2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00087

A Guaranteed Approximation Algorithm for Scheduling Fork-Joins with Communication Delay

Pierre-François Dutot
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG,*
38000 Grenoble, France
pierre-francois.dutot@univ-grenoble-alpes.fr

Yeu-Shin Fu, Nikhil Prasad and Oliver Sinnen
Parallel and Reconfigurable Computing Lab,
Dept. of Electrical, Computer, & Software Engineering
University of Auckland
Auckland, New Zealand
guyver.fu@anu.edu.au, nikhilprasad44@gmail.com,
o.sinnen@auckland.ac.nz

***Abstract*—Scheduling task graphs with communication delay is a widely studied NP-hard problem. Many heuristics have been proposed, but there is no constant approximation algorithm for this classic model. In this paper, we focus on the scheduling of the important class of fork-join task graphs (describing many types of common computations) on homogeneous processors. For this sub-case, we propose a guaranteed algorithm with a $(1+\frac{m}{m-1})$-approximation factor, where m is the number of processors. The algorithm is not only the first constant approximation for an important sub-domain of the classic scheduling problem, it is also a practical algorithm that can obtain shorter makespans than known heuristics. To demonstrate this, we propose adaptations of known scheduling heuristic for the specific fork-join structure. In an extensive evaluation, we then implemented these algorithms and scheduled many fork-join graphs with up to thousands of tasks and various computation time distributions on up to hundreds of processors. Comparing the obtained results demonstrates the competitive nature of the proposed approximation algorithm.**

***Index Terms*—task scheduling, communication delays, approximation algorithm, fork-join, DAGs**

I. Introduction

The problem of scheduling task graphs with communication delay on homogeneous processors is a classical scheduling problem, which is also used in practice. In the $\alpha|\beta|\gamma$ notation of scheduling problems [1] it is denoted as $P|prec, c_{ij}|C_{max}$. Its objective is to minimise the makespan (or schedule length), which is a strongly NP-hard optimisation problem [2]. While many dozens of heuristics have been proposed for this problem, e.g. [3], [4], [2], [5], [6], [7], there is no constant approximation algorithm for $P|prec, c_{ij}|C_{max}$ [8].

Task graphs with a fork-join structure (Figure 1) are an important subclass of the general directed acyclic graphs (DAGs) that represent task dependences. Fork-joins are a fundamental structure that is found in many other graph structures, in particular in series-parallel graphs. At the same time fork-joins represent a large and common class of computations, e.g. MapReduce frameworks, scatter-gather patterns in MPI, fork-join ExecutorService in Java etc. [9].

This work was partially funded by the REGALE project under the EuroHPC Programme grant agreement n. 956560.
*Institute of Engineering Univ. Grenoble Alpes

In this paper we will therefore study the scheduling of fork-join task graphs with communication delay on homogeneous processors, denoted as $P|fork-join, c_{ij}|C_{max}$. Related work on this particular problem includes an EPTAS [10], which, however, is rather of theoretical value and less useful in practice. Polynomial algorithms have been proposed for special cases of the problem where the computation time of the tasks is equal, i.e. $P|fork-join, p_j = p, c_{ij}|C_{max}$ [11]. In optimal scheduling, fork-join graphs are among the most difficult to schedule as there are communication costs, but no dependence among most of the tasks [12], [13].

In this paper we want to focus on designing practical algorithms for the fork-join scheduling problem, which at the same time give some theoretical guarantees. To that end we propose a $1+\frac{m}{m-1}$-approximation algorithm, with m being the number of processors. Beyond this guaranteed algorithm, we also study and adjust well known list scheduling algorithms [7] for the specific fork-join graph structure. These algorithms are simple and have low runtime complexity. In an extensive evaluation we then compare the performance of the proposed algorithms on a large set of randomly generated task graphs with up to 10000 tasks. Special attention is given to the generation of the task weights with appropriate distribution models from the literature. Thousands of graphs generated in this fashion are then scheduled on different processor numbers ranging from 3 to 512. Our proposed approximation algorithm demonstrates its competitiveness as a practical algorithm, while having a schedule length guarantee.

The rest of this paper is organized as follows. Section II formerly defines the scheduling model and used conventions. Our approximation algorithm, called ForkJoinSched, is proposed and analyzed in Section III. Section IV then proposes meaningful alternative heuristics for scheduling fork-join task graphs based on the list scheduling approach. To compare and evaluate all algorithms, we introduce our evaluation method and setup in Section V. The results we obtained in the evaluation are presented and discussed in Section VI before the paper concludes in Section VII.

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00087

II. Model

In this paper we target the scheduling of a program or application that can be represented by a fork-join graph as illustrated in Figure 1. It consists of a $source$ task and a $sink$ task and a set of V inner tasks, excluding $sink$ and $source$, which are independent of each other, but all children of $source$ and parents of $sink$. We denote the ith task of V as $n_i \in V$. Each task is associated with a computation weight denoted by w, where w_i is the weight of n_i. The computation weight corresponds to the execution time of the task. Weights in_i (out_i) associated with the incoming (outgoing) communications or edges of n_i represent the time it takes to transfer the corresponding data between different processors.

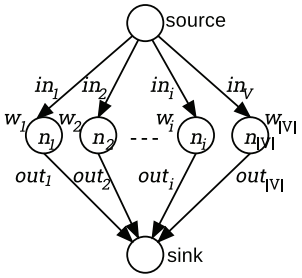

Fig. 1. Fork-join task graph

The problem to be addressed here is scheduling of the fork-join graph on m homogeneous processors, with the objective to minimize the makespan (or schedule length), i.e. the finish time of the $sink$ task. Formally, this scheduling problem is denoted by $P|fork-join, c_{ij}|C_{max}$ in the standard $\alpha|\beta|\gamma$ notation of scheduling problems, see e.g., [1]. This is a sub-domain of the classical scheduling problem $P|prec, c_{ij}|C_{max}$, where the task graph can have arbitrary precedence constraints ($prec$), hence can be any directed acyclic graph. It is assumed for this widely used model, e.g. [2], [3], [4], [5], [6], [7], [10], [11], [12], [14], [15], that the communication is contention free and can happen concurrently. Further, it is assumed that the communication is handled by a sub-system and the processors are not involved, thus computation can overlap with communication.

For each task $n_i \in V$ (and $source$ and $sink$) we need to determine a starting time and an allocated processor p_i, $1 \leq i \leq m$. We denote the starting time of task n_i as σ_i and its executing processor as π_i.

In this scheduling model there is no task preemption, i.e a task, once started, is executed until it is finished and at most one task is executed by each processor at any time. Hence $\forall n_i, n_j \in V; n_i \neq n_j$:

$$\pi_i = \pi_j \Rightarrow \sigma_j \geq \sigma_i + w_i \text{ or } \sigma_i \geq \sigma_j + w_j$$

A task can only start when the data from its parent has arrived. Hence, given the communication model assumptions stated above, the earliest start time of a task is constrained by the finish time of its parent plus the communication delay, if the parent and the child are executed on different processors. The reasonable assumption is made that the cost to communicate can be neglected if both tasks are executed on the same processor, thus set to 0. Formally $\forall n_i, n_j \in V$:

$$\sigma_i \geq \sigma_{source} + w_{source} + \begin{Bmatrix} in_i & \text{for } \pi_i \neq \pi_{source} \\ 0 & \text{for } \pi_i = \pi_{source} \end{Bmatrix} \quad (1)$$

And for $sink$:

$$\sigma_{sink} \geq \max_{i \in V} \left\{ \sigma_i + w_i + \begin{Bmatrix} out_i & \text{for } \pi_i \neq \pi_{sink} \\ 0 & \text{for } \pi_i = \pi_{sink} \end{Bmatrix} \right\} \quad (2)$$

For convenience we denote the finish time of the last task on p_i (excl. $sink$) as f_i.

A. Observations and conventions

We use the convention of $\sigma_{source} = 0$. Without loss of generality, we set $w_{source} = w_{sink} = 0$ for simplicity, as an obtained schedule with this assumption can be trivially adjusted to the real non-zero values. As we are targeting *homogeneous* processors, we also use the convention that $\pi_{source} = p_1$. As a consequence there are two options for the sink:

1) $sink$ and $source$ on same processor, i.e. $\pi_{sink} = \pi_{source} = p_1$, or
2) $sink$ and $source$ on different processors, and by convention we then set $\pi_{sink} = p_2$

All other processors, not hosting either $sink$ or $source$, are called *remote* processors.

III. Approximation algorithm

In this section we discuss and propose a constant approximation algorithm for scheduling fork-join task graphs on m homogeneous processors. We start with an overview of the algorithmic idea before presenting the (sub)-algorithms and discussing their properties.

A. Algorithmic idea

Based on the observation in the previous section we distinguish two cases:

1) $source$ and $sink$ on p_1
2) $source$ on p_1, $sink$ on p_2

For each case a best schedule is found separately and the overall best schedule that will be returned is the one with the shorter makespan.

These are the algorithmic steps for each case:

- **Order V tasks**, by sum of weights, into list
- **Divide ordered list** into two sets
- **Schedule separately**
  1) One set on p_1 (or p_1 and p_2)
  2) Other set on remote processors $p_2(p_3), \ldots, p_m$
- **Migrate tasks** from remote processors to p_1(and p_2) until no improvement

The proposed algorithm first orders all tasks V and then divides them into two sets. All possible division points are

considered. One set is scheduled on p_1 (case 1) or p_1 and p_2 (case 2). The other set is scheduled on the remote processors $p_2, \ldots, p_m$ (case 1) or $p_3, \ldots, p_m$ (case 2). As all V tasks are independent, the scheduling of each set can be done separately. In the last algorithmic step the algorithm migrates tasks from the remote processors to p_1(and p_2) until no further improvement is achieved, the start time σ_{sink} of $sink$ can not be reduced further.

B. Scheduling on remote processors

Let us start by looking at the scheduling of the tasks on the remote processors first. We do so by forcing all tasks to be scheduled on the remote processors and that $source$ and $sink$ are on p_1 (the considerations are the same for $source$ on p_1 and $sink$ on p_2).

A standard greedy list scheduling, see e.g. [7], here called REMOTESCHED (Algorithm 1), is used to schedule the tasks as early as possible on the remote processor. This greedy algorithm never leaves a processor idle if there is an available task, i.e. $\exists f_k \leq in_i$ for $2 \leq k \leq m$ and $i \in \{x \in V : x \text{ not scheduled yet}\}$.

Algorithm 1 REMOTESCHED(V_x, P_x) – list scheduling of tasks V_x on remote processors P_x

1: Order tasks of V_x into list L by non-decreasing in-order
2: **for all** $i \in L$ in order **do**
3: $p_{min} \leftarrow \arg\min_{j \in P_x}\{f_j\}$; $\pi_i \leftarrow p_{min}$; $\sigma_i \leftarrow \max\{f_{min}, in_i\}$

Figure 2 illustrates a schedule of the tasks only on the remote processors with REMOTESCHED, where the white boxes represent the scheduled tasks. Note that in this and the following schedule illustrations, the execution times of $source$ and $sink$ seem to be greater than 0, but this is only for better readability.

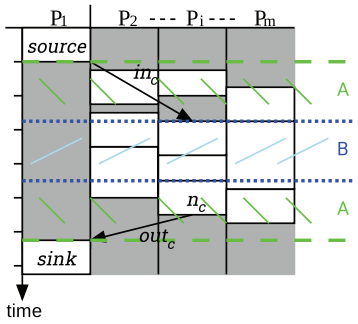

Fig. 2. Scheduling on remote processors only

REMOTESCHED gives a guarantee on the obtained makespan as established in the next lemma.

Lemma 1 (REMOTESCHED)**.** *For scheduling all tasks V on the remote processors,* REMOTESCHED *is a 2-approximation algorithm.*

Proof. Let C_{remote} be the makespan (i.e. σ_{sink}, with $w_{sink} = 0$) of the schedule obtained by REMOTESCHED and C^*_{remote} the optimal makespan for scheduling all tasks on the remote processors. We call a task a critical task n_c, when its outgoing communication arrives at the start of $sink$, i.e. $\sigma_{sink} = \sigma_c + w_c + out_c$, hence it inhibits an earlier start of $sink$. Let $A = in_c + w_c + out_c$ (illustrated as two areas in Figure 2)) and we have $A \leq C^*_{remote}$ by the constraints (1) and (2). We now let $B = \sigma_c - in_c$, in other words the time between the arrival of the input data of n_c and its execution start. Due to the greedy nature of REMOTESCHED, there cannot be any idle time slot on the remote processors between in_c and σ_c. If there were, REMOTESCHED would have started n_c in that slot. It follows that $B \leq \frac{\sum_{i \in V} w_i}{m-1}$ and $B \leq C^*_{remote}$. Finally we have $C_{remote} = A + B \leq 2C^*_{remote}$. □

C. Case 1: source and sink on p_1

This section now proposes the algorithm for case 1 where $source$ and $sink$ are allocated to p_1 and the V tasks are scheduled on all processors. Our intention is to reuse the same reasoning as in the remote only case of the previous sub-section. In order to enable that, we need to make sure that $A = in_c + w_c + out_c \leq C^*_{p_1}$, with $C^*_{p_1}$ being the optimal makespan when $source$ and $sink$ are on p_1. For this to be generally true, it must be guaranteed that the highest $in_k + w_k + out_k$, $k \in V$, on remote processors, i.e. $\pi_k \neq p_1$, is the same or less than in an optimal schedule with makespan $C^*_{p_1}$. Consequently, we try out all possibilities and with that propose FORKJOINSCHED-CASE1, given in Algorithm 2.

Algorithm 2 FORKJOINSCHED-CASE1 – $source$ and $sink$ on p_1

1: Index tasks from 1 to $|V|$ in non-decreasing $in + w + out$ order
2: **for all** $i, 1 \leq i \leq |V| - 1$ **do**
3: Allocate $source$ and $sink$ to p_1
4: $V_1 \leftarrow \{n_1, \ldots, n_i\}$; $V_2 \leftarrow \{n_{i+1}, \ldots, n_{|V|}\}$
5: Schedule tasks V_2 ASAP on p_1 in any order
6: REMOTESCHED$(V_1, \{p_2, \ldots, p_m\})$
7: MIGRATETOP1
8: **if** $makespan_i < makespan_{min}$ **then**
9: record schedule, update $makespan_{min}$

FORKJOINSCHED-CASE1 starts by ordering the tasks through re-indexing them such that $in_i + w_i + out_i \leq in_j + w_j + out_j$ is true $\forall i, j \in V, i < j$. The for-loop then iterates over the ordered tasks. In each iteration the tasks are divided into two sets: the set with the upper indices $V_2 = \{n_{i+1}, \ldots, n_{|V|}\}$ are put on p_1 in any order and the lower indices tasks $V_1 = \{n_1, \ldots, n_i\}$ are scheduled on the remote processors $p_2, \ldots, p_m$ with REMOTESCHED. This is repeated until we have considered all possible splits of the ordered tasks.

An illustration of an intermediate schedule obtained by FORKJOINSCHED-CASE1 for one iteration is given in Figure 3.

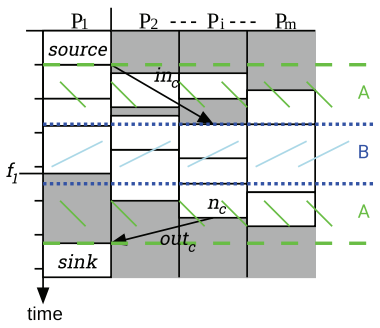

Fig. 3. FORKJOINSCHED-CASE1 intermediate schedule – *source* and *sink* on p_1

As we will see in Lemma 2, the procedure described so far is sufficient to obtain an approximation guarantee. Yet, as shown in the illustration, there might be idle time on p_1 and it makes heuristic sense to use it. To that end, we propose MIGRATETOP1, which is called from FORKJOINSCHED-CASE1 and detailed in Algorithm 3.

Algorithm 3 MIGRATETOP1 – Task migration to p_1

```
while f_1 < σ_c + out_c do
    Move n_c to p_1; σ_c ← f_1; f_1 ← σ_c + w_c
    REMOTESCHED({i ∈ V, π_i ≠ p_1}, {p_2, ..., p_m})
    Determine new n_c
```

MIGRATETOP1 repeatedly moves the (new) critical task n_c to p_1 until f_1 is greater than $\sigma_c + out_c$. After the move, the remote tasks are re-scheduled with REMOTESCHED to best use the free gap, because the removal of n_c from the remote task set might lead to other scheduling decisions. Last, the new critical task is determined for the next round.

Lemma 2 (FORKJOINSCHED-CASE1)**.** *For scheduling source and sink on p_1 and tasks V on all processors,* FORKJOINSCHED-CASE1 *is a $(1 + \frac{m}{m-1})$-approximation algorithm.*

Proof. Let $C^*_{p_1,i}$ be the optimal makespan (i.e. σ_{sink}, with $w_{sink} = 0$), for the case where *source* and *sink* are on p_1 and where the highest indexed task on the remote processors is n_i. The overall optimal makespan is then $C^*_{p_1} = \min_{i \in V} C^*_{p_1,i}$.

The start time of *sink* (i.e. the makespan) can be constrained by *(a)* the tasks on p_1 (i.e. $\sigma_{sink} = f_1$), or *(b)* the arrival of data from the tasks on the remote processors.

If *(a)* applies for an initial schedule produced by FORKJOINSCHED-CASE1 for iteration i, it holds that $f_1 \leq f_1^{*,p_1,i}$, because in the initial schedule produced by FORKJOINSCHED-CASE1 only tasks indexed higher than i are on p_1. There cannot be less in the optimal, hence the one produced by FORKJOINSCHED-CASE1 is optimal for the given iteration i.

For *(b)*, per definition of $C^*_{p_1,i}$ and n_c (see Lemma 1) it holds $A \leq in_c + w_c + out_c \leq in_i + w_i + out_i \leq C^*_{p_1,i}$. During the B period, there can be idle time only on p_1 (as illustrated in Figure 3). Hence it holds $B \leq \sum_{i \in V} \frac{w_i}{m-1} \leq \frac{m}{m-1} \sum_{i \in V} \frac{w_i}{m} \leq \frac{m}{m-1} C^*_{p_1,i}$. Together, under *(b)*, for the schedule obtained by FORKJOINSCHED-CASE1 for i we have $C_{ForkJoinSched-Case1,i} = A + B \leq (1 + \frac{m}{m-1}) C^*_{p_1,i}$,

Migration of tasks to p_1 cannot increase the initial makespan for iteration i: Tasks are only migrated to p_1 if the makespan is still constrained by the arrival of data from the remote processors, i.e. *(b)*, and there is sufficient idle time to accommodate the migrated critical task n_c (condition $f_1 < \sigma_c + out_c$). The rescheduling of the remote tasks with REMOTESCHED does not start any task later as before the rescheduling, because they are scheduled in the same order, but with one less task (the migrated n_c). Hence, the arrival time of the data of the *new* n_c (after rescheduling) will not be later than before.

As FORKJOINSCHED-CASE1 returns the overall best makespan, it follows $C_{ForkJoinSched-Case1} \leq (1 + \frac{m}{m-1}) C^*_{p_1}$ □

One might wonder if after the migration, when there is no idle time during the B period on p_1, the algorithm would not guarantee a 2-approximation (as in the remote only case). Unfortunately, this is not true as after the last migration the tasks on p_1 might constrain the makespan. Hence we would be in situation *(a)*, but without a meaningful insight about the tasks on p_1 in relation to those in an optimal schedule.

D. Case 2: *source* on p_1, *sink* on p_2

With the same general approach as in case 1, we now propose the algorithm for case 2 where *source* is allocated to p_1 and *sink* to p_2 and the V tasks are scheduled on all processors. Algorithm 4 outlines FORKJOINSCHED-CASE2 for this case 2. As in case 1, we iterate over all tasks and divide the tasks into two sets. Set V_2 is allocated to p_1 *and* p_2 in this case, and set V_1 goes to the remote processors. The difference to case 1 lies in the scheduling of V_2 on the two processors and the migration from the remote processors to them.

Algorithm 4 FORKJOINSCHED-CASE2 – *source* on p_1 and *sink* on p_2

```
Index tasks from 1 to |V| in non-decreasing in + w + out order
for all i, 1 ≤ i ≤ |V| − 1 do
    Allocate source to p_1 and sink to p_2
    V_1 ← {n_1, ..., n_i}; V_2 ← {n_{i+1}, ..., n_{|V|}}
    Schedule tasks {n_j ∈ V_2 : in_j ≥ out_j} ASAP on p_1 in non-increasing out_j-order
    Schedule tasks {n_j ∈ V_2 : in_j < out_j} ASAP on p_2 in non-decreasing in_j-order
    REMOTESCHED(V_1, {p_3, ..., p_m})
    MIGRATETOP1P2
    if makespan_i < makespan_min then
        record schedule, update makespan_min
```

Let us first look at the scheduling on the two processors p_1 and p_2. FORKJOINSCHED-CASE2 schedules the V_2 task

whose incoming communication is greater or equal to its outgoing communication on p_1 the others on p_2. The intuition is that the larger communications are zeroed, as local communication with $source$ on p_1 and $sink$ on p_2 is cost free. As-early-as-possible scheduling is used on both p_1 and p_2. On p_1 this gives us the finish time f_1 as before without any idle gaps between the tasks. On p_2, due to the incoming communication costs, there will be idle gaps, at least at the start. So here, instead of f_1, we define $g_2 = \sum_{n_j \in V_2 : \pi_j = p_2} w_j$, which is the total computation cost of all tasks on p_2. In an as-late-as-possible (without increasing the makespan) schedule, g_2 corresponds to the idle-time-free block before the execution of sink. This can be seen in an illustration of a schedule produced by FORKJOINSCHED-CASE2 in Figure 4.

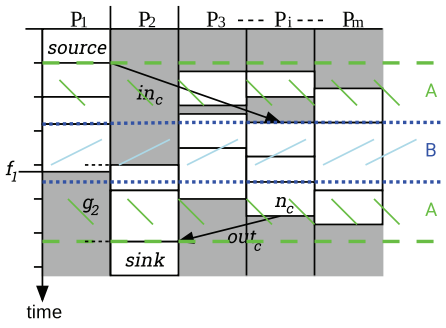

Fig. 4. Source on p_1, sink on p_2

After the initial schedule for iteration i, the migration to p_1 and p_2 (Algorithm 5) follows a similar approach as in the previous case. The while loop continues as long as a task can be migrated to either p_1 or p_2 without increasing the makespan. The if-statement then puts the critical task n_c on p_1 if its incoming communication is not less than its outgoing communication *or* if there is no space left on p_2. Otherwise it puts it on p_2. In all other aspects the migration behaves like in Algorithm 3.

Algorithm 5 MIGRATETOP1P2 Task migration to p_1 and p_2

```
1: while f_1 < σ_c ∨ g_2 < σ_c + out_c − in_c do
2:     if (in_c ≥ out_c ∨ g_2 ≥ σ_c + out_c − in_c) ∧ f_1 < σ_c
       then
3:         insert n_c on p_1, ordered
4:     else
5:         insert n_c on p_2, ordered
6:     REMOTESCHED({i ∈ V, π_i ≠ p_1, p_2}, {p_3 ... p_m})
7:     Determine new n_c
```

Remark. *For the special case with two processors, scheduling all the tasks on a single processor (including source and sink) is a 2-approximation (even when optimal schedules places source and sink on different processors).*

Straightforwardly, all optimal schedules have to complete the whole workload, and the best load balancing on two processors can only achieve half of the total workload on each processor. Placing all tasks on a single processor negates all communication costs, reducing the schedule to a continuous execution of tasks.

Theorem 1 (FORKJOINSCHED)**.** *Returning the best schedule obtained by* FORKJOINSCHED-CASE1 *and* FORKJOINSCHED-CASE2 *is a* $(1 + \frac{m}{m-1})$*-approximation algorithm for scheduling a fork-join graph on* m *processors.*

Proof. As we have shown in Lemma 2, if we assume that there exists an optimal schedule with sink and source on p_1, then FORKJOINSCHED-CASE1, provides a $(1 + \frac{m}{m-1})$-approximation solution. We now have to consider the case where all optimal schedules have source on p_1 and sink on p_2. Lets name i the task with the largest index on the remote processors in one such optimal schedule.

All tasks with indexes higher than i (and potentially some with indexes smaller than i) are thus scheduled in this optimal on either p_1 or p_2. The computational cost of these tasks is at least $\sum_{k>i} w_k$. As in the previous remark for the special case of two processors, putting all these tasks with source and sink on a single processor is a straightforward 2-approximation. If we now consider the solution for iteration i of FORKJOINSCHED-CASE1, we find that it is still a $(1 + \frac{m}{m-1})$ approximation even when all optimal schedules have source on p_1 and sink on p_2. Following Lemma 2's proof, the start time of $sink$ (i.e. the makespan) can be constrained by *(a)* the tasks on p_1 (i.e. $\sigma_{sink} = f_1$), or *(b)* the arrival of data from the tasks on the remote processors. We just showed that if *(a)* applies FORKJOINSCHED-CASE1 is a 2-approximation, and for *(b)* the arguments used in Lemma 2 still hold and prove that FORKJOINSCHED-CASE1 provides a $(1 + \frac{m}{m-1})$-approximation solution. □

It may come as a surprise to the reader that the FORKJOINSCHED-CASE2 is not directly used in the proof. Indeed, although this algorithm is a natural contender for best heuristic schedule, since we return the best schedule of both algorithms proving that one always provides a guaranteed solution is enough to prove the Theorem.

IV. OTHER HEURISTICS

Among the many proposed algorithms for task scheduling, e.g. [3], [8], [2], [6], list scheduling is a widely used heuristic for scheduling task graphs with communication delays that is also very competitive [7]. Hence, we use list scheduling as the base algorithm approach to compare FORKJOINSCHED against. Remember, this work studies *static* scheduling, consequentially we do not consider work-stealing algorithms, which are dynamic in nature and include task migration. Communication costs in that scheduling model usually refer to the synchronisation/protocol overhead. Instead, we here consider the time to communicate data, which is neglected in (early) work-stealing approaches and theoretical bounds [16]. As we are here targeting fork-join graphs only, we also propose specific variations of list scheduling algorithms that take the fork-join structure into account. In total we are presenting in the following seven list scheduling based algorithms.

A. List scheduling (LS)

A generic list scheduling algorithm is outlined in Algorithm 6. The algorithm consist of two parts. The first part of the algorithm is to sort the tasks based on a priority scheme and in topological order. The second part of the algorithm is to schedule the tasks on processors. A basic heuristic which the list scheduling employs is the earliest start time (EST) heuristic. The algorithm will iterate through each p_i of P to find which processor will give the earliest σ_i for task n_i while following the precedence constraints. The algorithm then schedules n_i on this processor, π_i. After all tasks have been scheduled, it will iterate through P to find the earliest possible start time for the sink. This is the schedule length of the schedule given by the list scheduling.

Algorithm 6 LS: Generic list scheduling

```
1: Sort nodes n ∈ V into List L with a priority scheme.
2: for n ∈ L do
3:     Schedule n on p ∈ P at EST of n.
```

This generic list scheduling algorithm will be employed in the evaluation with different priority schemes.

B. Priority schemes

Many different priority schemes have been proposed in the literature, usually based on task characteristics such as task weight, bottom-level, top-level, in-degree, out-degree etc. [8]. Given the specific nature of fork-join graphs, we propose to use three intuitive priority schemes (largest first).

- **CC:** $w_i + out_i$ - computation weight of task plus outgoing communication weight, i.e. bottom-level
- **CCC:** $in_i + w_i + out_i$ - computation weight of task plus outgoing communication weight, i.e. top-level plus bottom-level
- **C:** w_i - computation weight of task

We use these three priority schemes for LS and all the list scheduling variants discussed in the following.

C. List scheduling lookahead child (LS-LC)

Lookahead algorithms take into account the potential start time of a task's children for the scheduling decision [14]. In general this can be based on a specific child or multiple children. For fork-join graphs, every inner task or branch task has only one child, namely $sink$. Hence, we propose a lookahead, where the current task n_i under consideration is scheduled on the processor that results in the best potential start time for $sink$ This is outlined in Algorithm 7.

D. List scheduling lookahead neighbour (LS-LN)

For fork-join graphs the children lookahead approach is quite limited as $sink$ is the only child for all tasks. An alternative lookahead is to consider the start time of the current task's "neighbour" task. Neighbour task n_n is defined as being the next task to be scheduled according to the used priority scheme. The algorithm LS-LN is very similar to what is outlined in Algorithm 7 except now the chosen processor for the current task n_i to be scheduled on is the one that minimises the sum of its and the neighbours start times $\sigma_i + \sigma_n$.

Algorithm 7 LS-LC: List scheduling lookahead child

```
1: Sort nodes n_i ∈ V into List L with priority scheme.
2: for n_i ∈ L do
3:     for p_j ∈ P do
4:         Tentatively π_i ← p_j.
5:         Find best σ_sink on current partial schedule.
6:         If σ_sink is current min, p_min ← p_j
7:         Unset π_i to remove n_i from partial schedule.
8:     π_i ← p_min; σ_i ← EST on p_min.
```

E. List scheduling source and sink determined (LS-SS)

Section II-A observes that there are only two options for the scheduling of $source$ and $sink$ and establishes a convention for this. In the same way we use this in FORKJOINSCHED, we can also integrate this with list scheduling and predetermine the processors for $source$ and $sink$. With this information in place, the remaining nodes can be scheduled such that it will give the best estimate for the earliest start time of the $sink$. This algorithm is similar to the look ahead algorithm described previously. The difference is instead of allowing the sink node to be scheduled on any processor, it can only be scheduled on p_1 or p_2 depending on which case it is. This algorithm is outlined in Algorithm 8.

Algorithm 8 LS-SS: List scheduling source and sink fixed

```
1: Sort nodes n_i ∈ V into List L with priority scheme.
2: for π_sink ∈ {p_1, p_2} do
3:     π_source ← p_1
4:     for n_i ∈ L do
5:         for p_j ∈ P do
6:             Tentatively π_i ← p_j.
7:             Compute σ_sink on π_sink.
8:             If σ_sink is current min, p_min ← p_j
9:             Remove n_i from the partial schedule.
10:        π_i ← p_min; σ_i ← EST on p_min.
11:    Record schedule
12: return Better of two schedules
```

F. List scheduling – dynamic (LS-D)

Creating a static task list $\mathbf{L}$ in part 1 of the previous list scheduling algorithms is conceptually simple and has low runtime complexity. However, it can happen that these algorithms schedule a task in such a way that idle time is left between the current task and its preceding task on the same processor. Another unscheduled task (with lower priority) might be available that could have started earlier, which would reduce idle time. This situation is avoided by considering all free tasks (tasks whose parents have already been scheduled) and to choose the one that allows the earliest start time across all processors [5]. Sometimes the resulting task ordering is called a dynamic task priority. The outline of

the algorithm is shown in Algorithm 9. For fork-join graphs all tasks (except $sink$) are free once $source$ is scheduled and the earliest possible starting time is only constrained by in_i and the processor availability. Hence, the list scheduling with dynamic priority as of Algorithm 9 corresponds closely to the remote task scheduling of Algorithm 1, except that tasks are also scheduled on the processor where $sink$ is allocated.

Algorithm 9 LS-D: List scheduling – dynamic

$Q \leftarrow V$; $\sigma_{source} \leftarrow 0$; $\pi_{source} \leftarrow p_1$
while $Q \neq \varnothing$ **do**
    $n_i \leftarrow argmin_{n_j \in Q, p_i \in P}\{\sigma_j \text{ on } p_i\}$
    $Q \leftarrow Q \setminus n_i$
    Schedule n_i on processor with EST
Schedule $sink$ on EST processor

G. List scheduling - dynamic with variable priority (LS-DV)

In the specific case of fork-join graphs, the start time of the first tasks on each processor is constrained by the arrival of their incoming communication. Once a good number of tasks is already scheduled on a processor, the start time on that processor is likely constrained by the current finish time of that processor. Hence, it is not meaningful any more to prioritise the tasks by their incoming communication as LS-D essentially does. Recognising this Algorithm 10 changes the way it chooses a task for scheduling when the start time of the tasks is no longer constrained by incoming communication. Until then, it behaves like LS-D and then it changes and prioritises the task with the highest $w_i + out_i$, i.e. its bottom level (CC in Section IV-B).

Algorithm 10 LS-DV: List scheduling - dynamic with variable priority

$Q \leftarrow V$; $\sigma_{source} \leftarrow 0$; $\pi_{source} \leftarrow p_1$
while $Q \neq \varnothing$ **do**
    if σ's constrained by in **then**
        $n_i \leftarrow argmin_{n_j \in Q, p_i \in P}\{\sigma_j \text{ on } p_i\}$
        $Q \leftarrow Q \setminus n_i$
        Schedule n_i on processor with EST
    else
        $n_i \leftarrow argmax_{n_j \in Q}\{w_j + out_j\}$ {bottom-level}
        $Q \leftarrow Q \setminus n_i$
        Schedule n_i on processor with EST
Schedule $sink$ on EST processor

H. Algorithms complexity

Before the evaluation in the next section, a quick overview of the runtime complexity of the algorithms in Table I.

TABLE I
ALGORITHMS COMPLEXITY FOR FORK-JOIN GRAPH

Name	Summary	Complexity
LS	List scheduling	$\mathcal{O}(\lvert V\rvert(\log\lvert V\rvert + \log m))$
LS-D	LS – dynamic properties	$\mathcal{O}(\lvert V\rvert(\log\lvert V\rvert + \log m))$
LS-DV	LS – dynamic variable	$\mathcal{O}(\lvert V\rvert(\log\lvert V\rvert + m))$
LS-LC	LS – child lookahead	$\mathcal{O}(\lvert V\rvert(\log\lvert V\rvert + m^2))$
LS-LN	LS – neighbour LA	$\mathcal{O}(\lvert V\rvert(\log\lvert V\rvert + m\log m))$
LS-SS	LS – predet. procs for src./sink	$\mathcal{O}(\lvert V\rvert(\log\lvert V\rvert + m))$
FJS	FORKJOINSCHED	$\mathcal{O}(\lvert V\rvert^3 m)$

V. EVALUATION

To evaluate the proposed FORKJOINSCHED and the adapted list scheduling variants we implemented them and employed them to schedule a very large set of task graphs on different numbers of processors. The algorithms were implemented in Java and run on Linux machines with i7-4770 CPU and 16GB RAM. Reruns were done on other lab machines, however note that the platform has no influence on the produced schedules.

A. Task graph generation

Task graphs are generated by various properties. As the structure is fixed (fork-join), the parameters to vary are the number of tasks and the weights of the tasks and edges.

1) Number of tasks: The number of tasks ranged from tiny (4 tasks) to very large (10,000 tasks). The spacing between sizes also increased with size, e.g. 4-100: increment 1; 100-500: increment 10;...; 5000-10000: increment 500). In total there were 182 different task sizes.

2) Task weights: Task weights were randomly generated from three types of distributions, as shown in Figure 5, to better model real life programs from observations of real distributed systems [17], [18]. The uniform distribution shown by the blue line in Figure 5 models task weights that are generated with a range of values with equal probability [19], [20]. The dual Erlang distribution is shown by the orange line. This is to model task weights that are of the normal distribution without negative values [21], [22]. Lastly, the green line is the exponential Erlang distribution. This distribution is to model programs with many small tasks and other tasks which weights are modelled by the Erlang distribution [23], [24].

Five variations of the distribution are used to generate the task graphs as summarised in Table II. The range of the values is chosen as it has generally been found that there are both small tasks and large tasks present[21], [25]. Hence the range for uniform distribution and means for dual Erlang and exponential Erlang distributions are chosen to be at least one magnitude of difference.

TABLE II
DISTRIBUTIONS USED TO GENERATE TASK WEIGHTS

Distribution type	Properties
Uniform_1_1000	Uniform range from 1 to 1000
Uniform_10_100	Uniform range from 10 to 100
DualErlang_10_100	Erlang means are at 10 and 100
DualErlang_10_1000	Erlang means are at 10 and 1000
ExponentialErlang_1_1000	Decay start 1, Erlang mean 1000

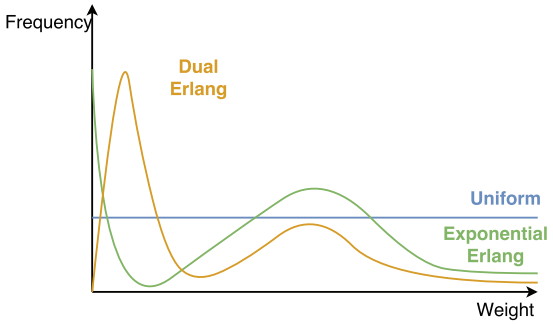

Fig. 5. Task weight distribution types

3) Edge weights: Edge weights are generated from a uniform distribution between 1 to 100. This is scaled to give the desired **communication to computation ratio (CCR)**, defined as the sum of all edge weights over the sum of all task weights. Four different CCR are chosen to model different types of programs [4], ranging from low to high communication: **CCR's : 0.1, 1, 2, 10**

B. Number of processors

The used number of target processors for the scheduling ranges from 3 processors to 512 processors. Starting at 4 processors the next number of processors is increased by a factor of two.

Three processors have been chosen to evaluate the theoretical worst guarantee of FORKJOINSCHED. For 512 processors, the guarantee approaches 2 times the optimal schedule length. Small processor numbers model computer hardware, whereas large processor numbers model distributed systems [26].

C. Lower bound

The comparison metric in the evaluation of the algorithms is schedule length. To make this metric comparable across different graphs and number of processors we normalise it. Given that the optimal length is unknown, we use a lower bound. The simplest would be the total task weights divided by the number of processors, but given the specific fork-join structure of the graphs we can easily improve this. Hence we used a lower bound calculation as proposed in [15] for fork-join graphs, where we also include the smallest incoming and outgoing communications that cannot be avoided when a certain number of processors are non-empty. Having that said, the relation between the different algorithms for a given graph is not influenced by the lower bound used, but only by the absolute values of schedule length.

VI. RESULTS AND DISCUSSIONS

A total of over 3500 task graphs were scheduled on nine different numbers of processors by all seven algorithms: FORKJOINSCHED, shortened as FJS, and the six algorithms proposed in Section IV. Four of those algorithms (LS, LS-D, LS-DV and LS-LC) were run with the three different priority schemes (CC, CCC, C) as defined in Section IV-B. All input graphs, the raw results and the generated charts for all results are provided on figshare [27].

A general trend that can be observed when plotting the normalised schedule length over the number of tasks for each algorithm is that the values start low, then quickly reach a peak where the number of tasks is around double the number of target processors and then slowly decline with increasing number of tasks. This can be observed in some form for all number of processors and all CCRs and can be seen in the scatter plots in the following. Having that said, for low numbers of processors (say 3,4,8), the peak coincides with the start of the plots and we can only observe the decline.

An intuitive explanation for this observation is that scheduling few tasks and many tasks (relative to the number of processors) is easier than scheduling a relative medium number of tasks. For very few tasks, many remote tasks can get their own processor, hence there is no decision to be made and akin to scheduling with an unlimited number of processors. For many tasks, the delay of the incoming communication is hidden for most tasks and the algorithms can concentrate on load balancing only.

A. Priority schemes

In order to reduce the number of algorithms to compare against, let us first have a look at the performance of the priority schemes for the four algorithms LS, LS-D, LS-DV and LS-LC.

While the trends are the same across all numbers of processors and CCRs, they are more pronounced for higher number of processors and higher CCRs and for illustration we will show plots with higher parameter values. Given the interesting observation that the results are qualitatively similar across different weight distribution approaches and for easier comparison we use the DualErlang_10_1000 distribution in the following to not vary too many parameters at the same time. The full set of results is available on figshare at the reference given at the start of the section.

List scheduling (LS) and list scheduling lookahead neighbour (LS-LN) both exhibit the same trend, exemplary shown in Figure 6 for LS-LN, 64 processors and CCR 2. The blue line, which is the CC bottom level, sorts the tasks in a order that allows the algorithm to generate the shortest schedule relative to the other bottom levels.

List scheduling with the source and sink determined and list scheduling lookahead children both exhibit a common trend, but differ from the previous two algorithms as, exemplary shown in Figure 7 for LS-SS, 512 processors and CCR 10. This is explained by both algorithms looking ahead to the sink task. The priority scheme CCC, green line, results in the shortest schedule length overall, even though the difference is small and CC, blue line, is lower for high numbers of tasks.

Overall, the CC priority performed the best overall and we only use that priority scheme in the following discussion.

B. Algorithm comparison

As already hinted at, the difference between the algorithms is strongly influenced by the CCR value and number of pro-

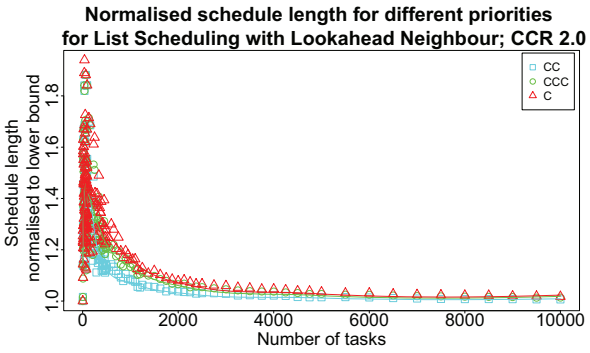

Fig. 6. Schedule length for different priorities for LS-LN, 64 procs, CCR 2

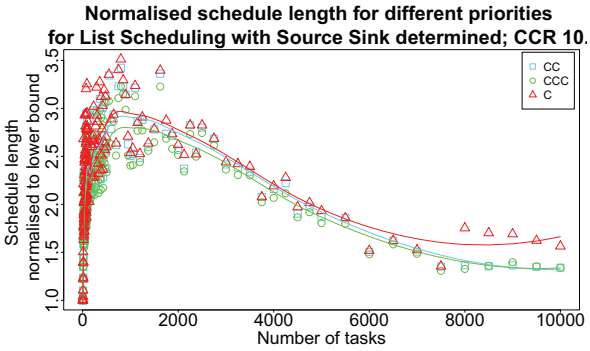

Fig. 7. Schedule length for different priorities for LS-SS, 512 procs, CCR 10

cessors. Higher values result in more significant differences. With this in mind, we start by looking at low CCRs and low numbers of processors and work up to the highest values.

1) Few processors: Figure 8 depicts boxplots of the schedule lengths for all algorithms for graphs with CCR 0.1 and Dual Erlang distribution on 3 processors. While there is some discernible difference between the algorithms, note that almost all schedules are within a very small percentage of the lower bound, so they are all very good and close to optimal.

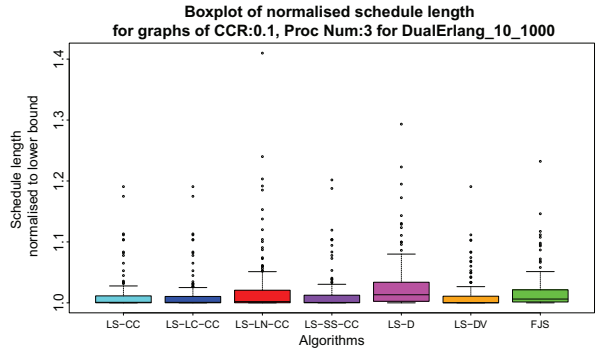

Fig. 8. Boxplot of schedule lengths for all algorithms, 3 procs, CCR 0.1

Increasing the CCR for the same number of processors makes some difference. Figure 9 shows the same boxplots as before, but now for a CCR of 10 instead of 0.1 in Figure 8. Both the absolute values have increased and the difference between the algorithms is more discernible. FJS now shows the best performance, but also the list scheduling algorithms with lookahead (LS-LN-CC and LS-SS-CC) perform well.

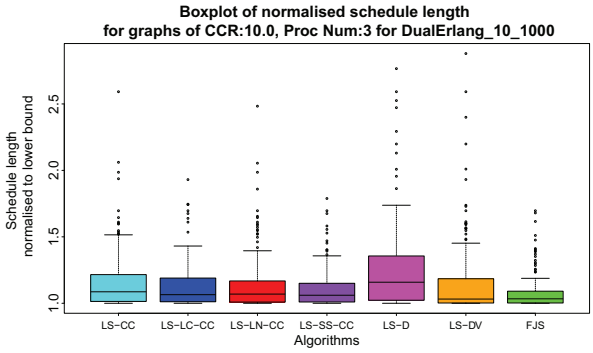

Fig. 9. Boxplot of schedule lengths for all algorithms, 3 procs, CCR 10

Looking at the corresponding scatterplot in Figure 10 shows that the difference mostly stems from the scheduling of graphs with very few tasks. For high task numbers, they behave very similarly.

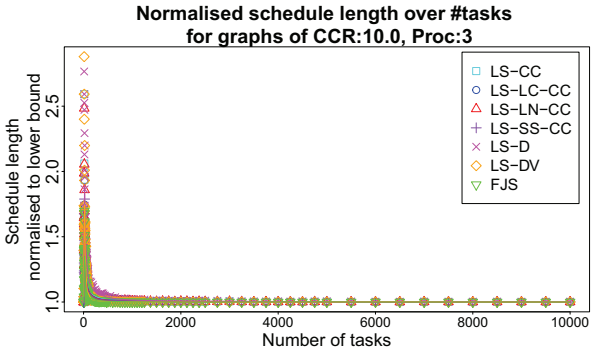

Fig. 10. Schedule lengths for all algorithms, 3 procs, CCR 10

2) Many processors: Let us now look at using a high number of processors, namely 512. Going back to a CCR of 0.1, Figure 11 shows the boxplots of the schedule length using Dual Erlang distribution. The observed performance is very similar to the case with 3 processors (Figure 8). It becomes clear that the two algorithms with dynamic priority (LS-D and LS-DV) are performing slightly worse than the other algorithms.

Looking at the corresponding scatterplot in Figure 12 shows that the "peak", as observed at the start of the discussion, is mild and only more pronounced for LS-D.

The tendencies we observed so far become clearer when scheduling graphs with high CCR of 10 with many processors. Figure 13 shows the corresponding boxplots. FJS is now setting itself apart with a lower average normalised schedule

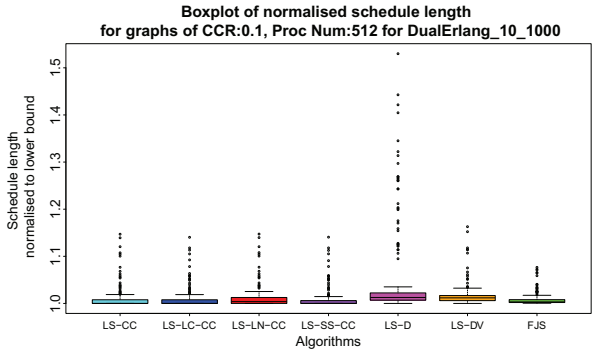

Fig. 11. Boxplot of schedule lengths for all algorithms, 512 procs, CCR 0.1

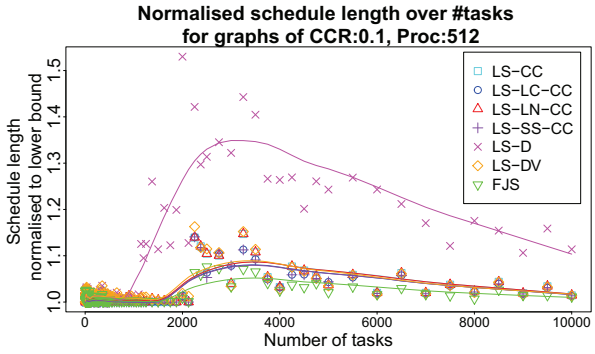

Fig. 12. Schedule lengths for all algorithms, 512 procs, CCR 0.1

length than the other algorithms. Again, the algorithms with dynamic priorities LS-D and LS-DV look worse than the other algorithms.

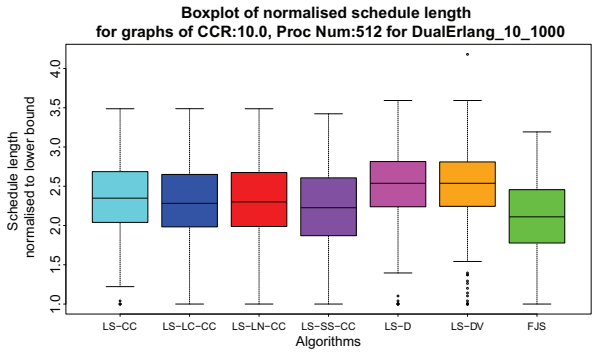

Fig. 13. Boxplot of schedule lengths for all algorithms, 512 procs, CCR 10

The corresponding scatterplot in Figure 14 now shows a pronounced peak for graphs between 500 to 1000 tasks. LS-D performs badly for low numbers of tasks, but is close to best for a high number, which is easily explained by its variable priority scheme. Nevertheless, simpler list scheduling approach perform better overall.

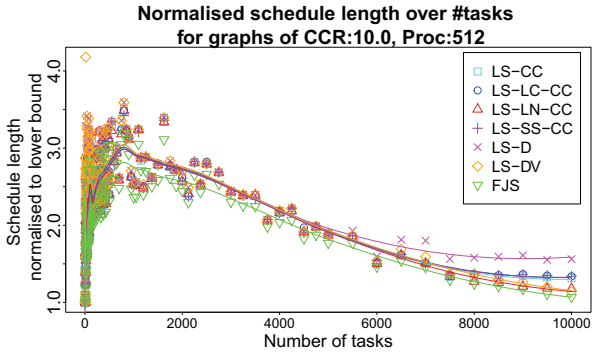

Fig. 14. Schedule lengths for all algorithms, 512 procs, CCR 10

C. Summary

The results presented in this section are representative examples for the qualitatively similar results for other weight distributions and processor numbers. We looked here at the smallest and largest number of processors and the results for other numbers of processors are somewhere in between.

We observe for high CCR that the normalised schedule length is significantly higher than for low CCR. Two explanations are possible: *(a)* The algorithms are not able to produce schedules close to optimal; or *(b)* the lower bound is not very tight for high CCR. It is likely that both are true, as even FJS with its guaranteed approximation factor of $1+\frac{m}{m-1}$ produces a few results with a normalised schedule length above 3.

While the performance of the algorithms overall is close, we can make some general observations. The weakest algorithms are LS-D and LS-DV with their dynamic priorities. The four list scheduling based approaches LS, LS-D, LS-DV and LS-LC perform quite similarly. Overall the strongest algorithm is the proposed FORKJOINSCHED (FJS), which is especially pleasing as it also gives a guarantee on the schedule length.

D. Runtime

The competitive performance and approximation guarantee of FORKJOINSCHED however come at the cost of significantly higher runtime. This should be clear from the complexities as discussed in Section IV-H. In practice, all the list scheduling based algorithms take at most a few seconds even for the largest problems in our Java implementations. In contrast, FORKJOINSCHED can take dozens of minutes or more for the large task graphs. Interestingly, the worst case is many tasks and very few processors (3 and 4). This can be explained by the migration of tasks after the initial schedule in each iteration, see Algorithm 2 and Algorithm 4. For a low number of processors there are many more beneficial migrations to p_1 (or p_1 and p_2) than for high number of processors. As a result there are many more rounds of rescheduling the remote tasks.

VII. CONCLUSION

In this paper we addressed the scheduling of task graphs with communication delay on homogeneous processors. Given

the importance of the fork-join graph structure and the lack of a known guaranteed approximation algorithm for the general case, we focused on the special case of scheduling arbitrary fork-join graphs with communication delay. For this problem, we proposed a novel $1+\frac{m}{m-1}$-approximation algorithm, which is also practical. To demonstrate this practicality, we adapted a set of list scheduling algorithms for the special case of scheduling fork-join graphs. In an extensive evaluation, we compared the algorithms with a large set of task graphs of different sizes and weight distributions and scheduled them on different numbers of processors. Our proposed FORKJOINSCHED algorithm demonstrated its competitiveness in comparison with all other algorithms, while giving a guarantee on the schedule length. In the future we will investigate if the approximation ratio can be further improved. Extending the algorithm to work with heterogeneous processors is also of strong interest.

REFERENCES

[1] E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," in *Logistics of Production and Inventory*, S. C. Graves, A. R. Kan, and P. Zipkin, Eds. North Holland, Amsterdam, 1993.

[2] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1989.

[3] M. I. Daoud and N. Kharma, "A hybrid heuristic genetic algorithm for task scheduling in heterogeneous processor networks," *Journal of Parallel and Distributed Computing*, no. 71, pp. 1518–1531, 2011.

[4] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[5] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE transactions on Parallel and Distributed systems*, vol. 4, no. 2, pp. 175–187, 1993.

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

[7] H. Wang and O. Sinnen, "List-scheduling vs. cluster-scheduling," *IEEE Transactions of Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1736–1749, 2018.

[8] M. Drozdowski, *Scheduling for Parallel Processing*. Springer, 2009.

[9] J. Dittrich and J.-A. Quiané-Ruiz, "Efficient big data processing in hadoop mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2014–2015, 2012.

[10] K. Jansen, O. Sinnen, and H. Wang, "An EPTAS for scheduling fork-join graphs with communication delay," *Theoretical Computer Science*, vol. 861, pp. 66–79, 2021.

[11] H. Wang and O. Sinnen, "Scheduling fork-join task graphs with communication delay and equal processing time," in *51st Int. Conference on Parallel Processing (ICPP 2022)*, Bordeaux, France, Aug. 2022.

[12] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović, "Towards the optimal solution of the multiprocessor scheduling problem with communication delays," in *In Proc. 3rd Multidisciplinary Int. Conf. on Scheduling: Theory and Application (MISTA)*, Aug. 2007, pp. 128–135.

[13] M. Orr and O. Sinnen, "Optimal task scheduling benefits from a duplicate-free state-space," *Journal of Parallel and Distributed Computing*, vol. 146, pp. 158–174, 2020.

[14] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, Feb 2010, pp. 27–34.

[15] S. Venugopalan and O. Sinnen, "Memory limited algorithms for optimal task scheduling on parallel systems," *Journal of Parallel and Distributed Computing*, vol. 92, pp. 35–49, 2016.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *PPoPP*. ACM, 1995, pp. 207–216.

[17] (2015) Parallel workloads archive: Metacentrum2. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/l_metacentrum2/index.html

[18] (2013) Parallel workloads archive: Intel netbatch. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/l_intel_netbatch/index.html

[19] Y. Gao, H. Rong, and J. Z. Huang, "Adaptive grid job scheduling with genetic algorithms," *Future Generation Computer Systems*, vol. 21, no. 1, pp. 151–161, 2005.

[20] M.-Y. Wu and D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, 1990.

[21] S. Ali, H. J. Siegel, M. Maheswaran, and D. Hensgen, "Task execution time modeling for heterogeneous computing systems," in *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*. IEEE, 2000, pp. 185–199.

[22] Y. A. Li and J. K. Antonio, "Estimating the execution time distribution for a task graph in a heterogeneous computing system," in *Heterogeneous Computing Workshop, 1997.(HCW'97) Proceedings., Sixth*. IEEE, 1997, pp. 172–184.

[23] D. G. Feitelson, "Packing schemes for gang scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1996, pp. 89–110.

[24] W. H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, and F. Zini, "Optorsim: A grid simulator for studying dynamic data replication strategies," *International Journal of High Performance Computing Applications*, vol. 17, no. 4, pp. 403–416, 2003.

[25] K. Aida, "Effect of job size characteristics on job scheduling performance," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2000, pp. 1–17.

[26] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit, "On advantages of grid computing for parallel job scheduling," in *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*. IEEE, 2002, pp. 39–39.

[27] P.-F. Dutot, Y.-S. Fu, N. Prasad, and O. Sinnen, "*Dataset of:* A Guaranteed Approximation Algorithm for Scheduling Fork-Joins with Communication Delay," Figshare, 10.6084/m9.figshare.21824934.v1, 2023. [Online]. Available: https://figshare.com/articles/dataset/A_Guaranteed_Approximation_Algorithm_for_Scheduling_Fork-Joins_with_Communication_Delay/21824934