An EPTAS for Scheduling Fork-Join Graphs with Communication Delay

Klaus Jansen[1]

Department of Computer Science, University of Kiel, 24118 Kiel, Germany

Oliver Sinnen

Department of Electrical, Computer, and Software Engineering, University of Auckland, Auckland 1010, New Zealand

Huijun Wang

Department of Electrical, Computer, and Software Engineering, University of Auckland, Auckland 1010, New Zealand

Abstract

This paper presents an EPTAS for scheduling fork-join task graphs with communication delay on homogeneous processors, denoted as $P|\text{fork-join}, c_{ij}|C_{max}$ in the $\alpha|\beta|\gamma$-notation. The fork-join structure is a basic structure found in many parallel computations. The algorithm uses an integer program as the feasibility test and searches for a solution which would be guaranteed to be within a $1+\epsilon$ factor of the optimum. It is shown that this runs in time exponential in terms of $1/\epsilon$ and polynomial in terms of the input size. Communication costs are dealt with effectively for this fork-join graph structure. The EPTAS is also adapted to scheduling independent tasks with release times and deadlines, which is denoted as $P|r_j|L_{max}$.

Keywords: Parallel Computing, Scheduling, EPTAS, Fork-Join, Approximation

[1]Supported in part by the DFG project JA 612 /20-1.

1. Introduction

In programming parallel systems, an optimisation problem is to schedule the computational workload, where units of work, called *tasks*, are each allocated to a processor and a time slot for execution, in such a way that optimises (i.e. minimises) the makespan of the schedule. There are data dependencies between the tasks, each one imposing a precedence constraint that must be satisfied by the schedule.

The structure of such a workload can be represented by a *task graph*, which is a directed acyclic graph (DAG) where the nodes represent the tasks and the edges represent their data communications. In addition to the processing times of tasks, there are data communication times, which are delays between the time when a sender finishes executing and when the receiver is ready to start, if they have to communicate between processors. Communication delays are spared between tasks on the same processor. This models systems where local communication between tasks on the same processor is negligible in comparison to remote communication.

This work focuses on single fork-join workloads, with a structure represented by a fork-join DAG, made up of many branch tasks receiving data from a common source, and sending data to a common sink (section 3). While fork-join graphs are structurally simple, they are an important DAG structure, both theoretically and practically. In theory they are the basic parallel sub-structure of series-parallel graphs and can represent master-slave types of computation. In essence all kinds of computations that can be split into independent tasks, but need data input for an initial task and then at the end the results need to be reduced (e.g. data is merged) to a single task. In practise they represent a significant subclass of parallel computations, e.g. MapReduce frameworks, like Apache Hadoop and Spark, or scatter/gather communication when using MPI parallelism. The modelled system has m identical processors, with unrestricted communications, meaning that communications can occur at the same time as each other and with the execution of tasks as well.

Finding a valid schedule with optimal makespan for such fork-join graphs is strongly NP-hard, because it is equivalent to scheduling independent tasks ($P||C_{max}$, which is strongly NP-hard [5]) if communication times are set to zero, as well as the processing times of the source and sink tasks.

An efficient polynomial time approximation scheme (EPTAS) for this problem is presented in this paper. An EPTAS is an algorithm which, for

an accuracy parameter $\epsilon > 0$, gives a solution that is within a $1 + \epsilon$ factor of the optimum in polynomial time in terms of the input size n. An EPTAS has complexity $\mathcal{O}(f(1/\epsilon) \times \text{poly}(n))$, rather than $\mathcal{O}(n^{f(1/\epsilon)})$ for a PTAS. The algorithm uses dual approximation [12] and a configuration ILP [8] (section 6), with techniques to make this work for communication times.

2. Related Work

For scheduling tasks with precedence constraints on limited processors, the problem denoted as $P|prec|C_{max}$, there cannot be a PTAS, because a lower bound of 4/3 for the best possible ratio of any polynomial time approximation algorithm has been proven in [20]. This is also true for the harder problem $P|prec, c_{ij}|C_{max}$ which includes communication costs. Even with unit processing and communication times, as in the problem denoted $P|p_j = 1, c_{ij} = 1|C_{max}$, there is a minimum polynomial time approximation ratio of 5/4 [13]. There are other theoretical results and approximations for more specific problems under $P|prec, c_{ij}|C_{max}$ [18, 14], especially for constant processing and communication times [1, 23, 21, 22, 10, 17, 11], and special DAG structures [24, 19, 3, 6], but no PTAS for these.

Scheduling a fork-join DAG has less to do with precedence constraints than the other problems under $P|prec, c_{ij}|C_{max}$, because most tasks in the fork-join structure are independent of each other. It is closer to scheduling independent tasks with release times and deadlines, as a feasibility problem of scheduling under time T. There has been some work done on such problems [2, 4, 9] (but no EPTAS that we know of), and they are not directly applicable to fork-join scheduling, which has to additionally consider the allocation of the source and sink (section 8).

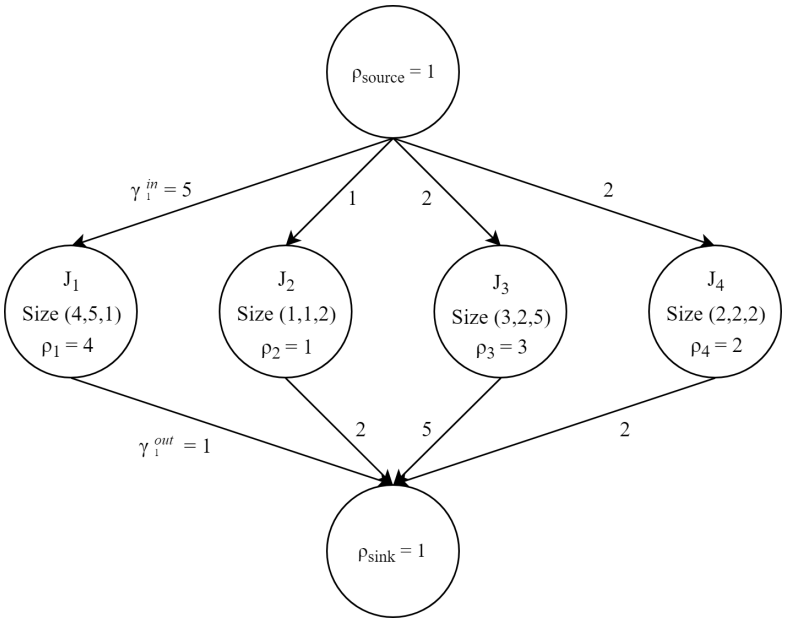
Configuration integer programs (section 6) are commonly used in the design of approximation algorithms [8, 15], and one of its first uses was in bin packing [7]. We also employ the dual approximation framework studied in [12], to search for an optimum makespan with a series of guesses, instead of solving for the solution directly.

3. Problem Definition

A DAG with fork-join structure is to be scheduled on m identical processors (machines), minimising the makespan. In this DAG there is a source task j_{source}, a sink task j_{sink}, and a set of branch tasks J. Each branch

task $j \in J$ has an associated size vector $(\rho_j, \gamma_j^{in}, \gamma_j^{out}) \in P \times \Gamma \times \Gamma$, which gives the time cost of processing, time cost of incoming communication from the source, and of outgoing communication to the sink, respectively. An incoming or outgoing communication cost is only incurred if the task is on a different processor from the source or sink task, respectively, otherwise it is 0. $P \subset \mathbb{N}_0$ is the set of all processing times, and $\Gamma \subset \mathbb{N}_0$ is the set of all communication times. Figure 1 shows an example fork-join DAG.

Figure 1: Fork-join DAG



A schedule consists of processor allocations $\pi : J \to [m]$ and start time allocations $\sigma : J \to \mathbb{N}_0$ for all tasks. The start time allocations are equivalent to a total order $\leq_\sigma$ of execution over each set of tasks scheduled to the same processor, because optimally, tasks start as early as possible given their order. Therefore, for $i, j \in J$ scheduled consecutively on the same processor (meaning $i \leq_\sigma j$ and $\nexists\, k \in J \colon i \leq_\sigma k \leq_\sigma j$), the start time of j is

$$\sigma(j) = \begin{cases} \max(\sigma(i) + \rho_i,\, \gamma_j^{in}), & \text{if } \pi(j) \neq \pi(j_{source}) \\ \sigma(i) + \rho_i, & \text{if } \pi(j) = \pi(j_{source}) \end{cases}$$

For $j = \min \leq_\sigma$, the first branch task on a processor,

$$\sigma(j) = \begin{cases} \gamma_j^{in}, & \text{if } \pi(j) \neq \pi(j_{source}) \\ 0, & \text{if } \pi(j) = \pi(j_{source}) \end{cases}$$

The objective is to find a schedule with optimal makespan. For this objective, without loss of generality, the processing times of the source and sink tasks can be set to 0, $\rho_{source} = \rho_{sink} = 0$. The makespan is then given by the start time of the sink:

$$\max_{j \in J} \left(\sigma(j) + \rho_j + \begin{cases} \gamma_j^{out}, & \text{if } \pi(j) \neq \pi(j_{sink}) \\ 0, & \text{if } \pi(j) = \pi(j_{sink}) \end{cases} \right)$$

Figure 2: Schedule of fork-join DAG

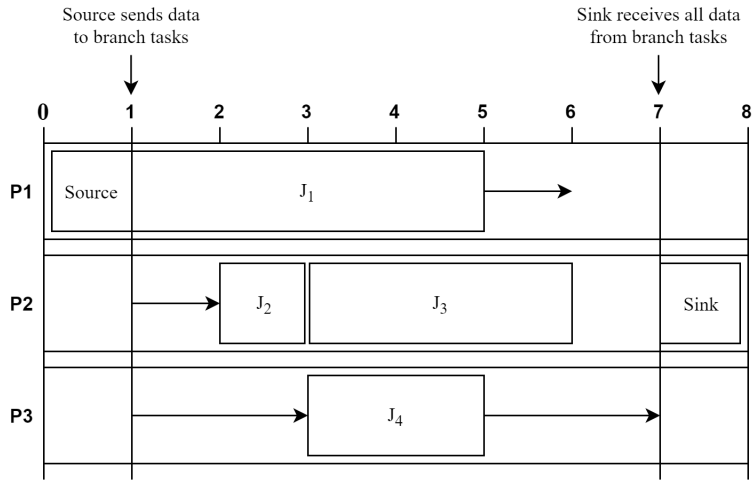


Figure 2 shows a schedule of the DAG in figure 1. The arrows represent communication delays. In this schedule, the source task executes on $P1$, so tasks on $P1$ have no incoming communication delays, and the sink task executes on $P2$, so tasks on $P2$ have no outgoing communication delays. Communications can occur at the same time as each other so only the latest communication delays will delay the schedule on each processor. Branch tasks not scheduled on the same processor with either the source or sink are said to be *remote*.

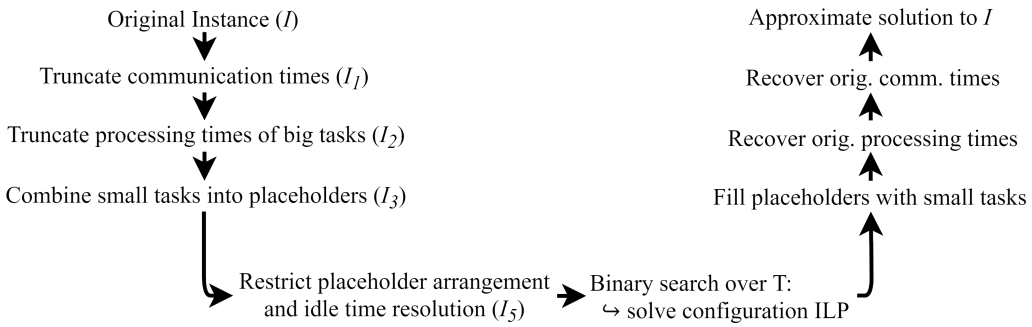
4. Algorithm overview

As stated, finding a valid schedule with optimal makespan for the fork-join graphs defined above is strongly NP-hard (section 1). We are therefore proposing an EPTAS using the following strategy.

At the top-level, a binary search over different time bounds T is conducted and for each T we test, using an ILP, whether a valid schedule with a makespan $\leq T$ exists (the technique used in [12]). The range for T is finite, as an upper bound for the optimal makespan is given by the fully sequential schedule, i.e. the sum of all processing times, and a lower bound is the sum of all processing times divided by the number of processors (i.e. perfect parallelism).

For each T, we set up an ILP in $\mathcal{O}(|J|)$ time to test if a feasible schedule exists. To control the runtime complexity, we are using a configuration ILP [7] with a fixed set of task size vectors normalised relative to T. The input to the ILP are the numbers of tasks of each normalised size vector at the scale set by T. To construct such an input that corresponds to the original input instance, we need to simplify and transform the original instance in various steps. Figure 3 illustrates our approach.

Figure 3: Overview of the scheme



The next section explains the simplification steps and analyses their impact on the makespan quality.

5. Simplification of the instance

The following simplification steps reduce task size vectors to a constant set, regardless of the input size, so that the size of the integer program is

bounded and constant. This is done by rounding (truncating) large tasks into standard values, and introducing placeholder tasks for small tasks (where rounding becomes unsuitable).

The variable ϵ (where $\epsilon \in \mathcal{R}$, $0 < \epsilon < 1$, and we can assume without loss of generality that $\frac{1}{\epsilon} \in \mathcal{N}$) decides the resolution of the schedule including task size vectors and start times, and the accuracy of the approximation. Tasks with processing times less than $\epsilon^2 T$ are said to be **small tasks**, and the rest are **big tasks**. The subsets of small and big tasks are denoted J_{small} and J_{big} respectively. Tasks are sometimes also categorised by communication times. The set of tasks with given communication times $(\gamma^{in}, \gamma^{out}) \in \Gamma^2$ is denoted $J^{\gamma^{in},\gamma^{out}}$.

Each of the following sections describes one simplification. Let I be the original instance, and OPT be its optimal makespan. The makespan T will be referenced throughout the definitions. It refers to the optimal makespan for the fully simplified instance (I_5), on which the integer program is applied, hence it is the parameter of the binary search and decides the inputs to the ILP. An L-schedule denotes a schedule with makespan at most L.

5.1. Communication Times

First, we limit the cardinality of Γ, by truncating all communication times to the nearest multiple of ϵT, including 0. Let I_1 be the instance obtained after this communication times truncation step.

Lemma 1 (Communication truncation)**.** *Let T_1 be the optimal makespan for I_1. Then*

$$T_1 \leq OPT \leq T_1 + 2\epsilon T$$

Proof. A T_1-schedule for I_1 is transformed into a $(T_1 + 2\epsilon T)$-schedule for I by recovering the truncated communication times, which extends each communication time by no more than ϵT.

Extending incoming communication times delays all branch tasks by no more than ϵT. Extending the outgoing communication times delays the sink by no more than ϵT. Together, these extend the schedule by at most $2\epsilon T$. A T_1-schedule for I_1 therefore guarantees a $(T_1 + 2\epsilon T)$-schedule for I. □

Now, the number of different communication costs is $|\Gamma| = \frac{1}{\epsilon}$.

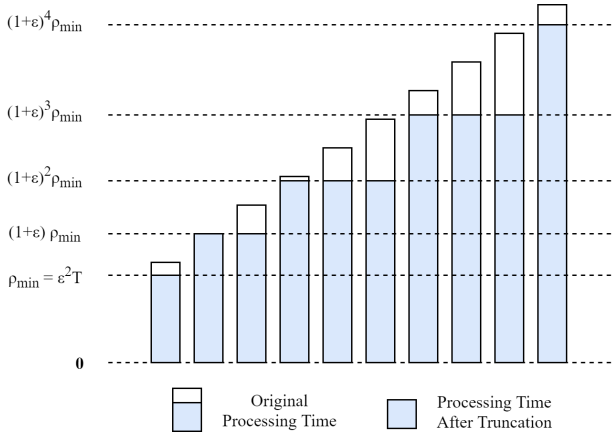
5.2. Big Tasks

Next, the processing times of big tasks are truncated to a nearest normalised value, from the following set, which decreases them by no more than a $1 + \epsilon$ factor.

$$\{(1 + \epsilon)^n \epsilon^2 T \mid n \in \mathbb{N}_0\} \tag{1}$$

To illustrate this, Figure 4 shows some processing times truncated to the nearest order of magnitude. Let I_2 be the instance obtained by applying this task time truncation step to I_1.

Figure 4: Truncation of big task processing times



Lemma 2 (Big task truncation)**.** *Let T_2 be the optimal makespan for I_2. Then*

$$T_2 \leq T_1 \leq T_2 + \epsilon T$$

Proof. A T_2-schedule for I_2 is transformed into a $(1 + \epsilon)T_2$-schedule for I_1 by recovering the truncated processing times. Let T_{big} be the total processing time of all big tasks. A processing time is at most truncated from a value approaching $(1+\epsilon)^{n+1}\epsilon^2 T$, to $(1+\epsilon)^n \epsilon^2 T$, where it was reduced by a factor of at most $1+\epsilon$. Returned to original values, these processing times are increased by no more than a factor of $1 + \epsilon$, and the entire schedule is increased by no more than ϵT_{big}. T_{big} is constant in all schedules for any instance, and $T_{big} \leq T$ where T is the optimal makespan for the final instance.

Therefore, a T_2-schedule for I_2 can guarantee a $(T_2 + \epsilon T)$-schedule for I_1. □

5.3. Small Tasks

For the rounding strategy in the previous step, each processing time can only be rounded to within some factor of itself. This would continue indefinitely into the smaller processing times (infinitesimally small compared to T) and cannot be used to bound the number of processing times. The number of processing times must be bound in another way, in this case by approximating the small tasks as uniform placeholder tasks. The task set J_{small} for each communication size vector $(\gamma^{in}, \gamma^{out}) \in \Gamma^2$, denoted by $J_{small}^{\gamma^{in},\gamma^{out}}$, is replaced by a set of placeholder tasks $\Theta^{\gamma^{in},\gamma^{out}}$, each of uniform processing time $\epsilon^3 T$, which covers almost the same total processing time. That is, all small tasks are removed, then, for all $(\gamma^{in}, \gamma^{out}) \in \Gamma^2$, the following (rounded down) number of placeholder tasks with size vector $(\epsilon^3 T, \gamma^{in}, \gamma^{out})$ are added:

$$|\Theta^{\gamma^{in},\gamma^{out}}| = \left\lfloor \frac{\sum_{j \in J_{small}^{\gamma^{in},\gamma^{out}}} \rho_j}{\epsilon^3 T} \right\rfloor$$

Let I_3 be the instance obtained after replacing small tasks in I_2 with placeholder tasks. A schedule for I_3, consisting of placeholder tasks instead of small tasks, can be transformed back into a schedule for I_2 by packing small tasks back into the spaces (time slots) occupied by their placeholder tasks, using Algorithm 1.

In Algorithm 1, each small task in $J_{small}^{\gamma^{in},\gamma^{out}}$ is packed into the corresponding placeholder spaces occupied by $\Theta^{\gamma^{in},\gamma^{out}}$, and this is done for all communication size vectors $\gamma^{in}, \gamma^{out}$. We arrange the placeholder tasks of the same γ^{in} in non-increasing order of γ^{out} (i.e. larger γ^{out} start earlier), because for any schedule, placeholder tasks may be rearranged this way without increasing the makespan. Hence the algorithm is applied in that order of γ^{out} to $\{J_{small}^{\gamma^{in},\gamma^{out}} \mid \gamma^{out} \in \Gamma\}$, for each $\gamma^{in} \in \Gamma$.

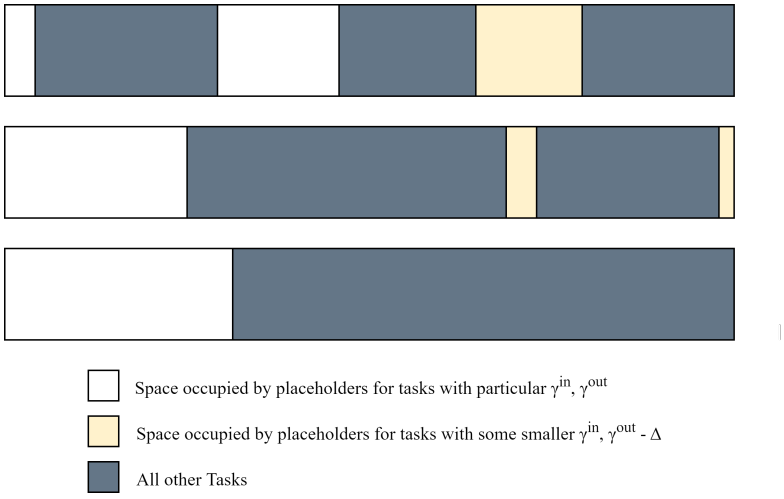
The packing uses a *next-fit* (next-cover) approach, which tries to fill each space until it is exceeded (overpacked). When a task from $J_{small}^{\gamma^{in},\gamma^{out}}$ exceeds a space, the rest of the schedule is shifted (pushed forward) to avoid overlaps. The amount by which the space is overpacked is subtracted from the next space on the same machine belonging to placeholders for $J_{small}^{\gamma^{in},\gamma^{out}}$ or for some $J_{small}^{\gamma^{in},\gamma^{out}-\Delta}$ (Δ would be a multiple of ϵT). If a space reduces to zero, we move to the next space, and continue until the amount of overpacking has been absorbed. If there are no more spaces to absorb this overpacking, the processing time of the schedule on this processor is extended, and this over-

packing is at most the processing time of an entire placeholder. If we run out of tasks of a given set $J_{small}^{\gamma^{in},\gamma^{out}}$ while filling some space, the remaining space is given to tasks in the next set in order $J_{small}^{\gamma^{in},\gamma^{out}-\Delta}$. (The amount of space subtracted from that set is no more than the amount given back to it.)

Generally, spaces for one communication size vector may not fit (by not leaving enough time for communication for) tasks of another communication size vector. However, looking at tasks with the same incoming communication, the spaces left unfilled by some $J_{small}^{\gamma^{in},\gamma^{out}}$ can become spaces for $J_{small}^{\gamma^{in},\gamma^{out}-\Delta}$. The shorter outgoing communication $\gamma^{out} - \Delta$ guarantees that no constraint is violated.

Figure 5 shows a schedule with some placeholder tasks. Figure 6 shows the same schedule with small tasks packed into placeholder spaces, starting with the larger outgoing communication size. Spaces for the smaller size may be reduced (top processor) or closed completely (middle processor), but are recouped with unused spaces (bottom processor). (It is also possible that all spaces are used when the packing is close to perfect.)

Figure 5: Schedule with placeholder tasks



Lemma 3 (Small task placeholder tasks)**.** *Let T_3 be the optimal makespan for I_3. Then*

$$T_3 - \epsilon T \leq T_2 \leq T_3 + 2\epsilon T$$

Proof. Given a schedule for I_3 we can construct a valid schedule for I_2 using Algorithm 1. It can at most extend the schedule by $\epsilon^2 T$ for each $\gamma^{in} \in \Gamma$

Figure 6: Schedule with small tasks recovered

where $|\Gamma| = \frac{1}{\epsilon}$, and in total ϵT, because the algorithm ensures that the overpacking of spaces for a given incoming communication size γ^{in} does not extend the schedule by more than the processing of a small task, which is $\epsilon^2 T$.

Additionally, the numbers of placeholder tasks were rounded down, ignoring up to $\epsilon^3 T$ processing time for each $J_{small}^{\gamma^{in},\gamma^{out}}$. Recovering them extends the schedule by no more than $\epsilon^3 T \cdot |\Gamma^2| = \epsilon T$. Therefore, a T_3-schedule for I_3 guarantees a $(T_3 + 2\epsilon T)$-schedule for I_2, giving

$$T_2 \leq T_3 + 2\epsilon T$$

On the other hand, any L-schedule for instance I_2, before the simplification, can be transformed into (thus guaranteeing) an $(L + \epsilon T)$-schedule for I_3. This can be done by using next-fit again to pack the placeholder tasks for each communication size vector $(\gamma^{in}, \gamma^{out}) \in \Gamma^2$ into the spaces occupied by the small tasks $J_{small}^{\gamma^{in},\gamma^{out}}$ which they will replace. In an optimal schedule for I_2, the small tasks cannot be assumed to start in any particular order with respect to communication times (unlike with placeholder tasks in I_3), due to their different processing times. The indeterminate order would cause conflicts if using a process like Algorithm 1. Instead, the next-fit and push-forward procedure found in Algorithm 1 is applied independently for each $J_{small}^{\gamma^{in},\gamma^{out}}$, extending the schedule by no more than the processing time of a

Algorithm 1: Packing Tasks into Spaces

```
for γ^in ∈ Γ in any order do
    Arrange placeholder tasks Θ^γin in decreasing γ^out order. Treat
     them as spaces for their respective small tasks (with the same
     γ^out).
    for γ^out ∈ Γ in decreasing value do
        for θ ∈ Θ^γin,γout in non-decreasing start order do
            Fill θ with unscheduled tasks in J_small^γin,γout until θ is full or
             exceeded
            if space θ is exceeded then
                t ← length exceeded beyond θ
                for each task (inc. placeholder tasks ∉ Θ^γin) or space
                 after θ on θ's processor, in increasing start time do
                    if task then
                        delay this task by t
                    if space then
                        Shorten and delay this space by t
                        t ← t − length of space
                        if t ≤ 0 then
                            break
        for each θ ∈ Θ^γin,γout not completely filled do
            θ becomes space for J_small^γin,γout−Δ, with the next smaller
             outgoing communication
```

placeholder task, $\epsilon^3 T$, for each $(\gamma^{in}, \gamma^{out}) \in \Gamma^2$, and in total ϵT. This means that $T_3 \leq T_2 + \epsilon T$, and

$$T_3 - \epsilon T \leq T_2$$

In other words, if T_3 is the optimal makespan for I_3, there cannot exist a schedule for I_2 with makespan $< T_3 - \epsilon T$, which would otherwise guarantee a smaller makespan for I_3 than T_3. □

5.4. Placeholder task Arrangement

In a further simplified instance I_4, placeholder tasks for each incoming communication size, denoted $\Theta^{\gamma^{in}}_{small}$ $\forall \gamma^{in}$ are forced to only exist in groups of $\frac{1}{\epsilon}$ such placeholder tasks (with total processing time $\epsilon^2 T$), or in a stub, which means a group of $\leq \frac{1}{\epsilon}$ placeholder tasks. At most $\frac{1}{\epsilon}$ stubs can exist

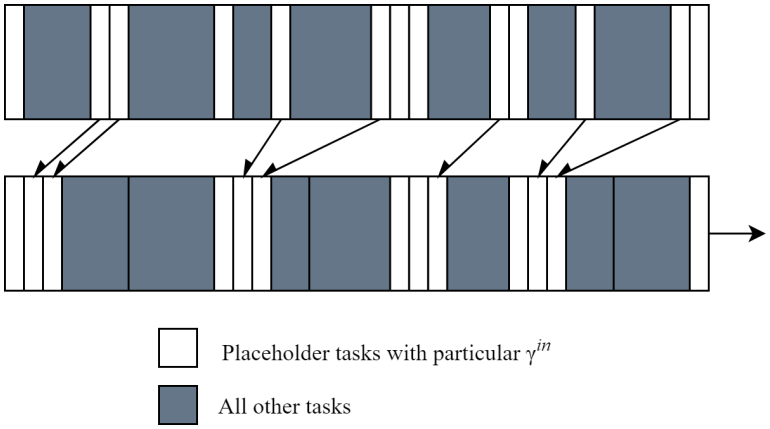
on one processor, as the stub for each incoming communication size can be reduced to one.

Algorithm 2 can do this placeholder task arrangement for a schedule where they are not grouped, by shifting placeholder tasks to earlier start times and delaying tasks between the old and new start times as needed. Figure 7 shows an example schedule on a processor before and after the transformation, for $\frac{1}{\epsilon} = 3$. The right arrow indicates delayed communications in the transformed schedule, as tasks were shifted to the right.

Algorithm 2: Shifting placeholder tasks into blocks

for $\gamma^{in} \in \Gamma$ **do**
  for *each placeholder task* $j \in \Theta^{\gamma^{in}}$*, in non-decreasing start time* **do**
    if *there is an unfinished block (with* $< \frac{1}{\epsilon}$ *placeholder tasks) before* j **then**
      Start j at the end of that block
      Delay all tasks (including other placeholder tasks) between j's old and new start times by $\epsilon^3 T$
    else
      j *is the start of a block*

Figure 7: Shifting placeholder tasks into blocks, here $1/\epsilon = 3$



In the eventual integer program, a group of placeholder tasks is scheduled as a single task. This reduces the variety of schedules and the size of the integer program needed to find them. Let I_4 be the instance of the problem with this grouping restriction to the solution.

Lemma 4 (Placeholder task arrangement)**.** *Let T_4 be the optimal makespan for I_4.*

$$T_4 - \epsilon T \leq T_3 \leq T_4$$

Proof. A schedule for I_3, where tasks are freely arranged, can be transformed into a schedule for I_4, where placeholder tasks are grouped together (into *blocks*) using Algorithm 2.

A schedule is transformed without shifting more than $\frac{1}{\epsilon}$ placeholder tasks past any one task, which does not delay the schedule by more than $\epsilon^2 T$ (due to delayed communications), for each $\Theta_{small}^{\gamma^{in}}$ $\forall \gamma^{in} \in \Gamma$, and in total ϵT.

After this process, each $\Theta_{small}^{\gamma^{in}}$ $\forall \gamma^{in} \in \Gamma$ could have left one stub, or $|\Gamma| = \frac{1}{\epsilon}$ stubs in total, which complies with the restriction on I_4.

Therefore, any L-schedule for I_3 guarantees an $(L + \epsilon T)$-schedule for I_4, and if T_4 is the optimal makespan for I_4, then the optimal makespan for I_3 cannot be less than $T_4 - \epsilon T$.

Lastly, T_4 is no smaller than T_3 as placeholder tasks in I_3 can be freely arranged, hence they can be in the same order as in T_4 if that is optimal. □

5.5. Gap Sizes

As with task sizes, we limit the resolution of gaps in the schedule. *Gaps* are idle times on a processor, before a branch task waiting for its communication to arrive. Their lengths are restricted to multiples of $\epsilon^2 T$. Let I_5 be the instance where idle time gaps are restricted in this way.

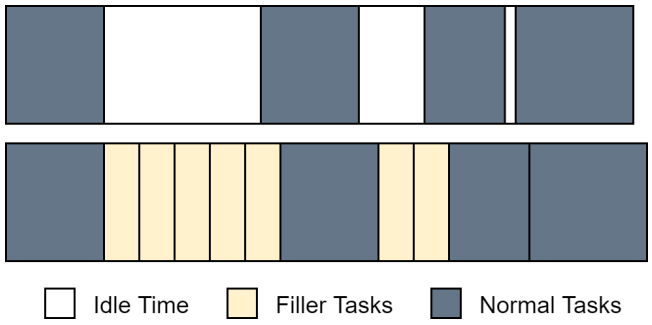
Lemma 5 (Gap sizes)**.** *Let T_5 be the optimal makespan for I_5.*

$$T_5 - \epsilon^2 T \leq T_4 \leq T_5$$

Proof. A schedule for I_4 can be transformed into a schedule for I_5, by adjusting the size of each of its gaps into a multiple of $\epsilon^2 T$, the processing time of a placeholder block, which is the smallest effective task (that can occur repeatedly, which excludes stubs).

This can be done by packing empty spaces of size $\epsilon^2 T$ (which can be thought of as filler tasks of size vector $(\epsilon^2 T, 0, 0)$, that are additional to the instance), into the existing gaps between tasks, filling each gap until exceeded (next-fit), delaying tasks scheduled after them as needed. Figure 8 shows this transformation. The schedule on top has gaps of any length, and the schedule

Figure 8: Representing empty spaces as filler tasks



beneath has the same gaps represented by filler tasks, or closed completely due to delays.

This way, the schedule is extended by no more than $\epsilon^2 T$, so an L-schedule for I_4 guarantees an $(L + \epsilon^2 T)$-schedule for I_5, and if T is the optimal makespan for I_5, then there cannot be a schedule for I_4 with makespan less than $T - \epsilon^2 T$.

Lastly, T_5 is no smaller than T_4 as all tasks are in the same order, and no task in T_4 starts later than in T_5.

□

I_5 is the instance for which the integer program can find an optimal solution. The smallest bound T found by the ILP is equal to its optimal makespan T_5.

6. Formulation of the Integer Linear Program

The integer linear program is formulated for instance I_5, consisting of big tasks and placeholder tasks, with restrictions on their placements which are needed in some of the following. (It is applied to the same inputs as I_3 but is only optimal for I_5).

The constructed ILP is a configuration ILP and a schedule is constructed by selecting processor configurations. Let $A = P \times \Gamma \times \Gamma$ be the set of all task size vectors after it has been bounded by the simplifications. A configuration C describes a set of tasks (multiset of task size vectors $(\rho, \gamma^{in}, \gamma^{out}) \in A$, defined $C : A \to \mathbb{N}_0$) to be scheduled onto a processor, and be the only tasks scheduled there. (For example, for a set A with three task size vectors a, b, c, a configuration $(1, 0, 2)$ would mean that one task of size vector a, none of

size vector b, and two of size vector c go onto a processor, which will execute these tasks only.) The tasks in a valid configuration must be able to execute on one processor in some sequence, as well as finish communicating, under the time bound T. The set of all needed configurations is denoted by $\boldsymbol{C}$.

Variables $x_C \ \forall C \in \boldsymbol{C}$ select the number of each configuration used. Hence, a solution to this ILP is a selection of configurations, one for each processor, which together can schedule all tasks in the given instance, respecting time bound T.

The configuration ILP that is constructed has a matrix that is constant (dimensions and elements) for each given ϵ. In other words, the constraints are constant, as well as the number of variables. Only numeric values of the input vector have increasing size $\mathcal{O}(|J|)$. In addition, the ILP is unaffected by the size of elements in J. This makes it small in terms of the input size.

Next we formulate the constraints for the ILP and then we discuss how to obtain the configurations.

6.1. Formulating Constraints

Let us first discuss task sizes. A task in a configuration can be thought of as a slot which fits a task of that size vector, or in fact a size vector with smaller communication times. A slot with size vector $(\rho, \gamma^{in}, \gamma^{out})$ can fit any task with size vector $(\rho, \gamma^{in}_{-}, \gamma^{out}_{-})$, where $\gamma^{in} \geq \gamma^{in}_{-} \wedge \gamma^{out} \geq \gamma^{out}_{-}$. This kind of generalisation is applied to communication times but not processing times, because fitting smaller communication times into slots does not change the configuration, unlike with processing times, where more tasks would fit into the configuration if processing times were reduced.

To make use of this, slots can be passed down to smaller size vectors, and only configurations with the largest possible slots are needed. Let $N_{\rho,\gamma^{in},\gamma^{out}}$ be the total number of tasks of size vector $(\rho, \gamma^{in}, \gamma^{out})$ in the instance, and let $C_{\rho,\gamma^{in},\gamma^{out}}$ be the number of slots of that size vector in configuration C. Normally, without reusing slots, we would have

$$\sum_{C \in \boldsymbol{C}} x_C C_{\rho,\gamma^{in},\gamma^{out}} \geq N_{\rho,\gamma^{in},\gamma^{out}} \quad \forall(\rho, \gamma^{in}, \gamma^{out})$$

which requires the number of slots to be enough for the given tasks at each individual size vector. Now, taking slot reuse into account, we can have the following constraint, where S denotes the number of slots passed on to smaller-communication size vectors, and S_+ denotes the number of slots passed down in the same way from larger-communication size vectors.

$$\sum_{C \in \boldsymbol{C}} x_C C_{\rho,\gamma^{in},\gamma^{out}} + S_+ - S \geq N_{\rho,\gamma^{in},\gamma^{out}}$$

To implement this, some variables are introduced. S_+ is replaced with $S^{>in}_{\rho,\gamma^{in}+\Delta,\gamma^{out}} + S^{>out}_{\rho,\gamma^{in},\gamma^{out}+\Delta}$, where $S^{>in}_{\rho,\gamma^{in}+\Delta,\gamma^{out}}$ is the number of slots passed down from the task size vector with the next bigger incoming communication, and $S^{>out}_{\rho,\gamma^{in},\gamma^{out}+\Delta}$ is the number of slots passed down from the size vector with next bigger outgoing communication. S is replaced with $S^{>in}_{\rho,\gamma^{in},\gamma^{out}} + S^{>out}_{\rho,\gamma^{in},\gamma^{out}}$, where $S^{>in}_{\rho,\gamma^{in},\gamma^{out}}$ is the number of slots to pass down to the size vector with the next smaller incoming communication, and $S^{>out}_{\rho,\gamma^{in},\gamma^{out}}$ is the number of slots to pass down to the size vector with the next smaller outgoing communication. The constraint becomes

$$\begin{aligned}\sum_{C \in \boldsymbol{C}} x_C C_{\rho,\gamma^{in},\gamma^{out}} + S^{>in}_{\rho,\gamma^{in}+\Delta,\gamma^{out}} + S^{>out}_{\rho,\gamma^{in},\gamma^{out}+\Delta} \\ - S^{>in}_{\rho,\gamma^{in},\gamma^{out}} - S^{>out}_{\rho,\gamma^{in},\gamma^{out}} \geq N_{\rho,\gamma^{in},\gamma^{out}} \\ \forall (\rho, \gamma^{in}, \gamma^{out}) \in A \end{aligned} \tag{2}$$

Figure 9 visualises the S variables. Each grid cell is a task communication size vector, and the arrows represent the directions in which slots are passed on, from large to small size vectors.

Only one each of the source and sink tasks can be included in all configurations. Let $C_{source} = 1$ if C contains the source, and likewise for the sink.

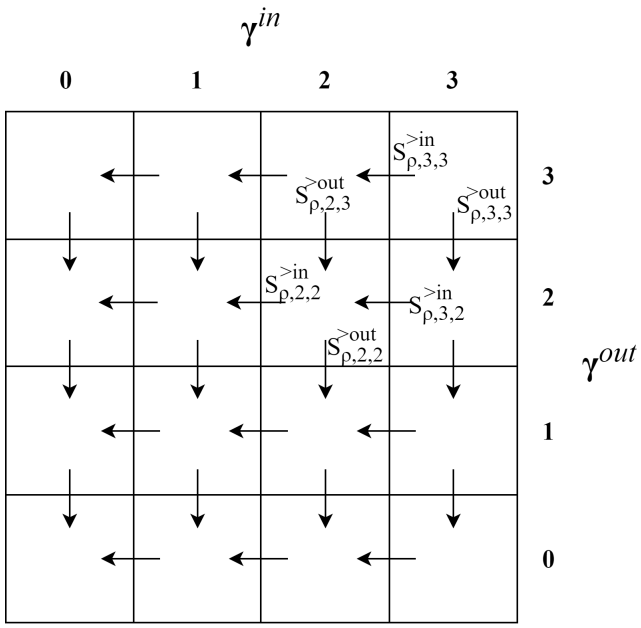
$$\sum_{C \in \boldsymbol{C}} x_C C_{source} = 1 \tag{3}$$

$$\sum_{C \in \boldsymbol{C}} x_C C_{sink} = 1 \tag{4}$$

The total number of configurations selected is equal to the number of processors.

$$\sum_{C \in \boldsymbol{C}} x_C = m \tag{5}$$

Figure 9: Transfer of slots from large to small size vectors



To summarise, the inputs (new for each T) are:

$$N_{\rho,\gamma^{in},\gamma^{out}} \quad \forall(\rho,\gamma^{in},\gamma^{out}) \in A$$

the variables are:

$$x_C \quad \forall C \in \boldsymbol{C}$$
$$S^{>in}_{\rho,\gamma^{in},\gamma^{out}},\ S^{>out}_{\rho,\gamma^{in},\gamma^{out}} \quad \forall(\rho,\gamma^{in},\gamma^{out}) \in A$$

and the constraints are the equations (2)-(5). These establish the ILP.

6.2. Obtaining the Configurations

Ignore the source and sink for a moment, and consider only configurations $C \in \boldsymbol{C}_{remote}$ of branch tasks scheduled remotely. Only maximal configurations (not subsets of other ones) are needed, as extra spaces provided by a configuration can be unused. In addition, configurations which can form other valid configurations by increasing the communication costs of its tasks, are not needed. This is because they can be represented by more general configurations where every task size vector is already maximised (in terms of

communication time), if the ILP is formulated in a way that allows each slot in a configuration to fit any smaller sized task (in terms of communication time), as was mentioned in the previous section. We formalise this as:

$$\boldsymbol{C}_{remote} = \{C \in \mathbb{N}_0^A \mid Valid(C) \wedge Max(C)\}$$
$$Max(C) \implies \forall x \colon C \cup \{x\} \notin \boldsymbol{C}_{remote}$$
$$Max(C) \implies \forall (\rho, \gamma) \colon C \setminus \{(\rho, \gamma)\} \cup \{(\rho, \gamma_+)\} \notin \boldsymbol{C}_{remote}$$
$$\text{where } \gamma = \gamma^{in}, \gamma^{out} \text{ and } \gamma_+^{in} > \gamma^{in} \vee \gamma_+^{out} > \gamma^{out}$$

Once $\boldsymbol{C}_{remote}$ is obtained, replicating $\boldsymbol{C}_{remote}$ with the source, sink, or both added to each configuration (with corresponding communication time changes) will give the rest of the configurations.

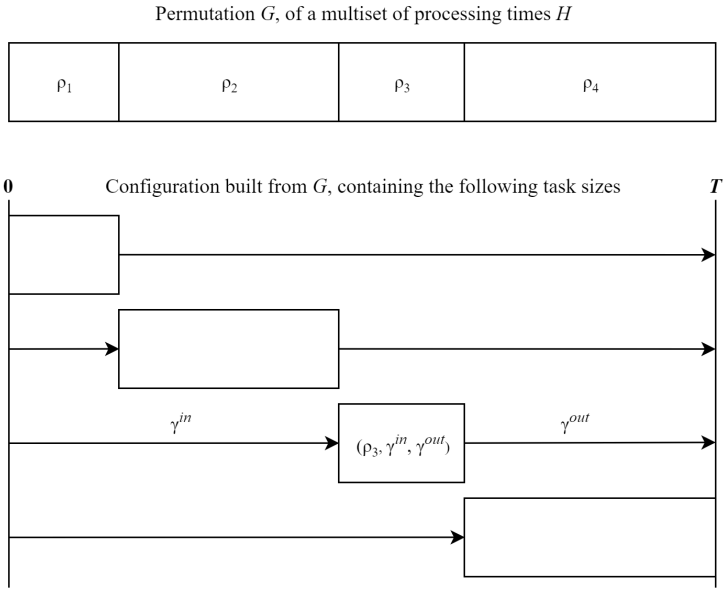
Each $C \in \boldsymbol{C}$ is encoded in the matrix of the ILP as a vector of multiplicities indexed by task sizes, where $\boldsymbol{C}$ contains the configurations needed to represent every possible schedule, of task size vectors in A. As A is constant and finite (section 5), so are their configurations $\boldsymbol{C}$.

$\boldsymbol{C}$ can be built with the following procedure. First, consider task processing times only, and ignore all communication costs. Let H be a (maximal) subset of branch tasks (including blocks of placeholder tasks, and containing at most $\frac{1}{\epsilon}$ stub blocks) that can fit onto one processor, which means having sum of processing times $\leq T$. $\boldsymbol{H}$ is the set of all possible maximal subsets H. For each $H \in \boldsymbol{H}$, add all its permutations G to set $\boldsymbol{G}$. Each $G \in \boldsymbol{G}$ implies start times for its tasks, assuming all communications arrive in time meaning that $\sigma(j) = \sigma(i) + \rho_i$ for $i \leq_G j$, and $\sigma(\min \leq_G) = 0$. Then, for every $G \in \boldsymbol{G}$, add $C = \{(\rho_j, \left\lfloor \frac{\sigma(j)}{\epsilon T} \right\rfloor \cdot \epsilon T, \left\lfloor \frac{T - \sigma(j) - \rho_j}{\epsilon T} \right\rfloor \cdot \epsilon T) \mid j \in G\}$ to $\boldsymbol{C}_{remote}$ (note that communication values are truncated to match those obtained after simplifications) In other words, each ordering becomes a configuration after assigning the maximum allowable communication times to each task slot given its position in the order and the resulting start time, as shown in figure 10.

All configurations $\boldsymbol{C}_{remote}$ created this way are part of the final set, $\boldsymbol{C} \supset \boldsymbol{C}_{remote}$. In addition, for every $C \in \boldsymbol{C}_{remote}$:

$$\{j_{source}\} \cup \{(\rho, \gamma_{max}, \gamma^{out}) \mid (\rho, \gamma^{in}, \gamma^{out}) \in C\} \in \boldsymbol{C}$$
$$\{j_{sink}\} \cup \{(\rho, \gamma^{in}, \gamma_{max}) \mid (\rho, \gamma^{in}, \gamma^{out}) \in C\} \in \boldsymbol{C}$$
$$\{j_{source}, j_{sink}\} \cup \{(\rho, \gamma_{max}, \gamma_{max}) \mid (\rho, \gamma^{in}, \gamma^{out}) \in C\} \in \boldsymbol{C}$$

Figure 10: Obtaining a configuration



These are configurations where the incoming communication, outgoing communication, or both, are not incurred because the source, sink, or both, have been scheduled to them. Therefore, they can fit tasks with any value of the corresponding communication time(s).

Lemma 6. *There exists $C \in \boldsymbol{C}$ to represent any possible schedule on one processor*

Proof. Let K be any set of tasks that can be scheduled on a processor without violating the time bound T, and K' be the same set of tasks with communication times ignored. The sum of processing times of these tasks is $\leq T$, so $\exists H \in \boldsymbol{H} : K' \subseteq H$. There is some permutation $G \in \boldsymbol{G}$ of this H that implies the same start times for those tasks in K as they would be viably scheduled. Lastly, the $C \in \boldsymbol{C}$ that is created from G has the maximum possible communication times for all of its tasks, including those which correspond to the ones in K, and when treated as slots, they will fit all corresponding tasks in K.

□

Lemma 7. *The number of configurations $|\boldsymbol{C}| = 2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)}$*

Proof. Let P_{big} be the set of simplified processing times for big tasks, including a block of $\frac{1}{\epsilon}$ placeholder tasks. Let P_{stub} be the set of processing times for stub blocks of $< \frac{1}{\epsilon}$ placeholder tasks. For the truncated processing times of the big tasks, equation (1), it holds $(1+\epsilon)^n \epsilon^2 T \leq T$, from which follows $n \leq \frac{2}{\log(1+\epsilon)} \log \frac{1}{\epsilon}$, hence $|P_{big}| = \mathcal{O}(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ for small $\epsilon < 1$, and $|P_{stub}| = \mathcal{O}(\frac{1}{\epsilon})$.

A set $H \in \boldsymbol{H}$ can be described as multisets of processing times $H_{big} : P_{big} \to [\frac{1}{\epsilon^2}]$ and $H_{stub} : P_{stub} \to [\frac{1}{\epsilon}]$, where $\frac{1}{\epsilon^2}$ is the most number of any $\rho \in P_{big}$ ($\min P_{big} = \epsilon^2 T$) a processor can execute, and $\frac{1}{\epsilon}$ the most number of stubs allowed on a processor. $|\boldsymbol{H}| = \mathcal{O}((\frac{1}{\epsilon^2})^{|P_{big}|} \times (\frac{1}{\epsilon})^{|P_{stub}|}) = 2^{\mathcal{O}(|P_{big}| \log \frac{1}{\epsilon^2} + |P_{stub}| \log \frac{1}{\epsilon})} = 2^{\mathcal{O}(1/\epsilon \log^2 1/\epsilon)}$.

There are $\mathcal{O}(\frac{1}{\epsilon^2})$ individual tasks in each $H \in \boldsymbol{H}$, of which there are $\mathcal{O}(\frac{1}{\epsilon^2})! = \mathcal{O}((1/\epsilon^2)^{1/\epsilon^2}) = 2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)}$ permutations. Together, these create $|\boldsymbol{C}_{remote}| = |\boldsymbol{G}| = |\boldsymbol{H}| \times 2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)} = 2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)}$, and $|\boldsymbol{C}| = 4 \cdot |\boldsymbol{C}_{remote}|$ is of the same order.

□

7. Result

An EPTAS can be formed using the processes (Figure 3) described above, which has the following bounds and complexity.

From the lemmas 1 to 5:

$$(1 - 2\epsilon - \epsilon^2)T \leq OPT \leq (1 + 5\epsilon)T$$

In other words, the optimum makespan is no shorter than $(1 - 2\epsilon - \epsilon^2)T$, while the final makespan obtained by the EPTAS is no longer than $(1+5\epsilon)T$. Therefore, the schedule obtained by the EPTAS has makespan no longer than

$$\frac{(1 + 5\epsilon)}{(1 - 2\epsilon - \epsilon^2)} OPT$$

Let $N = |J|$ be the number of tasks in the input. Simplifying the instance and obtaining inputs for the ILP takes $\mathcal{O}(N)$ time. Using a binary search to find T, the EPTAS has running time in the form $(\text{ILP} + \mathcal{O}(N)) \cdot \log(N)$.

The ILP has a number of constraints l given by the number of task size vectors $\mathcal{O}(\frac{1}{\epsilon^3} \log \frac{1}{\epsilon})$, a number of variables n dominated by the number of configurations $2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)}$, and elements in the matrix no bigger than $\mathcal{O}(\frac{1}{\epsilon^2})$

We make use of the result in [16], which has an algorithm that solves an ILP in time

$$O(H)^m \cdot \log(\Delta) \cdot \log(\Delta + \|b\|_\infty) + \mathcal{O}(nl)$$

where H is bounded by the maximum of the 1-norm of the columns, which corresponds to $\mathcal{O}(1/\epsilon^3)$ (the most number of tasks possible in a configuration), $\|b\|_\infty = \mathcal{O}(N \cdot \frac{1}{\epsilon})$ is the largest element in the input, which is bounded by the maximum number of placeholder tasks, and $\Delta = \mathcal{O}(\frac{1}{\epsilon^2})$ is the largest element in the matrix. This has a low cost relative to the number of variables, which is the largest dimension in this configuration ILP. Substituting the values, the ILP is solved in time

$$\begin{aligned}
&\mathcal{O}(1/\epsilon^3)^{(1/\epsilon^3 \log 1/\epsilon)} \cdot \log 1/\epsilon^2 \cdot \log(1/\epsilon^2 + N/\epsilon) \\
&\qquad + 2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)} \cdot \mathcal{O}(1/\epsilon^3 \log 1/\epsilon) \\
&= 2^{\mathcal{O}(1/\epsilon^3 \log^2 1/\epsilon)} \cdot (\log 1/\epsilon^2 + \log(1 + N\epsilon)) + 2^{\mathcal{O}(1/\epsilon^2 \log 1/\epsilon)} \\
&= 2^{\mathcal{O}(1/\epsilon^3 \log^2 1/\epsilon)} \cdot \mathcal{O}(\log(N) + \log(\epsilon)) \\
&= 2^{\mathcal{O}(1/\epsilon^3 \log^2 1/\epsilon)} \mathcal{O}(\log N)
\end{aligned}$$

Theorem 8. *The EPTAS finds a schedule with makespan no longer than:*

$$\frac{(1 + 5\epsilon)}{(1 - 2\epsilon - \epsilon^2)} OPT$$

in time:

$$2^{\mathcal{O}(1/\epsilon^3 \log^2 1/\epsilon)} \mathcal{O}(\log^2 N) + \mathcal{O}(N \log N)$$

This time complexity can be improved with the use of an efficient α-approximation for this problem, if one exists, and we believe it likely does (a result could be forthcoming). With the use of an α-approximation, a range $[\ell, \alpha\ell]$ can first be obtained, where $OPT \in [\ell, \alpha\ell]$. Then, apply simplifications to the processing and communication times of the original instance, using ℓ as the guessed makespan, to obtain a simplified instance I_0 with only $\mathcal{O}(1/\epsilon^3 \log 1/\epsilon)$ task size vectors (section 6.2). After this, all additional instances (for different guessed makespans) are obtained by simplifying from I_0 in $\mathcal{O}(1/\epsilon^3 \log 1/\epsilon)$ time. In addition, only a $\mathcal{O}(1/\epsilon)$ number of values in $[\ell, \alpha\ell]$

(in increments of $\mathcal{O}(\epsilon) \cdot \ell$) need to be considered, for the purpose of having an EPTAS (the approximation ratio will still be $1 + c\epsilon$ for some constant c). With these methods, the time complexity of the EPTAS will be given by the following, where α-APPROX is the time complexity of the α-approximation.

$$\begin{aligned}&(\text{ILP} + \mathcal{O}(1/\epsilon^3 \log 1/\epsilon)) \cdot \mathcal{O}(\log 1/\epsilon) + \mathcal{O}(N) + \alpha\text{-APPROX} \\ &= 2^{\mathcal{O}(1/\epsilon^3 \log^2 1/\epsilon)} \mathcal{O}(\log N) + \mathcal{O}(N) + \alpha\text{-APPROX}\end{aligned}$$

7.1. Scheduling Fork-Graphs and Join-Graphs

The EPTAS can be applied in the proposed form to a fork graph (or join graph, which reverses to become a fork-graph), by assigning 0 outgoing communication cost to all branch tasks. This results in a smaller set of task size vectors $P \times \Gamma \times \{0\}$, with cardinality reduced by a $\frac{1}{\epsilon}$ factor compared to the set of task size vectors for the fork-join problem. This would reduce the order of the number of constraints by a $\frac{1}{\epsilon}$ factor, resulting in a reduced total time complexity of:

$$2^{\mathcal{O}(1/\epsilon^2 \log^2 1/\epsilon)} \mathcal{O}(\log^2 N) + \mathcal{O}(N \log N)$$

Although the spare slot variables (e.g. $S^{>in}_{\rho,\gamma}$) could actually be eliminated entirely in this situation, it would not reduce the order of the total number of variables, which is dominated by the number of configurations and is not reduced.

8. The Release-Times and Deadlines Scheduling Problem

A similar problem to scheduling fork-join graphs is scheduling tasks with release times and deadlines (denoted $P|r_j|L_{max}$ in the $\alpha|\beta|\gamma$-notation), as described in the following. A set J of independent tasks are to be scheduled on m processors. Each task j has a release time r_j, which is the earliest time it may begin executing, and a deadline d_j before which it must finish. This is equivalent to scheduling a fork-join graph under time T (where T can be set to the latest deadline $\max_{j \in J} d_j$), with all communication delays incurred, so that a release time is effectively an incoming communication delay, and a deadline is effectively time T minus an outgoing communication delay. The objective is to minimise the maximum lateness of any task over its deadline, $L = \max_{j \in J}(\sigma(j) + \rho_j - d_j)$, which is equivalent to finding a

minimum schedule with length $T + L_{OPT}$ which gives a minimum lateness of L_{OPT}.

Therefore, the EPTAS can be adapted to approximate the minimum L to within a range, in the following way involving only a few modifications. Let an instance of the problem be called U, which requires a set J of independent tasks with release-times and deadlines to be scheduled onto m processors.

To begin with, create a fork-join graph with branch task j' for each $j \in J$, with $\gamma_{j'}^{in} = r_j$ and $\gamma_{j'}^{out} = T - d_j$, where T is the latest deadline, $\max_{j \in J} d_j$. Next, schedule this fork-join graph under a minimum amount of time $T + L$ on m processors, using the EPTAS. A modification must be made to the scheduling rules that all communication costs are incurred (meaning that effectively all tasks are scheduled remotely from the source and sink tasks). To implement this, remove all configurations containing the source or sink tasks, and assume they have been scheduled (i.e. remove the constraints $\sum_{\boldsymbol{C}} x_C C_{source} = 1$, $\sum_{\boldsymbol{C}} x_C C_{sink} = 1$). Let this adapted fork-join scheduling problem be called U'.

Let a T-schedule for U' denote a schedule with length at most T, and let an L-schedule for U denote a schedule with at most L lateness.

Lemma 9. *A $(T + L)$-schedule for U' implies an L-schedule for U, (vice versa) an L-schedule for U implies a $(T + L)$-schedule for U'.*

Proof. Given a $(T + L)$-schedule for U', create an equivalent schedule for U where each $j \in J$ in problem U is scheduled at the same time and on the same processor as its equivalent task j' in problem U'.

Task j' can finish as late as $T + L - \gamma_{j'}^{in}$ without extending its $(T + L)$-schedule. Therefore, task j will also finish no later than $T + L - \gamma_{j'}^{in}$, which would exceed its deadline, $T - \gamma_{j'}^{in}$, by no more than L.

In this way, an L-schedule for U can be obtained from a $(T + L)$-schedule for U'. The opposite result can be obtained in the reverse of this process. □

Therefore, an L-schedule for U exists if and only if a $(T + L)$-schedule exists for U'.

Lemma 10. *An approximate solution obtained by the EPTAS for U', and a guaranteed range for the optimal solution*

$$(1 - a\epsilon)(T + L) \leq OPT \leq (1 + b\epsilon)(T + L)$$

(where a, b are constants, from section 7), transform into an approximate solution for U, and a guaranteed range for the optimum lateness

$$L - a\epsilon(T + L) \leq L_{OPT} \leq L + b\epsilon(T + L)$$

Proof. Any $(1+b\epsilon)(T+L)$-schedule given by the EPTAS guarantees a schedule for U with lateness at most $(1 + b\epsilon)(T + L) - T = L + b\epsilon(T + L)$ (by lemma 9), which explains the upper bound.

If the EPTAS reports a lower bound of $(1 - a\epsilon)(T + L)$ for U', a schedule for U with lateness less than $L - a\epsilon(T + L)$ would guarantee a schedule for U' shorter than $T + L - a\epsilon(T + L) = (1 - a\epsilon)(T + L)$ (by lemma 9), which would be impossible, by the correctness of the EPTAS. □

9. Conclusion

This paper presents an EPTAS for scheduling fork-join task graphs with communication costs. At the centre of this is a procedure which, given a time bound T, either returns a schedule with makespan no longer than $(1 + b\epsilon)T$, or correctly determines that there is no schedule with makespan shorter than $(1 - a\epsilon)T$, where $a > 0$ and $b > 0$ are constants. Using this test, a binary search is carried out to reach a range for the optimum, and return a solution within it. The EPTAS can be used on a fork or join DAG individually with reduced time complexity. It also transfers to the problem of scheduling independent tasks with release times and deadlines, minimising the lateness.

References

[1] E. Bampis, A. Giannakos, and J.C. König. "On the complexity of scheduling with large communication delays". In: *European Journal of Operational Research* 94.2 (1996), pp. 252–260.

[2] M. Cieliebak, T. Erlebach, F. Hennecke, B. Weber, and P. Widmayer. "Scheduling With Release Times and Deadlines on A Minimum Number of Machines". In: *Exploring New Frontiers of Theoretical Informatics*. Ed. by J.J. Levy, E. W. Mayr, and J. C. Mitchell. Boston, MA: Springer US, 2004, pp. 209–222.

[3] J. Du, J.Y.-T. Leung, and G.H. Young. "Scheduling chain-structured tasks to minimize makespan and mean flow time". In: *Information and Computation* 92.2 (1991), pp. 219–236.

[4] N. Fisher and S. K. Baruah. "A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines". In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05)* (2005), pp. 117–126.

[5] M.R. Garey and D.S. Johnson. "Strong NP-completeness results: motivation, examples, and implications". In: *Journal of the Association for Computing Machinery* 25.3 (1978), pp. 499–508.

[6] M.R. Garey, D.S. Johnson, R.E. Tarjan, and M. Yannakakis. "Scheduling opposing forests". In: *SIAM Journal on Algebraic Discrete Methods* 4.1 (1983), pp. 72–93.

[7] P. C. Gilmore and R. E. Gomory. "A Linear Programming Approach to the Cutting-Stock Problem". In: *Operations Research* 9.6 (Dec. 1961), pp. 849–859.

[8] M. X. Goemans and T. Rothvoß. "Polynomiality for Bin Packing with a Constant Number of Item Types". In: *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '14. Portland, Oregon: Society for Industrial and Applied Mathematics, 2014, pp. 830–839.

[9] L. A. Hall and D. B. Shmoys. "Approximation schemes for constrained scheduling problems". In: *30th Annual Symposium on Foundations of Computer Science*. Oct. 1989, pp. 134–139.

[10] C. Hanen and A. Munier. "An approximation algorithm for scheduling dependent tasks on m processors with small communication delays". In: *Proceedings 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation. ETFA'95*. Vol. 1. Oct. 1995, 167–189 vol.1.

[11] C. Hanen and A. Munier. "Performance of Coffman-Graham schedules in the presence of unit communication delays". In: *Discrete applied mathematics* 81.1-3 (1998), pp. 93–108.

[12] D. S. Hochbaum and D. B. Shmoys. "Using Dual Approximation Algorithms for Scheduling Problems Theoretical and Practical Results". In: *Journal of the ACM* 34.1 (Jan. 1987), pp. 144–162.

[13] J. A. Hoogeveen, J. K. Lenstra, and B. Veltman. "Three, four, five, six, or the complexity of scheduling with communication delays". In: *Operations Research Letters* 16.3 (1994), pp. 129–137.

[14] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. "Scheduling precedence graphs in systems with interprocessor communication times". In: *SIAM Journal on Computing* 18.2 (1989), pp. 244–257.

[15] K. Jansen, K.M. Klein, and J. Verschae. "Closing the Gap for Makespan Scheduling via Sparsification Techniques". In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Vol. 55. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 72:1–72:13.

[16] K. Jansen and L. Rohwedder. *On Integer Programming, Discrepancy, and Convolution*. 2018. arXiv: `1803.04744 [cs.DS]`.

[17] S. Lam and R. Sethi. "Worst case analysis of two scheduling algorithms". In: *SIAM Journal on Computing* 6.3 (1977), pp. 518–536.

[18] E. L. Lawler, J. K. Lenstra, A.H.G. Rinnooy Kan, and D. B. Shmoys. "Sequencing and scheduling: Algorithms and complexity". In: *Handbooks in operations research and management science* 4 (1993), pp. 445–522.

[19] J. K. Lenstra, M. Veldhorst, and B. Veltman. "The complexity of scheduling trees with communication delays". In: *Journal of Algorithms* 20.1 (1996), pp. 157–173.

[20] J.K. Lenstra and A.H.G. Rinnooy Kan. "Complexity of scheduling under precedence constraints". In: *Operations Research* 26.1 (1978), pp. 22–35.

[21] A. Munier. "Approximation algorithms for scheduling trees with general communication delays". In: *Parallel Computing* 25.1 (Jan. 1999), pp. 41–48.

[22] A. Munier and C. Hanen. "Using duplication for scheduling unitary tasks on m processors with unit communication delays". In: *Theoretical Computer Science* 178.1-2 (1997), pp. 119–127.

[23] V. J Rayward-Smith. "UET scheduling with unit interprocessor communication delays". In: *Discrete Applied Mathematics* 18.1 (1987), pp. 55–71.

[24] T. A. Varvarigou, V. P. Roychowdhury, T. Kallath, and E. Lawler. "Scheduling in and out forests in the presence of communication delays". In: *IEEE Transactions on Parallel and Distributed Systems* 7.10 (1996), pp. 1065–1074.