Alcom-FT Technical Report Series
ALCOMFT-TR-03-148

# A $\frac{5}{4}$-approximation algorithm for scheduling identical malleable tasks ⋆

T. Decker, T. Lücking and B. Monien

University of Paderborn
Fürstenallee 11
33102 Paderborn, Germany
Fax: +49 (5251) 60 6697
{decker,luck,bm}@uni-paderborn.de

**Abstract.** We consider the problem of finding a schedule for $n$ identical malleable tasks on $p$ identical processors with minimal completion time. This problem arises while using the branch & bound or the divide & conquer strategy to solve a problem on a parallel system. If nothing is known about the sub-problems, then they are assumed to be identical. We assume that the execution time decreases with the number of processors while the computational work increases. We give an algorithm with running time exponential in $p$ which computes an optimal schedule. In order to approximate an optimal schedule, we use the concept of phase-by-phase schedules. Here schedules consist of phases in which every job uses the same number of processors. We prove that one can approximate an optimal schedule up to a factor $\frac{5}{4}$ using constant time, and we show that this is optimal. Furthermore, we give an $\varepsilon$-approximation algorithm if the speed-up is optimal up to a constant factor.

## 1 Introduction

**Motivation-Framework.** The multiprocessor scheduling problem for identical processors is well-known and has been studied extensively in many different variations. If the number of processors on which a specific job has to be executed is part of the input, then the tasks are called non-malleable. Otherwise they are called malleable. Likewise, if the jobs may be interrupted while being executed, then the resulting schedule is called preemptive, otherwise it is called non-preemptive. Furthermore, the specification depends on precedence constraints between jobs, and on the objective. Veltman et al. [22] give an overview of a multitude of works on various specifications.

In this paper we consider the problem of finding a non-preemptive schedule for $n$ identical jobs on $p$ identical processors with minimal total completion time, the so-called makespan. Since the jobs are identical, the time function is equal

⋆ Partly supported by the DFG-Sonderforschungsbereich 376 Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen, and by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

for all jobs, that is, the execution time on any certain number of processors is the same for all jobs.

Using the branch & bound or the divide & conquer strategy to solve a problem, the problem is split into smaller subproblems which have to be solved, that is, tasks which have to be executed. In many cases the parallelism given by the branch & bound tree or by the divide & conquer tree is sufficient to yield a good speed-up. But in many other cases this is not true and we have to parallelize the computations done at the tree nodes. All these computations are of the same kind and so we may assume that all these tasks are identical. In this case the scheduling problem we consider applies. Our motivation for doing research on this problem is that this scheduling problem arises isolating real roots in parallel using the Descartes method [8]. As far as we know there is no publication on this scheduling problem. We assume that the same properties for the execution time as in [3] hold. This implies that the execution of the jobs achieves some speed-up, but no super-linear speed-up. Up to now, it is not clear whether the corresponding decision problem is $\mathcal{NP}$-hard or not.

**Related work.** There exist many results on the complexity of various scheduling problems [5]. Du and Leung [10] showed that both the non-malleable and the malleable scheduling problem are $\mathcal{NP}$-hard, so researchers are interested in approximation algorithms. If the tasks are non-malleable, then the scheduling problem is a special case of the resource constraint scheduling problem. Hence the optimal schedule can be approximated up to factor 2 using list planning [12].

Krishnamurti and Ma [17] were the first who did research on the scheduling problem with malleable tasks. Belkhale and Banerjee [3] introduced an algorithm with approximation factor $2/(1 - 1/p)$, assuming that execution time decreases with the number of processors while the computational work increases. Turek et al. [21] improved this result, using no assumptions, and showed an approximation factor 2. The latest result is from Mounié et al. [18]. Using the same assumptions for the execution time as Belkhale and Banerjee [3], they proved that an optimal schedule for malleable tasks can be approximated up to the factor $\sqrt{3}$. Blazewicz et al. [4] improved this result to 2, and a proof for the factor $\frac{3}{2}$ has been submitted for publication [19].

The best known approximation algorithm for schedules where jobs are only allowed to be executed on processors with successive indices is from Steinberg [20]. He showed an approximation factor 2. Note that this scheduling problem is closely related to the orthogonal packing problem of rectangles first investigated by Baker et al. [1]. See [11, 2, 16] for a typology of cutting and packing problems and results on approximation algorithms.

Jansen and Porkolab [14] invented the first polynomial approximation algorithm for a constant number of processors using linear programming. Note that this problem is related to orthogonal strip packing of rectangles [6, 15]. Furthermore, Jansen [13] proposed an asymptotic fully polynomial time approximation scheme if $p$ is part of the input.

**Results.** In Section 2 we formally define the scheduling problem and introduce an algorithm which computes a schedule with minimal makespan. We show that its running time is exponential in the number $p$ of processors. If $p$ is constant, then this yields an algorithm polynomial in the number $n$ of jobs. However, the algorithm is not suitable for practical purposes. In order to approximate an optimal schedule, we introduce phase-by-phase schedules in Section 3. Here schedules have a simple structure. They consist of phases in which each job uses the same number of processors. A new phase can not be started until the last phase has finished. We illustrate with help of an example that the quotient of the makespan of an optimal phase-by-phase schedule and the makespan of an optimal schedule can be $\frac{5}{4}$. Furthermore, we give a constant time algorithm which only uses certain phase-by-phase schedules providing an approximation factor $\frac{5}{4}$. In order to prove this approximation factor, we prove two technical lemmas in Section 4, providing good lower bounds on the makespan of an optimal schedule as well as restricting the set of schedules which have to be considered to dominating schedules. In Sections 5 we use these lemmas to prove the correctness of the algorithm for the case $p < n \leq \lfloor \frac{3}{2}p \rfloor$ by case analysis. Due to lack of space the proofs of the other cases are omitted here. They can be done in a similar way and can be found in [9]. Finally, we give an $\varepsilon$-approximation algorithm in the case that the speed-up is optimal up to a constant factor in Section 6.

## 2 Optimal Schedules

For the malleable scheduling problem the input consists of $n$, $p$ and $t$: $n$ is the number of identical jobs which can be executed on different number of processors in parallel. $p$ is the number of identical processors on which the jobs are to be scheduled. $t$ is the time function given by $t : \{1, \ldots, p\} \longrightarrow \mathbb{R}^+$. Here $t(i)$ is the running time needed to compute a job on $i$ processors. For $1 \leq i \leq j \leq p$ the time function $t$ must have the following properties:

- **monotony**: $t(j) \leq t(i)$
- **speed-up property**: $i \cdot t(i) \leq j \cdot t(j)$

These properties imply that the execution of the jobs may achieve some speed-up, but no super-linear speed-up.

A **schedule** $S$ assigns a set $\sigma_i$ of processors and a starting time $\tau_i$ to every job $i$ such that all jobs are executed and every processor executes at most one job at any time. We call $(\sigma_i, \tau_i)$ the **plan** according to job $i$. The objective is to minimize the **makespan** defined by

$$\max\{\tau + t(|\sigma|) \mid (\sigma, \tau) \in S\}.$$

A schedule with minimal makespan is called **optimal schedule**.

Our next goal is to find an algorithm to compute an optimal schedule. Some of these results are also in [7]. In order to proof the correctness of our algorithm, we first show that there always exists an optimal schedule with certain properties.

**Definition 1.**

1. *The* **load vector** $l(S) = (l_1, \ldots, l_p)$ *of a schedule* $S$ *is defined by*

$$l_i = \max\{\tau + t(|\sigma|) \mid (\sigma, \tau) \in S \wedge i \in \sigma\},\ 1 \le i \le p.$$

2. *The* **sorted load vector** $\lambda(S) = (\lambda_1, \ldots, \lambda_p)$ *is given by* $\lambda_i = l_{\pi(i)}$, $1 \le i \le p$, *where* $\pi$ *is a permutation with* $l_{\pi(i)} \le l_{\pi(j)}$ *for all* $1 \le i < j \le p$. *Note that* $\lambda_p$ *is the makespan of* $S$.

We will show that for each schedule there exists another schedule with the same makespan and certain properties. One of this properties is that each job in a schedule starts either at time $\tau = 0$ or directly subsequent to another job. We now define this property formally.

**Definition 2.** *A schedule* $S$ *is* **packed** *if for all plans* $(\sigma, \tau) \in S$ *either* $\tau = 0$ *holds, or there exists another pair* $(\sigma', \tau') \in S$ *with* $\sigma \cap \sigma' \neq \emptyset$ *and* $\tau' + t(|\sigma'|) = \tau$.

**Definition 3.** *Let* $S$ *be a schedule for* $n$ *jobs. Then* $S$ *contains a schedule* $S'$ *if* $S' \subseteq S$. *The schedules* $S_1, \ldots, S_{n-1}$ *are called* **intermediary schedules of** $S$ *if* $S_1 \subseteq \ldots \subseteq S_{n-1} \subseteq S$ *and* $|S_i| = i$ *for all* $1 \le i < n$.

In general we can compute a schedule for $i > 1$ jobs by using a schedule for $i-1$ jobs and assigning a set of free processors to the $i$th job. We use this fact to give an algorithm. Every step leads to a set of intermediary schedules.

**Lemma 1.** *For each optimal schedule* $S$ *for* $n$ *jobs there exists a schedule* $S_n$ *and intermediary schedules* $S_1, \ldots, S_{n-1}$ *of* $S_n$ *such that for* $1 \le i \le n$ *we have*

1. $S$ *and* $S_n$ *have the same makespan.*
2. $S_i$ *is packed, and for the sorted load vector* $(\lambda_1^{(i)}, \ldots, \lambda_p^{(i)})$ *of* $S_i$ *the inequality* $\lambda_p^{(i)} - \lambda_1^{(i)} \le t(1)$ *holds. No job in* $S_i$ *is finished later than in* $S$.

*Proof.* Let $1 \le i \le n$, and let $S_{i-1}$ be a packed intermediary schedule. Let $j$ be a processor with $l_j^{i-1} = \lambda_1^{i-1}$, i.e. $j$ has minimal load among all processors in $S_{i-1}$. Choose a plan $(\sigma, \tau) \in S$ with $j \in \sigma$ and minimal starting time which has not been considered yet. Denote $S' = S_{i-1} \cup \{(\sigma, \tau)\}$.

If $S'$ is packed then we set $S_i := S'$, and we are done. Otherwise let $\tau^\star$ be the earliest time at which all processors in $\sigma$ in $S_{i-1}$ are ready. Then $S^\star := S_{i-1} \cup \{(\sigma, \tau^\star)\}$ is packed, and $S^\star \cup (S \setminus S')$ is an optimal schedule. Hence $S_i := S^\star$ is an intermediary schedule.

If the makespan of $S'$ is larger than $\lambda_1^{i-1} + t(1)$, then remove $(\sigma, \tau)$ from the schedule and add the plan $(\{j\}, l_j^{i-1})$. Proceed in the same way if there exists no $(\sigma, \tau) \in S \setminus S_{i-1}$ with $j \in \sigma$ and the makespan of $S$ is larger than $\lambda_1^{i-1} + t(1)$. If no such $(\sigma, \tau)$ exists and the makespan of $S$ is at most $\lambda_1^{i-1} + t(1)$ then we may proceed with any plan from $S \setminus S_{i-1}$ with minimal starting time. □

We get optimal schedules by computing intermediary schedules of optimal schedules iteratively. Due to Lemma 1 we can restrict our search to packed schedules with sorted load vector $(\lambda_1, \ldots, \lambda_p)$ with $0 \le \lambda_p - \lambda_1 \le t(1)$.

In the following we will present optimal algorithms. We do so for time functions $t : \mathbb{N} \to \mathbb{R}^+$ and $t : \mathbb{N} \to \mathbb{N}$. In both cases we assume that manipulating numbers can be done in constant time. Applying the Turing machine model would result in an additional factor $\mathcal{O}(\log(\text{makespan})) = \mathcal{O}(\log(n) + \log(t(1)))$ if $\mathbb{N}$ is the range of the time function. Note that the jobs are identical and therefore the number $n$ of jobs contributes only $\log(n)$ bits to the input size.

**Lemma 2.** *If a schedule $S$ is packed, then each entry of the load vector $\lambda(S) = (\lambda_1, \ldots, \lambda_p)$ can be written as a linear combination*

$$\lambda_i \in \{a_1 t(1) + \cdots + a_p t(p) \mid a_1, \ldots, a_p \in \{0, \ldots, n\}\}.$$

**Theorem 1.**

1. *If the range of the time function $t$ is $\mathbb{R}^+$, then there exists an algorithm which computes an optimal schedule using $\mathcal{O}(n(n+1)^{p^2} 2^p)$ time.*
2. *If the range of the time function $t$ is $\mathbb{N}$, then there exist algorithms which compute an optimal schedule using $\mathcal{O}(\log(n) \cdot |V|^3)$ or $\mathcal{O}(n \cdot |V| \cdot 2^p)$ time, where $|V| = t(1)^p$.*

*Proof.* We can view the scheduling problem as a constrained shortest path problem in a directed graph. For a sorted load vector $\lambda(S) = (\lambda_1, \ldots, \lambda_p)$ of a schedule $S$ we denote with $\Lambda(S) = (\lambda_i - \lambda_1)_{i=1,\ldots,p}$ the pattern of $S$. In the directed graph every possible pattern corresponds to a node. Let $V$ be the set of nodes. A directed edge from node $i$ to node $j$ exists if and only if we can get a schedule with the pattern corresponding to node $j$ by adding a plan to a schedule with the pattern corresponding to node $i$. We assign to each edge $(i, j)$ the augmentation of the makespan gained by adding the corresponding plan. Now we can compute the minimal makespan by searching a path with $n$ edges having minimal weight.

1. If the range of the time function $t$ is $\mathbb{R}^+$, then Lemma 2 yields $|V| \le (n+1)^{p^2}$. Furthermore we know that the graph has an out-degree at most $2^p$. We use dynamic programming to compute all paths with $i$ edges for $1 \le i \le n$. So we need at most $\mathcal{O}(n(n+1)^{p^2} 2^p)$ time.
2. If the range of the time function $t$ is $\mathbb{N}$, then we have $|V| = t(1)^p$. We can compute a minimal weight path with $n$ edges in $\mathcal{O}(\log(n) \cdot |V|^3)$ time using the doubling-technique. The time complexity of the second claim can be proved in the same way as in part (1) using the fact that we only have to consider $|V| = t(1)^p$ nodes for $1 \le i \le n$. $\square$

We have proved that it is possible to compute an optimal schedule using time polynomial in $n$ if the number of processors is constant. However, the running time depends exponentially on the number of processors.

## 3 Phase-By-Phase Schedules (PPS)

We will use a **phase-by-phase schedule (PPS)** to approximate an optimal schedule. Here a schedule consists of phases in which each job uses the same number of processors. A new phase can not be started until the last phase has finished. Hence we do not have to store plans for all jobs but may write a phase-by-phase schedule with $k$ phases as $PPS = (i_1, \ldots, i_k)$ where $i_j$ denotes the number of processors used in phase $j$. The makespan is

$$\text{makespan}(PPS) = \sum_{j=1}^{k} t(i_j).$$

Decker and Krandick [8] introduced an algorithm which computes an optimal phase-by-phase schedule using $\mathcal{O}(n^2)$ time. The running time of this algorithm can be improved to $\mathcal{O}(n \cdot \min\{n, p\})$. We now show that there exists an algorithm which computes an optimal phase-by-phase schedule using $\mathcal{O}(p^3 + \log(n))$ time.

**Theorem 2.** *There exists an algorithm which computes an optimal phase-by-phase schedule using $\mathcal{O}(p^3 + \log(n))$ time.*

*Proof.* Let $x_i$ be the number of phases using $i$ processors. Due to the speed-up property $x_i \leq i - 1$ holds for $i \geq 2$. Furthermore at most $\lfloor \frac{p}{i} \rfloor$ jobs are executed during such a phase. Therefore at most

$$\sum_{2 \leq i \leq p} x_i \left\lfloor \frac{p}{i} \right\rfloor \leq \sum_{2 \leq i \leq p} (i-1) \left\lfloor \frac{p}{i} \right\rfloor \leq (p-1)p$$

jobs are executed in phases using more than one processor.

The algorithm works as follows. If $n > (p-1)p$, then compute the minimal $r$ such that $n - r \cdot p \leq (p-1)p$. The schedule starts with $r$ sequential phases. The improved algorithm from [8] is used to compute the schedule for the remaining $n - r \cdot p$ jobs. □

Decker [7] showed that the makespan of an optimal phase-by-phase schedule is at most twice as large as the makespan of an optimal schedule. The following example illustrates that computing an optimal phase-by-phase schedule can not lead to an approximation factor lower than $\frac{5}{4}$.

*Example 1.* Let $p = 5$ and $n = 3$. Furthermore let $t(1) = 1$, $t(2) = \frac{1}{2}$ and $t(i) = \frac{1}{3}$ for all $3 \leq i \leq 5$. The optimal phase-by-phase schedule is $(2, 5)$ with makespan $t(2) + t(5) = \frac{5}{6}$. However, $\sigma = \{(\{1,2,3\}, 0), (\{1,2,3\}, \frac{1}{3}), (\{4,5\}, 0)\}$ is an optimal schedule with makespan $2t(3) = \frac{2}{3}$. So the approximation factor via phase-by-phase schedules is $\frac{5}{4}$. Note that this example can be extended to $p = 3j + 2$ and $n = 2j + 1$ with $j \in \mathbb{N}$.

Our next goal is to find an algorithm with approximation factor $\frac{5}{4}$ using constant time. The idea of our algorithm is as follows: we compute a solution only depending on the relation between $p$ and $n$. We do not know anything about the time function (up to the monotony and the speed-up property), and we will consider a phase-by-phase schedule. In the following we show that this algorithm provides an approximation factor of $\frac{5}{4}$. For the sake of readability we only return the makespan of the phase-by-phase schedule which we choose. Note that the phase-by-phase schedule is returned, too. The algorithm is as follows:

**Algorithm 1** *Phase-by-Phase Schedule (PPS)*
**Input:** *number of jobs $n$, number of processors $p$, time function $t$*
**Output:** *phase-by-phase schedule*

**begin**
**if** $n > p$ **then**
    **if** $p < n \leq \lfloor \frac{3}{2}p \rfloor$ **then**
        $a = \left\lfloor \frac{p}{n-p} \right\rfloor$
        **return** $t(1) + t(a)$
    **if** $\lfloor \frac{3}{2}p \rfloor < n \leq 2p$ **then**
        $a = \left\lfloor \frac{p}{n-\lfloor \frac{3}{2}p \rfloor} \right\rfloor$
        **return** $\min\{t(1) + t(2) + t(a), 2t(1)\}$
    **if** $2p < n \leq \lfloor \frac{5}{2}p \rfloor$ **then**
        $a = \left\lfloor \frac{p}{n-2p} \right\rfloor$
        **return** $2t(1) + t(a)$
    **if** $\lfloor \frac{5}{2}p \rfloor < n \leq 3p$ **then**
        $a = \left\lfloor \frac{p}{n-\lfloor \frac{5}{2}p \rfloor} \right\rfloor$
        **return** $\min\{2t(1) + t(2) + t(a), 3t(1)\}$
    **else**
        $a = \left\lfloor \frac{p}{n-\lfloor \frac{n}{p} \rfloor \cdot p} \right\rfloor$
        **return** $\lfloor \frac{n}{p} \rfloor \cdot t(1) + t(a)$
**else**
    **if** $\lfloor \frac{p}{2} \rfloor < n \leq p$ **then**
        $a = \left\lfloor \frac{p}{n-\lfloor \frac{p}{2} \rfloor} \right\rfloor$
        **if** $2\lfloor \frac{p}{3} \rfloor < n \leq p$ **then**
            **return** $\min\{t(1), t(2) + t(a)\}$
        **if** $\lfloor \frac{p}{3} \rfloor + \lfloor \frac{p}{4} \rfloor < n \leq 2\lfloor \frac{p}{3} \rfloor$ **then**
            **return** $\min\{t(1), t(2) + t(a), 2t(3)\}$
        **else**
            **return** $\min\{t(1), t(2) + t(a), t(3) + t(4)\}$
    **if** $\lfloor \frac{p}{3} \rfloor < n \leq \lfloor \frac{p}{2} \rfloor$ **then**
        $a = \left\lfloor \frac{p}{n-\lfloor \frac{p}{3} \rfloor} \right\rfloor$
        **if** $\frac{2}{5}p \leq n \leq \lfloor \frac{p}{2} \rfloor$ **then**
            **return** $\min\{t(2), t(3) + t(a)\}$
        **if** $2\lfloor \frac{p}{5} \rfloor < n < \frac{2}{5}p$ **then**
            **if** $p \leq 17$ **then**
                **return** $\min\{t(2), t(3) + t(a), t(4) + t(6)\}$
            **else**
                **return** $\min\{t(2), t(3) + t(a), t(4) + t(5)\}$

$\quad\quad$ **else**
$\quad\quad\quad$ **return** $\min\{t(2), t(3) + t(a), 2t(5)\}$
$\quad$ **if** $\lfloor \frac{p}{4} \rfloor < n \leq \lfloor \frac{p}{3} \rfloor$ **then**
$\quad\quad$ $a = \left\lfloor \frac{p}{n - \lfloor \frac{p}{4} \rfloor} \right\rfloor$
$\quad\quad$ **if** $\frac{2}{7}p \leq n \leq \lfloor \frac{p}{3} \rfloor$ **then**
$\quad\quad\quad$ **return** $\min\{t(3), t(4) + t(a)\}$
$\quad\quad$ **if** $2\lfloor \frac{p}{7} \rfloor < n < \frac{2}{7}p$ **then**
$\quad\quad\quad$ **if** $\mathrm{rem}(p, 7) = 5$ **and** $p \leq 36$ **then**
$\quad\quad\quad\quad$ **return** $\min\{t(3), t(4) + t(a), t(5) + t(9), t(6) + t(7)\}$
$\quad\quad\quad$ **if** $\mathrm{rem}(p, 7) = 6$ **and** $p \leq 37$ **then**
$\quad\quad\quad\quad$ **return** $\min\{t(3), t(4) + t(a), t(5) + t(11), t(6) + t(7)\}$
$\quad\quad\quad$ **else**
$\quad\quad\quad\quad$ **return** $\min\{t(3), t(4) + t(a), t(6) + t(7)\}$
$\quad\quad$ **else**
$\quad\quad\quad$ **return** $\min\{t(3), t(4) + t(a), 2t(7)\}$
$\quad$ **if** $\lfloor \frac{p}{k} \rfloor < n \leq \lfloor \frac{p}{k-1} \rfloor$ **and** $k \geq 5$ **then**
$\quad\quad$ $a = \left\lfloor \frac{p}{n - \lfloor \frac{p}{k} \rfloor} \right\rfloor$
$\quad\quad$ **return** $\min\{t(k-1), t(k) + t(a)\}$
**end**

We now show that Algorithm 1 approximates an optimal schedule up to a factor $\frac{5}{4}$ using constant time. Note that applying the Turing machine model would result in a time bound $\mathcal{O}(\log(n) + \log(p) + \log(t(1)))$. The proof for $n \leq p$ is more complex, mostly since we aimed to get the real approximation factor for this interval. Thus the algorithm for $n \leq p$ is rather complex. The approximation factor for the simple algorithm we use for $n > p$ is $\frac{6}{5}$. However, we think that by using a more complex algorithm, we could get a better result. The example shown in Figure 1 demonstrates that for $n > p$ the approximation factor is at least $\frac{8}{7}$.

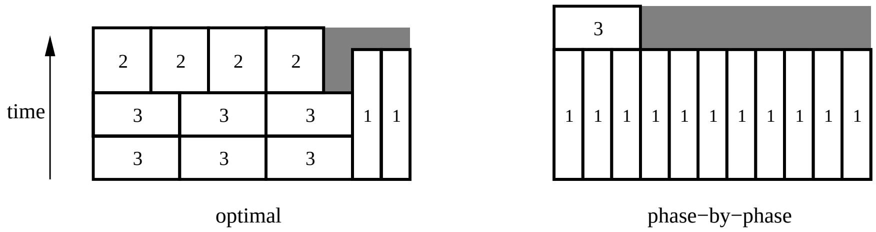


**Fig. 1.** Optimal schedule and optimal phase-by-phase schedule for $p = 11$, $n = 12$, and $t(1) = 1$, $t(2) = \frac{1}{2}$ and $t(i) = \frac{1}{3}$ for all $3 \leq i \leq 11$. The makespan of the optimal schedules is $t(2) + 2t(3) = \frac{7}{6}$, and the makespan of the optimal phase-by-phase schedule is $t(1) + t(3) = \frac{4}{3}$. So the approximation factor is $\frac{8}{7}$.

**Theorem 3.** *Algorithm 1 computes a phase-by-phase schedule which is an optimal schedule up to a factor $\frac{5}{4}$ using constant time. If $n > p$ then the approximation factor is $\frac{6}{5}$.*

*Proof.* For $p < n \leq \lfloor \frac{3}{2} p \rfloor$ the claim follows with Lemma 5. Due to lack of space the proofs for the other cases are omitted here. They can be done in a similar way and can be found in [9].

## 4 Technical lemmas

In the following we denote with $T_{OPT}$ the makespan of an optimal schedule. In order to prove Theorem 3 we mainly consider the quotient of the upper bound provided by Algorithm 1 and a lower bound for $T_{OPT}$. The following lemma will help us to find good lower bounds.

**Lemma 3.** *Let $1 \leq u_1 \leq u_2 \leq p$, $\frac{p}{u_1} + \frac{p}{u_2} \leq n$, and let the makespan of an optimal schedule consist of at least two addends.*

1. *If $t(u_1)$ is an addend in the makespan of an optimal schedule, then $T_{OPT} \geq t(u_1) + t(u_2)$.*
2. *If $u_2 \leq u_1 + 1$ and no $t(i)$, $i < u_1$, is an addend in the makespan of an optimal schedule, then $T_{OPT} \geq t(u_1) + t(u_2)$.*

*Proof.*

1. Let $t(u_1) + \sum_{1 \leq i \leq l} t(k_i)$ be the makespan of an optimal schedule. If there exists $k_i \leq u_2$, then we are done. Otherwise the speed-up property yields
$$t(u_1) + \sum_{1 \leq i \leq l} t(k_i) \geq t(u_1) + \sum_{1 \leq i \leq l} \frac{u_2}{k_i} \cdot t(u_2) = t(u_1) + \frac{u_2}{p} \cdot t(u_2) \sum_{1 \leq i \leq l} \frac{p}{k_i}$$
$$\geq t(u_1) + \frac{u_2}{p} \cdot t(u_2) \cdot \left( n - \frac{p}{u_1} \right) \geq t(u_1) + t(u_2).$$
2. Let $\sum_{1 \leq i \leq l} t(k_i)$ be the makespan of an optimal schedule. If $t(u_1)$ is an addend in this makespan, then we are done. Otherwise we have $k_i \geq u_2$ for all $1 \leq i \leq l$. We get
$$\sum_{1 \leq i \leq l} t(k_i) \geq \sum_{1 \leq i \leq l} \frac{u_2}{k_i} \cdot t(u_2) = \frac{u_2}{p} \cdot t(u_2) \cdot \sum_{1 \leq i \leq l} \frac{p}{k_i} \geq \frac{u_2}{p} t(u_2) \cdot n$$
$$\geq \frac{u_2}{p} \cdot t(u_2) \left( \frac{p}{u_1} + \frac{p}{u_2} \right) = t(u_2) + \frac{u_2}{u_1} \cdot t(u_2) \geq t(u_1) + t(u_2).$$

This completes the proof of the lemma. □

We have to consider all makespans of possible schedules. Unfortunately the number of such makespans can become very large. So finding a good lower bound is difficult. The next definition helps us to restrict the number of makespans we have to consider.

**Definition 4.** *Let $S$,$T$ be two packed schedules, and let the sums $\sum_{1 \le l \le k_1} t(i_l)$ and $\sum_{1 \le l \le k_2} t(j_l)$ be the makespans of $S$ and $T$, respectively. Then $S$* **dominates** *$T$ if for all valid time functions $t$, that is, time functions for which the monotony and the speed-up property hold,*

$$\sum_{1 \le l \le k_1} t(i_l) < \sum_{1 \le l \le k_2} t(j_l).$$

**Lemma 4.** *Let $S$ be a schedule, $r \in \mathbb{Q}^+$, $n > p \cdot r$, and let $I \subseteq \{m \in \mathbb{N} \mid m \le p\}$ be the set of number of processors which are allowed to be used. Then*

$$T_{OPT} \ge \min \left\{ \sum_{i \in I} x_i t(i) \,\middle|\, \sum_{i \in I} \frac{x_i}{i} > r \right\}.$$

*Proof.* Let $1 \le j \le p$, and let $y_i(j)$ be the number of jobs in an optimal schedule which use $i$ processors including processor $j$. Now consider a solution to the problem. If there exists a $j$ with $\sum_{i \in I} \frac{y_i(j)}{i} > r$, then we are done. Otherwise we have

$$\sum_{1 \le j \le p} \sum_{i \in I} \frac{y_i(j)}{i} = \sum_{i \in I} \sum_{1 \le j \le p} \frac{y_i(j)}{i} \le p \cdot r < n,$$

which contradicts the solvability. □

Due to Lemma 4 we only have to consider the makespans of dominating schedules for which $\sum_{i \in I} \frac{x_i}{i} > r$.
In order to find another way to get a lower bound on $T_{OPT}$, we use the speed-up property as follows: if each job in the schedule uses at least $j$ processors, then each job needs at least $j \cdot t(j)$ area. This leads to the lower bound

$$T_{OPT} \ge \frac{n}{p} \cdot (j \cdot t(j)).$$

In the following this technique will be helpful.

## 5 Proof for the case $p < n \le \lfloor \frac{3}{2} p \rfloor$

**Lemma 5.** *Let $p < n \le \lfloor \frac{3}{2} p \rfloor$. Then Algorithm 1 computes a phase-by-phase schedule which is an optimal schedule up to a factor $\frac{6}{5}$.*

*Proof.* Let $a = \lfloor \frac{p}{n-p} \rfloor$. Since $n \le \lfloor \frac{3}{2} p \rfloor$ we have $a \ge 2$. Algorithm 1 returns $t(1) + t(a)$. Note that $t(1)$ is obviously a lower bound on the makespan of an optimal schedule.
If $t(a) \le \frac{1}{5} t(1)$ we get

$$\frac{t(1) + t(a)}{t(1)} \le \frac{6}{5}.$$

So let $t(a) > \frac{1}{5}t(1)$.
**First case:** There exist no more than $p$ jobs each being executed on at most $a$ processors.
Using the speed-up property yields

$$\begin{aligned}
T_{OPT} &\geq \frac{1}{p}(p \cdot t(1) + (n-p)(a+1) \cdot t(a+1)) = t(1) + \frac{n-p}{p} \cdot (a+1) \cdot t(a+1) \\
&= t(1) + \frac{n-p}{p}\left(\left\lfloor \frac{p}{n-p} \right\rfloor + 1\right) t(a+1) > t(1) + t(a+1) \\
&\geq t(1) + \frac{a}{a+1} t(a).
\end{aligned}$$

Due to monotony of the time function we get

$$\frac{t(1)+t(a)}{t(1)+\frac{a}{a+1}t(a)} \leq \frac{t(1)+t(a)}{t(1)+\frac{2}{3}t(a)} \leq \frac{t(1)+t(1)}{t(1)+\frac{2}{3}t(1)} \leq \frac{2}{1+\frac{2}{3}} = \frac{6}{5}.$$

**Second case:** There exist at least $p+1$ jobs each being executed on at most $a$ processors.
We only have to consider dominating schedules for which $t(a)$ is the smallest addend which appears in the makespan. Since $t(a) > \frac{1}{5}t(1)$, we have $T_{OPT} \geq 6t(a) > t(1) + t(a)$ for all makespans of a schedule with more than 5 addends. If we have 5 addends then $T_{OPT} \geq 5t(a) > \frac{4}{5}t(1) + t(a)$, and we get

$$\frac{t(1)+t(a)}{\frac{4}{5}t(1)+t(a)} \leq \frac{1+\frac{1}{5}}{\frac{4}{5}+\frac{1}{5}} = \frac{6}{5}.$$

Hence we only have to consider makespans with at most 4 addends. The dominating schedules are those corresponding to the makespans $t(1)+t(a)$, $t(2)+t(3)+t(a)$, $t(2)+t(5)+2t(a)$ for $a \geq 4$, and $2t(3)+2t(a)$ for $a \geq 3$.
<u>$t(1)+t(a)$:</u>
We immediately get

$$\frac{t(1)+t(a)}{t(1)+t(a)} = 1.$$

<u>$t(2)+t(3)+t(a)$:</u>
For $a \leq 2$ we immediately get the approximation factor 1. For $a = 3$ we have

$$\frac{t(1)+t(3)}{t(2)+2t(3)} \leq \frac{t(1)+t(3)}{\frac{1}{2}t(1)+2t(3)} \leq \frac{1+\frac{1}{3}}{\frac{1}{2}+\frac{2}{3}} \leq \frac{8}{7}.$$

For $a \geq 4$ we get

$$\frac{t(1)+t(a)}{t(2)+t(3)+t(a)} \leq \frac{t(1)+t(a)}{(\frac{1}{2}+\frac{1}{3})t(1)+t(a)} < \frac{t(1)+\frac{1}{5}t(1)}{\frac{5}{6}t(1)+\frac{1}{5}t(1)} \leq \frac{6}{5} \cdot \frac{30}{31} < \frac{6}{5}.$$

<u>$t(2)+t(5)+2t(a)$:</u>
We have

$$\frac{t(1)+t(a)}{t(2)+t(5)+2t(a)} \leq \frac{t(1)+t(a)}{(\frac{1}{2}+\frac{1}{5})t(1)+2t(a)} < \frac{t(1)+\frac{1}{5}t(1)}{\frac{7}{10}t(1)+\frac{2}{5}t(1)} = \frac{6}{5} \cdot \frac{10}{11} < \frac{6}{5}.$$

$\underline{2t(3) + 2t(a)}$:
It follows that

$$\frac{t(1)+t(a)}{2t(3)+2t(a)} \leq \frac{t(1)+t(a)}{\frac{2}{3}t(1)+2t(a)} < \frac{t(1)+\frac{1}{5}t(1)}{\frac{2}{3}t(1)+\frac{2}{5}t(1)} = \frac{6}{5}\cdot\frac{15}{16} < \frac{6}{5}.$$

This completes the proof. □

## 6 An $\varepsilon$-approximation algorithm

In the previous section we proved that there exists a $\mathcal{O}(\log(n))$-time algorithm with approximation factor $\frac{5}{4}$. We now show that if the speed-up is good enough and $p$ is large enough, then an optimal phase-by-phase schedule approximates an optimal schedule up to a factor $1+\varepsilon$. Using $r = \frac{n}{p}$ we denote

$$T_k = \min\left\{ \sum_{i=1}^{k} y_i t(i) \,\middle|\, \sum_{i=1}^{k} \frac{y_i}{i} \geq r \right\}.$$

Note that $T_p$ is a lower bound on the makespan of an optimal schedule (see Section 5). The following lemma is the key to our algorithm.

**Lemma 6.** *For $1 \leq k \leq p$ we have $T_p \geq T_k - t(k)$.*

*Proof.* Denote $\sum_{i=k+1}^{p} \frac{y_i}{i} = \frac{a}{k} + b$ with $a \in \mathbb{N}_0$ and $0 \leq b < \frac{1}{k}$. Furthermore, let

$$z_i = \begin{cases} y_i & \text{if } i < k \\ y_k + a + 1 & \text{if } i = k \end{cases}$$

This implies

$$\sum_{1\leq k\leq p} \frac{z_i}{i} = \sum_{1\leq i\leq k-1} \frac{y_i}{i} + \frac{y_k + a + 1}{k} \geq \sum_{i=1}^{k} \frac{y_i}{i} + \frac{k\sum_{k+1\leq i\leq p} \frac{y_i}{i}}{k} = \sum_{1\leq i\leq p} \frac{y_i}{i} \geq r.$$

The speed-up property yields

$$\begin{aligned} T_p &= \sum_{i=1}^{p} y_i t(i) \geq \sum_{i=1}^{k} y_i t(i) + \left(\sum_{i=k+1}^{p} \frac{y_i}{i}\right) k\cdot t(k) = \sum_{i=1}^{k} y_i t(i) + \left(\frac{a}{k} + b\right) k\cdot t(k) \\ &= \sum_{i=1}^{k} z_i t(i) - (a+1)t(k) + (a+bk)t(k) = \sum_{i=1}^{k} z_i t(i) + (bk-1)t(k) \\ &\geq T_k - t(k). \end{aligned}$$

This completes the proof of the lemma. □

If $t(i\cdot j) \leq \frac{c}{i} t(j)$, $c$ constant, then the previous lemma yields an approximation algorithm.

**Theorem 4.** *Let $t(i \cdot j) \leq \frac{c}{i} \cdot t(j)$, $c$ constant. Then the optimal phase-by-phase schedule is optimal up to a factor $1 + \sqrt[6]{\frac{4c^3}{p}}$.*

*Proof.* Let $\varepsilon > 0$. We will show that the optimal phase-by-phase schedule is optimal up to a factor $1 + \varepsilon$ if $p \geq 4 \cdot \frac{c^3}{\varepsilon^6}$ holds.
$\underline{n < \varepsilon \cdot p}$: We choose the minimal $i \in \mathbb{N}$ with $\frac{p}{i+1} < n \leq \frac{p}{i}$. Hence $t(i)$ is an upper bound and $t(i+1)$ is a lower bound on the makespan of an optimal schedule. This yields

$$\frac{t(i)}{t(i+1)} \leq \frac{t(i)}{\frac{i}{i+1} \cdot t(i)} = 1 + \frac{1}{i},$$

and we are done.
$\underline{n \geq \varepsilon \cdot p}$: The coefficients $y_1, \ldots, y_p$ in the definition of $T_k$ define a phase-by-phase schedule. With this schedule we can not handle all jobs. However, due to speed-up property we have at most $i-1$ phases using $i$ processors. So we have at most $\sum_{1 \leq i \leq k}(i-1) = \frac{1}{2}(k-1)k$ phases using more than one processor. If $p \geq \frac{1}{2}k^3$, then the remaining jobs can be executed in exactly one more phase using $k$ processors. For an optimal phase-by-phase schedule with makespan $T_{PPS}$ using at most $k$ processors we have $T_{PPS} \leq T_k + t(k)$. We choose the minimal $k \in \mathbb{N}$ with $k \geq \frac{2c}{\varepsilon^2}$ and get

$$\begin{aligned} T_{PPS} &\leq T_k + t(k) \leq T_p + 2t(k) \leq T_p + \frac{2c}{k}t(1) \\ &\leq T_p + \varepsilon^2 t(1) \leq T_p + \varepsilon \cdot \frac{n}{p}t(1) \leq (1+\varepsilon) \cdot T_p. \end{aligned}$$

This completes the proof. □

*Remark 1.*

1. Combining Theorems 1, 2 and 4 leads to an $\varepsilon$-approximation algorithm. Use the optimal phase-by-phase algorithm if $p \geq 4 \cdot \frac{c^3}{\varepsilon^6}$ holds and the optimal algorithm from Theorem 1 otherwise.
2. Theorem 4 can be generalized to other time functions fulfilling $t(i) \to 0$ for $i \to \infty$.

## 7 Acknowledgment

The authors would like to thank Denis Trystram for Example 1 and Norbert Sensen for part (2) of Theorem 1, and both of them for the fruitful discussions.

## References

1. B. Baker, E. Coffman, and R. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

2. B.S. Baker, D.J. Brown, and H.P. Katseff. A 5/4 algorithm for two-dimensional packing. *Journal of Algorithms*, 2:348–368, 1981.
3. K.P. Belkhale and P. Banerjee. Partitionable independent task scheduling problem. In *Proc. of the 1990 International Conference on Parallel Processing (ICPP'90)*, volume 1, pages 72–75, 1990.
4. J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram. Approximation algorithms for scheduling independent malleable tasks. In *Proc. of the 7th European Conference on Parallel Computing (Euro-Par'01)*, pages 191–197, 2001.
5. P. Brucker. *Scheduling Algorithms.* Springer Verlag, 1995.
6. W. Fernandez de la Vega and V. Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Applied Mathematics*, 82:98–101, 1998.
7. T. Decker. *Ein universelles Lastverteilungssystem und seine Anwendung bei der Isolierung reeller Nullstellen.* Dissertation, 2000.
8. T. Decker and W. Krandick. Parallel real root isolation using the descartes method. In *Proc. of the 6th High Performance Conference (HiPC'99)*, pages 261–268, 1999.
9. T. Decker, T. Lücking, and B. Monien. A 5/4-approximation algorithm for scheduling identical malleable tasks. Technical report, tr-rsfb-02-071, University of Paderborn, 2002.
10. J. Du and Y-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.
11. H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.
12. M. Garey and R. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975.
13. K. Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial-time approximation scheme. In *Proc. of the 10th Annual European Symposium on Algorithms (ESA'02)*, pages 562–573, 2002.
14. K. Jansen and L. Porkolab. Linear time approximation schemes for scheduling malleable parallel tasks. *Algorithmica*, 32(3):507–520, 2002.
15. C. Kenyon and E. Remila. Approximate strip packing. In *Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 142–154, 1996.
16. S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangular tiling and packing. In *Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, pages 384–393, 1998.
17. R. Krishnamurti and E. Ma. The processor partitioning problem in special-purpose partitionable systems. In *Proc. of the 1988 International Conference on Parallel Processing (ICPP'88)*, volume 1, pages 434–443, 1988.
18. G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proc. of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 23–32, 1999.
19. G. Mounié, C. Rapine, and D. Trystram. A $\frac{3}{2}$-approximation algorithm for independent scheduling malleable tasks. *submitted for publication*, 2001.
20. A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.
21. J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, pages 323–332, 1992.
22. B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.