Efficient Parallel Scheduling of Malleable Tasks

Peter Sanders
Department of Informatics
Karlsruher Institut für Technologie
Karlsruhe, Germany
sanders@kit.edu

Jochen Speck
Department of Informatics
Karlsruher Institut für Technologie
Karlsruhe, Germany
speck@kit.edu

***Abstract*—We give an $\mathcal{O}(n + \min\{n, m\} \log m)$ work algorithm for scheduling n tasks with flexible amount of parallelism on m processors, provided the speedup functions of the tasks are concave. We give efficient parallelizations of the algorithm that run in polylogarithmic time. Previous algorithms were sequential and required quadratic work. This is in some sense a best-possible result since the problem is NP-hard for more general speedup functions.**

I. Introduction

As parallellism becomes ubiquitous and cheap with modern many-core processors, we need more flexible ways to exploit this parallelism. One interesting model are *malleable tasks* (see [12]) that can adapt the parallelism they use to the ressources currently available. Following standard assumptions of classical scheduling theory, in this paper we study the makespan optimization problem for independent malleable tasks for the case where we have complete information. This information specifies the sequential work w_j required for each of n jobs, as well as a speedup function $f_j : 1..m \to [1, m]$[1] specifying how much faster job j can be executed when run on multiple processors (m being the number of processors (PEs)). Unfortunately, the problem is NP-hard for general speedup functions – In Appendix A we show that the problem is NP-hard even for step functions with just two possible speedups. The good news is that there are polynomial time algorithms for the case of concave speedup functions. We naturally get concave speedup functions for parallel programs that work with any number of PEs but loose efficiency with a growing number of them, e.g., due to some sequential component in the code. It is also allowed that there is a maximal useful amount of parallelism where the speedup function remains flat beyond a certain point.

Our main contribution is to make these algorithms fast. The best previous result [3] requires running time $\Theta\big(\max\{nm, n^2 \log^2 m\}\big)$ which may be too slow for large machines. Parallelization is not discussed. After introducing basic ideas and notation in Section II we give an algorithm based on binary search with running time $\mathcal{O}(n + \min\{n, m\} \log^2 m)$ in Section III. At least a randomized version of the algorithm is fairly simple and practical. Then, in Section IV, we reduce the factor $\log^2 m$ to $\log m$. In Section V we then discuss how these algorithms can be parallelized. It turns out that we can achieve polylogarithmic execution time with near linear speedup. Section VI summarizes the results and discusses further issues.

More Related Work

In classical scheduling theory (e.g., [12]) there has been extensive work on scheduling sequential jobs. We view this as too specialized since it allows parallelism only at the most coarse level. For jobs running on multiple PEs, three different variants can be considered:

Tasks with a fixed number of PEs are also too inflexible since many problems can be parallelized with a variable number of PEs and since the optimal number of PEs to use depends on the number of PEs available and on the total set of jobs to be scheduled. Finding optimal schedules in this case is difficult because in case of nonpreemptive tasks the problem is NP-hard (generalisation from classical scheduling theory) and even strongly NP-hard for $m \geq 5$ [6]. Even in case of preemptive tasks the problem is NP-hard for m being part of the input [5] but it becomes polynomially solvable if m is fixed [2].

Moldable jobs allow a flexible number of PEs that has to be fixed once and for all.[2] While this is an attractive model, the resulting scheduling problem is NP-hard in the preemptive and nonpreemptive case [6]. The problem is even strongly NP-hard for $m \geq 5$ in the nonpreemptive case and for m being part of the input in the preemptive case [6]. As far as we know, the best approximation algorithm known has approximation ratio $\frac{3}{2} + \epsilon$ [15] (there are approximation schemes for fixed m [9]). [13] connects the approximation of moldable and fixed size jobs.

Malleable jobs offer the most flexibility. The more general problem with speedup functions without any additional properties was studied by Jansen [8]. But the general problem seems to be far more difficult. Accordingly only an approximate solution (FPTAS) with time complexity cubic in n could be obtained in [8].

[1]In this paper we use $i..j$ as a shorthand for $\{i, \dots, j\}$.

[2]Care should be taken since some works use the term malleable also in this situation (e.g. [9]).

There are astonishingly few result on parallel algorithms for classical scheduling problems. A notable exception is the bin packing approximation algorithm given in [1] that can be used to schedule nonpreemptible single processor jobs.

II. Preliminaries

The tasks are malleable, which means that a job can be preempted at any time to be moved to a different set of PEs that may even contain a different number of PEs. Preemptions are free but we will compute schedules with few preemptions. For each task j in the task set $\mathcal{J}$, we are given the amount of work w_j that has to be done to complete the task and a speedup function $f_j : 1..m \to \mathbb{Q}_{\geq 0}$ with $f_j(1) = 1$. Running job j for a time interval of length t using k processors will get work $f_j(k)t$ done. Let $w_j(k) := w_j / f_j(k)$ denote the *work function* of job j. All speedup functions are required to be concave.[3] Wlog we can also assume that speedup functions are monotonically increasing since a nonmonotonic concave speedup function reaching its maximum at k PEs could be changed into a monotonic speedup functions with value $f_j(i) := f_j(k)$ for all $i \geq k$ – this is equivalent to not using PEs that do not contribute to speedup. Our default assumption is that speedup functions are represented by the vector of their m function values. In this case, the total description length of a scheduling instance is $\Theta(nm)$. Note that we will give scheduling algorithms running in time sublinear in this input size. This makes sense in application contexts where multiple scheduling problems with only slightly changed job sets have to be solved and when we actually have different, more compact representations of speedup functions. For example, we could have a closed form expressions for the parallel execution times of the programs that work on the jobs. Note that in this case, inverting work functions may be possible in constant time.

III. A Binary Search Algorithm

Our scheduling problem can be solved in two main steps. In the first step, we solve the fractional version where the domain of the speedup functions is extended to the real interval $[0, m]$ by linear interpolation between the speedup values for adjacent integer numbers of PEs (setting $f_j(0) = 0$). In the second step, the fractional schedule is transformed into a dicrete one – see Section III-B.

The nice thing about the fractional version of the problem is that it has a very simple structure:

Lemma 1 *There are optimal schedules for fractional problems where every job runs with a fixed number of processors during the entire makespan.*

[3] f is concave iff $\forall x < y < z : f(y) \geq \frac{y-x}{z-x} f(x) + \frac{z-y}{z-x} f(z)$.

```
**Function** findMakeSpan($\mathcal{J}$)
    $\underline{T} := \max(\max_{j \in \mathcal{J}} w_j(m), \sum_{j \in \mathcal{J}} \frac{w_j}{m})$
    $\overline{T} := \max(\max_{j \in \mathcal{J}} w_j, \sum_{j \in \mathcal{J}} \frac{w_j}{m})$
    $L := \{j \in \mathcal{J} : w_j > \underline{T}\}$                    -- large jobs
    $g := \sum_{j \in \mathcal{J} \setminus L} w_j$        -- work for small, sequential jobs
    while $\exists j \in L : \#(w_j^{-1}(\underline{T}), w_j^{-1}(\overline{T})) \geq 1$ do
        $T :=$ findPivot($L, \underline{T}, \overline{T}$)
        if $\frac{g}{T} + \sum_{j \in L} w_j^{-1}(T) > m$ then $\underline{T} := T$
        else $\overline{T} := T$
    Assert $\forall j \in \mathcal{J} : w_j$ is linear on $[\underline{T}, \overline{T}]$
    return solution of $\frac{g}{T} + \sum_{j \in L} w_j^{-1}(T) = m$
```

Figure 1. Finding the makespan of the fractional problem using binary search.

Proof: Let

$$\mathcal{R} := \{r \in \mathbb{R}^n | \sum_{i=1}^{n} r_i \leq m, r_i \geq 0\} \ .$$

Each $r \in \mathcal{R}$ defines a ressource allocation of the PEs to the jobs (job j gets r_j PEs). Let

$$\mathcal{S} := \{s \in \mathbb{R}^n | s_i = f_i(r_i), r \in \mathcal{R}\} \ .$$

$\mathcal{S}$ is the set of possible speeds for the jobs. Chapter 3 of [18] Lemma 1 shows that $\mathcal{S}$ is a convex set. Let $w := (w_1, \ldots, w_n)$ be the vector of work for all jobs. Then Theorem 1 (in Chapter 3 of [18]) tells us that the minimal makespan for all jobs is

$$T^* = \min\{T > 0 | \frac{w}{T} \in \mathcal{S}\} \ .$$

Thus there is an optimal solution in which each task runs with speed $s_j^* := \frac{w_j}{T^*}$ all the time and thus uses $r_j^* := f_j^{-1}(s_j^*)$ PEs all the time. ■

This lemma reduces the fractional scheduling problem to the problem of finding the optimal makespan. In the following we will discuss several increasingly sophisticated algorithms for this purpose.

A. Binary Search

A straight forward idea is to start with simple upper and lower bounds for the makespan and then to use binary search for narrowing down this range:

Lemma 2 $\max(\max_{j \in \mathcal{J}} w_j(m), \sum_{j \in \mathcal{J}} \frac{w_j}{m})$ *is a lower bound for the makespan of the fractional schedule.*

Proof: Since the speedup functions can be assumed to be monotone, an obvious lower bound is the time $\max_{j\in\mathcal{J}} w_j(m)$ needed to process any job using full parallelism. Since the speedup functions are concave, the most efficient way to execute the jobs is to do this sequentially. The total resulting work of $\sum_{j\in\mathcal{J}} w_j$ can only be split among m PEs. ■

Lemma 3

$$\max(\max_{j\in\mathcal{J}} w_j, \sum_{j\in\mathcal{J}} \frac{w_j}{m})$$

is a upper bound for the makespan of the fractional schedule.

Proof: If we compare the length of an (yet unknown) optimal schedule with the w_j we get two cases:

In the *first case* all w_j are smaller than the length of the optimal schedule. Then we can consider all jobs as sequential ones with work w_j. Hence we can use McNaughton's wrap-around rule [14] to get a schedule with makespan $\sum_{j\in\mathcal{J}} w_j/m$. And

$$w_i \leq \sum_{j\in\mathcal{J}} \frac{w_j}{m}$$

holds for all i. So in this case

$$\max(\max_{j\in\mathcal{J}} w_j, \sum_{j\in\mathcal{J}} \frac{w_j}{m}) = \sum_{j\in\mathcal{J}} \frac{w_j}{m} \quad .$$

In the *second case* the maximal w_i is bigger than the length of the optimal schedule. Then it is clear that the maximal w_i is an upper bound for the length of the optimal schedule. We also get

$$w_i > \sum_{j\in\mathcal{J}} \frac{w_j}{m}$$

for the maximal w_i because $\sum_{j\in\mathcal{J}} w_j/m$ is a lower bound for the optimal schedule. ■

Here, we maintain the invariant that the optimal makespan is in $[\underline{T}, \overline{T}]$. Figure 1 gives pseudocode for this approach. After computing the starting range, the algorithm first indentifies the set L of jobs that are sufficiently large that they may possibly require more than one PE in an optimal solution. This is mainly important for the running time, since there can be at most $m-1$ jobs in L.

When we want to check whether some $T \in [\underline{T}, \overline{T}]$ is a makespan sufficient to process all jobs, we just have to check whether the total required ressources for achieving makespan T are bounded by m. Using Lemma 1 this is easy because we just have to find out how many processors are needed to achieve exactly makespan T for the large jobs. The latter task amounts to inverting the work function w at T. Note that $w^{-1}(T)$ always has a well defined value since we are only using values $T \geq \underline{T} \geq \max_{j\in\mathcal{J}} w_j(m)$.

What we said above already suffices to quickly obtain good approximations for the optimal makespan. But when can we actually stop the binary search and how do we finally get the optimal value? The key insight here is that we can give a closed form formula for the optimal makespan, once all the speedup functions are linear for the entire range $[\underline{T}, \overline{T}]$. Appendix B derives the actual formula needed for this purpose. A speedup function becomes linear for $[\underline{T}, \overline{T}]$ once its corresponding domain $[w^{-1}(\underline{T}), w^{-1}(\overline{T})]$ does not properly contain any integers (where it can bend). Defining $\#(a, b)$ as the number of integers properly contained in $[a, b]$, we obtain the stopping criterion used in Figure 1.

There remains one crucial problem: We can construct instances , where convergence would be very slow using the usual interval halving technique for choosing the next makespan T (refer to the end of this section). Rather than halving the range of possible makespans, we should try to half the number of bend points $\sum_{j\in\mathcal{J}} \#(w_j^{-1}(\underline{T}), w_j^{-1}(\overline{T}))$. This means that we should take a bend point corresponding to a median value of the work function with respect to the work function value at all bend points. This sounds expensive since there are up to mn bend points overall and up to $m|L|$ bend points for large jobs. However, we can exploit that the bend points are organized into $|L|$ sorted sequences of length up to m. This multisequence selection problem can be solved in time $\mathcal{O}(|L| \log m)$ (see [7]).

With multisequence selection for implementing the function findPivot, we arrive at the following result:

Theorem 1 *n malleable jobs with concave speedup function can be scheduled optimally in time*

$$\mathcal{O}\big(n + \min(n, m) \log^2 m\big) \quad .$$

Proof: Function findMakeSpan finds an optimal makespan for the fractional problem and in Section III-B it is explained how this can be used to construct a discrete schedule with the same makespan. Since the number of bend points is halved in each iteration of the main loop, it performs at most $\log(|L|m) = \mathcal{O}(\log m)$ iterations. Each iteration takes time $\mathcal{O}(m \log m)$ if pivot selection uses an efficient algorithm for multisequence selection and binary search for computing inverses of work functions. ■

Multisequence selection is relatively complicated and involves significant constant factors hidden in the $\mathcal{O}(\cdot)$-notation. The algorithm becomes faster by a constant factor if we replace exact multisequence selection with a computation that is guaranteed to have a constant fraction of the bend points on either side of T. This can be done like this: For each large job, we take the median of the bend points in $I_j = [w_j^{-1}(\underline{T})..w_j^{-1}(\overline{T})]$ and compute the weighted median of these medians, where the weight is the number of bend points contained in I_j [7].

We can get even faster an more simple by choosing a random bend point as pivot. This can be done in time $\mathcal{O}(m)$ and still guarantees an expected number of $\mathcal{O}(\log m)$ iterations of the main loop. We obtain the same asymptotic performance as in Theorem 1, now as an expected time bound. If the work function can be inverted in constant time, this randomized algorithm achieves running time $\mathcal{O}(n+\min(n,m)\log m)$.

Example: Here we present a family of scheduling problems, where a simple bisection technique can't find the optimal optimal schedule in a logarithmic number of executions of the main loop.

Let the number of jobs be 6 for all members of the family. The number of processors $m \geq 50$ can be any multiple of 10 which is not divisible by 6. All jobs have the same amount of work w and all jobs have the same speedup function f which is defined as follows:

$$f(k)=\begin{cases} k, & \text{for } 0\leq k\leq \frac{m}{10} \\ \frac{m}{10}+2^{-m}(k-\frac{m}{10}), & \text{for } \frac{m}{10}<k\leq\frac{2m}{10} \\ \frac{m}{10}+2^{-m}\frac{m}{10} & \text{else} \end{cases}$$

The optimal fractional schedule is giving every job $\frac{1}{6}$ of the processors.

For all members of the familiy

$$\underline{T}=\max(\frac{w}{\frac{m}{10}+2^{-m}\frac{m}{10}},\frac{6w}{m})\leq\frac{11w}{m}$$

and

$$\overline{T}{:=}\ \max(w,\frac{6w}{m})=wpunkt$$

Hence, the length of the starting interval $[\underline{T},\overline{T}]$ is at least $\overline{T}-\underline{T}\geq\frac{w}{2}$. Now we compute the length of the largest interval that contains T^* but doesn't contain any bend point. The length is

$$\begin{aligned} & \frac{w}{f(\lfloor\frac{m}{6}\rfloor)}-\frac{w}{f(\lceil\frac{m}{6}\rceil)} \\ = & \frac{w}{f(\lfloor\frac{m}{6}\rfloor)f(\lceil\frac{m}{6}\rceil)}(f(\left\lceil\frac{m}{6}\right\rceil)-f(\left\lfloor\frac{m}{6}\right\rfloor)) \\ \leq & \frac{w}{\frac{m}{10}\frac{m}{10}}(2^{-m}(\left\lceil\frac{m}{6}\right\rceil-\frac{m}{10})-2^{-m}(\left\lfloor\frac{m}{6}\right\rfloor-\frac{m}{10})) \\ = & \frac{100w}{m^2}(2^{-m}(\left\lceil\frac{m}{6}\right\rceil-\left\lfloor\frac{m}{6}\right\rfloor)) \\ = & \frac{100w}{m^2}2^{-m} \end{aligned}$$

Hence one needs at least $\Omega(m)$ bisection steps to get an interval without bend points that contains T^*.

B. Solving the Discrete Problem

The pseudocode in Figure 2 summarizes the computation of a complete schedule once the optimal makespan is given. This algorithm is basically a simplified description of the main result of [3]. We use $\langle x\rangle$ to denote the fractional part $x-\lfloor x\rfloor$ of x. The schedule is output as a sequence of tuples $(j,[a,b],i..k)$ saying that during time interval $[a,b]$ job j runs on PEs $i..k$. We use $T^*[a,b]$ as a shorthand for $[T^*a,T^*b]$. The algorithm schedules one job after the other and maintains the invariant that when job j is scheduled, PEs $1..\lfloor c\rfloor$ are completely filled with the previous jobs and PE $\lfloor c\rfloor+1$ is filled during time span $[0,T^*\langle c\rangle]$ where c is the sum of the ressource requirement of the previously scheduled jobs. Let $r=w_j^{-1}(T^*)$ denote the ressources required to finish job j in time T^* in the fractional solution. The basic idea is to emulate the fractional schedule by running job j with $\lceil r\rceil$ PEs for a duration $T^*\langle r\rangle$ and with $\lfloor r\rfloor$ PEs for the remaining duration of $T^*(1-\langle r\rangle)$. To do this in such a way that the invariant is mainained, we need to distinguish two cases. If $\langle r\rangle+\langle c\rangle\leq 1$ then the "big" part of j requiring $\lceil r\rceil$ PEs is scheduled just after the lower part of the previous schedule begins. Otherwise, the big part is split into two pieces in such a way that the second piece just fills the lower part of the old schedule and the remaining piece starts at time zero. The pictures in the pseudocode illustrate these cases.

Procedure findSchedule has running time $\mathcal{O}(n+\min(n,m)\log m)$ – taking into account that $w_j^{-1}(T^*)$ can be computed in constant time for the majority of jobs that have $w_j^{-1}(T^*)\leq 1$ and in logarithmic time for the at most $m-1$ larger jobs.

The produced schedules have several nice properties: Each job is preempted at most twice. A job with ressource requirement r in the fractional solution is executed on $\lceil r\rceil+1$ consecutively numbered PEs. No PE has to work on more than two large jobs. A description of a schedule has length $\mathcal{O}(n)$.

Example for a complete solution:

Let $m=5$, $n=3$ and $w_1=12, w_2=12, w_3=8$. The speedup funktions are given in the following table:

$\#m$	1	2	3	4	5
job 1	1	$\frac{3}{2}$	2	2	2
job 2	1	2	3	4	4
job 3	1	2	$\frac{8}{3}$	$\frac{8}{3}$	$\frac{8}{3}$

These speeds lead to the following execution times on different numbers of processors:

$\#m$	1	2	3	4	5
job 1	12	8	6	6	6
job 2	12	6	4	3	3
job 3	8	4	3	3	3

When we compute T^* with the algorithm from Figure 1 and Appendix B we get $T^*=\frac{22}{3}$ and $r_1^*=\frac{25}{11}$, $r_2^*=\frac{18}{11}$, $r_3^*=\frac{12}{11}$ and the fractional depicted in Figure 3.

The algorithm from Figure 2 then produces the discrete schedule shown in Figure 4.

IV. A Faster Algorithm

The binary search algorithm from Section III-A has quadratic dependence on $\log m$ since it uses two binary searches nested within each other – the main loop and inversions of work functions. The inversion is needed in order to compute the exact ressource requirements for the

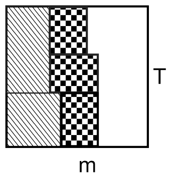
```
Procedure findSchedule($\mathcal{J}, T^*$)
    $c$:= 0                                          -- currently consumed PEs
    foreach $j \in \mathcal{J}$ do
        $r$:= $w_j^{-1}(T^*)$                          -- ressources needed for job $j$
        if $\langle r \rangle + \langle c \rangle \leq 1$ then output

            $(j, T^*[0, \langle c \rangle], \lfloor c \rfloor + 1 .. \lfloor c \rfloor + \lfloor r \rfloor)$
            $(j, T^*[\langle c \rangle, \langle c \rangle + \langle r \rangle], \lfloor c \rfloor .. \lfloor c \rfloor + \lfloor r \rfloor)$
            $(j, T^*[\langle c \rangle + \langle r \rangle, 1], \lfloor c \rfloor .. \lfloor c \rfloor + \lfloor r \rfloor - 1)$

        else output

            $(j, T^*[0, \langle c \rangle + \langle r \rangle - 1], \lfloor c \rfloor + 1 .. \lfloor c \rfloor + \lfloor r \rfloor + 1)$
            $(j, T^*[\langle c \rangle + \langle r \rangle - 1, \langle r \rangle], \lfloor c \rfloor + 1 .. \lfloor c \rfloor + \lfloor r \rfloor)$
            $(j, T^*[\langle r \rangle, 1], \lfloor c \rfloor + 1 .. \lfloor c \rfloor + \lfloor r \rfloor)$

        $c$:= $c + r$
```



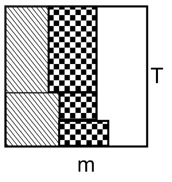


Figure 2. Computing a discrete schedule given the optimal makespan.

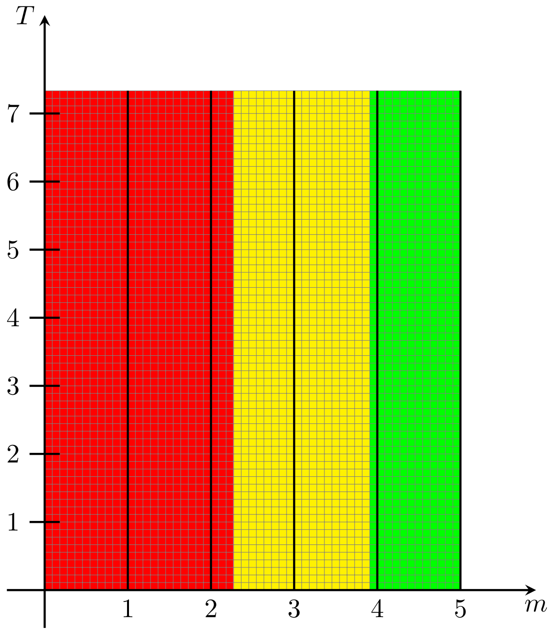


Figure 3. Fractional schedule for our example.

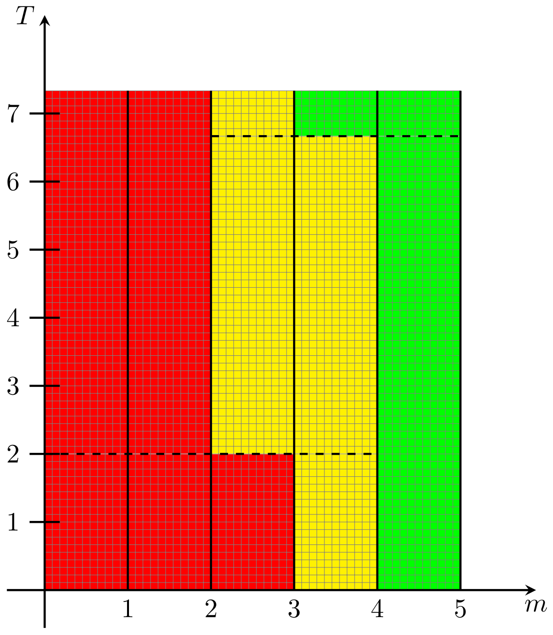


Figure 4. Discrete schedule for our example.

pivot makespan T. We will now describe an algorithm which becomes asymptotically faster by avoiding exact inversions of work functions (we assume $m \geq 17$ as otherwise the inversion of work functions is possible in constant time). We show how already approximate ressource requirements can be used to eliminate a constant fraction of the remaining bend points. This reduces the total running time to $\mathcal{O}(n + \min(n, m) \log m)$, because we still need only $\mathcal{O}(\log m)$ iterations of the main loop, each of which will only need time $\mathcal{O}(\min(n, m))$. The fast algorithm replaces the global interval $[\underline{T}, \overline{T}]$ of possible makespans with per job intervals $[\underline{T}_j, \overline{T}_j]$ which are associated with bend points of the speed function of job j. Each of these intervals still contains the optimal makespan T^*. The small tasks which will surely be executed sequentially are considered through g which is the sum of their w_j-s. This is analogous to

```
Function findMakeSpanFast($\mathcal{J}$)
    $L{:=} \left\{ j \in \mathcal{J} : w_j > \sum_{j \in \mathcal{J}} \frac{w_j}{m}) \right\}$                -- large jobs
    $g{:=} \sum_{j \in \mathcal{J} \setminus L} w_j$                -- work for small, sequential jobs
    foreach $j$ do $B_j{:=} 1..m-1$ (or $\underline{b}_j{:=} 0$, $\overline{b}{:=} m$)
    $W{:=} L$                -- wide jobs ($|B_j| \geq 16$)
    $N{:=} \emptyset$                -- narrow jobs ($|B_j| < 16$)
    while $W \neq \emptyset$ do
        compute $p(\frac{1}{3})$ and $p(\frac{2}{3})$ for the wide jobs                -- weighted selection
        if $\frac{g}{t(1/3)} + \sum_{j \in L} \overline{s}_j^{(1/3)} \leq m$ then
            foreach $j \in W$ with $w_j(m_j(\frac{1}{3})) \geq t(\frac{1}{3})$ do
                $\underline{b}_j{:=} m_j(\frac{1}{3})$
        if $\frac{g}{t(2/3)} + \sum_{j \in L} \underline{s}_j^{(2/3)} \geq m$ then
            foreach $j \in W$ with $w_j(m_j(\frac{2}{3})) \leq t(\frac{2}{3})$ do
                $\overline{b}_j{:=} m_j(\frac{2}{3})$
        update $W$ and $N$
    finish using binary search similar to Section III-A
    return solution (by solving the remaining linear equation)
```

Figure 5. Finding the makespan of the fractional problem using advanced binary search.

Section III-A. If the interval $[\underline{T}_j, \overline{T}_j]$ contains less than 16 bend points we can compute $w_j^{-1}(T)$ in constant time. If this is the case for all jobs we can use an algorithm similar to the one in Section III-A which takes the border values $\underline{b}_j$ and $\overline{b}_j$ as additional input.

To describe the algorithm in more detail, we need some additional notation: Let $B_j = (\underline{b}_j + 1)..(\overline{b}_j - 1)$ denote the interval of *active* bend points for job j, i.e., those bend points of which it is still unknown if they are bigger or smaller than the ressource requirement for the optimal makespan T^*. We define W as the set of wide jobs with $B_j \geq 16$ and N as the set of narrow jobs with $B_j < 16$. The decision in which set a job is can be done in constant time.

For $c \in [0,1]$ let

$$m_j(c){:=} \lceil c|B_j|\rceil + \min B_j - 1,$$

i.e., $m_j(c)$ is a c-quantile that splits B_j in the ratio c to $1-c$. One can compute $m_j(c)$ in constant time. Let

$$U = \sum_{j \in W} |B_j|$$

denote the total number of bend points of the wide jobs left. Let

$$M_c{:=} \{(m_j(c), j) : j \in W\}$$

denote the set of c-quantiles. We consider the elements (i,j) of M_c to be weighted with $|B_j|$ and ordered like this:

$$(i,j) < (\tilde{i}, \tilde{j}) \Leftrightarrow w_j(i) > w_{\tilde{j}}(\tilde{i}) \ .$$

Let $p(c) = (r(c), j(c))$ be the weighted c-quantile of M_c. This means $p(c)$ is the smallest element of M_c such that

$$\sum_{\{j \in W : (m_j(c), j) \leq p(c)\}} |B_j| \geq c \cdot \sum_{j \in W} |B_j| \ .$$

We also introduce the appreviation $t(c){:=} w_{j(c)}(r(c))$. One can compute $p(c)$ in time $|W|$ using the weighted selection algorithm from [10].

Our fast algorithm computes approximations of $w_j^{-1}(t(c))$ for two different values of c (1/3 and 2/3). We will show that for one of these splits it must be possible to cut the set of remaining bend points by a constant fraction. Let $\overline{s}_j^{(c)}$ denote an upper bound for $w_j^{-1}(t(c))$ that is exact when $|B_j| < 16$ and based on five iterations of binary search otherwise (the first iteration takes $m_j(c)$ as pivot). Note that in either case, $\overline{s}_j^{(c)}$ can be computed in constant time. Let $\underline{s}_j^{(c)}$ denote an analogously defined lower bound. If $t(c) < w_j(m)$ we set

$$\underline{s}_j^{(c)}{:=} \overline{s}_j^{(c)}{:=} m+1 \ .$$

This will prevent us from infeasible solutions. We assume that $T^* > \max\{w_j(m) | j \in \mathcal{J}\}$. We can check this condition if we compute $\max\{w_j(m)|j \in \mathcal{J}\}$ in $\mathcal{O}(n)$ and then test in $\mathcal{O}(n + \min(n, m)\log m)$ (like in Section III-A) if this solution is feasible. If our assumption is wrong this will give us a solution immediately.

With these definitions in place, the pseudocode given in Figure 5 is fairly straightforward. What remains to be shown is that one of the if-statements in the main loop will always apply, that either one will reduce U by a constant factor and that T^* is always inside $[\underline{T}_j, \overline{T}_j]$.

Lemma 4 *In each iteration of Algorithm 5 at least one of the two* **if***-cases is true.*

Proof: The lemma is true if

$$\frac{g}{t(1/3)} + \sum_{j\in L} \overline{s}_j^{(1/3)} \le \frac{g}{t(2/3)} + \sum_{j\in L} \underline{s}_j^{(2/3)} \tag{1}$$

because if both **if**-cases are false, we must have

$$\frac{g}{t(1/3)} + \sum_{j\in L} \overline{s}_j^{(1/3)} > m$$

and

$$m > \frac{g}{t(2/3)} + \sum_{j\in L} \underline{s}_j^{(2/3)}$$

which cannot be true simultaneously with inequality 1.

From the definition of $t(\frac{1}{3})$ we know

$$\sum_{\{j\in W: w_j(m_j(\frac{1}{3}))\le t(\frac{1}{3})\}} |B_j| \ge \frac{2}{3}U$$

and from the definition of $t(\frac{2}{3})$ we know

$$\sum_{\{j\in W: w_j(m_j(\frac{2}{3}))\ge t(\frac{2}{3})\}} |B_j| \ge \frac{2}{3}U \ .$$

Thus

$$\sum_{\{j\in W: w_j(m_j(\frac{2}{3}))\ge t(\frac{2}{3}) \wedge w_j(m_j(\frac{1}{3}))\le t(\frac{1}{3})\}} |B_j| \ge \frac{1}{3}U \ .$$

Hence, there has to exist a $j \in W$ with

$$t(\frac{2}{3}) \le w_j(m_j(\frac{2}{3}))$$

and

$$w_j(m_j(\frac{1}{3})) \le t(\frac{1}{3}) \ .$$

We know $w_j(m_j(\frac{2}{3})) \le w_j(m_j(\frac{1}{3}))$ and hence we get $t(\frac{2}{3}) \le t(\frac{1}{3})$.

This immediately gives $\frac{g}{t(1/3)} \le \frac{g}{t(2/3)}$. Because the w_j-s are monotonically decreasing we get

$$w_j^{-1}(t(\frac{1}{3})) \le w_j^{-1}(t(\frac{2}{3}))$$

for all $j \in L$. For $j \in N$ we have

$$\overline{s}_j^{(1/3)} = w_j^{-1}(t(\frac{1}{3}))$$

and $\underline{s}_j^{(2/3)} = w_j^{-1}(t(\frac{2}{3}))$. We get

$$\frac{g}{t(1/3)} + \sum_{j\in N} \overline{s}_j^{(1/3)} \le \frac{g}{t(2/3)} + \sum_{j\in N} \underline{s}_j^{(2/3)},$$

thus we only have to care about tasks that are in $W = L \backslash N$.

For tasks in $j \in W$ there are 4 cases (orderings in $M_{\frac{1}{3}}$ or $M_{\frac{2}{3}}$):

1) $(m_j(\frac{1}{3}), j) < p(\frac{1}{3})$ and $(m_j(\frac{2}{3}), j) \le p(\frac{2}{3})$
2) $(m_j(\frac{1}{3}), j) \ge p(\frac{1}{3})$ and $(m_j(\frac{2}{3}), j) \le p(\frac{2}{3})$
3) $(m_j(\frac{1}{3}), j) \ge p(\frac{1}{3})$ and $(m_j(\frac{2}{3}), j) > p(\frac{2}{3})$
4) $(m_j(\frac{1}{3}), j) < p(\frac{1}{3})$ and $(m_j(\frac{2}{3}), j) > p(\frac{2}{3})$

Let us consider a j in case 2. We know that $m_j(\frac{1}{3}) \ge w_j^{-1}(t(\frac{1}{3}))$ and $m_j(\frac{1}{3}) \ge \overline{s}_j^{(1/3)}$ as we start the binary search with $m_j(\frac{1}{3})$. Analogously we know $m_j(\frac{2}{3}) \le \underline{s}_j^{(2/3)}$. Now we can compute a lower bound for $d_j^\ell := \underline{s}_j^{(2/3)} - \overline{s}_j^{(1/3)}$. First we get

$$d_j^\ell \ge m_j(\frac{2}{3}) - m_j(\frac{1}{3}) = \left\lceil \frac{2}{3}|B_j| \right\rceil - \left\lceil \frac{1}{3}|B_j| \right\rceil \ge \frac{1}{3}(|B_j| - 1)$$

and as $|B_j| \ge 16$ we get

$$d_j^\ell \ge \frac{1}{3} \cdot \frac{15}{16}|B_j| = \frac{5}{16}|B_j| \ .$$

Now consider a j in case 1,3 or 4. We overestimate the worst case if we start the binary search with borders $\underline{b}_j$ and $\overline{b}_j$ after step 1. After 4 more steps, the remaining search interval has length

$$\ell_j := \frac{1}{16}(\overline{b}_j - \underline{b}_j) \le \frac{1}{16}(|B_j| + 2)$$

and as $|B_j| \ge 16$ we get

$$\ell_j \le \frac{1}{16} \cdot \frac{18}{16}|B_j| = \frac{9}{128}|B_j| \ .$$

Now we can compute a upper bound for

$$d_j^u := \overline{s}_j^{(1/3)} - \underline{s}_j^{(2/3)} \ .$$

We know $w_j^{-1}(t(\frac{1}{3})) \le w_j^{-1}(t(\frac{2}{3}))$ thus

$$\begin{aligned} d_j^u &\le \overline{s}_j^{(1/3)} - w_j^{-1}(t(\frac{1}{3})) + w_j^{-1}(t(\frac{2}{3})) - \underline{s}_j^{(2/3)} \\ &\le \frac{9}{128}|B_j| + \frac{9}{128}|B_j| = \frac{9}{64}|B_j| \ . \end{aligned}$$

We know that

$$\sum_{j\in W} \underline{s}_j^{(2/3)} - \sum_{j\in W} \overline{s}_j^{(1/3)} \ge \sum_{j\in \text{case 2}} d_j^\ell - \sum_{j\in \text{case 1,3 or 4}} d_j^u \ .$$

Because of the definition of $p(\frac{1}{3})$ we get

$$\sum_{j\in \text{case 1 or 4}} |B_j| \le \frac{1}{3}\sum_{j\in W} |B_j| \ .$$

Because of the definition of $p(\frac{2}{3})$ we get

$$\sum_{j \in \text{case 3 or 4}} |B_j| \leq \frac{1}{3} \sum_{j \in W} |B_j| \ .$$

Thus we know

$$\sum_{j \in \text{case 2}} |B_j| \geq \frac{1}{3} \sum_{j \in W} |B_j|$$

. With $U = \sum_{j \in W} |B_j|$ we get:

$$\sum_{j \in W} \underline{s}_j^{(2/3)} - \sum_{j \in W} \overline{s}_j^{(1/3)} \geq \frac{5}{16} \cdot \frac{1}{3} U - \frac{9}{64} \cdot \frac{2}{3} U = \frac{1}{96} U > 0$$

This gives the assertion. ■

Lemma 5 *The execution time of Algorithm 5 is* $\mathcal{O}(n + \min\{n, m\} \log m)$.

Proof: Every step of the main loop takes time $\mathcal{O}(\min\{n, m\})$. Thus we only have to show that the main loop needs $\mathcal{O}(\log m)$ steps to terminate. To do this, it suffices to show, that in every step of the main loop the number of bend points in $\bigcup_{j \in W} B_j$ is reduced by a constant factor.

We know from Lemma 4 that at least one of the **if**-conditions in Algorithm 5 is true in every step. Wlog we assume that

$$\frac{g}{t(1/3)} + \sum_{j \in L} \overline{s}_j^{(1/3)} \leq m \ .$$

From the definition of $p(\frac{1}{3})$ we know that at least $\frac{1}{3}$th of all bend points in $\bigcup_{j \in W} B_j$ are in a set B_j with $(m_j(\frac{1}{3}), j) \leq p(\frac{1}{3})$. As we set $\underline{b}_j := m_j(\frac{1}{3})$ we delete at least $\frac{1}{3}$th of the bend points in B_j. So in every step of the main loop at least $\frac{1}{9}$th of all bend points in $\bigcup_{j \in W} B_j$ are deleted. This proves the running time.

In the end we can use binary search similar to Section III-A because computing $w_j^{-1}(T)$ takes only constant time if $|B_j| < 16$ and the binary search algorithm with constant time work function inversion runs in time $\mathcal{O}(n + \min\{n, m\} \log m)$. ■

Lemma 6 *The invariant* $\underline{b}_j \leq w_j^{-1}(T^*) \leq \overline{b}_j$ *holds for all* j *at the end of the main loop in Algorithm 5.*

Proof: The invariant holds before the first step of the main loop. We now have to show that the invariant before one step implies the invariant after the step.

Recall that

$$\frac{g}{T^*} + \sum_{j \in L} w_j^{-1}(T^*) = m \ .$$

Now we consider the case

$$\frac{g}{t(1/3)} + \sum_{j \in L} \overline{s}_j^{(1/3)} \leq m \ .$$

Then,

$$\frac{g}{t(1/3)} + \sum_{j \in L} w_j^{-1}(t(\frac{1}{3})) \leq \frac{g}{T^*} + \sum_{j \in L} w_j^{-1}(T^*)$$

and because of the monotonicity of the w_j-s we have $t(\frac{1}{3}) \geq T^*$. From the definition of the ordering in $M_{\frac{1}{3}}$ we get for $j \in W$ with $(m_j(\frac{1}{3}), j) \leq p(\frac{1}{3})$ that $w_j(m_j(\frac{1}{3})) \geq t(\frac{1}{3})$ and thus $w_j(m_j(\frac{1}{3})) \geq T^*$. The monotonicity of the w_j-s proves the lemma in this case because setting $\underline{b}_j$ to $m_j(\frac{1}{3})$ for $j \in W$ with $(m_j(\frac{1}{3}), j) \leq p(\frac{1}{3})$ does not change the invariant.

The case of

$$\frac{g}{t(2/3)} + \sum_{j \in L} \underline{s}_j^{(2/3)} \geq m$$

can be proven analogously. ■

Overall we obtain:

Theorem 2 *n malleable jobs with concave speedup function can be scheduled optimally in time*

$$\mathcal{O}(n + \min(n, m) \log m) \ .$$

V. Parallelization

We now explain how our algorithms can be parallelized. We assume a machine where all PEs have access to the work functions and where the collective operations broadcast, barrier, reduction, and prefix sum can be computed in time $\mathcal{O}(\log m)$. For example, an EREW PRAM [11] has these properties. But note that this also holds for more realistic shared memory models with asynchronous behavior and even for distributed memory machines. An adaptation to distributed memory is possible if the descriptions of the work functions are available on all nodes of the system or if a work function description can be represented with constant space.

We first parallelize the binary search algorithm from Section III-A. Initially, we assign n/m jobs to each PE. $\overline{T}$, $\underline{T}$, and g can then easily be computed in time $\mathcal{O}(n/m + \log m)$ using local computations, reduction, and broadcast. The large jobs in L can then be computed locally. Since $|L| < m$ we can from now on assign a full PE to each large job (using prefix sums).

Inverting work functions of large jobs can be done in parallel for all work functions and takes time $\mathcal{O}(\log m)$ also. The decisions in the main loop can be implemented using broadcast and reduction in time $\mathcal{O}(\log m)$. For routine findPivot, we have to decide which implementation to choose. A random pivot is relatively easy to find in logarithmic time: compute an (exclusive) prefix sum over the bend point counts $c_j = \#(w_j^{-1}(\underline{T}), w_j^{-1}(\overline{T}))$. Broadcast a random number r between 1 and the total number of bend points. The PE whose prefix sum s_j is below r but where $s_j + c_j \geq r$ broadcasts the work required at its $r - s_j$-th local bend point. Even the deterministic algorithm at the end of

Section III-A can be parallelized in logarithmic time on an EREW PRAM: Sort the local medians using Cole's merge sort [4]. A weighted median is then easy to determine using a prefix sum.

Computing a discrete schedule given a feasible makespan is easy once one notices that the computations in the main loop of Figure 2 only depend on the sum of the ressource requirements of the previously scheduled jobs. This information can be computed using a prefix sum over the individual ressource requirements. Overall, we obtain the following result:

Theorem 3 *n malleable jobs with concave speedup function can be scheduled optimally in time $\mathcal{O}\big(n/m + \log^2 m\big)$ on m PEs of an EREW PRAM.* □

Parallelizing the more efficient algorithm from Section IV is also possible. However, on m PEs we do not get faster than the above result since we still need a logarithmic number of global operations. However, we can can get more efficient by reducing the number of PEs to $m/(\log m \log\log m)$ and aiming for execution time $\mathcal{O}\big(\log^2 m \log\log m\big)$. The only nontrivial change to the simple binary search algorithm is that we have to replace Cole's mergesort by a more efficient algorithm that directly finds weighted medians in time $\mathcal{O}(\log m \log\log m)$ [16]. In [17, Section 3.2] an even faster weighted selection algorithm is claimed which would reduce the factor $\log\log m$ to the iterated logarithm $\log^* m$. Unfortunately, the analysis of the algorithm is wrong since at one place a positive value ϵ assumed to be constant is set to a value with asymptotic behavior $o(1)$. [17]

VI. Conclusion

We have shown that malleable tasks with concave speedup functions can be scheduled in near linear time sequentially and in polylogarithmic time in parallel. At least the randomized version of the binary search algorithm is fairly simple and should be easy to implement, even in parallel. This means that rather than considering this scheduling problem as a complicated offline process, we can use our fast algorithms to recompute schedules dynamically within milliseconds. Thanks to parallelization this even holds for a large parallel computer. This might open up new ways to use malleable tasks in practice, e.g., in real time systems.

More generally, we find it astonishing how few results there are on parallel algorithms for classical scheduling problems. It looks like a promising direction of research to find more results in this direction. We might also look at nonconvex speedup functions with somehow weaker restrictions. As the NP-hardness proof from Appendix A only works with speedup functions which contain big steps maybe one can find polynomial time algorithms for the optimal solution if one controls the step size. It also remains unclear if the problem with general speedup functions is only NP-hard for m exponential in the input. We might find good approximation algorithms or even algorithms with running time polynomial in both m and n in case of m being polynomial in n.

The makespan minimization problem we consider does not fully exploit the potential of malleable tasks which allows arbitrary changes in number of processors. This will change if we consider more complicated settings, e.g., where jobs have arrival times and deadlines.

Acknowledgment

Partially supported by the German Science Foundation (DFG), Collaborative Research Center SFB/Transregio 89.

References

[1] R. J. Anderson, E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277, 1989.

[2] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *Computers, IEEE Transactions on*, C-35(5):389 –393, may. 1986.

[3] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz. Preemptable malleable task scheduling problem. *IEEE Transactions on Computers*, 55:486–490, 2006.

[4] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[5] M. Drozdowski. On the complexity of multiprocessor task scheduling. *Bull. Pol. Acad. Sci., Tech. Sci.*, 43(3):381–392, 1995.

[6] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[7] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in X+Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24:197–208, 1982.

[8] K. Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, 2004.

[9] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. *Algorithmica*, 32(3):507–520, 2002.

[10] D. B. Johnson and T. Mizoguchi. Selecting the kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$. *SIAM J. Comput.*, 7:147–153, 1978.

[11] J. JJ. *An introduction to parallel algorithms*. Addison-Wesley, Reading, Mass. [u.a.], 1992.

[12] J. Y.-T. Leung, editor. *Handbook of Scheduling*. CRC, 2004.

[13] W. Ludwig and P. Tiwari. Scheduling malleable and non-malleable parallel tasks. In *5th ACM SIAM Symposium on Discrete Algorithms*, pages 167–176, 1994.

[14] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):pp. 1–12, 1959.

[15] G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM Journal on Computing*, 37(2):401–412, 2007.

[16] E. Ruppert. Finding the k shortest paths in parallel. *Algorithmica*, 28(2):242–254, 2000.

[17] H. Shen. Optimal parallel weighted multiselection. volume 1, pages 323 – 326 vol.1, oct. 2002.

[18] S. G. H. Tzafestas, editor. *Optimisation and control of dynamic operational research models*. North Holland systems and control series ; 4. North-Holland, Amsterdam [u.a.], 1982. Includes bibliographies and index.

Appendix A. Malleable Scheduling with Arbitrary Speedup Functions is NP-Hard

The idea used here is similar to the one used in [5] to prove that scheduling fixed size parallel jobs is NP-hard. We will reduce PARTITION on MALLEABLE SCHEDULING. Consider $a_1, \dots, a_{2n}$ and B with $\sum_{i=1}^{2n} a_i = 2B$ and the question Exists a $J \subseteq \{1, \dots, 2n\}$ with $\sum_{i \in J} a_i = \sum_{i \notin J} a_i = B$? be our PARTITION instance.

Then we construct a MALLEABLE SCHEDULING instance with: $m := B$ and $2n$ jobs with $w_j = 2a_j$ and speedup function $f_j(k) = \begin{cases} 1, & k < a_j \\ 2a_j, & k \geq a_j \end{cases}$ and the question: Is there a schedule with makespan 2?

If there is a yes-solution for the PARTITION instance we get a yes-solution of the MALLEABLE SCHEDULING instance by running the jobs in J first (each with a_j PEs and with time 1) and then the jobs in $\{1, \dots, 2n\} \setminus J$.

If we have a yes-instance of MALLEABLE SCHEDULING every task has to run with maximal efficiency, thus has to use exactly a_j PEs during its complete computation time. Also there can't be idle PEs. Thus a set $J \subseteq \{1, \dots, 2n\}$ with $\sum_{i \in J} a_i = B$ must exist. So we get a yes-instance of PARTITION.

Appendix B. Solution of the Linear Problem

Assume we have computed border values $\underline{b}_j$ and $\overline{b}_j$ for each job. If we have only computed $\underline{T}$ and $\overline{T}$ we can compute $\overline{b}_j = w_j^{-1}(\underline{T})$ and $\underline{b}_j = w_j^{-1}(\overline{T})$ for large jobs and set $\overline{b}_j = 1$ and $\underline{b}_j = 0$ for small jobs (which are executed sequentially).

We know that all speedup functions are linear between $\underline{b}_j$ and $\overline{b}_j$. As the speedup functions are linear between $\underline{b}_j$ and $\overline{b}_j$ there exist $k_j, c_j \quad \forall j \in \{1, \dots, n\}$ such that $f_j(r_j) = k_j r_j + c_j$ for each $r_j \in [\underline{b}_j, \overline{b}_j]$. The k_j and c_j can be computed for each task in constant time. For the optimal solution $r_1^*, \dots, r_n^*$, the equation

$$w_j = f_j(r_j^*)T^* = (k_j r_j^* + c_j)T^*$$

holds. Some algebra gives:

$$\begin{aligned} w_j &= (k_j r_j^* + c_j)T^* \\ \frac{w_j}{k_j} &= (r_j^* + \frac{c_j}{k_j})T^* \\ \sum_{j=1}^{n} \frac{w_j}{k_j} &= (\sum_{j=1}^{n} r_j^* + \sum_{j=1}^{n} \frac{c_j}{k_j})T^* \\ &= (m + \sum_{j=1}^{n} \frac{c_j}{k_j})T^* \\ T^* &= \frac{\sum_{j=1}^{n} \frac{w_j}{k_j}}{m + \sum_{j=1}^{n} \frac{c_j}{k_j}} \end{aligned}$$

After we have computed T^*, we compute r_i^* using

$$w_j = (k_j r_j^* + c_j)T^*$$

for each task. This solves the continuous problem. All computations in this Section can easily be parallelized through reduce (to compute T^*) and broadcast (broadcast T^* to compute the r_j^*).