solution to this problem is hard. To overcome this hurdle, we propose an approximation algorithm running in polynomial time that guarantees a near-optimal solution. For an in-depth analysis of the complexity of the algorithm, see Section 5.

The input for the algorithm is a list S of N integer weights and the number of bins B these elements must be assigned to. The idea of the algorithm can be explained by the following three steps:

1. Sort S in ascending order.
2. Starting from the first (solid) element in the sorted list S, assign elements from S to the B bins in a cyclic manner (at any time no two bins can have a difference of more than one element) until any bin can not entirely hold the proposed next item.
3. Break the remaining elements from S to fill the remaining space in the bins.

Fig. 1 presents the pseudocode for the first two phases, while Fig. 2 illustrates phase 3. The output of this algorithm is an assignment, $list = (list[1], \ldots, list[p])$, of –possibly fractional– elements to bins. Each entry in *list* is a set of triplets that specify which portion of an integer sized element is assigned to a bin. Let $(j, i, k) \in list[l]$ be one such triplet for bin number l. We interpret this triplet as follows: bin l is assigned the fraction of element j that starts at i and ends at k (including i and k).

For the application in phylogenetics, each triplet specifies which portion (how many sites) of a partition is assigned to which processor. Again, let $(j, i, k) \in list[l]$ be one such triplet for some processor l. We interpret this triplet as follows: processor l is assigned sites i through k of partition j.

If $i \neq 1$ or $k \neq s_j$ (recall s_j is the size of element j), we say that element j is partially assigned to bin i, that is, only a fraction of element j is assigned to

```
LoadBalance(N, B, S)
▷ Phase 1 — Initialization
 1. Sort S in ascending order and let S = (s_1, s_2, ..., s_N)
 2. σ = Σ_{i←1}^{N} s_i
 3. c ← ⌈σ/B⌉
 4. r ← c · B − σ
 5. for i ← 1 to B do
 6.       size[b] ← 0; items[b] ← 0; list[b] ← ∅
 7. full_bins ← 0; b ← 0;
▷ Phase 2 — Initial filling
 8. for i ← 1 to N do
 9.       if size[b] + s_i ≤ c then
10.            size[b] ← size[b] + s_i
11.            items[b] = items[b] + 1
12.            Enqueue(list[b], (i, 1, s_i))
13.            if size[b] = c then
14.                 full_bins ← full_bins + 1
15.                 if full_bins = B − r then c ← c − 1
16.       else
17.            break
18.       b ← (b + 1)  mod B
```

Fig. 1. The algorithm accepts three arguments N, B and S, where N is the number of items in list S, and B is the number of bins

```
▷ Phase 3 — Partitioning items into bins
19. low ← B; ℓ ← B; high ← 1; h ← 1; add ← 0
20. while i ≤ N do
21.     while size[ℓ] ≥ c do
22.         low ← low − 1; ℓ ← low
23.     while size[h] ≥ c do
24.         high ← high + 1; h ← high
25.     if size[h] + add ≥ c then
26.         items[h] ← items[h] + 1
27.         ENQUEUE(list[h], (i, s_i − add + 1, s_i − add − size[d] + c))
28.         add ← size[h] + add − c
29.         size[h] ← c
30.         full_bins ← full_bins + 1
31.         if full_bins = B − r then c ← c − 1
32.     else
33.         items[ℓ] ← items[ℓ] + 1
34.         if size[ℓ] + add < c then
35.             size[ℓ] ← size[ℓ] + add
36.             ENQUEUE(list[ℓ], (i, s_i − add + 1, s_i))
37.             add ← 0
38.             high ← high − 1; h ← ℓ
39.             low ← low − 1; ℓ ← low
40.         else
41.             ENQUEUE(list[ℓ], (i, s_i − add + 1, s_i − add − size[d] + c))
42.             add ← size[ℓ] + add − c
43.             size[ℓ] ← c
44.             full_bins ← full_bins + 1
45.             if full_bins = B − r then c ← c − 1
46.     if add = 0 then
47.         i ← i + 1; add ← s_i
```

Fig. 2. Phase 3 of the algorithm

bin i. Otherwise, if $i = 1$ and $k = s_j$, then the triplet represents a solid element, i.e., element j is fully assigned to bin i.

For applications that allow any fraction of an integer to be assigned to a bin, not just whole integer values (that is, problem LB$\mathbb{R}$), we redefine the variable c, i.e. the maximum capacity of the bins, to be exactly σ/B, without rounding. Additionally, the output (*list*) must correctly state which ranges of the elements are assigned to which bin and not give integer lower and upper bounds. Next, we give two examples of how LoadBalance works on specific sets of integers.

Example 1. Consider the set $\{2, 2, 3, 5, 9\}$ and three bins. During initialization (phase 1) we have $c = 7$ and $r = 0$. Phase 2 makes the following assignments: $list[1] = \{(1, 1, 2), (4, 1, 5)\}$, $list[2] = \{(2, 1, 2)\}$, $list[3] = \{(3, 1, 3)\}$. Adding the next element of size 9 is not possible since $size[2] + 9 = 2 + 9 = 11 > c$. Thus, phase 2 ends. Phase 3 splits the last element of size 9 among bins 2 and

3, and the solution is $list[1] = \{(1,1,2),(4,1,5)\}$, $list[2] = \{(2,1,2),(5,1,5)\}$, $list[3] = \{(3,1,3),(5,6,9)\}$. With $\max\{|list[1]|, |list[2]|, |list[3]|\} = 2$. This is also an optimal solution.

Example 2. Consider the set $\{1,1,2,3,3,6\}$ and two bins. During the initialization (phase 1) we have $c = 8$ and $r = 0$. Phase 2 generates the following assignments: $list[1] = \{(1,1,1),(3,1,2),(5,1,3)\}$, $list[2] = \{(2,1,1),(4,1,3)\}$. The last element of size 6 can not be fully assigned to bin 2, thus phase 2 terminates. Finally, phase 3 splits the last element of size 6 among the two bins, and the solution is $list[1] = \{(1,1,1),(3,1,2),(5,1,3),(6,1,2)\}$, $list[2] = \{(2,1,1),(4,1,3),(6,3,6)\}$. We get $\max\{|list[1]|, |list[2]|\} = 4$. However, an optimal solution $list_1^\star = \{(1,1,1),(2,1,1),(6,1,6)\}$, $list_2^\star = \{(3,1,2),(4,1,3),(5,1,3)\}$ with $max\{|list_1^\star|, |list_2^\star|\} = 3$ exists.

As we can see in Example 2, algorithm LoadBalance fails to find the optimal solution in certain cases. However in the next section we show that the difference of 1, as observed in Example 2, represents the worst case scenario.

5 Algorithm analysis

We now show that the score obtained by algorithm LoadBalance, for any given set of integers and any number of bins, is at most one above the optimal solution. We then give the asymptotic time and space complexities.

5.1 Near-optimal solution

Before we start with the proof, we make three observations associated with the algorithm that facilitate the proof. We use the same notation as in the description of the algorithm. That is, $items[i]$ indicates the number of items in bin i, $size[i]$ the sum of sizes of items in bin i, and $list[i]$ is a list of records per item in bin i, describing which fraction of the particular item is assigned to bin i.

Observation 1 *During phase 2 of algorithm* LoadBalance*, for any two bins j and i, it holds that $size[i] > size[j]$, such that $items[i] = items[j] + 1$.*

The list of integers was sorted in Phase 1 of the algorithm to a non-decreasing sequence. Hence, any item added to a bin during the i-th cyclic iteration over bins, must be smaller or equal to an item that is added during iteration $i + 1$. Following directory from Observation 1, we obtain the next observation.

Observation 2 *For all bins i and j during phase 2 of algorithm* LoadBalance*, it holds that $items[j] \leq items[i] + 1$.*

Observation 3 *Phase 3 appends at most 2 more (fractional) items to a bin.*

Any remaining (unassigned) item of size s in this phase satisfies the condition $size[j]+s > c$, for any bin j and capacity c as computed in Fig. 1. Therefore, each bin will be assigned at most one fractional item that does not fill it completely, and one new element that is guaranteed to fill it up.

Lemma 2. *Let $OPT(S, B)$ be the score for the optimal solution for a set S distributed to B bins. Let list be the solution produced by algorithm* LoadBalance *for the same set S and B bins. Then it holds that* $\max\{\, |list[i]| \mid i = 1, 2, \ldots, B \,\} \leq OPT(S, B) + 1$

Proof. Let $\hat{j}$ be the bin that terminates phase 2. That is, $\hat{j}$ is the last bin considered for any assignment in phase 2. After phase 2, if there exists a bin j with $items[j] = items[\hat{j}]+1$ we get, by Observation 1 and the *pigeonhole principle*, that $OPT(S, B) \geq items[\hat{j}]+1$. Otherwise, if no such bin exists, $OPT(S, B) \geq items[\hat{j}]$. Let K be the number of unassigned elements at the beginning of phase 3. Let J be the number of bins j with $items[j] = items[\hat{j}]$. We distinguish between three cases. First assume that $items[j] = items[\hat{j}]$ (after phase 2) for all bins j and $K > 0$. Clearly, $OPT(S, B) \geq items[\hat{j}] + 1$. By observation 3 we know that $items[j] \leq items[\hat{j}] + 2$ (after phase 3). Thus the lemma holds for this case. Now consider $K > J$ and $items[j] \neq items[\hat{j}]$ for some bin j, that is, there are more unassigned elements than there are bins with only $items[\hat{j}]$ elements assigned to them. By the pigeonhole principle, $OPT(S, B) \geq items[\hat{j}] + 2$. By observation 3 we get that $items[j] \leq items[\hat{j}] + 1 + 2 = items[\hat{j}] + 3$ for all j. Thus the lemma holds for this case as well. For the last case assume $K \leq J$ and $items[j] \neq items[\hat{j}]$ for some bin j. After a bin is assigned a fractional element that does not fill it completely, it is immediately filled up with the next element. Since preference is given to any bin j with $items[j] = items[\hat{j}]$ and there are at least as many such bins as remaining elements to be added ($K \leq J$), we get that $items[j] \leq items[\hat{j}]+2$. Since we have seen above that $OPT(S, B) \geq items[\hat{j}]+1$, the lemma holds. As this covers all cases, the lemma is proven. □

5.2 Run-time

The runtime analysis is straight forward. Phase 1 of the algorithm consists of initializing variables, sorting N items by size in ascending order and computing their sum. Using an algorithm such as Merge-Sort, Phase 1 requires $\mathcal{O}(N \log(N))$ time. Phase 2 requires $\mathcal{O}(N)$ time to consider at most N items, and assign them to B bins in a cyclic manner. Phase 3 appends at most 2 items to a bin (see Observation 3), and hence has a time complexity of $\mathcal{O}(B)$. This yields an overall asymptotic run-time complexity of $\mathcal{O}(N \log(N)+B)$. Note that, if we are already given a sorted list of partitions, the algorithm runs in linear time $\mathcal{O}(N + B)$. Finally, LoadBalance requires $\mathcal{O}(B)$ space due to the arrays *items*, *size* and *list*, that are each of size B.

6 Practical Application

As mentioned before, the scheduling problem arises for parallel phylogenetic likelihood calculations on large partitioned multi-gene or whole-genome datasets. This type of partitioned analyses represent common practice at present. The number of multiple sequence alignment partitions, the number of alignment sites

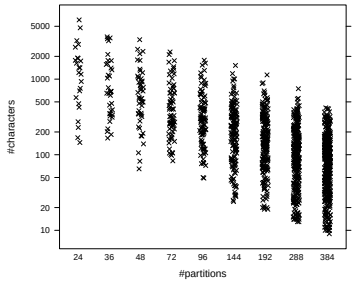


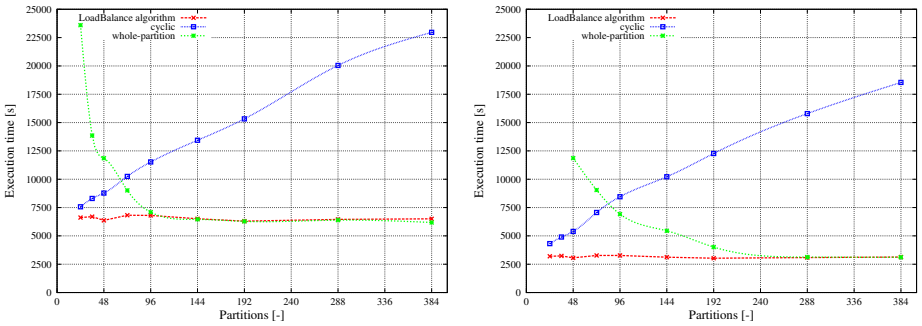
Fig. 3. Number of characters/sites in each partition for the partitioning schemes.

per partition, and the number of available processors are the input to our algorithm. The production-level maximum likelihood based phylogenetic inference software ExaML for supercomputers implements two different data distribution approaches: The *cyclic data distribution* scheme that does not balance the number of unique partitions per processor, but just assigns single sites to processors in a cyclic fashion. The second approach is the *whole-partition data distribution* scheme. Here, the individual partitions are not considered divisible and are assigned monolithically to processors using the longest processing time heuristic for the 'classic' multi-processor scheduling problem [10]. This ensures that the total and maximum number of initialization steps (substitution matrix calculations) is minimized, at the cost of not being balanced with respect to the sites per processor. Nonetheless, using this scheme instead of the cyclic distribution already yielded substantial performance improvements. In order to evaluate the new distribution scheme, we compare it to these two previous schemes, in terms of total ExaML runtime. Note that, our algorithm has also been implemented in ExaBayes[4] which is a code for large-scale Bayesian phylogenetic inference.

6.1 Methods

We performed runtime experiments on a real-world alignment. The alignment is made of 144 species and 38 400 amino acid characters[5] used the alignment to create 9 distinct partitioning schemes with an increasing number of partitions. For each scheme, partition lengths were drawn at random, while the number of partitions per scheme was fixed to 24, 36, 48, 72, 96, 144, 192, 288, 384, and 768, respectively. To generate n partition lengths, we drew n random numbers $x_1, \ldots, x_n$ from an exponential distribution $\exp(1) + 0.1$. For a partition p, the value of $x_p / \sum_{i=1..n} x_i$ then specifies the proportion of characters that belong to partition p. The offset of 0.1 was added to random numbers to prevent partition lengths from becoming unrealistically small, since the exponential distribution strongly favors small values. Fig. 3 displays the distributions of the partition

[4] Available at http://www.exelixis-lab.org/web/software/exabayes/index.html
[5] Data from the 1KITE (www.1kite.org) project.



(a) Runtimes on 24 cores.

(b) Runtimes on 48 cores.

Fig. 4. Runtime comparison for ExaML employing algorithm LoadBalance, the cyclic data distribution scheme, or the whole-partition data distribution scheme.

lengths for each of the 9 partition schemes. As expected, partition lengths are distributed uniformly on the log-scale.

We executed ExaML using 24 and 48 processes, respectively, to assess performance with our new data distribution algorithm and compare it with the cyclic site and whole-partition data distribution performance. We used a cluster equipped with Intel SandyBridge nodes (2×6 cores per node) and an Infiniband interconnect. Thus, a total of 2 nodes was needed for runs with 24 processes and 4 nodes for runs with 48 processes (inducing higher inter-node communication costs). In Fig. 4.b, the run-times for the whole-partition distribution approach with less than 48 partitions are omitted, since they are identical to executing the runs on 24 processes. The reason is that this method does not divide partitions and thus, in case the number of partitions is smaller than the number of available processors, the extra processors will remain unused.

6.2 Results

As illustrated by Fig. 4, with algorithm LoadBalance ExaML always runs at least as fast as the two previous data distribution strategies with one minor exception. Compared to the cyclic data distribution, LoadBalance is 3.5× faster for 24 processes and up to 5.9× faster for 48 processes. Using LoadBalance, ExaML requires up to 3.6× less runtime than with the whole partition distribution scheme for 24 processes and for 48 processes the runtime can be improved by a factor of up to 3.9×. For large numbers of partitions, the runtime of the whole partition distribution scheme converges against the runtime of LoadBalance. This is expected, since by increasing the number of partitions we break the alignment into smaller chunks and the chance of any heuristic to attain a near-optimal load/data distribution increases. However, if the same run is executed with more processes (i.e., 48 instead of 24), this break-even point shifts towards a higher number of partitions, as shown in Fig. 4.

The results show that, cyclic data distribution performance is acceptable for many processes and few partitions, whereas monolithic whole-partition data distribution is on par with our new heuristic for analyses with few processes and many partitions. Both figures show, that there exists a region where neither of the previous strategies exhibits acceptable performance compared to LoadBalance and that this performance gap widens, as parallelism increases.

Finally, employing LoadBalance, ExaML executes twice as fast with 48 processes than with 24 processes and thus exhibits an optimum scaling factor of about 2.07 in all cases. For comparison, under the cyclic data distribution, scaling factors ranged from 1.24 to 1.75 and under whole partition distribution, scaling factors ranged from 1.00 (i.e., no parallel runtime improvement) to 2.04. The slight superlinear speedups are due to increased cache efficiency.

7 Conclusion

We have introduced an approximation algorithm for solving a NP-hard scheduling problem with an acceptable worst-case performance guarantee. This theoretical work was motivated by our efforts to improve parallel efficiency of phylogenetic likelihood calculations. By implementing the approximation algorithm in ExaML, a dedicated code for large-scale maximum likelihood-based phylogenetic analyses on supercomputers, we show that (i) the data distribution is near-optimal, irrespective of the number of partitions, their lengths, and the number of processes used and (ii) substantial run time improvements can be achieved, thus saving scarce supercomputer resources. The data distribution algorithm is generally applicable to any code that parallelizes likelihood calculations.

References

1. Bharadwaj, V., Ghose, D., Robertazzi, T.: Divisible load theory: A new paradigm for load scheduling in distributed systems. Cluster Computing 6(1), 7–17 (2003)
2. Błażewicz, J., Drozdowski, M.: Distributed processing of divisible jobs with communication startup costs. Discrete Appl. Math. 76(1-3), 21–41 (Jun 1997)
3. Cook, S.A.: The complexity of theorem-proving procedures. STOC '71 Proceedings of the third annual ACM symposium on Theory of computing pp. 151 – 158 (1971)
4. Felsenstein, J.: Inferring phylogenies. Sinauer Associates (2003)
5. Gonzalez, T.F.: Handbook of Approximation Algorithms and Metaheuristics. Chapman & Hall/CRC (2007)
6. Karp, R.: Reducibility among combinatorial problems. Complexity of Computer Computations pp. 85–103 (1972)
7. Stamatakis, A., Aberer, A.J.: Novel parallelization schemes for large-scale likelihood-based phylogenetic inference. In: IPDPS. pp. 1195–1204 (2013)
8. Veeravalli, B., Li, X., Ko, C.C.: On the influence of start-up costs in scheduling divisible loads on bus networks. Parallel and Distributed Systems, IEEE Transactions on 11(12), 1288–1305 (Dec 2000)
9. Yang, Z.: Computational Molecular Evolution. Oxford University Press (2006)
10. Zhang, J., Stamatakis, A.: The multi-processor scheduling problem in phylogenetics. In: IPDPS Workshops. pp. 691–698. IEEE Computer Society (2012)