LOS: Level Order Sampling for Task Graph Scheduling on Heterogeneous Resources

Carl Witt
Humboldt-Universität zu Berlin
Berlin, Germany
wittcarl@informatik.hu-berlin.de

Sam Wheating
University of Victoria
Victoria, Canada
samwheating@gmail.com

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

***Abstract*—List scheduling is an approach to task graph scheduling that has been shown to work well for scheduling tasks with data dependencies on heterogeneous resources. Key to the performance of a list scheduling heuristic is its method to prioritize the tasks, and various ranking schemes have been proposed in the literature. We propose a method that combines multiple random rankings instead of a using a deterministic ranking scheme.**

We introduce L-Orders, which are a subset of all topological orders of a directed acyclic graph. L-Orders can be used to explore targeted *regions* of the space of all topological orders. Using the observation that the makespans in one such region are often approximately normal distributed, we estimate the expected time to solution improvement in certain regions of the search space. We combine targeted search and improvement time estimations into a time budgeted search algorithm that balances exploration and exploitation of the search space. In 40,500 experiments, our schedules are 5% shorter on average and up to 40% shorter in extreme cases than schedules produced by HEFT.

***Index Terms*—Distributed Computing, Task Graph Scheduling, Scientific Workflows, Adaptive Sampling**

I. Introduction

The amount of data routinely analyzed in scientific or business contexts is ever growing. To keep up with the development and to scale analyses to large input sets, distributed and parallel computing is used.

A common approach to parallelizing compute workloads is task-based computing. An application manages a set of tasks representing discrete, often atomic units of work that can be run in parallel on distributed and possibly heterogeneous compute resources. We focus on scientific workflows, a paradigm for defining data analyses where tasks produce output data that is passed as input data to downstream tasks. This yields a set of precedence constraints on the tasks that can be described as a directed acyclic graph (dag). Passing data between tasks also causes communication delays when tasks are executed on different processors.

Dag scheduling is the problem of deciding which tasks to run on which resources, and when. With precedence constraints and communication times, the problem of minimizing the overall execution time is NP-complete, even on an unlimited number of processors [1]. List scheduling methods are a well-known class of scheduling heuristics that compute a schedule for a given dag with known task runtimes and data transfer times. They proceed in two phases: First, each task is assigned a rank, often representing an estimate of how critical the task is for advancing the execution of the dag. Tasks are then considered for scheduling in order of their ranks, usually greedily optimizing some criterion like the earliest start time or the earliest finish time.

The Heterogeneous Earliest Finish Time (HEFT) algorithm [2] is a well-known list scheduling method to solve the scheduling problem with precedence constraints, communication times, and heterogeous resources. HEFT first averages the computation and communication costs over all available resources and then propagates combined computation and communication costs from the bottom of the dag to the entry tasks. This gives the length of a partial critical path starting at each task that is used to prioritize tasks with more downstream work. However, it has been shown that different ranking schemes can yield significantly different results and may be used to reduce schedule lengths by up to 3%-6%, although no single ranking scheme always performs best [3].

The idea behind our proposed Level Order Sampling (LOS) method is to replace the ranking scheme in HEFT by a randomized method that both allows for exploring a large number of alternative rankings as well as considering long-term dependencies between scheduling decisions by repeatedly evaluating schedule variations affecting different portions of the task graph.

Our contributions are the following: We present an efficient way to randomly select topological orders of a dag from specific regions of the search space and use it to build a time-budgeted adaptive search algorithm for task graph scheduling on heterogeneous resources. We propose a time time-budgeted method to offer users more flexibility in balancing the conflicting goals of scheduling quality and speed. We show that our method outperforms HEFT by 5% on average to 40% and also beats alternative ranking schemes from the literature. The code is available on demand from the corresponding author.

The paper is structured as follows. Section 2 introduces the scheduling problem and notation, as well as basics of list scheduling and the HEFT algorithm in particular. Section 3 presents the LOS algorithm for randomized scheduling on a time budget. Section 4 presents the results of the experimental evaluation. Section 5 reviews related work. Finally, Section 6 discusses design alternatives and future work.

II. Background

A. Scheduling Problem Terms and Definitions

We consider the problem of scheduling a communication dag on heterogeneous resources subject to minimizing the overall execution time (makespan). Let $P = \{p_1, p_2, ..., p_p\}$ be a set of processors. A communication dag is a directed acyclic graph $G = (V, E, w, c)$ that describes a workflow's tasks $V = \{T_1, T_2, ..., T_n\}$, the dependency relationships $E \subset V \times V$ between tasks, their computation times $w\colon V \times P \to \mathbb{R}^+$, and the amount of data to be transferred from a task to another $c\colon E \to \mathbb{R}^+$. Note that the task execution times do not have to be consistent [4] with respect to processors, i. e., certain processor might be better suited to certain tasks, e. g., p_1 could be the fastest processor for T_1 but another processor is fastest for T_2.

We use the standard network model in dag scheduling, a fully connected network with heterogeneous data transfer rates between processors, but without network contention. Data transfer rates and communication startup times between all pairs of processors can be defined and affect the data transfer times, but we do not introduce symbols for them here, as we do not use them later on. Tasks being executed on the same processor communicate for free.

The transitive closure of the precedence constraints E gives a strict partial order $\prec$. $T_i \prec T_j$ denotes that there is a path from T_i to T_j. We say T_j is a descendant of T_i. Equivalently, we may write $T_j \succ T_i$ to denote that T_i is an ancestor of T_j. If neither $T_i \prec T_j$ nor $T_i \succ T_j$, the tasks are called incomparable. In the context of scheduling, $T_i \prec T_j$ means that T_i has to finish execution before T_j can start. If $(T_i, T_j) \in E$ then $c(T_i, T_j)$ data has to be transferred between the processors on which T_i and T_j are executed. If T_i and T_j are incomparable, they can potentially be executed in parallel.

We use $\prec$ to define the lastest precedence value [5], or level for short, of each task. The level of a task is defined recursively based on the levels of its descendants. A task without descendants is called a leaf.

$$\text{level}(T_i) = \begin{cases} 0 & T_i \text{ is a leaf} \\ 1 + \max_{T_j \succ T_i} \text{level}(T_j) & \text{otherwise} \end{cases} \tag{1}$$

According to this definition, level(T_i) equals the longest path from T_i to a leaf. Note that in the literature, the level of a task often takes into account the runtimes of the tasks on a longest path to a leaf node, e. g., the b-level [6]. Here, the level of a task is based only on the number of tasks on a longest path to a leaf.

We define an *order* of the tasks as a permutation $\pi\colon V \to \{1, ..., n\}$ such that tasks with higher ranks are either ancestors of tasks with lower ranks or can be executed in parallel with them, i. e., $\forall i, j\colon \pi(T_i) > \pi(T_j) \Rightarrow T_i \prec T_j \vee (T_i \not\prec T_j \wedge T_j \not\prec T_i)$. We refer to $\pi(T_i)$ as the rank of T_i. An order can be found using topological sorting. Thus, there is usually more than one order of the tasks of a dag. We refer to the set of all orders as the order space. Figure 1 exemplifies these concepts.

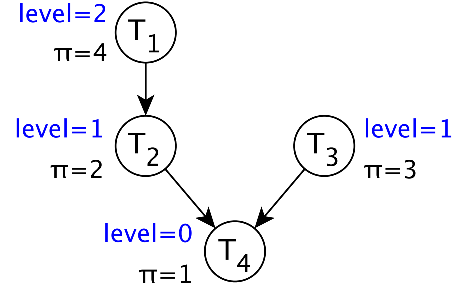

Fig. 1: Example dag, its task levels, and an ordering π. We have $T_1 \prec T_2, T_1 \prec T_4, T_2 \prec T_4, T_3 \prec T_4$. For example, since neither $T_1 \prec T_3$ nor $T_3 \prec T_1$, both tasks can be executed in parallel. The order space consists of the following three permutations: $(T_1, T_2, T_3, T_4), (T_1, T_3, T_2, T_4), (T_3, T_1, T_2, T_4)$.

B. List Scheduling and Task Weighting

List scheduling [6] is a well-known approach to dag scheduling that divides the problem into a task prioritization phase and a processor assignment phase. During the prioritization phase, a priority for each of the tasks is computed according to some weighting method. In the processor assignment phase, the scheduler considers the tasks in decreasing order of priority to decide on which processor the processor should run.

A weighting scheme $V \to \mathbb{R}$ is a method to map the tasks to weights representing their priority. Weighting method are usually chosen such that sorting by descending weight produces an order of the tasks. In case tasks have equal weights, ties are often broken arbitrarily.

Sometimes, a slightly different definition of a list scheduling algorithm is used [1], which requires the scheduler to always run as many tasks as possible in parallel, i. e., not holding tasks back when processors are idle. For dags without communication times, such a strategy can be shown to always produce a schedule of length at most $2 - \frac{1}{p}$ times the optimal makespan [7], where p is the number of processors. However, this does not hold for the more complex problem that arises when considering communication times between processors. With communicaton times, it can be better to keep processors idle in case the incurred communication times are larger than the gains from parallelizing computation.

C. Heterogeneous Earliest Finish Time Algorithm

A tried and tested popular list scheduling algorithm for solving the scheduling problem at hand is the Heterogeneous Earliest Finish Time (HEFT) algorithm. We briefly explain it here, because the basic idea of the proposed level order sampling method is to replace the task ordering method used in HEFT with a randomized ordering scheme.

During the task prioritization phase, HEFT uses a weighting scheme that computes the weight of each task recursively based on the weights of its children. Let $\bar{w}$ and $\bar{c}$ denote the computation/communication costs averaged over all proces-

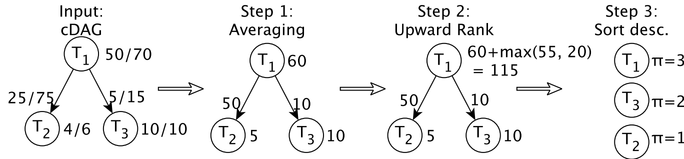

Fig. 2: The HEFT weighting method to obtain a task order. For step 1, different schemes have been proposed [3] to which we compare our method. This example uses two processors and two network links, the corresponding computation and communication times are annotated to the input as x/y. In Step 1, averaging is used to remove heterogeneity. Step 2 uses the upward rank from Equation 2, and Step 3 obtains an order of the tasks by sorting by urank descending.

sors/network links, respectively. The upward rank of a task is defined as:

$$\text{urank}(T_i) = \bar{w}(T_i) + \max_{(T_i,T_j)\in E} \bar{c}\,(T_i, T_j) + \text{urank}(T_j) \quad (2)$$

Using this definition, the weight of a task equals the maximal sum of the average computation and communication on a path to a leaf. In contrast, our definition of the task level as the maximal number of tasks on a path to a leaf serves the purpose of capturing the dependency constraints and finding valid execution orders (see Section III-A). Resorting to average computation and communication times is a simple solution to the problem that we do not yet know where tasks will eventually be executed and thus do not know their actual execution times in a heterogeneous environment. Figure 2 illustrates HEFT's overall process of weighting, ranking, and computing a task list.

Note that the upward rank allows HEFT to trade-off between communication overheads and degree of parallelism, e. g., if the communication costs suggest that running two tasks on the same processor would be faster than transferring the input data to another processor. Consequently, HEFT can choose a lower degree of parallelism to avoid excessive communication overhead. HEFT does, however, not have a reliable way to take into account the long term effects of such a decision, which we address with our level order sampling approach.

In the processor assignment phase, HEFT selects for each task, in order of decreasing upward rank (ties arbitrarily broken), the processor that minimizes the earliest finish time of that task. The earliest finish time of a task T_i on a given processor p_j depends on T_i's parents' finish times, the data transfer times from the parent's scheduled processors to p_i, and T_i's execution time $w(T_i, p_j)$. Since sorting by decreasing upward rank yields a topological order, it is certain that when scheduling a task, all its parent tasks have already been scheduled. HEFT uses an insertion-based strategy that schedules a task T_i between tasks already scheduled on p_j if T_i's input data can be transferred to p_i in time and the gap has a length of at least $w(T_i, p_j)$, i. e., the already scheduled tasks are not delayed.

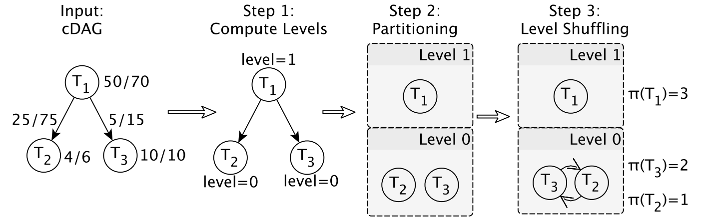

Fig. 3: The L-Order sampling method for obtaining a task order.

III. Adaptive Level Order Sampling

In this section, we describe our randomized approach to dag scheduling. We first introduce the notion of L-Orders, which allow us to partition the space of all topological ordering into regions and efficiently sample from the regions. Second, we propose a scheme to estimate how promising a region is with respect to finding better solutions. We then combine the two into a budgeted adaptive randomized search algorithm.

A. L-Order Sampling

Definition 1. An L-Order π_L is an order of the tasks such that the levels of the tasks are non-increasing with respect to their rank:

$$\pi_L(u) > \pi_L(v) \Rightarrow \text{level}(u) \geq \text{level}(v) \quad (3)$$

Lemma 1. Every L-Order is a topological order. Otherwise, a pair of tasks with $\pi_L(v) > \pi_L(u)$ could exist such that $u \prec v$. Since $\text{level}(u) = 1 + \max_{v \succ u} \text{level}(v)$ we have $\text{level}(u) > \text{level}(v)$, which contradicts the assumption that the levels are non-increasing.

Definition 2. To shuffle the k-th level of an L-Order, replace the order of the nodes of level k with a random order of these nodes. More formally, pick a random bijection $r\colon R \to R$ on the set of the ranks R of the T_i with level k and let the new L-Order π'_L be:

$$\pi'_L(T_i) = \begin{cases} r(\pi_L(T_i)) & \text{if level}(T_i) = k \\ \pi_L(T_i) & \text{otherwise} \end{cases} \quad (4)$$

For example, in Figure 3, the task order is $\pi(T_1) = 3, \pi(T_3) = 2, \pi(T_2) = 1$, where $\text{level}(T_1) = 1, \text{level}(T_2) = 0, \text{level}(T_3) = 0$. To shuffle Level 0, we consider the set of ranks of the tasks with level 0: $R = \{\pi(T_2) = 1, \pi(T_3) = 2\}$. A random bijection could swap the ranks of tasks T_2, T_3: $r = \{(1, 2), (2, 1)\}$. The new order π'_L is then $\pi(T_1) = 3, \pi(T_2) = 2, \pi(T_3) = 1$.

The set of L-Orders is closed under the operation of shuffling any level, since changing the order of tasks with equal levels maintains the requirement of non-increasing task levels. We refer to the set of all possible L-Orders resulting from shuffling a specific level as a *region* of the order space. Thus an L-Order π_L with k levels covers k regions in search space, one for each level in π_L.

A solution to the scheduling problem can be obtained by passing an L-Order to the HEFT algorithm and computing the

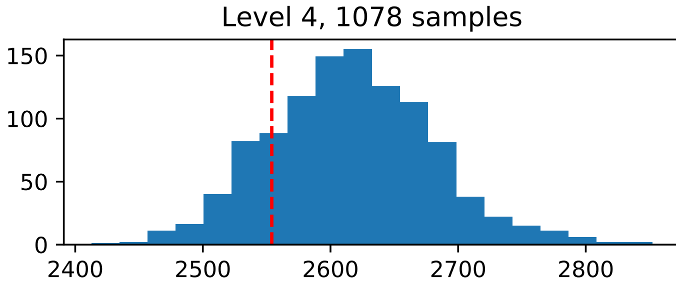

Fig. 4: Distribution of the makespans of L-Orders sampled from a single region. HEFT's makespan is marked in red.

schedule using the processor assignment method described in Section II-C. The quality of the L-Order is defined as the makespan of the resulting schedule, which equals the latest finish time of a task in the schedule. Since the schedule computation step is simple, we will refer to an L-Order as solution where the context does not require a distinction.

Figure 4 shows an exemplary distribution of makespans obtained when passing L-Orders from a specific region to the HEFT algorithm. Here, 1078 permutations of the tasks with a level of 4 have been randomly generated. Note how a substantial fraction of the L-Orders yields makespans shorter than HEFT's, which is marked with a red dashed line.

B. Adaptive Search for L-Orders

We now describe a randomized adaptive budgeted algorithm for searching an L-Order that minimizes makespan based on the L-Order sampling idea from Section III-A. The algorithm consists of exploitation and an exploration phases, which are executed alternately, as shown in Algorithm 1. The overall LOS algorithm takes a communication dag to schedule and a point in time until scheduling shall finish. It then repeatedly tries to replace the current best solution with a better variation of the solution. For parallelization, we simply run several instances of the algorithm and select the best solution.

```
Algorithm: Level Order Sampling
Method Schedule(dag G, deadline):
    solution ← random L-Order of G;
    while current time < deadline do
        //exploit
        best improvement ← exploit(solution);
        //explore
        solution ← max(solution, best improvement);
    end
    return solution;
end
```

Algorithm 1: The level order sampling algorithm combines exploration and exploitation into a budgeted search for an L-Order that minimizes makespan. The adaptiveness of the algorithm stems from variable amounts of time spent in the exploit subroutine (see Section III-C).

We refer to exploitation as the process of evaluating a number of variations π'_L of a reference L-Order π_L, which we obtain by shuffling one of its levels (we shuffle different levels, but each variation differs only in level from the reference π_L). In the exploitation phase, we need to decide how many variations to obtain and from which levels. Intuitively, the level selection process should favor promising levels, i. e., spend compute time in proportion to the probability of improving the current solution. We propose an estimator for the improvement probability in Section III-C. As a second requirement, the amount of time spent on improving a reference solution should be limited to sufficiently balance exploration and exploitation within the time budget. We do this by deriving an estimated time to improvement based on the improvement probabilities of all regions spanned by the current reference order (see Section III-D). The combination of both makes our algorithm adaptive.

Exploration is the process of changing the reference L-Order, which makes new regions eligible for sampling. Our exploration method simply replaces the current reference order π_L by the best variation π^*_L found during the previous exploitation phase. By fixing a favorable variation of a level in the reference and again moving on to exploitation, the algorithm explores interaction effects between variations of different levels. This provides an important conceptual advantage over HEFT, which takes into account only short term consequences of scheduling decisions.

The alternation of exploitation and exploration phases is illustrated in Figure 5. First, a reference order π_L is generated. Here, the dag comprises tasks of three levels and we denote the orders of the tasks of those levels as A, B, and C. During the first exploitation phase, several variations π'_L are generated by shuffling one of the three levels. The best order π^*_L was found in level C. In the exploration phase, the reference solution is set to π^*_L. Note that since level C has changed, the makespan distributions for all other levels change, because of interaction effects between different parts of the schedule. However, all solutions sampled so far from level C are still valid, since sampling shuffles from the shuffled Level C* is the same as sampling shuffles from the original level C. Reusing those solutions is here indicated with the term carryover.

C. Improvement Probability

In this section, we describe how to estimate the improvement probability, i. e., the probability of finding a solution with a makespan better than a given reference makespan r in a certain region of the order space. The basic idea is to analyze the distribution of makespans obtained by sampling L-Orders from that region. We use the observation that the makespans are often normally distributed (as shown in Figure 4), but we do not rely on this assumption. If, however, the makespans are approximately normal distributed (definition follows), the cumulative distribution function of a fitted normal distribution can be used to estimate improvement probability.

Let μ denote the average makespan of the solutions obtained so far from a specific region. Let $\sigma_{95\%}$ denote the upper end-

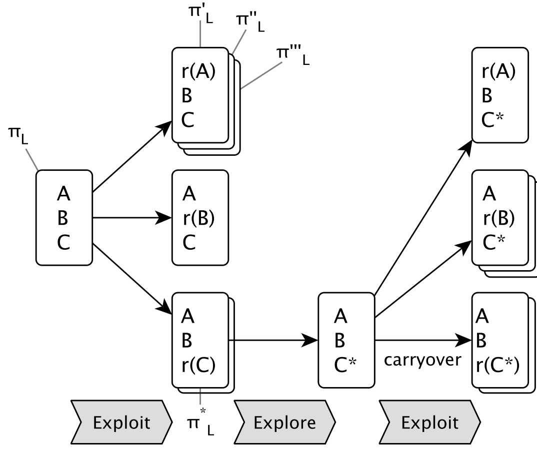

Fig. 5: The interplay between exploitation and exploration. Exploitation refers to the process of sampling variations $\pi'_L, \pi''_L, ...$ of an L-Order π_L. Exploration refers to picking the best variation π^*_L as a new reference to exploit. This way, the search considers L-Orders that combine arbitrary combinations of level shuffles relative to the initial L-Order. Here, A, B, and C refer to permutations of the tasks of a dag with three levels, and $r(\cdot)$ denotes the shuffle operation used to obtained new permutations of a level. C* is the permutation that yields the best variation in the first exploration phase. Note that shuffling twice yields the same population of variations as shuffling once, so the evaluated solutions for Level C can be carried over to the next exploitation phase.

point of a 95% confidence interval for the standard deviation of the makespans. In case of approximately normally distributed makespans, the probability to randomly select an L-Order with makespan m at least as good as a reference makespan r can be estimated by consulting the cumulative distribution $F_{\mathcal{N}}$ of a normal distribution with the according parameters:

$$p_{\mathcal{N}}(m \leq r) = \frac{1}{2} F_{\mathcal{N}}(r; \mu, \sigma^2_{95\%}) \tag{5}$$

By using a confidence interval for the standard deviation rather than the sample standard deviation, we take into account the number of solutions collected so far. For few solutions, we are not as certain about the actual standard deviation, so we assume a larger value. When adaptively selecting regions based on their $p_{\mathcal{N}}$, this encourages sampling from regions with few observations. We set $p_{\mathcal{N}} = 1$ for levels that have less than 2 evaluated solutions. This enforces sampling until the variance can be computed. Should all samples have the same makespan, we change the makespan of the first sample by 1% to avoid division by zero.

We next formalize the notion of approximately normal distributed makespans, and describe our fallback method for estimating the improvement probability. Let n denote the number of solutions sampled from a region. We compare the number k of solutions better than r to the number of better solutions that we would expect to see, given a certain improvement probability. We assume that sampling a solution is a Bernoulli trial that succeeds with the assumed improvement probability $p_{\mathcal{N}}(m \leq r)$. The cumulative distribution function F_B of a Binomial distribution then gives the probability of observing at most k successes among n trials. If we find an improbably ($p < 5\%$) low number of improvements in a region, we assume that Equation 5 does not apply in the current region and use the simpler empirical probability $p_r(m \leq r)$ instead.

$$p_r(m \leq r) = \frac{k}{n} \tag{6}$$

In summary, the improvement probability is based on the cumulative distribution function of a fitted normal when improvement expectations have not been violated, and on a simple success rate otherwise.

$$p(m \leq r) = \begin{cases} p_r(m \leq r) & \text{if } F_B(k; n, p_{\mathcal{N}}(m \leq r)) < 5\% \\ p_{\mathcal{N}}(m \leq r) & \text{otherwise} \end{cases} \tag{7}$$

The improvement probability is used in the next section to adaptively select regions during exploitation and decide upon the fraction of the remaining time budget spent on a single exploitation phase.

D. Exploiting Adaptively

Here we explain how an exploitation phase is structured in detail, which consists of generating multiple variations of the reference order by shuffling different levels. This entails from which levels to sample and how many samples to obtain.

During an exploitation phase, the current reference order spans several regions, one for each level in the dag. We select the level to shuffle next at random, in proportion to each level's improvement probability. Let $p(k = l)$ denote the *level selection probability*, i. e., the probability to pick level l to generate the next variation of the reference order. The level selection probability is the level's share of the summed improvement probabilities. Let $p^l(m \leq r)$ denote the improvement probability for a specific level l.

$$p(k = l) = \frac{p^l(m \leq r)}{\sum_l p^l(m \leq r)} \tag{8}$$

By employing probabilistic level selection, we ensure that most of the compute time is spent on promising levels, but also allow sampling from a less promising level from time to time. The reason is that our estimates of the improvement probabilities might not be extremely accurate, and randomizing gives less promising levels a chance to reveal improved solutions. The level selection process is illustrated in Figure 6.

We base the length of an exploitation phase on the remaining time budget and the estimated improvement probabilities per level. We use the latter to estimate the expected number of additional solutions that we need to sample until improving over the current best found variation π^*_L of the reference order π_L.

Let L be the number of levels with $p^l(m \leq r) > 0$. The expected required number of additional samples for a level is $1/p^l(m \leq r)$ multiplied by the level's selection probability

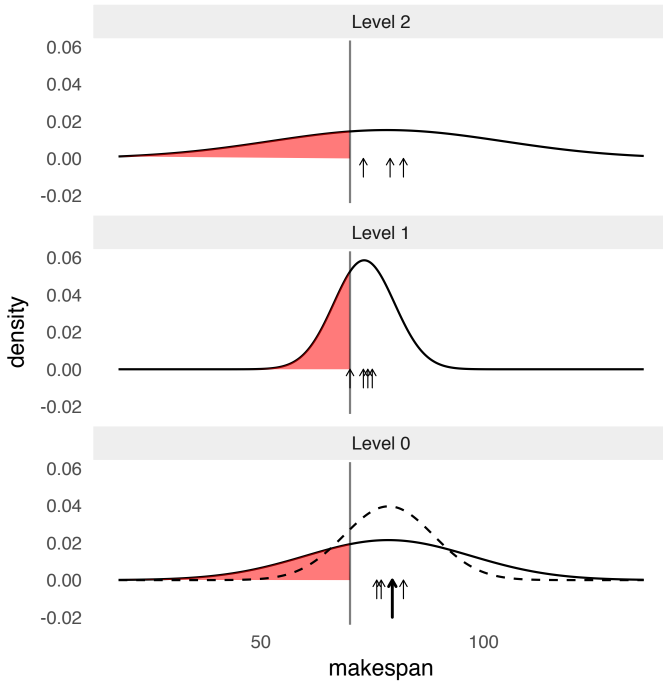

Fig. 6: The level selection process uses the estimated improvement probabilities (red area) to favor promising levels for sampling new variations. The makespans of the L-Orders sampled so far (black arrows on the x axis), are used to fit a normal distribution for each level. Here Level 0 has the highest improvement probability, which is reduced by the new solution (bold arrow, dashed distribution). In the next step, Level 1 has the highest estimated improvement probability. Note that standard deviations are based on 95% confidence intervals, such that the fitted distributions are much wider when few solutions are available, as in Level 2.

$p^l(k = l)$. The overall expected number of additional samples until improvement s_{improv} results from summing over all levels.

$$s_{\text{improv}}(r) = \frac{L}{\sum_l p^l(m \leq r)} \quad (9)$$

If all improvement probabilities are 0, we assume an infinite amount of samples until improvement. Given an estimated compute time t per solution sample, we calculate the wall clock time until improvement t_{improv} as $t \cdot s_{\text{improv}}(r)$. In practice, we use the average of all measured solution evaluation times so far as the estimated compute time.

The time limit on each exploitation phase relative to the algorithm's time budget defines whether the algorithm favors exploitation (spend more time finding good variations of a reference order) or exploration (moving on to different reference orders). In our experiments, we found that sometimes more thorough exploitation is favorable, while in other cases, moving on more quickly to other solutions gives better results. We thus settled on a random fraction in range $[5\%, 50\%]$. We found that randomization does a good job in automatically selecting good values. The downside is that introducing more sources of randomness potentially also increases the variance of the performance of the method, which, however, was found to be sufficiently low during evaluation. The overall design of the exploitation phase is summarized in Algorithm 2.

Algorithm: Adaptive L-Order Exploitation
Method *Exploit(π_L)*:
  best order $\pi^*_L \leftarrow \pi_L$;
  carry over collected samples from last exploit phase;
  partial budget $\leftarrow \mathcal{U}(0.05, 0.5)$ remaining time;
  deadline $\leftarrow$ current time + partial budget;
  $t_{\text{improv}} \leftarrow 0$;
  while *current time*+t_{improv} < *deadline* **do**
    if $0 = \sum p^l(m \leq r)$ **then** break;
    $p^l_s \leftarrow p^l(m \leq r) / \sum_l p^l(m \leq r)$;
    randomly select level l with probability p^l_s;
    $\pi'_L \leftarrow \pi_L$ with level l shuffled;
    new solution = evaluate(π'_L);
    update p^l_s with makespan of new solution;
    $\pi^*_L \leftarrow \max(\pi^*_L, \pi'_L)$;
  end
  return π^*_L
end

Algorithm 2: The exploitation algorithm adaptively samples new L-Orders from the most promising levels as long as the expected time to improvement is below a fraction of the remaining time budget.

For small levels, we do not sample but enumerate all orders for the tasks in that level and then randomly draw without replacement. This avoids evaluating the same solution twice, whereas for levels with more tasks, the factorial grows so fast that the probability of sampling the same solution twice approaches 0. We avoid further considering a level after having evaluated all of its permutations by setting its improvement probability to 0.

IV. Experimental Results

In this section, we summarize our experimental setup, the competitor method, and the improvements and behavior of LOS.

A. Input Instances and Experiments

For evaluation, we generate random dags using the random dag generation method described in [16]. The compute time $w(T_i, p_j)$ of a task on a processor is sampled uniformly at random in range $[1, 100]$. The edge weights $w(T_i, T_j)$ are also sampled uniformly in range $[1, 100]$. The data transfer rates of the network links are set to 1. We evaluated dags of size $n \in \{32, 64, 128, 256, 512\}$ on $p \in \{3, 10, 30\}$ processors.

We refer to a combination of a dag and a number of processors as an *input instance*. For each input instance, we compute a schedule using the original HEFT algorithm and use its makespan as a reference to assess the quality of the schedules delivered by the baseline method and LOS. Since LOS relies on randomization and thus exhibits variance in

performance, we run LOS three times on each input instance, and each run is referred to as an *experiment*.

In total, we generated 4500 random dags, 900 of each size. We combined each of the dags with each number of processors, giving 13,500 input instances. Using three repetitions, this results in 40,500 experiments. The time budget of LOS has been set to 30 seconds for dags of size 32 and 64, and to 5 minutes for dags of size 128, 256, and 512. In each experiment, 4 instances of the LOS algorithm are run in parallel and the best solution is chosen. Overall, 426,072,399 schedules have been considered during the experiments by LOS.

B. Baseline Ranking Methods

As a baseline method, we compute 11 schedules using each of the 11 alternative weighting methods proposed in [3] and select the schedule with the shortest makespan. We briefly refer to this scheduler as "Zhao" (see Figure 7). The weighting methods are: mean, median, minimum, maximum, simple minimum, and simple maximum, each combined with either upward ranking or downward ranking. The proposed minimum/maximum weighting methods calculate the weight of an edge using the data transfer rate between the processors that result in the minimum/maximum compute time. The simple minimum/maximum method calculate the weight of an edge using the highest/smallest data transfer rate among network links. In our case, data transfer rates are homogeneous, such that these weighting methods give the same results. The first weighting method, averaging with upward rank, corresponds to the original HEFT algorithm and is part of the ensemble, such that the baseline never performs worse than HEFT. The same fallback mechanism has been added to LOS.

C. Makespan Reductions

We compute the improvements of LOS over standard HEFT and Zhao's alternative ranking methods on different dag sizes and numbers of processors. For each input instance we compute the *relative makespan* as the ratio between the makespan of the best schedule delivered by a method (Zhao, LOS) and the makespan of HEFT's schedule. Figure 7 shows the distributions of the relative makespans using violin plots. LOS clearly outperforms the baseline method, giving shorter median makespans and longer tails towards short makespans in almost all of the cases. A striking effect is that both the baseline and LOS deliver diminishing improvements for larger dags. An interesting exception is the case of 512-task dags on 30 processors that suggests that LOS could perform well on even larger input instances. However, the relationship between dag size and number of processors is complex, as for example the baseline is better on dags with 256 tasks on 10 processors, which gives a similar task/processor ratio.

Since LOS uses randomization, we analyzed the amount of variation in the relative makespans returned across three different runs of LOS (for each input instance). Mean and standard deviation are computed across the 3 runs and the medians of both the means and the standard deviations are reported in Table I. For instance, in 50% of the experiments with 128 tasks

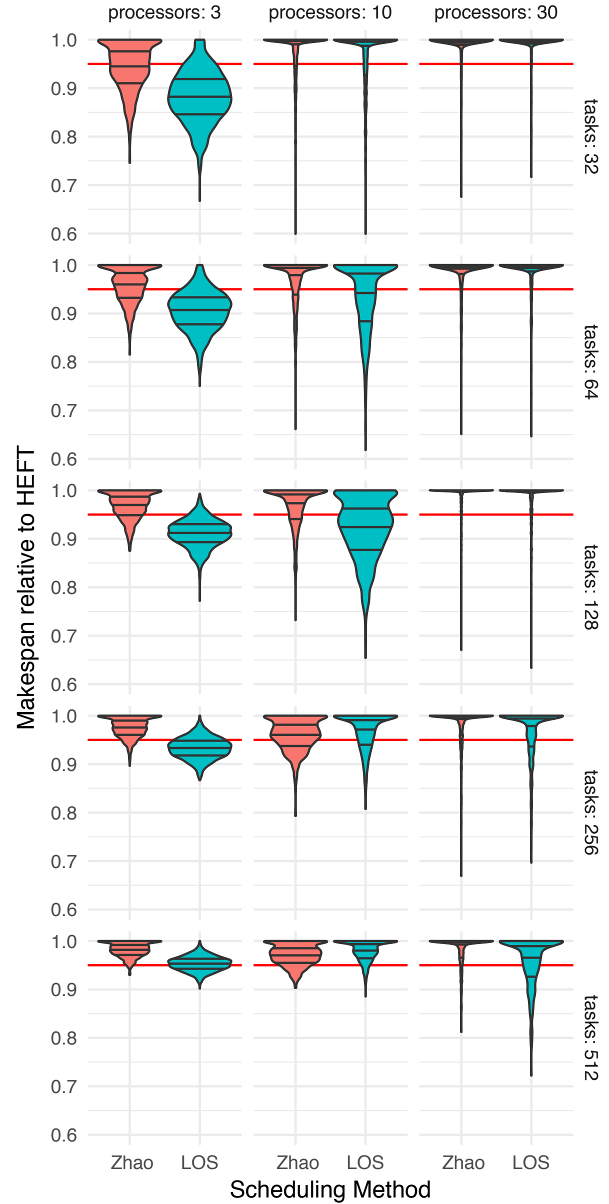

Fig. 7: Distribution of makespans relative to HEFT's makespan for different numbers of processors and dag sizes. The baseline method (red) is consistently outperformed by LOS on 3 processors. The violin plots summarize both the distribution of the relative makespans and their quartiles, as indicated by the three horizontal lines within each colored area. The red reference line indicates 95% of HEFT's makespan.

on 10 processors, the mean relative makespan seen across three runs was ≤ 0.932 with a standard deviation of ≤ 0.003. More concretely, three LOS runs on a specific input instance with 128 tasks and 10 processors gave makespans of 310, 306, and 297. The makespan of HEFT's schedule in this case was 347. The relative makespans of LOS are thus 0.89, 0.88, and 0.85, giving a mean relative makespan of 0.877 and the standard

TABLE I: Median values of the mean and standard deviation of LOS relative makespans for an input instance (3 repetitions).

	3 Processors		10 Processors		30 Processors	
Tasks	LOS	Zhao	LOS	Zhao	LOS	Zhao
32	**0.884** $\pm$ **0.006**	0.954	1 $\pm$ 0	1	1 $\pm$ 0	1
64	**0.908** $\pm$ **0.008**	0.968	**0.964** $\pm$ **0**	0.995	1 $\pm$ 0	1
128	**0.912** $\pm$ **0.006**	0.975	**0.932** $\pm$ **0.003**	0.985	1 $\pm$ 0	1
256	**0.933** $\pm$ **0.005**	0.979	0.983 $\pm$ 0.004	**0.964**	**0.996** $\pm$ **0**	1
512	**0.954** $\pm$ **0.004**	0.984	0.986 $\pm$ 0.005	**0.974**	**0.979** $\pm$ **0**	1

deviation across the relative makespans is 0.019. In this case, the mean is below the median mean of 0.932 and the standard deviation is above the median standard deviation of 0.003. Since input instances cannot easily be ordered with respect to both quantities, the medians reported in Table I are computed independently, one over the average relative makespans and one over the standard deviations of the relative makespans.

D. Progress and Convergence

In this section, we analyze the progress made by LOS during its adaptive search for a schedule. We report the following metrics: average improvement ratio, the number of improvements, and the latest improvement time.

The *average improvement ratio* is the average of the improvement ratios obtained between exploitation phases during a LOS run. The improvement ratio is the makespan of the best found improvement π_L^* and the makespan of the current best solution at that time. For instance, when improving the makespan in two exploitation phases first from 100 to 90 and then from 90 to 45, the first improvement ratio is 0.9 and the second is 0.5, giving an average improvement ratio of 0.7. The average improvement ratio indicates whether the method finds better schedules using a sequence of small improvements or using fewer but larger improvements. The median average improvement across all experiments is 0.95, considering only the cases where LOS finds an improvement over HEFT. Figure 8 summarizes the distribution of average improvement ratios for all combinations of dag sizes and processors. For small dags on few processors, larger average improvement ratios are found, and the distributions shift towards smaller average improvement ratios with increasing dag size. Note that some of the diagrams, e. g., for 30 processors and 32 tasks, summarize fewer input instances, as the experiments for which no improvement has been found have been removed, as explained in greater detail in the next paragraph.

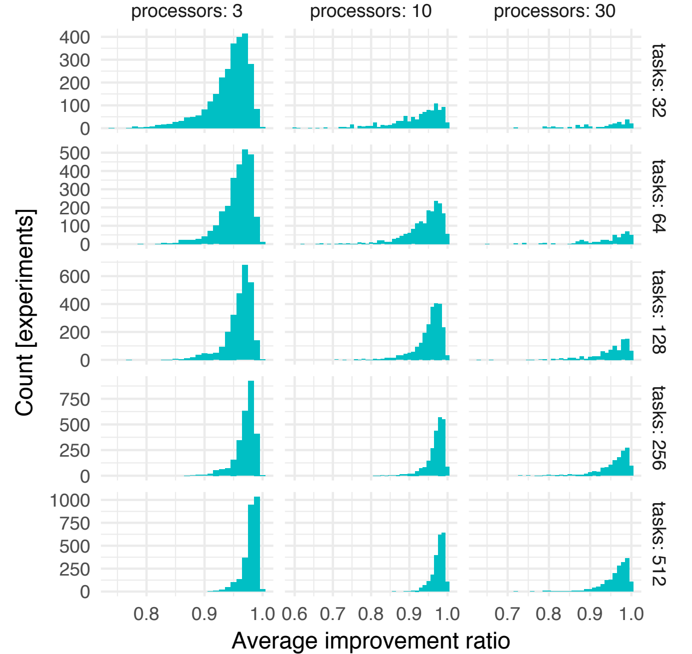

Fig. 8: The average relative improvement between exploitation phases. Small ratios indicate that solutions are improved in small steps. Note that the number of experiments (sum of the bins) differs from panel to panel since in some experiments, LOS was not able to improve over the initial solution.

The *number of improvements* is the number of exploitation phases that yield a better solution than the reference solution. It can be zero, if none of the exploitation phases finds a schedule that is better than the initial solution. The number of improvements combined with the average improvement ratio gives an estimate of the overall improvement. Figure 9 shows the distribution of the number of improvements over all input instances and sample seeds. We distinguish input instances in which the best found solution was better (shown in green) than HEFT's solution and those in which LOS did not find a better schedule than HEFT's schedule (red). This was observed in 60 out of 13440 cases for 3 processors, but happens quite often for larger processor counts. In some cases, e. g., for 32 tasks on 30 processors, the input instances might be too simple, but in general, adding more processors seems to result in significantly harder input instances. Among the unsuccessful runs (red) we sometimes observe runs with a non-zero number of improvements, e. g., for 512 tasks on 30 processors. This indicates that LOS makes progress, but not fast enough to outperform HEFT within the given time budget.

The *latest improvement time* refers to the elapsed wall clock time until LOS does not find more improvements. This metric indicates at which portion of the allocated time budget LOS converges to its solution. For instance, a latest improvement time of 50 seconds on a time budget of one minute indicates that improvements are found close until the budget expires, which suggests that LOS could further improve on a larger time budget. Figure 10 shows the distribution of latest improvement times for dags with at least 128 tasks. For

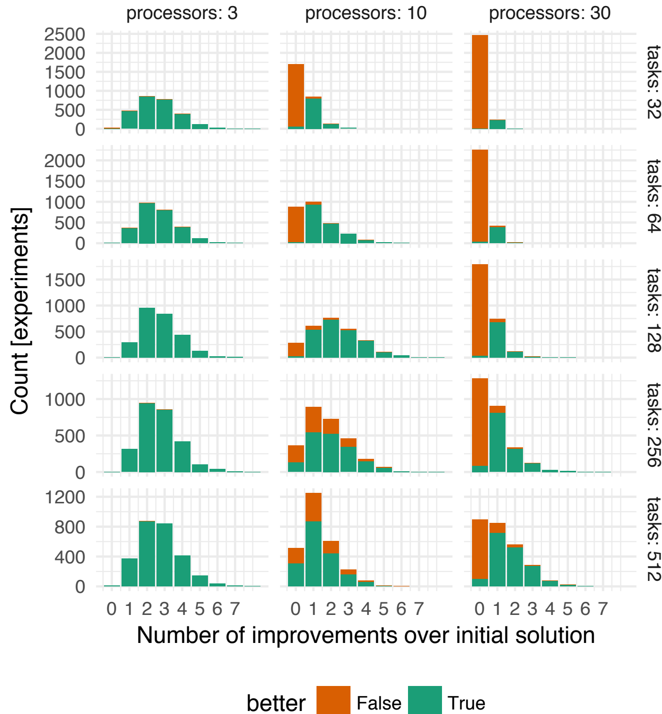

Fig. 9: The number of improvements refers to the number of times LOS was able to improve upon its current best solution. The *better* variable refers to whether LOS was able in a given experiment to find a schedule that outperforms HEFT.

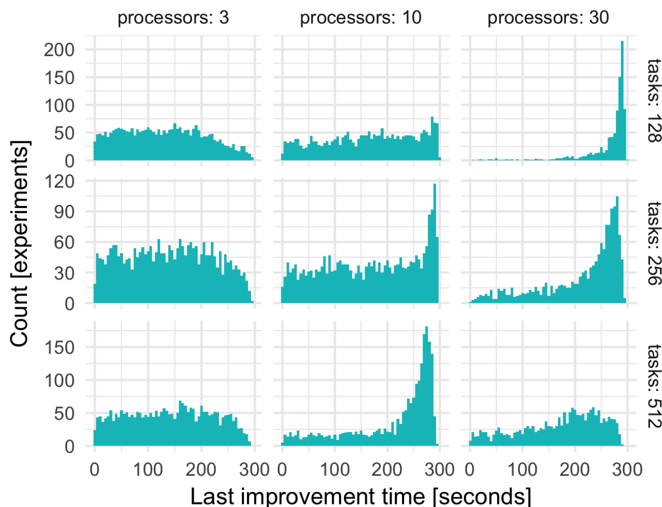

Fig. 10: The latest improvement time refers to the elapsed wall clock time when LOS was last able to improve upon its previous solution. For 3 processors, convergence times are relatively uniform. In some configurations (e. g., tasks ≤ 256, processors ≥ 30) LOS seldom converges before the end of the time budget, indicating that schedules could further be improved on a higher time budget.

these input instances, LOS has been allotted a time budget of 5 minutes, whereas for the smaller input instances, a time limit of 30 seconds was set.

E. Non-significant Factors

We evaluated a few alternative designs that showed very similar behavior to the results shown above. For instance, we evaluated the effect of sorting the tasks in each level of the initial L-Order according to HEFT's ranks. The idea is to provide a better starting point than a random permutation. However, improvements have been marginal.

Second, we tried a simpler processor assignment method which always appends tasks to the end, instead of using HEFT's insertion-based policy that looks for gaps between already scheduled tasks. The idea is to reduce the time needed to compute a schedule for a single input instance, which is a limiting factor on the number of evaluated solutions when facing a time budget. The question was whether LOS could replace the insertion mechanism by considering more orders, orders that may not require insertion in the first place. However, both methods did not yield significant differences, such that we selected HEFT's default (but slower) insertion-based policy.[1]

Finally, we evaluated the performance of our method when using different rejection probabilities $p \in \{1\%, 5\%, 20\%\}$ for switching between $p_{\mathcal{N}}(m \leq r)$ and $p_r(m \leq r)$. We did not observe significant differences, which indicates that our method is robust with respect to this parameter.

V. Related Work

Improving upon HEFT's performance is an ongoing effort. Several alternative weighting schemes have been proposed, which we briefly summarize below. In addition, several alternatives to HEFT have been proposed. Since these usually compare to HEFT as the baseline, we focused in our evaluation on the performance of LOS relative to HEFT.

As described in Section II-C, HEFT uses averaged computation and communication times before computing upward ranks. In [3], it has been empirically shown that the weighting method can have a significant impact on the makespan of HEFT's schedule. The authors evaluate simple alternative weighting methods, such as the worst case computation time $\max_p w(T_i, p)$ to compute the weights of tasks and edges. In contrast to this method, we search the order space rather than trying to come up with a single weighting method that results in a favorable order most of the times. In addition, we consider variations of orders that affect only certain parts of the dag, which is similar to applying different weighting methods to different parts of the dag.

In [8] a weighting method for series parallel graphs is proposed. Instead of using the maximum of the child weights,

[1]We also evaluated the impact of using the insertion-based policy on HEFT's performance. In about 5% of the cases, gap search even resulted in longer schedules (by up to 10%), but most of the times, it did not have an impact. However, in 20% of the cases, using the insertion-based policy resulted in 1%-10% shorter schedules compared to simply appending tasks at the end of a processor's partial schedule.

the child weights are summed up. The idea is to consider more than one path (the critical path) to assess the downstream work of a task. However, the authors focus on a special class of graphs, whereas we do not impose a restriction on the structure of the workflows.

In [9], a weighting method is proposed that adds the sum $\sum_{(u,v)\in E} c(u,v)$ of outgoing data transfers from a task u to its weight. It is assumed that the network links have the same speed, although averaging could be used to handle heterogeneous transfer speeds.

In [10] a weighting method using a sufferage metric [11] is proposed. Sufferage refers to the increase in computing time when scheduling a task on the second-best processor compared to the fastest processor for that task. Using sufferage allows to prioritize tasks that would potentially run much slower when scheduled with lower priority. The scheme is used to apply HEFT in CPU-GPU environments.

In [12], the BMCT heuristic is proposed, which is supposed to be more robust with respect to the choice of the weighting method. First, HEFT's weighting method (averaging) is applied. Then, the dag is partitioned into a sequence of independent task sets which are then scheduled using a heuristic for independent task scheduling. The method is computationally more expensive than HEFT because it iteratively optimizes the placement of the tasks in each set.

In [13], a lookahead version of the HEFT algorithm is proposed that considers the earliest finish times of the children of a task for a given placement of the task, in addition to the task's own earliest finish time. This also aims in a similar direction as our method, i. e., trading scheduling time for better schedules. However, the approach is based on a single weighting method. In [14] an improved version of the lookahead-HEFT is proposed that achieves similar effects at lower computational costs.

VI. Discussion

Here, we briefly summarize our findings and discuss future work. LOS employs an innovative randomized approach to dag scheduling on heterogeneous resources with communication times. It uses L-Orders to partition the space of possible task orders into regions and estimates the probability for various regions of randomly sampling a better order than the current best order. LOS schedules for 3 processors and less than 256 tasks outperform HEFT by at least 10% in 50% of the experiments, whereas the baseline method gives only a median 3% improvement. However, we see various opportunities to further improve the method and gain insights into the scheduling problem at hand. Most importantly, a more principled approach to balancing exploration and exploitation, partitioning the order space into more fine grained regions, comparing to more baseline methods, and an in-depth analysis of the computed rankings could be pursued.

LOS uses a randomized approach to balance exploration and exploitation. The length of an exploit phase depends on a random fraction of the remaining time budget between 0.05 and 0.5. Comparing to experiments using a fixed fraction of the remaining time budget for exploitation phases, the randomized method performs favorably. However, more sophisticated methods would probably be able to further reduce the makespan of the schedules found by LOS. For instance, one could introduce a minimum improvement ratio and rate. Intuitively, LOS should not spend too much time on getting a 1% improvement over the current solution, even if it is currently the most promising known region in order space. Rather, the algorithm should then favor exploration over exploitation to find regions where either larger improvements or quicker improvements are possible.

Second, for dags with larger numbers of task per level, the regions in order space might be too coarse-grained to quickly make progress. A simple solution would be to allow LOS to apply a binary search scheme to identify promising portions of levels, by recursively splitting levels in half. The partial levels could then be treated as separate levels to sample from and estimate improvement probabilities for. A related aspect is the carryover of samples between exploitation phases. Currently, we reuse the samples collected from the level that gave the best variation of the reference order, but in fact, not all the distributions of the other levels have to change a lot when changing one level in the reference order. Carrying over all samples and validating their distributions could maybe reduce the number of L-Order evaluations needed until a level's improvement probability can be reliably estimated. In addition, reusing partial schedules and an implementation that is optimized for performance would speed up the scheduling and thus the number of solutions LOS is able to consider within its time budget.

Third, comparing to more baseline methods would be interesting. Since HEFT was evaluated against a broad range of scheduling methods, including genetic algorithms, comparing to HEFT is a good benchmark. However, new scheduling methods often use custom dag generators and it would be interesting to compare to these methods in their respective evaluation scenarios. One could even use the shuffle operator as a mutation operator in a genetic algorithm (a crossover operator is also straightforward) and compare the improvement rates of our adaptive search to the improvement rates given by a classical genetic algorithm. Note however, that our method has no hyperparameters except the budget, whereas genetic algorithms usually need tuning, e. g., to select population size, mutation rates, etc., which complicates a fair comparison.

Finally, comparing well performing to poorly performing orders could yield further insights into the scheduling problem.

Acknowledgements

Carl Witt received funding by Deutsche Forschungsgemeinschaft through the SOAMED graduate school (GRK 1651). Sam Wheating received funding for a research intern at Humboldt-Universität zu Berlin by Deutscher Akademischer Austauschdienst through the RISE program.

REFERENCES

[1] H. Casanova, A. Legrand, and Y. Robert, *Parallel Algorithms*. CRC Press, 2008.

[2] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[3] H. Zhao and R. Sakellariou, "An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm," in *Euro-Par 2003 Parallel Processing*. Berlin, Heidelberg: Springer, Berlin, Heidelberg, Aug. 2003, pp. 189–194.

[4] S. Ali, H. J. Siegel, M. Maheswaran, D. A. Hensgen, and S. Ali, "Task Execution Time Modeling for Heterogeneous Computing Systems." *Heterogeneous Computing Workshop*, pp. 185–199, 2000.

[5] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, vol. C-21, no. 2, pp. 137–146, Feb. 1972.

[6] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[7] E. G. Coffman and J. L. Bruno, *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[8] K.-C. Huang, Y. L. Tsai, and H. C. Liu, "Task ranking and allocation in list-based workflow scheduling on parallel computing platform," *The Journal of Supercomputing*, vol. 71, no. 1, pp. 217–240, Sep. 2014.

[9] E. Ilavarasan, P. Thambidurai, and R. Mahilmannan, "Performance Effective Task Scheduling Algorithm for Heterogeneous Computing System," in *The 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)*. IEEE, 2005, pp. 28–38.

[10] K. R. Shetti, S. A. Fahmy, and T. Bretschneider, "Optimization of the HEFT Algorithm for a CPU-GPU Environment," in *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, Nanyang Technological University, Singapore City, Singapore. IEEE, Jan. 2014, pp. 212–218.

[11] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," *9th Heterogeneous Computing Workshop (HCW 2000)*, pp. 349–363, 2000.

[12] R. Sakellariou and H. Zhao, "A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems." *IPDPS*, vol. 18, pp. 1571–1583, 2004.

[13] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm," in *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society, Feb. 2010.

[14] H. Arabnejad and J. G. Barbosa, "List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table." *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, 2014.

[15] W. Zheng and R. Sakellariou, "Stochastic DAG scheduling using a Monte Carlo approach," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1673–1689, Dec. 2013.

[16] P. L. Krapivsky and S. Redner, "Organization of growing random networks," *Physical Review E*, vol. 63, no. 6, p. 066123, May 2001.