Using High Performance Computing for Unrelated Parallel Machine Scheduling with Sequence-Dependent Setup Times: Development and Computational Evaluation of a Parallel Branch-and-Price Algorithm

Gerhard Rauchecker[a,*], Guido Schryen[a]

[a] *Universitaetsstrasse 31, 93053 Regensburg, Germany*
[b] *Warburger Strasse 100, 33098 Paderborn, Germany*

Abstract

Scheduling problems are essential for decision making in many academic disciplines, including operations management, computer science, and information systems. Since many scheduling problems are NP-hard in the strong sense, there is only limited research on exact algorithms and how their efficiency scales when implemented on parallel computing architectures. We address this gap by (1) adapting an exact branch-and-price algorithm to a parallel machine scheduling problem on unrelated machines with sequence- and machine-dependent setup times, (2) parallelizing the adapted algorithm by implementing a distributed-memory parallelization with a master/worker approach, and (3) conducting extensive computational experiments using up to 960 MPI processes on a modern high performance computing cluster. With our experiments, we show that the efficiency of our parallelization approach can lead to superlinear speedup but can vary substantially between instances. We further show that the wall time of serial execution can be substantially reduced through our parallelization, in some cases from 94 hours to less than six minutes when our algorithm is executed on 960 processes.

Keywords: parallel machine scheduling with setup times, parallel branch-and-price algorithm, high performance computing, master/worker parallelization

*Corresponding author
Email addresses: gerhard.rauchecker@ur.de (Gerhard Rauchecker), guido.schryen@upb.de (Guido Schryen)

1. Introduction

In this article, we study the parallel machine scheduling problem on unrelated machines with sequence- and machine-dependent setup times, machine eligibility restrictions, and the total weighted completion time as objective function. This problem is well-known in the scheduling literature and classified as $R/s_{ijk}, M_j/\sum w_j C_j$ (Pinedo, 2012) in the established $\alpha/\beta/\gamma$-notation (Graham et al., 1979). For convenience, we refer to this problem as UPMSP (Unrelated Parallel Machine Scheduling Problem) in the remainder of this article. UPMSP is NP-hard in the strong sense since the more specific problem $P//\sum w_j C_j$ of minimizing the total weighted completion time on identical machines is NP-hard in the strong sense (Skutella & Woeginger, 2000).

The problem can be described as follows: A set of jobs has to be processed on a set of machines, where (i) each job has to be processed exactly once, (ii) the processing of a job must not be interrupted (non-preemption), and (iii) a machine may be capable of processing a job or not (machine eligibility restrictions). Processing times depend on the job and the processing machine, while sequence-dependent setup times depend on the job, the preceding job, and the processing machine. Furthermore, each job has a priority level (weight). The goal of UPMSP is to find a feasible set of machine schedules with minimal total weighted completion time.

There are several real-life settings where decision makers face UPMSP. In disaster response, rescue units (machines) are scheduled to process emergency incidents (jobs) with different priorities (weights). Setups are required by rescue units for traveling between incidents' locations (Wex et al., 2014). Another application are traffic flow networks where repairmen (machines) have to repair broken toll plazas or toll bridges (jobs) with different traffic throughput rates (weights). Setups are represented by travel times of repairmen between toll plazas or bridges (Weng et al., 2001). UPMSP is also found in injection molding departments where machines are used to produce different components (jobs) with certain importance (weights) and for which setup times are required for dies or molds (Chen, 2015).

Solving medium- or large-scale instances of UPMSP - and even of the more specific problem where no machine eligibility restrictions apply and where setup times are not machine-dependent - to optimality is computationally challenging as recent studies show (Arnaout & Rabadi, 2005; Chen, 2015; Rauchecker & Schryen, 2015; Schryen et al., 2015; Tsai & Tseng, 2007; Weng et al., 2001; Wex et al., 2014). In order to overcome efficiency limitations due to sequentially executed algorithms, researchers can rely on recent technological developments in high performance computing (HPC),

which refers to the use of parallel computing architectures. HPC is particularly relevant since speed improvement on a single core is limited because of technological reasons (Hager & Wellein, 2010, p. 23). Modern PCs and even smartphones have multiple cores, which allow for parallel code execution. At the extreme, computer clusters and supercomputers - containing up to several millions of cores - are pushing the boundaries of HPC (TOP500, 2017). HPC has been successfully applied to a broad range of problems in many scientific disciplines, including biology, chemistry, physics, geology, weather forecasting, aerodynamic research, and computer science (Bell & Gray, 2002; Vecchiola et al., 2009) - but there is little research on using HPC for scheduling problems. Nowadays, taking advantage of HPC does not require having access to a supercomputer; it can also be done on computing clusters, which have become commodity IT resources. For example, they are available at many universities and are provided by some cloud providers, for example, as part of the Amazon Web Services (Mauch et al., 2013). To sum up, HPC has not only become technologically feasible but also economically affordable (Hager & Wellein, 2010, p. 1).

However, in order to exploit the capabilities of HPC, algorithms have to be parallelized. In this work, we adapt a serial branch-and-price algorithm, which was suggested by Lopes & de Carvalho (2007), to solve UPMSP. Their algorithm was designed for the parallel machine scheduling problem on unrelated machines with sequence-dependent setup times, machine availability dates, release dates, due dates, and the total weighted tardiness as objective function. We suggest an algorithmic parallelization of the adapted b&p algorithm and conduct extensive computational experiments on an HPC cluster to analyze the scalability of our parallel approach on a large number of cores.

1.1. Literature Review

Scheduling problems appear in many forms and have attracted thousands of research papers. In order to structure this large body of research, comprehensive literature reviews (e.g., Allahverdi et al. (1999, 2008); Allahverdi (2015); Cheng & Sin (1990)) and books (e.g., Brucker (2007); Pinedo (2012); Rabadi (2016)) have been published. We can divide scheduling problems into problems which account for setup times and those which do not. Allahverdi et al. (1999, 2008) and Allahverdi (2015) provide comprehensive surveys about all scheduling problems accounting for setup times. They further divide these into problems with sequence-independent setup times and problems with sequence-dependent setup times. In our overview, we focus on problems that can be represented as $R/ST_{SD}/\gamma$, i.e., scheduling on unrelated parallel machines with sequence-dependent setup times. We

further restrict γ to objective functions that are at least as general as the total weighted completion time.

According to Allahverdi et al. (1999, 2008) and Allahverdi (2015), the first research focusing on this type of problems was conducted by Zhu & Heady (2000), who considered due dates and the total weighted earliness/tardiness ($\sum w'_j E_j + \sum w''_j T_j$) objective function, which is equivalent to the total weighted completion time when all earliness weights and due dates are 0. They modeled $R/ST_{SD}/\sum w'_j E_j + \sum w''_j T_j$ by a mixed-integer program (MIP) and were capable of finding optimal solutions for up to 9 jobs and 3 machines. Akyol & Bayhan (2008) present an exact Artificial Neural Network algorithm but they were not able to solve larger instance sizes. A Tabu Search was presented by Bozorgirad & Logendran (2012) while Zeidi & Mohammad Hosseini (2015) have chosen a hybrid Genetic Algorithm / Simulated Annealing approach.

In the absence of earliness weights, $\sum w'_j E_j + \sum w''_j T_j$ turns into the total weighted tardiness objective function ($\sum w_j T_j$), which is equivalent to the total weighted completion time when all due dates are 0. Tavakkoli-Moghaddam & Aramon-Bajestani (2009), Lopes & de Carvalho (2007), and Lopes et al. (2014) present branch-and-bound (b&b) and branch-and-price (b&p) algorithms based on MIP formulations for $R/ST_{SD}/\sum w_j T_j$, with the b&b algorithm being capable of solving instances with up to 10 jobs and 4 machines and the b&p algorithms being capable of solving instances with up to 180 jobs and 50 machines. The total weighted tardiness objective was further tackled with Genetic Algorithms (Joo & Kim (2012)), Tabu Search (Logendran et al. (2007)), Simulated Annealing (Kim et al. (2003)), and several other heuristic approaches (Alvelos et al. (2016); de Paula et al. (2010); Lin & Hsieh (2014); Rauchecker & Schryen (2015); Zhang et al. (2007)). Our problem $R/ST_{SD}/\sum w_j C_j$ has been formulated as a quadratic binary program and tested with the off-the-shelf solver *Gurobi* by Wex et al. (2014) and Schryen et al. (2015). Their computational studies indicate that this formulation and strategy is not efficient as it fails to compute optimal solutions for small-sized instances consisting of 40 jobs and 10 machines within several hours. Other approaches approximate this problem with Genetic Algorithms (Tsai & Tseng, 2007), b&p based heuristics (Rauchecker & Schryen, 2015), and several problem-specific heuristics (Arnaout & Rabadi, 2005; Chen, 2015; Weng et al., 2001).

The common problem of (meta-) heuristic approaches is the fact that they usually cannot guarantee optimality. Consequently, efficient exact algorithms are not only desirable for solving large real-world problems to optimality, but also they are an important tool for benchmarking (meta-)

heuristics. However, our literature review shows that (i) in contrast to heuristics, only very few research on exact algorithms for UPMSP and related problems exists and (ii) none of the suggested exact algorithms has been parallelized in order to leverage the potential of modern HPC capabilities. However, the most promising of the presented exact approaches use b&b and especially b&p algorithms, which offer a high potential for parallelization (e.g., Eckstein (1994), Migdalas et al. (2013), or Ralphs et al. (2003)). B&p algorithms were conceptualized by Barnhart et al. (1998) and use both a b&b algorithm and a column generation procedure (Dantzig & Wolfe, 1960; Desaulniers et al., 2006; Lübbecke & Desrosiers, 2005) to solve integer programs with many variables. This kind of algorithm has widely been used for tackling scheduling problems, including studies by Bard & Rojanasoonthon (2006), Fei et al. (2008), van den Akker et al. (1999), and Chen & Powell (1999, 2003). However, only a few studies use parallel implementations of b&b or b&p algorithms to tackle scheduling problems, see, for instance, Perregaard & Clausen (1998), Clausen & Perregaard (1999), Crespo Abril & Maroto Alvarez (2005), Aitzai & Boudhar (2013), and Chakroun et al. (2013). Just recently, a new parallel b&p-based heuristic for UPMSP has been suggested (Rauchecker & Schryen, 2015). We contribute to closing the aforementioned research gap by suggesting and computationally validating an exact parallel b&p algorithm for UPMSP as outlined in Section 1.2.

1.2. Contribution and Structure

In Section 2, we present the mathematical formulation of our scheduling problem. In Section 3, we propose an adaptation of the serial b&p algorithm suggested by Lopes & de Carvalho (2007) to UPMSP. Section 4 presents a parallelized version of the adapted b&p algorithm. To the best knowledge of the authors, this is the first time that an exact b&p algorithm for a scheduling problem has been parallelized. In Section 5, we demonstrate the applicability of the parallelized algorithm on a Linux-based HPC cluster with extensive numerical experiments and measure its performance using established scalability metrics. With our experiments, we show that our parallelization approach achieves high efficiencies with even superlinear speedups for some instances. We further show that the wall time of our tested instances is reduced from up to 94 hours for the most difficult test instance in serial execution to less than six minutes when the algorithm is executed on 960 processes. We discuss those finding is Section 6 before we finally conclude in Section 7.

2. Problem Formulation

We presented an overview of articles on problems which are at least as general as UPSMP in Section 1.1. Regarding exact algorithms, the computational results from Lopes & de Carvalho (2007) and Lopes et al. (2014) for the scheduling problem $R/ST_{SD}/\sum w_jT_j$ are most promising in terms of efficiency and, in addition, they solve their model by a highly parallelizable b&p algorithm. Consequently, we adapt their binary linear formulation to UPMSP. We introduce the notation for our formulation in the following while keeping all notations from the introduction.

Let $\{1,\dots,n\}$ be the set of jobs and $\{1,\dots,m\}$ be the set of machines. For a job j, let M_j denote the set of machines that are capable of processing j. A schedule $\omega = (j_1,\dots,j_h)$, with $0 \leq h \leq n$, is a tuple of pairwise different jobs $j_1,\dots,j_h$. A schedule $\omega = (j_1,\dots,j_h)$ is feasible for a machine k if and only if $k \in M_{j_l}$ for all $l = 1,\dots,h$. The tuple represents the order in which the jobs $j_1,\dots,j_h$ are processed on machine k. The set of all feasible schedules for machine k is denoted by Ω^k. The parameter $a_{j\omega} \in \{0,1\}$ represents the number of times job j is contained in schedule ω. We further denote by c_ω^k the weighted completion time of a schedule ω when it is processed on machine k. For each machine k and each schedule $\omega \in \Omega^k$, we introduce a binary decision variable x_ω^k being 1 if ω is operated on k and 0 otherwise. Then, we can formulate UPMSP as a binary linear program:

$$\min \quad \sum_{k=1}^{m} \sum_{\omega\in\Omega^k} c_\omega^k \cdot x_\omega^k \tag{1}$$

$$\text{s.t.} \quad \sum_{k=1}^{m} \sum_{\omega\in\Omega^k} a_{j\omega} \cdot x_\omega^k = 1 \quad \forall j = 1,\dots,n \tag{2}$$

$$\sum_{\omega\in\Omega^k} x_\omega^k \leq 1 \qquad \forall k = 1,\dots,m \tag{3}$$

$$x_\omega^k \in \{0,1\} \qquad \forall k = 1,\dots,m; \omega \in \Omega^k \tag{4}$$

The objective function (1) represents the total weighted completion time of a tuple $(\omega^1,\dots,\omega^m)$ of schedules with $\omega^k \in \Omega^k$ for all machines k. Constraints (2) guarantee that each job is processed exactly once while constraints (3) ensure that each machine is assigned at most one schedule. Note that since our definition allows for empty schedules as well, it would be more natural to replace the inequality in (3) by an equality resulting in exactly one schedule per machine. However, the inequality formulation is more useful for b&p algorithms (Section 3) and leads to the same optimal solution since it makes no difference whether a machine operates no schedule or the empty schedule.

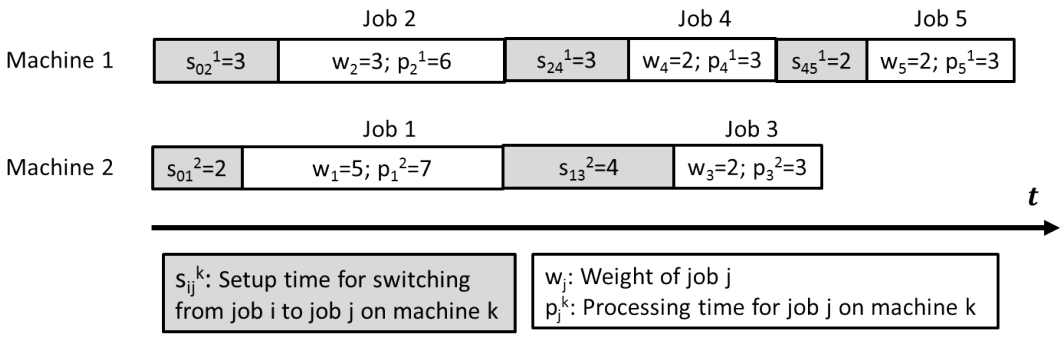


Figure 1: Sample solution for UPMSP

In our formulation, the processing time p_j^k of a job j on a machine k, the setup time s_{ij}^k between jobs i and j on a machine k, and the weight w_j of a job j are included in the weighted completion time c_ω^k of a schedule $\omega = (j_1, \ldots, j_h) \in \Omega^k$ on a machine k. The relationship can be described as

$$c_\omega^k = \sum_{l=1}^{h} w_{j_l} \cdot \left(\sum_{r=1}^{l} s_{j_{r-1} j_r}^k + p_{j_r}^k \right), \tag{5}$$

where s_{0j}^k is the initial setup time for machine k to process job j.

A graphical visualization of a sample solution for UPMSP with two machines and five jobs is provided in Figure 1. Here, machine 1 processes job 2 first, then job 4 and finally job 5. Furthermore, machine 2 processes job 1 before job 3. This translates to $x_{(2,4,5)}^1 = 1$ and $x_{(1,3)}^2 = 1$. The weighted completion times can be calculated as $c_{(2,4,5)}^1 = 3 \cdot (3+6) + 2 \cdot (3+6+3+3) + 2 \cdot (3+6+3+3+2+3) = 97$ and $c_{(1,3)}^2 = 5 \cdot (2+7) + 2 \cdot (2+7+4+3) = 77$ leading to a total weighted completion time of $\sum_{k=1}^{2} \sum_{\omega \in \Omega^k} c_\omega^k \cdot x_\omega^k = 97 + 77 = 174$.

There are three minor differences of our formulation to the model presented by Lopes & de Carvalho (2007). First, the structure of our sets Ω^k is different since machine eligibility restrictions are taken into account. Second, we use the total weighted completion time as objective function instead of the total weighted tardiness. Third, we take care of the machine-dependence of setup times in equation (5).

It should be noted that problem formulation (1)-(4) includes a low number of $n + m$ constraints but a high number of variables. Assuming that the probability with which an arbitrary machine is capable of processing an arbitrary job is given by $\theta \in [0, 1]$, the expected number of variables

$\sum_{k=1}^{m} |\Omega^k|$ is bounded by

$$m \cdot \lfloor \theta n \rfloor ! \cdot \sum_{r=0}^{\lfloor \theta n \rfloor} \frac{1}{r!} \leq \sum_{k=1}^{m} |\Omega^k| \leq m \cdot \lceil \theta n \rceil ! \cdot \sum_{r=0}^{\lceil \theta n \rceil} \frac{1}{r!} \tag{6}$$

and thus grows linearly with m and factorially with n. An appropriate mechanism to address a high number of variables in linear programs is column generation (Dantzig & Wolfe, 1960), which is used in the b&p algorithm presented in Section 3.

3. Serial Branch-And-Price Algorithm for UPMSP

In this section, we present and discuss an adaptation of a b&p algorithm - introduced by Lopes & de Carvalho (2007) for the problem $R/ST_{SD}/\sum w_j T_j$ - to UPMSP. A b&p algorithm is a specific form of a b&b algorithm where all linear relaxations are solved by a column generation procedure (Dantzig & Wolfe, 1960). A first overview on solving integer programs (IPs) with b&p algorithms was provided by Barnhart et al. (1998). A high-level pseudo code of a b&p algorithm that solves general IPs is presented in Algorithm 1.

Algorithm 1 B&p algorithm (Barnhart et al., 1998)

```
solve linear relaxation of root node using column generation
initialize set of active nodes
repeat
    select an active node
    branch on selected node
    solve new nodes' linear relaxations using column generation
    update set of active nodes
until set of active nodes is empty
```

When solving an UPMSP instance, the root node of the b&b tree is the problem formulated in (1)-(4). In the linear relaxation, constraints (4) are relaxed to $0 \leq x_\omega^k \leq 1$ for all $k = 1, \ldots, m$ and $\omega \in \Omega^k$. The relaxation is solved by column generation, which is described in Section 3.2. If the relaxation has an integer optimal solution, this solution is also optimal for UPSMP. Otherwise, the set of active nodes is initialized with the root node.

After selecting an active node, the branching on the selected node is conducted by constructing two child nodes. Both the node selection and the branching rule is explained in Section 3.1. The parent node is removed from the set of active nodes and the two child nodes are added. After branching, the linear relaxations of both child nodes are again solved using column generation. At

the end of each iteration of the *repeat* loop, the set of active nodes of the b&b tree is updated based on the optimal solutions of both child nodes' relaxations - depending on whether they have fractional, integer, or no optimal solutions (infeasibility). In this procedure, an active node is marked as inactive when (i) its relaxation has an integer optimal solution, (ii) it has no feasible solution at all, or (iii) the optimal solution of its relaxation is higher than the best integer optimal solution from any node solved so far. The latter is called *bounding* (note that in a minimization problem the optimal solution of a node's relaxation is a lower bound for the optimal integer solution of the node). Finally, when there are no more active nodes left, the current best integer solution is an optimal solution for UPMSP.

Algorithm 1 has also been used by Lopes & de Carvalho (2007). Their problem turns into UPMSP if we (i) specify all release dates, due dates, and machine availability dates to be 0, (ii) allow setup times to be machine dependent, and (iii) add machine eligibility restrictions. We present the specifications of the algorithm and our adaptations to UPMSP in the rest of this section.

3.1. Node Selection and Branching Rule

There are two main node selection strategies for branch-and-bound algorithms: depth-first search and best-first search (Clausen & Perregaard, 1999). While depth-first search is suitable to find good feasible solutions early (Rauchecker & Schryen, 2015), it is clearly inferior to best-first search when it comes to finding optimal solutions as our pretests showed. Our pretests also showed that a combination of depth-first and best-first search (finding a good feasible solution with depth-first before improving it with best-first) is still inferior to a pure best-first strategy. Therefore, our node selection strategy is best-first search (also called *best lower bound rule*), in which we select the active node with the lowest lower bound being the next node to be explored.

As suggested by Lopes & de Carvalho (2007), the branching on a selected node is conducted along variables X_{ij}^k which are defined as

$$X_{ij}^k = \sum_{\omega \in \Omega^k} \delta_{ij\omega} \cdot x_\omega^k \leq \sum_{\omega \in \Omega^k} a_{j\omega} \cdot x_\omega^k \overset{(2)}{\leq} 1 \tag{7}$$

for every $i = 0, \ldots, n$, $j = 1, \ldots, n$, and $k = 1, \ldots, m$, where $\delta_{ij\omega} \in \{0, 1\}$ is the number of times job i is processed directly before job j in schedule ω and $(x_\omega^k)_{k=1,\ldots,m;\omega\in\Omega^k}$ is an optimal solution of the selected node's relaxation. Here, $\delta_{0j\omega}$ is set to 1 if job j is processed first in schedule ω and 0 otherwise. If x_ω^k is binary for all machines k and schedules $\omega \in \Omega^k$, then X_{ij}^k indicates whether

job i is processed directly before job j on machine k or not (i.e., branching corresponds to adding job ordering restrictions to the child nodes). Consequently, X^k_{0j} indicates whether job j is processed first on machine k or not.

Equation (7) is used to calculate the branching variable. We branch on the variable $X^{k^*}_{i^*j^*}$ with the largest integer infeasibility, i.e., (i^*, j^*, k^*) is the argument of

$$\min_{i=0,\ldots,n;j=1,\ldots,n;k=1,\ldots,m} |X^k_{ij} - 0.5|. \tag{8}$$

In the first child node, $X^{k^*}_{i^*j^*}$ is set to one, i.e., i^* has to be processed by k^* directly before j^*. In the second child node, $X^{k^*}_{i^*j^*}$ is set to zero, i.e., i^* is forbidden to be processed by k^* directly before j^*. All sets Ω^k of feasible schedules on machine k are updated accordingly. Consequently, all nodes in the b&b tree are of structure (1)-(4) with different node-specific sets Ω^k.

3.2. Column Generation Process

In this section, we adapt the column (i.e., variable) generation procedure for solving the linear relaxations of each b&b node as suggested by Lopes & de Carvalho (2007) to our problem. According to Section 3.1, all nodes in the b&b tree are of structure (1)-(4) with different node-specific sets of feasible schedules Ω^k on each machine k. We refer to a node's linear relaxation as the *original LP* in this section.

The column generation procedure is described by Algorithm 2. The pricing problem (step 2 in Algorithm 2) as part of the column generation procedure is solved by a dynamic programming algorithm (Algorithm 3) which we adapt from Lopes & de Carvalho (2007) and outline in the remainder of this section. The dynamic programming algorithm may return schedules that process jobs more than once. To ensure that generated schedules are still elements of Ω^k, we artificially enlarge Ω^k by allowing jobs to be processed arbitrarily often (or alternatively - to keep Ω^k finite - a very large number of times) in each schedule. Consequently, the ranges of $a_{j\omega}$ and $\delta_{ij\omega}$ for jobs i, j and a schedule ω are no longer binary. Note that this does only affect optimal solutions of linear relaxations and not the optimal solution for UPMSP because constraint (2) forces each feasible solution of the integer model (1)-(4) to process each job exactly once.

For a machine k and a job j, let P^k_j be the set of all possible predecessor jobs of j on machine k. These sets are node-specific and depend on machine eligibility and in particular on the job ordering restrictions induced by the branching history (Section 3.1). For a machine k, a time $t \in \mathbb{Z}$, and a

Algorithm 2 Column Generation Procedure

1: Solve a restricted form of the original LP by considering only a (typically small) subset of variables, i.e., setting all other variables to 0. We refer to this as the *restricted LP*. The initial set of variables can be obtained by a solution heuristic (for the root node) or be adopted from the parent node (for all other nodes in the b&b tree).

2: Let (π, σ) denote the optimal dual solution of the restricted LP from step 1, i.e., π_j is the dual variable corresponding to job j in constraint (2) and σ_k is the dual variable corresponding to machine k in constraint (3). Determine if there is any variable x_ω^k in the original LP that has a negative reduced cost

$$r_\omega^k = c_\omega^k - \sum_{j=1}^{n} a_{j\omega}\pi_j - \sigma_k \tag{9}$$

with respect to the optimal dual solution of the restricted LP. This is equivalent to solving

$$r^* = \min_{k=1,\ldots,m} \min_{\omega \in \Omega^k} c_\omega^k - \sum_{j=1}^{n} a_{j\omega}\pi_j - \sigma_k \tag{10}$$

and is called the *pricing problem*. For algorithmic details, see Algorithm 3.

3: If there is a variable with negative reduced cost in step 2 (i.e., $r^* < 0$), add one (or a fixed number of) variable(s) x_ω^k with least reduced cost(s) to be additionally considered (i.e., these variable(s) is (are) not fixed to 0 anymore) in the restricted LP and go back to step 1. Otherwise, an optimal solution of the restricted LP is also an optimal solution of the original LP, which corresponds to the linear relaxation of that b&b node.

job j, we define $f^k(t,j)$ as the minimum reduced cost of all variables x^k_ω where ω finishes processing exactly at time t and processes j last. We set a time limit $T \geq 1$ and use a recursive procedure to calculate those minimum reduced costs for all $t \leq T$, see Algorithm 3.

Algorithm 3 Solving the Pricing Problem

1: Initialize $f^k(t,j) = \infty$ for each machine k, time $t \leq 0$, and job j.

2: For each machine k, initialize $f^k(0,0) := -\sigma_k$ and $f^k(t,0) = \infty$ for each time $0 \neq t \leq T$.

3: For each machine k, time $1 \leq t \leq T$, and job j, set

$$f^k(t,j) = \min_{i \in P^k_j} f^k(t - s^k_{ij} - p^k_j, i) + w_j t - \pi_j. \tag{11}$$

4: The *minimum reduced cost under the value* T is defined as

$$r^*_T = \min_{k=1,\ldots,m} \min_{t=0,\ldots,T} \min_{j=0,\ldots,n} f^k(t,j). \tag{12}$$

A schedule corresponding to a variable with minimum reduced cost r^*_T under the value of T can be determined by reversing the recursion path. The complexity of Algorithm 3 is bounded by $\mathcal{O}(mn^2T)$ since taking the minimum in equation (12) requires at most mnT calculations of $f^k(t,j)$, which in turn requires $\mathcal{O}(n)$ elementary calculations each. Therefore, its performance depends heavily on the time quantity T. Lopes & de Carvalho (2007) argue that it is possible to start with a low value of T (for example the makespan of a heuristic solution) and iteratively adjust T by adding

$$T_{max} := \max_{i=0,\ldots,n} \max_{j=1,\ldots,n} \max_{k=1,\ldots,m} s^k_{ij} + p^k_j. \tag{13}$$

If there are neither variables with negative reduced cost under a value of T nor under the value of $T + T_{max}$, then there are no more variables with negative reduced cost under any value of T, which implies $r^* \geq 0$ in (10) and terminates the column generation procedure. This keeps T in (12) relatively low and also works in our adapted version.

Lopes & de Carvalho (2007) also argue that when setup times fulfill the triangle inequality, it is sufficient to focus only so called *decreasing reduced cost variables* during column generation, i.e., it is sufficient to only take into account jobs j with $t \geq \frac{\pi_j}{w_j}$ when solving equation (12). In addition to their result, we prove that it is also sufficient to focus on decreasing reduced cost variables whenever all setup times are lower than any processing times. Definitions and a detailed proof can be obtained from Appendix A.

Our algorithms in this section differ in two ways from the procedure suggested by Lopes & de Carvalho (2007). First, the definition of negative reduced costs in equation (9) depends on the total weighted completion time instead of the total weighted tardiness and second, we account for machine-dependence of setup times in Algorithm 3. These adaptations are necessary to make the procedure of Lopes & de Carvalho (2007) applicable to our problem UPMSP.

4. Parallelization of the Branch-and-Price Algorithm

In this section, we explain how we parallelize our b&p algorithm from Section 3. According to Gendron & Crainic (1994), there are three types of parallelism for b&b algorithms: First, the solution of single b&b nodes can be executed in parallel (type 1). Second, the b&b tree itself can be parallelized by solving concurrently active nodes of the b&b tree simultaneously (type 2). Third, multiple b&b trees can be built and explored in parallel (type 3). It is also possible to combine those types. In our pretests, we focus on parallelization within one single b&b tree, i.e., we exclude parallelism of type 3. We found that parallelism of type 1 is not a promising approach for effectively reducing execution times of our b&p algorithm, neither when being applied as sole parallelism nor when being used jointly with type 2 parallelism. A detailed reasoning can be obtained from Appendix B. Consequently, we use only parallelism of type 2 for parallelizing our b&p algorithm, i.e., we solve concurrently active b&b tree nodes in parallel.

Algorithm 4 Code of master process

```
1: solve linear relaxation of root node
2: initialize set of active nodes
3: repeat
4:     if not all active nodes are at workers then
5:         increase loop counter by +1
6:         select an active node (if loop counter is odd)
7:         branch on selected node (if loop counter is odd)
8:     endif
9:     communicate with an idle worker process by
10:        receiving a solved node from worker process (if available)
11:        sending child node 1 to worker process (if available and loop counter is odd)
12:        sending child node 2 to worker process (if available and loop counter is even)
13:    update set of active nodes
14: until set of active nodes is empty
15: send termination flag to all worker processes
```

We use a centralized master/worker approach to enable the simultaneous solving of concurrently active nodes of the b&b tree on different processes. With regard to the serial version of the b&p

Algorithm 5 Code of worker processes

```
1: repeat
2:     communicate with master process by
3:         sending a solved node to the master process (if available)
4:         receiving a new node from the master process (if available)
5:     solve linear relaxation of received node (if received a node)
6: until termination flag is received
```

algorithm as shown in Algorithm 1, we parallelize the execution of lines 3-8. The master process is responsible for the tree management and executes Algorithm 4, while the worker processes are responsible for solving the linear relaxations of problems at the b&b tree nodes and execute Algorithm 5. After each branching, the master process sends the two child problems to different worker processes via message passing. When a worker process receives a problem from the master process, it solves the node's relaxation and sends the solution back to the master process via message passing. Note that one iteration of the *repeat* loop in Algorithm 1 (lines 3 to 8) corresponds to two iterations of the *repeat* loop in Algorithm 4 (lines 3 to 14). Therefore, we use a loop counter since selecting an active node and branching on the selected node (lines 6 and 7 in Algorithm 4) is only necessary in every second iteration.

5. Computational Experiments

In this section, we report the results of an extensive computational study to test the efficiency of the parallelized b&p algorithm from Section 4 when it is executed using multiple processes in an HPC environment.

5.1. HPC Environment

Our experiments were conducted on the Linux-based CLX-MPI cluster of the RWTH Aachen University, which consists of 600 network-connected computing nodes. Each node is a two-socket Intel Broadwell EP E5-2650v4 shared-memory system with 12 cores per socket and a clock speed of 2.2 GHz per core. This is a standard architecture in modern high performance systems.

The b&p algorithm was coded in C++ and we used the *mpicxx* wrapper compiler with optimization flag *-O3*. The *Gurobi 8* API is used to solve the restricted linear programs during column generation (step 1 of Algorithm 2), and *MPI 3* is used for passing messages between the master process and its worker processes (lines 9-12 in Algorithm 4 and lines 2-4 in Algorithm 5). Each MPI

Table 1: Instance sizes (n, m) tested in our computational evaluation

$\frac{n}{m} = 10$	$(100, 10)$	$(200, 20)$	$(300, 30)$	-	-	-	-
$\frac{n}{m} = 7.5$	$(75, 10)$	$(150, 20)$	$(225, 30)$	$(300, 40)$	-	-	-
$\frac{n}{m} = 5$	$(50, 10)$	$(100, 20)$	$(150, 30)$	$(200, 40)$	$(250, 50)$	$(375, 75)$	$(500, 100)$
$\frac{n}{m} = 2.5$	$(25, 10)$	$(50, 20)$	$(75, 30)$	$(100, 40)$	$(125, 50)$	$(188, 75)$	$(250, 100)$
$\frac{n}{m} = 1$	$(10, 10)$	$(20, 20)$	$(30, 30)$	$(40, 40)$	$(50, 50)$	$(75, 75)$	$(100, 100)$
	$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	$m = 75$	$m = 100$

process is bound to its own core. To enable scalability of our master/worker approach on many processes, it is crucial to keep the activity of the master process at a minimum. This demands for a subtle implementation, which is presented in detail in Appendix C.

5.2. Instance Generation and Algorithm Parametrization

As it is common practice in the literature, our evaluation is based on simulated data. We investigated different instances sizes where the number of machines m is set to $m \in \{10, 20, 30, 40, 50, 75, 100\}$ and where the ratio of the number of jobs n to the number of machines m is set to $\frac{n}{m} \in \{1, 2.5, 5, 7.5, 10\}$. For each instance size, we randomly generated and solved five different instances. An overview of our instance sizes is given in Table 1. For example, the entry $(375, 75)$ in the column of $m = 75$ and the row of $\frac{n}{m} = 5$ represents an instance size with $n = 375$ jobs on $m = 75$ machines. [1] Combinations with no entry turned out to be too complex to be investigated in our computational environment as we will see in the pretest section 5.3.

We generated processing times p_j^k, setup times s_{ij}^k, and job weights w_j using the following distributions: $p_j^k \sim U(10, 100, 1)$, $s_{ij}^k \sim U(1, 10, 1)$, and $w_j \sim U(1, 10, 1)$. We set the probability θ of a machine being capable to process a certain job to 0.2. Similar settings have been used in the literature (Lopes & de Carvalho, 2007; Schryen et al., 2015; Wex et al., 2014). Note that our random distributions assure all setup times to be at most as high as any processing times. This guarantees that all assumptions of Theorem 2 (Appendix A) are fulfilled.

[1] In the column of $m = 75$ and the row of $\frac{n}{m} = 2.5$, we obtain an instance size of $n = 187.5$ jobs on $m = 75$ machines. To obtain an integer number of jobs, we round up to $n = 188$ jobs on $m = 75$ machines.

Table 2: Maximum wall time in seconds on 24 processes during pretests

$\frac{n}{m} = 10$	$26_{(5)}$	$5848_{(5)}$	$79165_{(1)}$	-	-	-	-
$\frac{n}{m} = 7.5$	$< 10_{(5)}$	$37_{(5)}$	$74552_{(5)}$	$22508_{(3)}$	-	-	-
$\frac{n}{m} = 5$	$< 10_{(5)}$	$< 10_{(5)}$	$130_{(5)}$	$91_{(5)}$	$861_{(5)}$	$10375_{(5)}$	$60936_{(1)}$
$\frac{n}{m} = 2.5$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$
$\frac{n}{m} = 1$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$	$< 10_{(5)}$
	$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	$m = 75$	$m = 100$

We generated 20 columns in each iteration of the column generation procedure. [2] As an initial set of variables for the restricted LP of the root node (step 1 of Algorithm 2), we applied a greedy heuristic (Schryen et al., 2015), which provides a solution in fractions of a second. As an initial set of variables for all other nodes of the b&b tree, we used all columns that were generated for the solution of the parent node - except those that have become infeasible because of new job ordering restrictions induced by branching on the parent node.

5.3. Pretests

To efficiently utilize our computation quota at the RWTH Aachen University cluster and to predict the size of instances which can be solved with serial execution (i.e., on one process) in reasonable time, we have conducted pretests to gain insights into the behavior of our instances. We have solved all five instances of all instance sizes from Table 1 on 24 processes (one computing node) with a wall time limit of 24 hours. [3] Table 2 lists the maximum wall time for each instance size. The maximum is taken among all instances that terminated sucessfully within the time limit. The subscripts in brackets represent the number of instances of an instance size that did terminate successfully within the time limit.

For the computational evaluation of the scalability of our parallel algorithm, we excluded those

[2] Pretests showed that this is a sweet spot. The wall times of the serial algorithm tend to decrease when the number of generated columns per iteration is increased. However, further increasing this number above 20 does not lead to a substantial decrease of wall times anymore. In order to save memory and communication time, we fixed the number of generated columns per iteration to 20.

[3] By *wall time* we refer to the elapsed time from the beginning of algorithm execution until the termination of the algorithm.

instance sizes with a maximum wall time of less than 10 seconds since there is little to no potential to decrease wall times using a higher number of processes. This criterion applies to instance sizes $(75, 10)$, $(50, 10)$, $(100, 20)$, and all instance sizes with $\frac{n}{m} \in \{1, 2.5\}$. The instances of the remaining instance sizes are divided into two subsets for two different types of computational experiments. The first subset consists of instances that we will, in particular, solve in serial mode and the second subset consists of more difficult instances that we will not solve in serial mode. The detailed selection of the subsets for both types of computational experiments is explained in the following.

For the first type, we excluded all single instances which did not terminate successfully within the time limit or have a predicted serial wall time of more than 120 hours (wall time limit of the RWTH Aachen University cluster). [4] This applies to all five instances of the instance sizes $(300, 30)$ and $(500, 100)$, to three instances of instance size $(300, 40)$ and to one instance of instance size $(225, 30)$. In summary, our computational evaluation of the first type investigates the following instance sizes: $(100, 10)$, $(200, 20)$, $(150, 20)$, $(225, 30)$, $(300, 40)$, $(150, 30)$, $(200, 40)$, $(250, 50)$, $(375, 75)$. The respective instances are solved on one process (serial execution), 24 processes (1 computer node), 120 processes (5 computing nodes), 240 processes (10 computing nodes), 480 processes (20 computing nodes), 720 processes (30 computing nodes), and 960 processes (40 computing nodes) to investigate scalability on a high number of processes.

For the second type, we investigate those four single instances, which successfully terminated within 24 hours but which have a predicted serial wall time of more than 120 hours. This is exactly one instance from each of the instance sizes $(300, 30)$, $(225, 30)$, $(300, 40)$, and $(500, 100)$. We solved those instances on 24 processes (1 computer node), 120 processes (5 computing nodes), 240 processes (10 computing nodes), 480 processes (20 computing nodes), 720 processes (30 computing nodes), and 960 processes (40 computing nodes).

5.4. Results

First, we present the results for our computational evaluation of the first type. The average serial wall times (in seconds) of all instance sizes are presented in Table 3. The numbers in brackets show the coefficients of variation (CVs). The average numbers of explored b&b tree nodes (including CVs) in serial execution are shown in Table D.8 in Appendix D. To show to which extent parallel

[4] For predicing serial wall times, we assume a linear slowdown from execution on 24 processes to serial execution.

Table 3: Average wall times in seconds for serial computation (CVs in brackets)

$\frac{n}{m} = 10$	98 (1.5)	86077 (1.5)	-	-	-	-	-
$\frac{n}{m} = 7.5$	-	125 (1.2)	6436 (1.4)	74393 (0.9)	-	-	-
$\frac{n}{m} = 5$	-	-	567 (1.9)	463 (1.3)	3765 (1.8)	97356 (1.2)	-
	$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	$m = 75$	$m = 100$

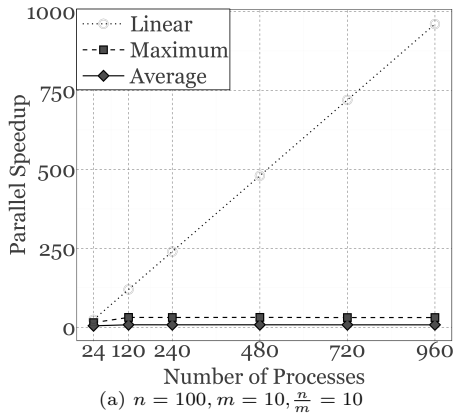
execution is capable of reducing wall times of the algorithm, we present results on the parallel speedup and the parallel efficiency of our master/worker approach (Hager & Wellein, 2010, pp. 123–126). Parallel speedup on R processes is defined as the ratio of the serial wall time to the wall time on R processes. Parallel efficiency on R processes is defined as the parallel speedup on R processes divided by the number of processes R. The parallel speedups are presented in Figures 2 and 3. The parallel efficiencies are presented in Figures D.7 and D.8 in Appendix D. The average parallel speedup is taken over all instances of an instance size and is represented by solid lines. The maximum parallel speedup is the speedup achieved for the best-scaling instance of an instance size and is represented by dashed lines. [5] The dotted lines are reference lines for so called *linear speedup*, i.e., a speedup of R on R processes (equivalent: with an efficiency of 100%).

To understand the algorithmic potential for parallelizing the b&p tree, we present for each instance size the average and minimum (among all instances of an instance size) share of the time to solve the root node from the total wall time in serial mode in Table 4. Since the root node represents the non-parallel part of our master/worker approach, it is also refered to as *serial part*. Another indicator for the scalability potential of our master/worker approach is the mean number of concurrently active nodes during algorithm execution in serial mode, which is presented in Table 5, since at any time there can only be as many busy worker processes as there are concurrently active nodes. [6] An important factor for the scalability of master/worker approaches is the utilization of the master process, which we present in D.9 and D.10 in Appendix D, since high contention at the master can cause waiting times for workers when they request a new node to solve.

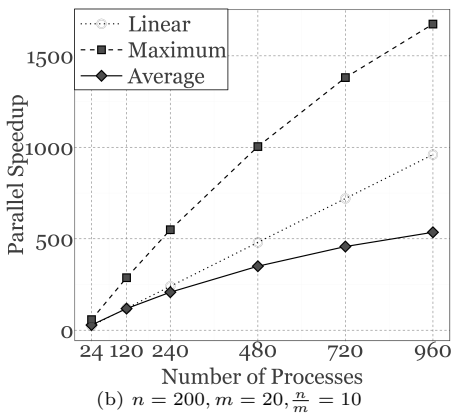
For those four instances where an optimal solution has been found in our pretests within 24

[5] For each instance size, the best-scaling instance was the same instance on any number of processes.

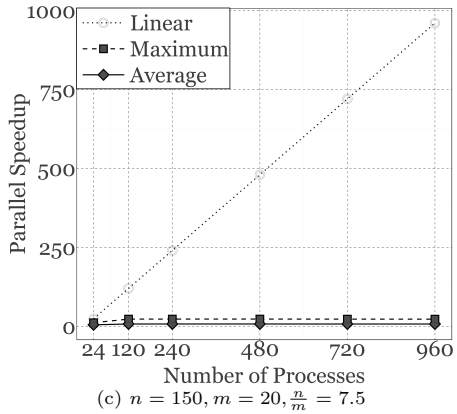
[6] Note that the *mean number of concurrently active nodes* refers to a single instance. It refers to the mean number of nodes that are active at the end of every iteration of the *repeat* loop in Algorithm 1.



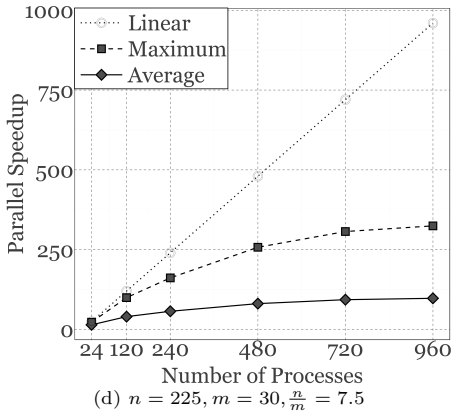
(a) $n = 100, m = 10, \frac{n}{m} = 10$



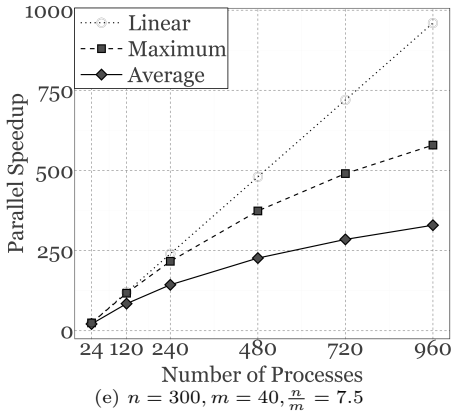
(b) $n = 200, m = 20, \frac{n}{m} = 10$



(c) $n = 150, m = 20, \frac{n}{m} = 7.5$

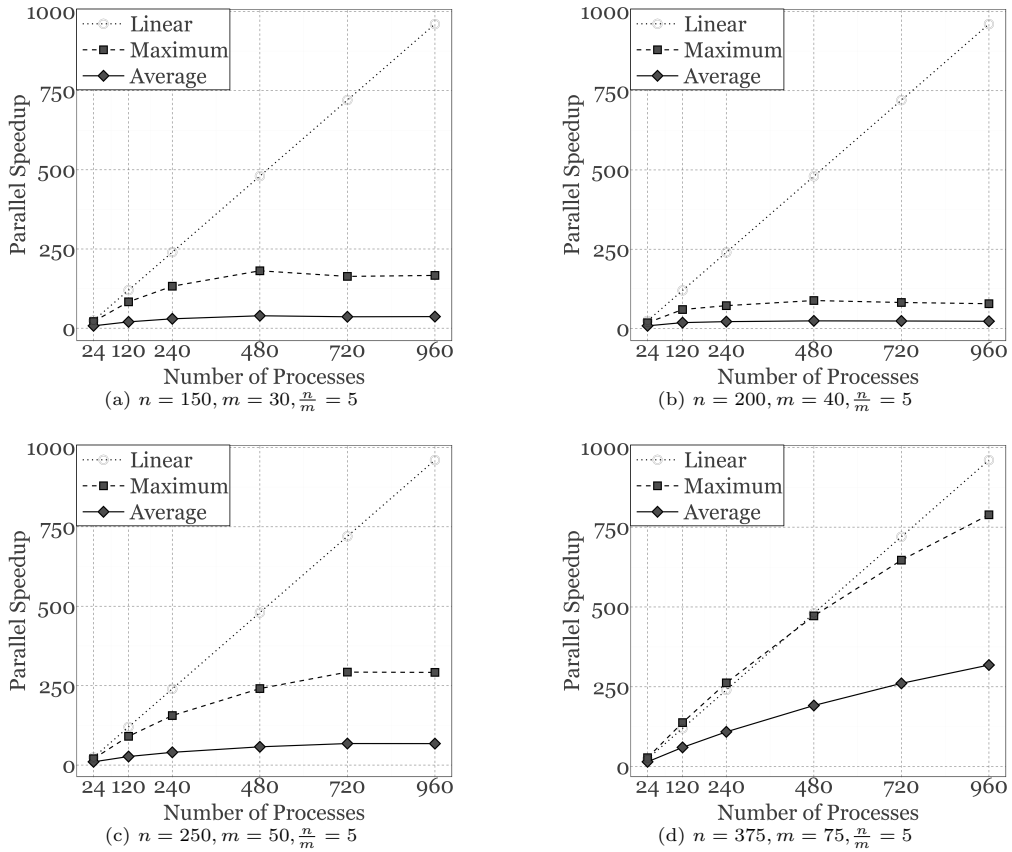


(d) $n = 225, m = 30, \frac{n}{m} = 7.5$



(e) $n = 300, m = 40, \frac{n}{m} = 7.5$

Figure 2: Parallel speedups part 1



(a) $n = 150, m = 30, \frac{n}{m} = 5$ (b) $n = 200, m = 40, \frac{n}{m} = 5$

(c) $n = 250, m = 50, \frac{n}{m} = 5$ (d) $n = 375, m = 75, \frac{n}{m} = 5$

Figure 3: Parallel speedups part 2

Table 4: Share of root node time from total wall time in serial mode

$\frac{n}{m} = 10$	Avg	34.2%	1.0%	-	-	-	-	-
	(Min)	(1.2%)	(0.01%)					
$\frac{n}{m} = 7.5$	Avg	-	33.4%	2.7%	0.4%	-	-	-
	(Min)		(2.2%)	(0.1%)	(0.05%)			
$\frac{n}{m} = 5$	Avg	-	-	12.4%	25.9%	15.4%	18.0%	-
	(Min)			(0.2%)	(0.5%)	(0.1%)	(0.02%)	
		$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	$m = 75$	$m = 100$

Table 5: Mean number of concurrently active nodes during algorithm execution in serial mode

$\frac{n}{m} = 10$	Avg	32	2435	-	-	-	-	-
	(Max)	(139)	(8105)					
$\frac{n}{m} = 7.5$	Avg	-	27	603	4000	-	-	-
	(Max)		(97)	(2118)	(7060)			
$\frac{n}{m} = 5$	Avg	-	-	211	171	1255	6392	-
	(Max)			(985)	(663)	(5957)	(18101)	
		$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	$m = 75$	$m = 100$

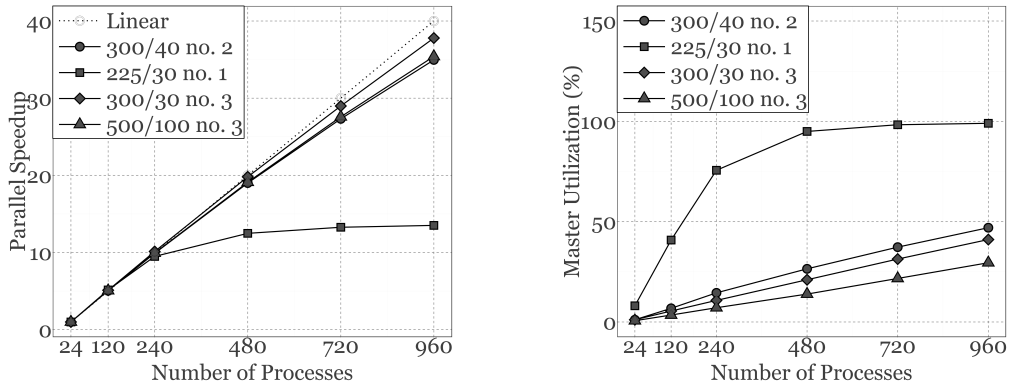


Figure 4: Parallel speedups (w.r.t. 24 processes) and master utilizations in experiments of the second type

hours on 24 processes but the predicted serial wall time was more than 120 hours (experiments of the second type) the parallel speedups from 24 processes to 120, 240, 480, 720, and 960 processes are presented in Figure 4(a). [7] The master utilization for those four instances is shown in Figure 4(b). Finally, the share of the root node and the mean number of concurrently active nodes for algorithm execution on 24 processes are presented in Table 6.

[7] In contrast to the speedups presented in Figures 2 and 3, the basis for speedup calculation is 24 processes. For example, a linear speedup on 960 processes would be equivalent to a speedup of 960/24=40.

Table 6: Further information on experiments of the second type

Instance	Root node share	Mean number of concurrently active nodes
300/40 no. 2	0.34%	40855
225/30 no. 1	0.04%	112209
300/30 no. 3	0.16%	39005
500/100 no. 3	0.32%	50985

6. Discussion

6.1. Scalability in Experiments of the First Type

From the speedup curves in Figures 2 and 3, we can see that the average speedup and the maximum speedup can differ substantially within instance sizes. This is driven by the fact that serial wall times differ substantially within our instance sizes as the coefficients of variation (CV) for the wall times lie between 0.9 and 1.8, see Table 3. The size of the b&b tree, which is another indicator for parallelization potential, varies to the same extent with CVs between 0.8 and 1.9, see Table D.8. It has to be emphasized that for any instance size, the instance with the highest speedup on any number of processes was always the instance with the highest serial wall time. This is a very attractive observation, however not surprising since instances with a high serial wall time also have many explored b&b nodes and therefore a high potential for parallel scalability.

The average speedup is sublinear in all tested instance sizes. For single instances, however, our parallel b&p algorithm can lead to superlinear speedup as observed in the instance sizes with $n = 375$ jobs on $m = 75$ machines, where the maximum speedup is slightly superlinear on 24, 120, and 240 processes and with $n = 200$ jobs on $m = 20$ machines where the maximum speedup is clearly superlinear for up to 960 processes. This is in accordance with studies from the literature (Ponz-Tienda et al., 2017; Borisenko et al., 2011; Galea & Le Cun, 2011) that report superlinear speedups, which can be achieved, for example, when the parallel executed algorithm provides good bounds that allow pruning large parts of the search tree at early stages.

As a consequence of the high volatility of speedups within instance sizes, we focus on comparing the average speedup curves between instance sizes and do not compare single instances among different instance sizes. We can see that the speedup curves in Figures 2 and 3 show different behaviors: The slopes of the speedup curves in Figures 2(a) and 2(c) are almost 0. The speedups

in Figures 2(d) and 3(a), 3(b), and 3(c) are higher but still the curves are flat. In constrast, the speedups curves reported in Figures 2(b), 2(e), and 3(d) are much steeper and show a good scaling behavior. An important observation is that the speedups increase with an increasing ratio $\frac{n}{m}$ of the number of jobs to machines when the number of machines m is fixed. Furthermore, the speedups increase with an increasing number of machines m when the ratio $\frac{n}{m}$ is fixed.

We see different factors that may have an impact on the average speedups of our instance sizes: (a) a large serial part (i.e., when the root node has a relatively high solution time compared to the rest of the b&b tree) leads to a bad scaling potential; (b) a low mean number of concurrently active nodes has a bad influence on scalability due to worker idleness; (c) high master activity leads to waiting times of idle worker processes, which has a negative impact on scaling. In the following, we analyze (the occurrance of) each of these possible factors based on data collected in our experiments.

Share of the serial part. The share of the serial part of the algorithm (i.e., solving the root node) from the total wall time provides us with an upper bound on which speedup is possible on an infinite number of processes. For example, when the share of the serial part from the total wall time of the algorithm is 1%, there cannot be a speedup of more than 100 – regardless of the number of processes used. For the instance sizes $(100, 10)$ and $(150, 20)$, even the minimum root node share is 1.2% and 2.2%, respectively, and therefore none of the instances can even theoretically scale beyond the factor of $1/0.012 = 83.3$ and $1/0.022 = 45.5$, respectively. In sharp constrast, for the instance size of $n = 375$ and $m = 75$, the minimum root node share is 0.02% which enables theoretical scaling up to the factor of $1/0.0002 = 5000$. We conclude that for some instance sizes, the serial part is already a bottleneck that prohibits scaling from parallelizing the b&b tree. However, there are also instance sizes where this is not an issue and scalability on thousands of processes is possible. As future work, we recommend to develop techniques to reduce the share of the serial part, for example by parallelizing the solution of the root node. Appealing candidates for root node parallelization are the solving of the pricing problem (Algorithm 3), solving the restricted LPs during column generation (Algorithm 2), and the calculation of the branching variable (equation 8).

Mean number of concurrently active nodes. The mean number of concurrently active nodes is important for analyzing scalability of our parallel approach since at any time of algorithm execution, some workers will be idle when there are less active nodes than worker processes. We see that for the instance sizes $(100, 10)$ and $(150, 20)$, there are on average 32 and 27 concurrently active nodes.

Therefore, good scalability on more than 24 processes cannot be expected. At the other extreme, for the instance sizes $(200, 20)$, $(300, 40)$, and $(375, 75)$, there is an average of 2435, 4000, and 6392 concurrently active nodes. This enables scalability of our parallel algorithm on many thousands of processes. Again, we can conclude that for some instance sizes, the number of concurrently active nodes detains our parallel algorithm from good scaling while there are instance sizes where this is not an issue within our computational range up to 960 processes. For those instance sizes with a low number of concurrently active nodes, we recommend to develop methods for solving single b&b tree nodes in parallel as future work. As mentioned above for the root node, appealing candidates for intra-node parallelization are again the pricing problem, the restricted LPs during column generation and the branching variable calculations.

Activities of the master. A third important factor for master/worker scalability is the amount of work done by the master. When there is much contention at the master process, worker processes may have to wait for a certain time until they receive their next node to solve. Once the master is occupied (close) to 100%, further scaling of the parallel algorithm is impossible. As the average master utilization is well below 20% for all instances sizes – with one exception being $(200, 20)$ – even on 960 processes (see Figures D.9 and D.10), this seems to be a minor issue in our experiments. However, there is one single instance with a master utilization of around 70% on 960 processes. As Figures D.9 and D.10 suggest, master utilization increases linearly with the number of proceses. Therefore, this instance will not scale to substantially more than 1,000 processes. Since there is no more scaling potential once the master utilization reaches 100% (a significant decrease of the slope of the speedup curve will probably occur much earlier), we recommend to implement and test hierarchical master/slave approaches as future work in order to relieve the master process. However, we stress that a hierarchical master/worker scheme would not have significantly improved scalability in our computational experiments, since the master activity was not the limiting factor for our tested instances on up to 960 processes.

6.2. Scalability in Experiments of the Second Type

We can see that three out of four instances show almost linear speedup on up to 960 processes in Figure 4 while the fourth instance does not scale beyond 480 processes. As in the previous subsection, we discuss the influence of the factors (a) root node share, (b) mean number of concurrently active nodes, and (c) master utilization on scalability. From Table 6, we obtain that the share of the root

node from the total wall time on 24 processes is between 0.04% and 0.34% for all four instances. This upper-bounds the speedup of our parallelization to between $1/0.0034 \approx 294$ and $1/0.0004 = 2500$. Since the basis for this speedup bound is 24 processes, this enables scaling on thousands or even tens of thousands of processes. The mean number of concurrently active nodes (see Table 6) lies between 39005 and 112209 and also enables speedup on tens of thousands of processes. The master activity, however, is a bottleneck for one instance ($n = 225$, $m = 30$, random instance 2) in which the master is almost 100% busy when executing the parallel algorithm on 480 processes. This explains why there is no further scaling beyond this point.

The near-linear speedup in three instances is a highly desirable behavior especially because those were three of the four hardest instances in our computational pretests. One direction of future work is therefore to investigate whether this promising behavior occurs systematically for instances of more difficult instance sizes than those investigated in our computational experiments. Since we had memory overflows while solving some of the harder instances, this requires hardware with higher memory and/or a highly memory-efficient algorithm implementation. Furthermore, the instance with a master activity of almost 100% on 480 processes could benefit from the implementation of hierarchical master/worker approaches, especially because the root node share and the number of concurrently active nodes theoretically allow for scaling on tens of thousands of processes for this particular instance.

6.3. Wall Time Reduction

As we can see from the serial wall times in Table 3, instances become more difficult to solve (ceteris paribus) the higher the number of jobs m or the ratio $\frac{n}{m}$ of the number of jobs to machines gets. Particularly difficult instance sizes in our computational experiments are $(200, 20)$, $(300, 40)$, and $(375, 75)$ with average serial wall times of 86077s (approx. 24 hours), 74393s (approx. 21 hours), and 97356s seconds (approx. 27 hours), respectively. The average wall times for those instance sizes have been reduced to 101s, 184s, and 186s, repsectively, on 960 processes. Furthermore, the three highest wall times in serial mode were approximately 94 hours, 73 hours, and 60 hours while all instances were solved within six minutes on 960 processes.

In the computational experiments of the second type, we have seen almost linear speedup for three out of four instances on up to 960 processes. This results in a decrease of wall times from approximately 6 hours, 17 hours, and 22 hours on 24 processes to approximately 11 minutes, 29

minutes, and 35 minutes, respectively, on 960 processes. The predicted serial wall time of these three instances is as high as approximately 143 hours, 388 hours, and 505 hours, respectively. In summary, our parallel b&p algorithm is capable of substantially reducing wall time – especially of those instances that are hard to solve.

7. Conclusions

In this paper, we adapt a b&p algorithm, which was originally developed by Lopes & de Carvalho (2007), to the strongly NP-hard scheduling problem $R/s_{ijk}, M_j/\sum w_j C_j$. We suggest, implement, and computationally validate a master/worker parallelization strategy for the adapted algorithm, thereby bridging the gap between the largely unconnected fields of scheduling problems and HPC. We use multiple processes to solve concurrently active nodes of the b&b tree simultaneously. Our computational experiments show that our parallelization strategy can achieve high efficiencies and in some instances even superlinear speedups.

We find that speedups increase with an increasing ratio of the number of jobs to machines when the number of machines is fixed. Furthermore, speedups also increase with an increasing number of machines when the ratio of the number of jobs to machines is fixed. We present and discuss several factors that explain this speedup behavior: the share of the serial part of the algorithm, the mean number of concurrently active nodes during algorithm execution and the activity of the master process. Finally, we show that parallel execution substantially reduces wall times of the algorithm from up to 94 hours in serial execution for the most difficult tested instance to less than six minutes on 960 processes.

There are several directions for future work. First, hierarchical master/slave approaches could be developed to improve scalability for those instances where our conventional master/slave approach shows high contention at the master process. Second, especially for instances with a complicated root node or only a few concurrently active nodes, strategies for solving single b&b nodes in parallel should be developed. Such intra-node parallelization approaches could be combined with our inter-node master/worker approach in a hybrid implementation. Third, fault tolerance and load balancing issues should be included into the parallelization design, especially when further increasing the number of processes used. Fourth, the near-linear scaling in most of the hard instances should be further evaluated on larger instance sizes. For this, more computing resources and a more efficient memory management are crucial.

Acknowledgments

This research has been funded by the Federal Ministry of Education and Research of Germany in the framework of KUBAS (project number 13N13942). Simulations were performed with computing resources granted by RWTH Aachen University under project prep0011.

References

Aitzai, A., & Boudhar, M. (2013). Parallel branch-and-bound and parallel pso algorithms for job shop scheduling problem with blocking. *International Journal of Operational Research*, *16*, 14–37.

van den Akker, J. M., Hoogeveen, J. A., & van de Velde, S. L. (1999). Parallel machine scheduling by column generation. *Operations Research*, *47*, 862–872.

Akyol, D. E., & Bayhan, G. M. (2008). Multi-machine earliness and tardiness scheduling problem: An interconnected neural network approach. *The International Journal of Advanced Manufacturing Technology*, *37*, 576–588.

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, *246*, 345–378.

Allahverdi, A., Gupta, J. N. D., & Aldowaisan, T. (1999). A review of scheduling research involving setup considerations. *Omega*, *27*, 219–239.

Allahverdi, A., Ng, C. T., Cheng, T. C. E., & Kovalyov, M. Y. (2008). A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, *187*, 985–1032.

Alvelos, F., Lopes, M., & Lopes, H. (2016). A matheuristic based on column generation for parallel machine scheduling with sequence dependent setup times. In *Computational Management Science* (pp. 233–238). Springer.

Arnaout, J.-P. M., & Rabadi, G. (2005). Minimizing the total weighted completion time on unrelated parallel machines with stochastic times. In *Proceedings of the IEEE Winter Simulation Conference*.

Bard, J. F., & Rojanasoonthon, S. (2006). A branch-and-price algorithm for parallel machine scheduling with time windows and job priorities. *Naval Research Logistics*, *53*, 24–44.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., & Vance, P. H. (1998). Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, *46*, 316–329.

Bell, G., & Gray, J. (2002). What's next in high-performance computing? *Communications of the ACM*, *45*, 91–95.

Borisenko, A., Kegel, P., & Gorlatch, S. (2011). Optimal design of multi-product batch plants using a parallel branch-and-bound method. In *International Conference on Parallel Computing Technologies* (pp. 417–430). Springer.

Bozorgirad, M. A., & Logendran, R. (2012). Sequence-dependent group scheduling problem on unrelated-parallel machines. *Expert Systems with Applications*, *39*, 9021–9030.

Brucker, P. (2007). *Scheduling Algorithms*. Springer.

Chakroun, I., Melab, N., Mezmaz, M., & Tuyttens, D. (2013). Combining multi-core and gpu computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing*, *73*, 1563–1577.

Chen, J.-F. (2015). Unrelated parallel-machine scheduling to minimize total weighted completion time. *Journal of Intelligent Manufacturing*, *26*, 1099–1112.

Chen, Z.-L., & Powell, W. B. (1999). Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing*, *11*, 78–94.

Chen, Z.-L., & Powell, W. B. (2003). Exact algorithms for scheduling multiple families of jobs on parallel machines. *Naval Research Logistics*, *50*, 823–840.

Cheng, T., & Sin, C. (1990). A state-of-the-art review of parallel-machine scheduling research. *European Journal of Operational Research*, *47*, 271–292.

Clausen, J., & Perregaard, M. (1999). On the best search strategy in parallel branch-and-bound: Best-first search versus lazy depth-first search. *Annals of Operations Research*, *90*, 1–17.

Crespo Abril, F., & Maroto Alvarez, C. (2005). Scheduling resource-constrained projects using branch and bound and parallel computing techniques. *International Journal of Operational Research*, *1*, 172–187.

Dantzig, G. B., & Wolfe, P. (1960). Decomposition principle for linear programs. *Operations Research*, *8*, 101–111.

Desaulniers, G., Desrosiers, J., & Solomon, M. (Eds.) (2006). *Column Generation*. Springer.

Eckstein, J. (1994). Parallel branch-and-bound algorithms for general mixed integer programming on the cm-5. *SIAM Journal on Optimization*, *4*, 794–814.

Fei, H., Chu, C., Meskens, N., & Artiba, A. (2008). Solving surgical cases assignment problem by a branch-and-price approach. *International Journal of Production Economics*, *112*, 96–108.

Galea, F., & Le Cun, B. (2011). A parallel exact solver for the three-index quadratic assignment problem. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (pp. 1940–1949). IEEE.

Gendron, B., & Crainic, T. G. (1994). Parallel branch-and-bound algorithms: Survey and synthesis. *Operations research*, *42*, 1042–1066.

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy Kan, A. H. G. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, *5*, 287–326.

Hager, G., & Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press.

Joo, C. M., & Kim, B. S. (2012). Parallel machine scheduling problem with ready times, due times and sequence-dependent setup times using meta-heuristic algorithms. *Engineering Optimization*, *44*, 1021–1034.

Kim, D.-W., Na, D.-G., & Chen, F. F. (2003). Unrelated parallel machine scheduling with setup times and a total weighted tardiness objective. *Robotics and Computer-Integrated Manufacturing*, *19*, 173–181.

Lin, Y.-K., & Hsieh, F.-Y. (2014). Unrelated parallel machine scheduling with setup times and ready times. *International Journal of Production Research*, *52*, 1200–1214.

Logendran, R., McDonell, B., & Smucker, B. (2007). Scheduling unrelated parallel machines with sequence-dependent setups. *Computers & Operations Research*, *34*, 3420–3438.

Lopes, M., Alvelos, F., & Lopes, H. (2014). Improving branch-and-price for parallel machine scheduling. In *Computational Science and Its Applications–ICCSA 2014* (pp. 290–300). Springer.

Lopes, M. J. P., & de Carvalho, J. M. V. (2007). A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times. *European Journal of Operational Research*, *176*, 1508–1527.

Lübbecke, M. E., & Desrosiers, J. (2005). Selected topics in column generation. *Operations Research*, *53*, 1007–1023.

Mauch, V., Kunze, M., & Hillenbrand, M. (2013). High performance cloud computing. *Future Generation Computer Systems*, *29*, 1408–1416.

Migdalas, A., Pardalos, P. M., & Storøy, S. (2013). *Parallel Computing in Optimization*. Springer Science & Business Media.

de Paula, M. R., Mateus, G. R., & Ravetti, M. G. (2010). A non-delayed relax-and-cut algorithm for scheduling problems with parallel machines, due dates and sequence-dependent setup times. *Computers & Operations Research*, *37*, 938–949.

Perregaard, M., & Clausen, J. (1998). Parallel branch-and-bound methods for the job-shop scheduling problem. *Annals of Operations Research*, *83*, 137–160.

Pinedo, M. L. (2012). *Scheduling: Theory, Algorithms, and Systems*. Springer.

Ponz-Tienda, J. L., Salcedo-Bernal, A., & Pellicer, E. (2017). A parallel branch and bound algorithm for the resource leveling problem with minimal lags. *Computer-Aided Civil and Infrastructure Engineering*, *32*, 474–498.

Rabadi, G. (2016). *Heuristics, Meta-heuristics and Approximate Methods in Planning and Scheduling* volume 236. Springer.

Ralphs, T. K., Ladanyi, L., & Saltzman, M. J. (2003). Parallel branch, cut, and price for large-scale discrete pptimization. *Mathematical Programming*, *98*, 253–280.

Rauchecker, G., & Schryen, G. (2015). High-performance computing for scheduling decision support: A parallel depth-first search heuristic. In *Proceedings of the 26th Australasian Conference on Information Systems*.

Schryen, G., Rauchecker, G., & Comes, T. (2015). Resource planning in disaster response. *Business & Information Systems Engineering*, *7*, 1–17.

Skutella, M., & Woeginger, G. J. (2000). A ptas for minimizing the total weighted completion time on identical parallel machines. *Mathematics of Operations Research*, *25*, 63–75.

Tavakkoli-Moghaddam, R., & Aramon-Bajestani, M. (2009). A novel b and b algorithm for a unrelated parallel machine scheduling problem to minimize the total weighted tardiness. *IJE transactions A: Basics*, *22*, 269–286.

TOP500 (2017). Top500 supercomputer sites. retrieved september 05, 2017. URL: `http://www.top500.org/list/2017/06/`.

Tsai, C., & Tseng, C. (2007). Unrelated parallel-machines scheduling with constrained resources and sequence-dependent setup time. In *Proceedings of the 37th International Conference on Computers and Industrial Engineering* (pp. 20–23).

Vecchiola, C., Pandey, S., & Buyya, R. (2009). High-performance cloud computing: A view of scientific applications. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks* (pp. 4–16).

Weng, M. X., Lu, J., & Ren, H. (2001). Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective. *International Journal of Production Economics*, *70*, 215–226.

Wex, F., Schryen, G., Feuerriegel, S., & Neumann, D. (2014). Emergency response in natural disaster management: Allocation and scheduling of rescue units. *European Journal of Operational Research*, *235*, 697–708.

Zeidi, J. R., & Mohammad Hosseini, S. (2015). Scheduling unrelated parallel machines with sequence-dependent setup times. *The International Journal of Advanced Manufacturing Technology*, *81*, 1487–1496.

Zhang, Z., Zheng, L., & Weng, M. X. (2007). Dynamic parallel machine scheduling with mean weighted tardiness objective by q-learning. *The International Journal of Advanced Manufacturing Technology*, *34*, 968–980.

Zhu, Z., & Heady, R. B. (2000). Minimizing the sum of earliness/tardiness in multi-machine scheduling: a mixed integer programming approach. *Computers & Industrial Engineering*, *38*, 297–305.

Appendix A. Decreasing Reduced Cost Variables

In this section, we present a result that allows us to consider only a subset of all variables during column generation - so called *decreasing reduced cost variables* - whenever none of the setup times is larger than any processing times. Lopes & de Carvalho (2007) propose a similar result claiming that all setup times have to meet the triangle inequality.

Definition 1. *Let k be a machine and $\omega = (j_1, \ldots, j_H) \in \Omega^k$ be a schedule. For a job j, we define $j \in \omega$ if and only if $j = j_h$ for some $h = 1, \ldots, H$. For all jobs $j \in \omega$, we define $c^k_{j\omega}$ as the completion time of j when ω is operated on k. Finally, we define the variable x^k_ω to be a* decreasing reduced cost variable *if and only if $c^k_{j\omega} < \frac{\pi_j}{w_j}$ for all jobs $j \in \omega$.*

If we reformulate equation (9) by

$$r^k_\omega = -\sigma_k + \sum_{j \in \omega} w_j \cdot c^k_{j\omega} - \pi_j \tag{A.1}$$

then Definition 1 means that each job j_h for $1 \leq h \leq H$ decreases the reduced cost when added to the schedule $(j_1, \ldots, j_{h-1})$. This leads us to the following result.

Theorem 2. *Let k be an arbitrary machine and $\omega \in \Omega^k$ be a schedule such that x^k_ω is a variable with negative reduced cost. If $s^k_{ij} \leq p^k_j$ holds true for all $i = 0 \ldots, n$, $j = 1, \ldots, n$, and $k = 1, \ldots, m$ then there is at least one schedule $\omega' \in \Omega^k$ such that $x^k_{\omega'}$ is a decreasing reduced cost variable with $r^k_{\omega'} \leq r^k_\omega$.*

Proof. Taking into account $\sigma_k \leq 0$ (from duality theory) and (A.1), we can conclude that ω is not the empty schedule (i.e., $H \geq 1$) and has at least one job $j \in \omega$ with $c^k_{j\omega} < \frac{\pi_j}{w_j}$ since otherwise ω would have non-negative reduced cost according to (A.1). Let $\omega' \in \Omega^k$ be the non-empty schedule which emanates from ω by discarding all jobs j with

$$c^k_{j\omega} \geq \frac{\pi_j}{w_j}. \tag{A.2}$$

The discarding of a job does never lead to a higher completion time of the following jobs because the vanishing processing time is not lower than the additional setup time as we assumed all setup times to be not higher than any processing times. This leads to

$$c^k_{j\omega'} \leq c^k_{j\omega} \quad \forall j \in \omega', \tag{A.3}$$

which shows that $x^k_{\omega'}$ is a decreasing reduced cost variable since

$$c^k_{j\omega'} \leq c^k_{j\omega} < \frac{\pi_j}{w_j} \tag{A.4}$$

for all $j \in \omega'$. The latter inequality holds because of the construction of ω'. Finally, we calculate

$$\begin{aligned} r^k_\omega &\overset{(A.1)}{=} -\sigma_k + \sum_{j\in\omega} w_j \cdot c^k_{j\omega} - \pi_j \\ &\overset{(A.2)}{\geq} -\sigma_k + \sum_{j\in\omega'} w_j \cdot c^k_{j\omega} - \pi_j \\ &\overset{(A.3)}{\geq} -\sigma_k + \sum_{j\in\omega'} w_j \cdot c^k_{j\omega'} - \pi_j \\ &\overset{(A.1)}{=} r^k_{\omega'} \end{aligned} \tag{A.5}$$

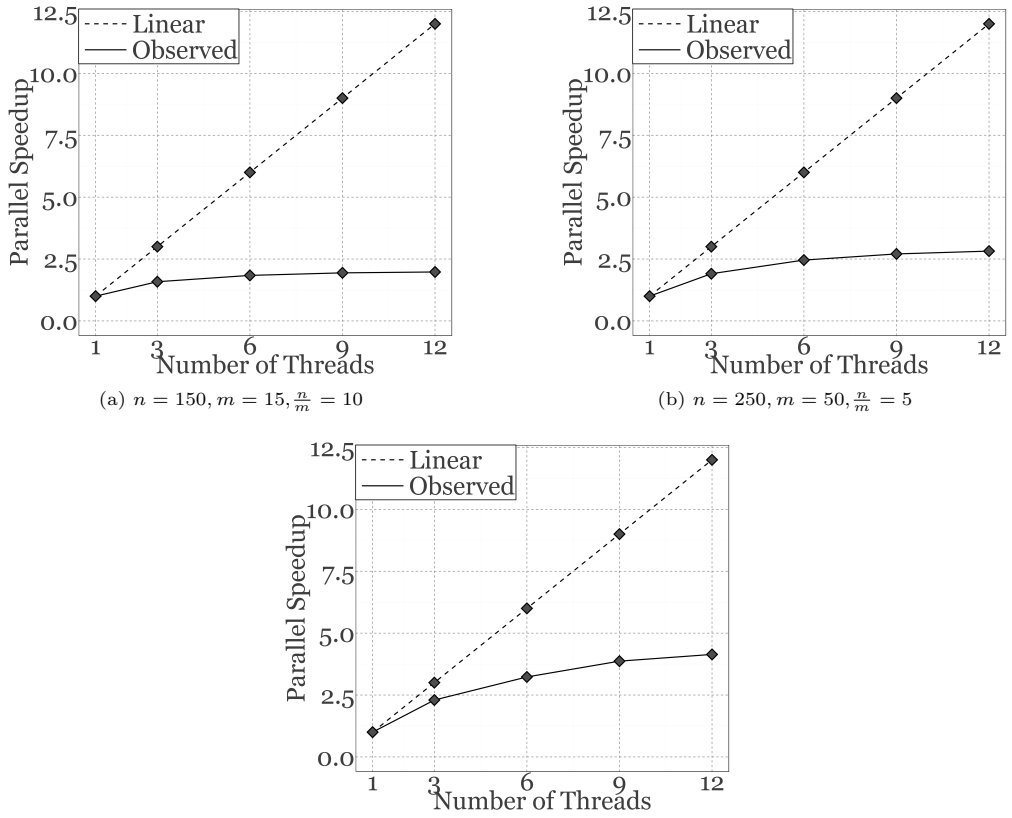
which proves the result. □

According to this result, we have to consider only decreasing reduced cost variables during the column generation procedure (Algorithm 2). This means that when solving equation (12), for each pair (k, t), we only have to take into account jobs j with $t < \frac{\pi_j}{w_j}$.

After branching, the presence of job ordering restrictions may prohibit the discarding of jobs as used in the proof of Theorem 2. Consequently, we also have to consider jobs j that have appeared in any branching variable leading to the current node - regardless whether their insertion leads to a decreasing reduced cost variable or not.

Appendix B. Pretests on Parallelization of Type 1

In this section, we present the results of our pretests on parallelization of type 1, i.e., executing the solving of single b&b nodes in parallel. We identified potential for parallelization of type 1 for the following parts of our b&p algorithm, which in combination comprised almost the entire execution time of the algorithm: First, the solving of restricted LPs (step 1 of Algorithm 2) can be parallelized by the LP solver. We used GUROBI for this purpose. Second, the values $f^k(t, j)$ (step 3 of Algorithm 3) can be calculated independently for different machines k. Third, the calculation of values X^k_{ij}, see equation (7), can be performed simultaneously for different machines k. Our parallelization of type 1 consists of executing all these three parts of the algorithm in parallel.



(a) $n = 150, m = 15, \frac{n}{m} = 10$

(b) $n = 250, m = 50, \frac{n}{m} = 5$

(c) $n = 600, m = 300, \frac{n}{m} = 2$

Figure B.5: Speedups for parallelization of type 1

In our pretests, we have tested ten instances for all the three different instance sizes

$$(n, m) \in (150, 15), (250, 50), (600, 300)$$

which reflect different ratios $\frac{n}{m}$ of the number of jobs to the number of machines. For each instance, we parallelized the algorithm by executing the above mentioned parts of the algorithm on 3, 6, 9, and 12 threads. Figure B.5 presents the achieved speedups (parallel execution time divided by serial execution time) averaged over the ten instances of each instance size. The dashed lines symbolize
675 linear speedup (i.e., a speedup of R on R threads)

We can see that there is not much execution time reduction as the speedups on 12 threads are 2.0 for $(n, m) = (150, 15)$, 2.8 for $(n, m) = (250, 50)$, and 4.1 for $(n, m) = (600, 300)$ with little potential for further improvement due to the flatness of the speedup curves. The main contributor

Table B.7: Speedup ratios when doubling number of threads per process

(n, m)	$(150, 15)$	$(150, 15)$	$(250, 50)$	$(250, 50)$	$(600, 300)$	$(600, 300)$
	$\frac{SU_{hyb}(R,6)}{SU_{hyb}(R,3)}$	$\frac{SU_{hyb}(R,12)}{SU_{hyb}(R,6)}$	$\frac{SU_{hyb}(R,6)}{SU_{hyb}(R,3)}$	$\frac{SU_{hyb}(R,12)}{SU_{hyb}(R,6)}$	$\frac{SU_{hyb}(R,6)}{SU_{hyb}(R,3)}$	$\frac{SU_{hyb}(R,12)}{SU_{hyb}(R,6)}$
$R = 1$	1.16	1.08	1.29	1.15	1.41	1.28
$R = 5$	1.17	1.07	1.28	1.16	1.40	1.28
$R = 10$	1.17	1.08	1.28	1.15	1.40	1.28
$R = 20$	1.16	1.10	1.29	1.15	1.40	1.28
$R = 50$	1.19	1.08	1.29	1.14	1.40	1.28

to poor speedup is that the restricted LPs (repeatedly solved by GUROBI in step 1 of Algorithm 2), the solution of which represents up to 40% of the total execution time, seem to be too small for an effective parallelization.

There are potential interdependencies between our parallelization approaches of type 1 (solving each b&b node using multiple threads) and type 2 (solving different b&b nodes using multiple processes). For example, when those types are jointly applied, using a certain number of threads per process may have different speedup effects on the solving of different b&b nodes and consequently, the order in which the b&b nodes are investigated may change when using more/less threads per process. This may finally lead to a different number of nodes explored during the b&p algorithm.

Therefore, on the same set of instances as above, we have pretested a hybrid approach, i.e., solving concurrently active nodes of the b&b tree simultaneously on different processes (as described in Section 4) and parallelizing the solving of each of the single b&b nodes on multiple threads (as described above).

Table B.7 lists the speedup ratios

$$\frac{SU_{hyb}(R, 6)}{SU_{hyb}(R, 3)} \quad \text{and} \quad \frac{SU_{hyb}(R, 12)}{SU_{hyb}(R, 6)}$$

when doubling the number of threads per process (from 3 to 6 and from 6 to 12 threads per process) for each number $R \in \{1, 5, 10, 20, 50\}$ of processes. All values are averaged over all ten instances per instance size.

We see that the average speedup ratios from 3 to 6 threads and from 6 to 12 threads are almost entirely independent of the number of processes R. Doubling the number of threads from 3 to 6 results in a speedup ratio of roughly 1.2 for $(n, m) = (150, 15)$, 1.3 for $(n, m) = (250, 50)$, and 1.4 for

$(n, m) = (600, 300)$. This means, for example, that for $(n, m) = (150, 15)$ the speedup on 6 threads is only 20% higher than the speedup on 3 threads although the number of computing resources is doubled. Further doubling the number of threads from 6 to 12 results in a speedup ratio of roughly 1.1 for $(n, m) = (150, 15)$, 1.2 for $(n, m) = (250, 50)$, and 1.3 for $(n, m) = (600, 300)$. In summary, using parallelization of type 1 is still not promising for effectively reducing execution times when jointly applied with parallelization of type 2 in a hybrid approach.

Appendix C. Details on Parallel Implementation

In this section, we present details on the implementation of the introduced master/worker parallelization concept from Section 4. We take special care of an efficient master/worker communication and aim at keeping the work done by the master process at a minimum to enable scalability of our approach. A graphical overview of the interaction between the master process and the worker processes is shown in the sequence diagram in Figure C.6, which we outline in the following.

Messages are indicated by arrows and are passed between processes using MPI. All communication is blocking and unbuffered. At the beginning of the execution of the parallelized algorithm, the master process calculates a starting set of columns to initialize column generation, solves the root node using column generation and calculates the corresponding variable to branch on. After that, the master process sends the root node data (including generated columns and branching variable) to all worker processes and initializes the set of active nodes. Sending the root node data to all worker processes substantially reduces the volume of data to be communicated during the rest of the algorithm, which is explained in the following.

Since the master process is responsible for tree administration while the worker processes only have to solve nodes of the b&b tree which they receive from the master process, see Algorithms 4 and 5, the master process has to provide a worker process with information about the next node it has to solve. One part of this information is the data-intensive set of variables x_ω^k generated for solving the parent node (these serve as the initial set of variables in step 1 of the column generation procedure). Our pretests show that the number of additional columns generated during the solution of a b&b node is low compared to the number of columns generated for the solution of the parent node and even compared to the number of columns generated for the root node solution. Therefore, we instruct the master process to distribute the solution of the root node to all worker processes. Consequently, whenever a b&b node has to be transferred from the master process to a worker

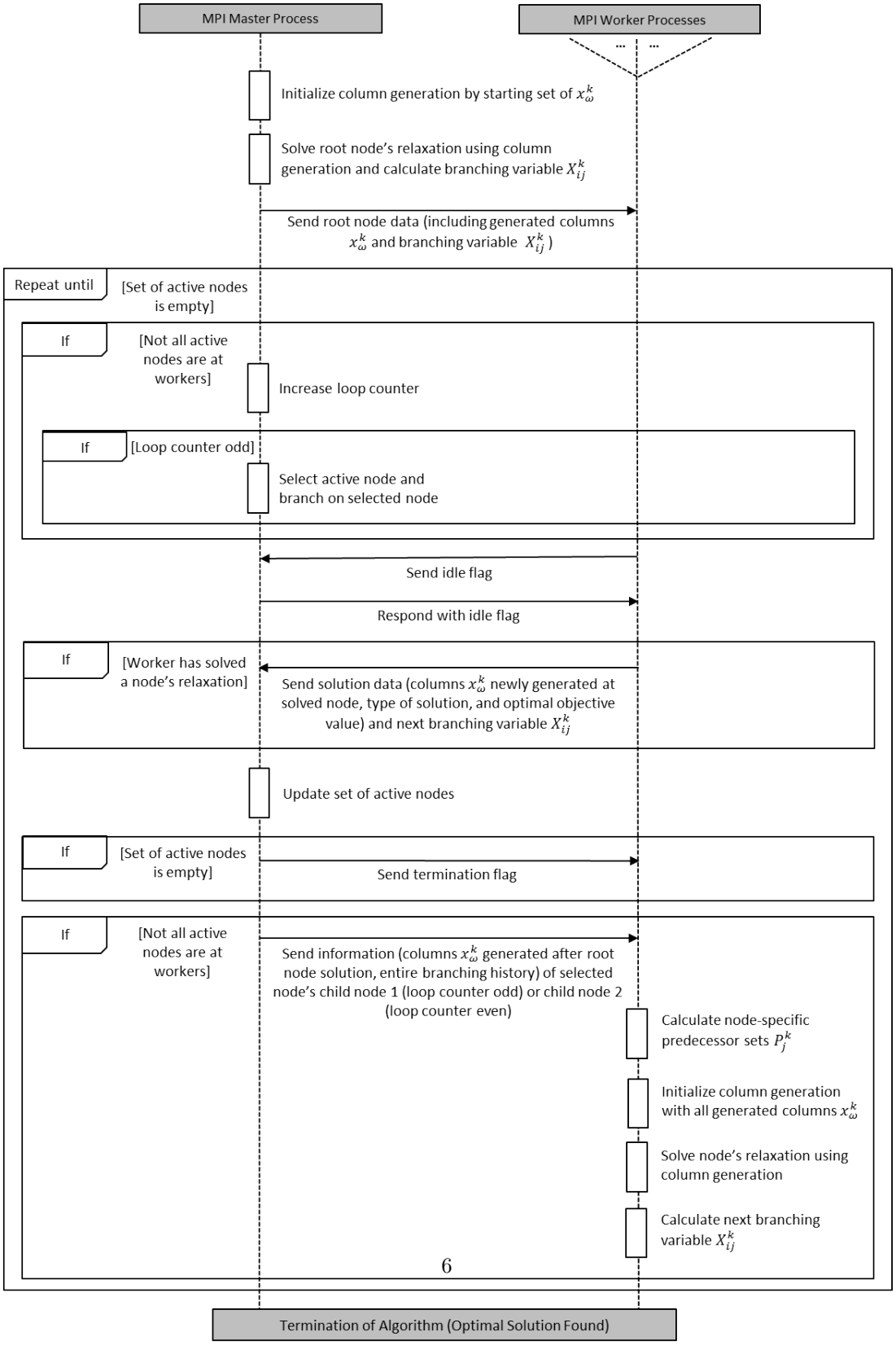


Figure C.6: Interaction between MPI master process and MPI worker processes

process, only columns that have not already been generated at the root node have to be transferred - because the columns generated at the root node are already stored in the memory of each worker process.

At the beginning of the *repeat* loop in Figure C.6, the master process increases the loop counter, which is used to determine whether two new child nodes have to be generated in the current iteration through branching (loop count odd) or not (loop count even). If the loop counter is odd, the master process selects an active node and branches on the selected node. Each time a worker process has solved the problem assigned to it by the master process and becomes idle, it sends a flag to the master process to initiate communication of solution data and waits for a response. Once the master process responds with a flag, the worker process sends solution data and the next branching variable (the reason for sending the latter is explained later) of the node that it finished solving. Note that only columns x_ω^k which have been newly generated at the solved node have to be transferred since all columns generated for the parent node's solution are already stored in the memory of the master process.

The master process updates the set of active nodes and sends a termination flag in case that there are no more active nodes left. [8] When there are active nodes that are not already being processed by another worker, the master process responds to the worker process by sending data that specify the next b&b node (child node 1 if loop counter is odd and child node 2 otherwise) that has to be processed by the worker process, which starts solving the received node's relaxation and determining the node's next branching variable.

In order to provide a worker process with information to solve a new node, the master process transfers all columns that were generated for the parent node's solution - but not already at the root node (each worker process has already stored this information in its memory). Furthermore, the branching history $X_{i_1 j_1}^{k_1}, \ldots, X_{i_r j_r}^{k_r}$ from the root node to the new node is transferred where r is the depth of the new node in the b&b tree. The node-specific sets P_j^k can be constructed by the worker process from the corresponding sets at the root node (already stored in the memory of the worker process) using the transferred branching history, which is much cheaper to communicate than the entire sets P_j^k. This results in the consumption of little more CPU time (sum of the computing

[8] Termination flags are only sent when there are no more active nodes – including nodes currently being solved by worker processes. In particular, no key tasks are interrupted on any worker process.

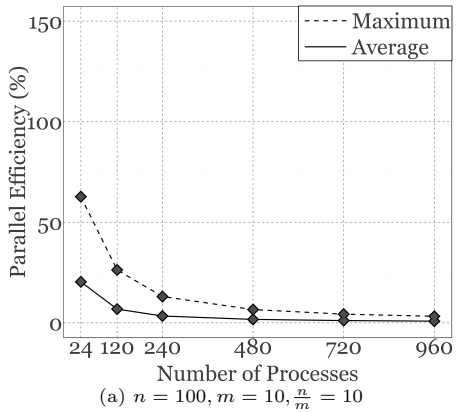
times of *all* processes) but leads to a decrease of wall time (total time from start to termination of the algorithm) since all additional work is done by the worker processes which thereby relieve the master process by reducing data transfer volume. The variables for the initialization of the column generation procedure (i.e., all columns that were generated for the solution of the parent node) compound from the stored columns generated at the root node and the transferred columns generated after the root node.

The calculation of the branching variable, i.e., solving equation (8), is a non-trivial step that would consume wall time on the master process (line 7 in Algorithm 4). We fixed this issue by forcing each worker process to calculate the branching variable of its current node immediately after the solution of its linear relaxation and sending back this information to the master process. This, again, leads to a low increase of CPU time since some branching variables are calculated but not used later on. However, the benefit is a decrease of wall time since the master process is relieved from calculating branching variables.

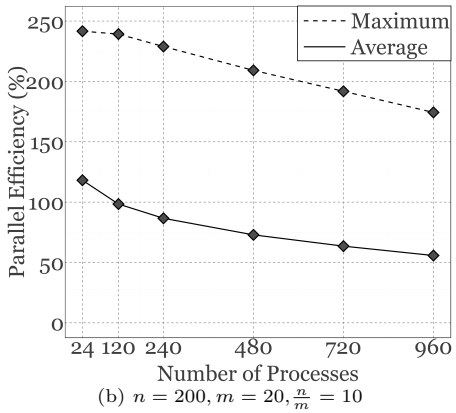
Appendix D. Further Results for Computational Experiments

Table D.8: Average number of explored nodes for serial computation (CVs in brackets)

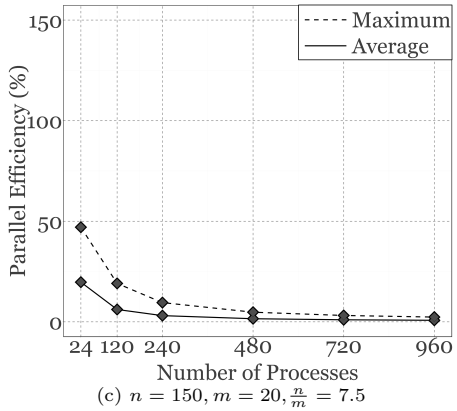
$\frac{n}{m} = 10$	225 (1.6)	22521 (1.2)	-	-	-	-	-
$\frac{n}{m} = 7.5$	-	243 (1.4)	6042 (1.4)	32855 (0.8)	-	-	-
$\frac{n}{m} = 5$	-	-	2069 (1.9)	1273 (1.4)	5335 (1.8)	47089 (1.2)	-
	$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	$m = 75$	$m = 100$



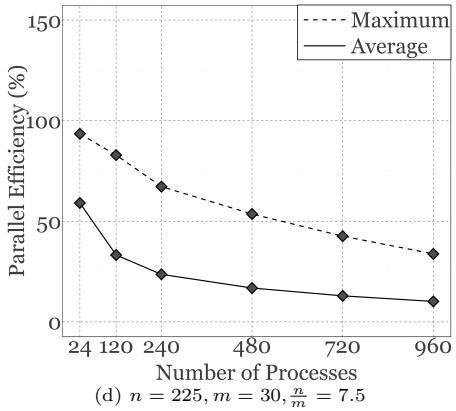
(a) $n = 100, m = 10, \frac{n}{m} = 10$



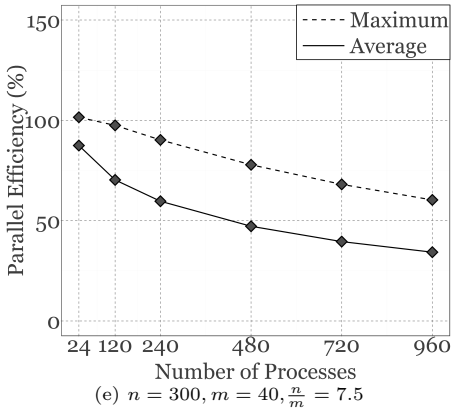
(b) $n = 200, m = 20, \frac{n}{m} = 10$



(c) $n = 150, m = 20, \frac{n}{m} = 7.5$

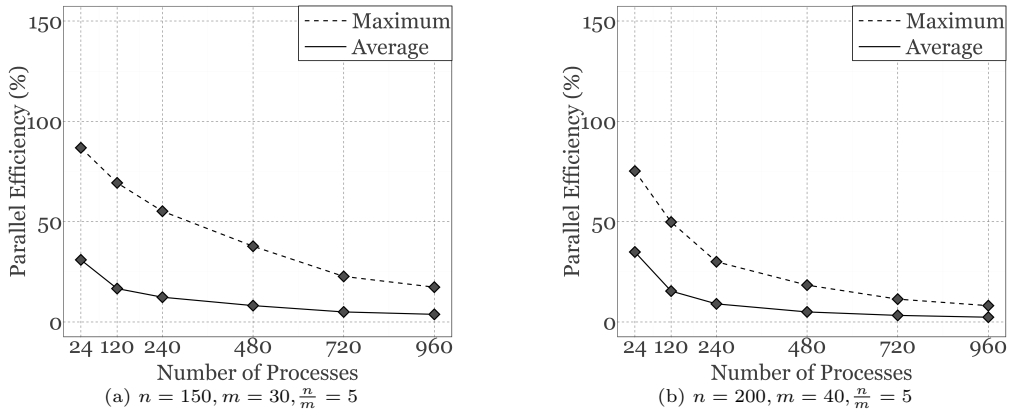


(d) $n = 225, m = 30, \frac{n}{m} = 7.5$



(e) $n = 300, m = 40, \frac{n}{m} = 7.5$

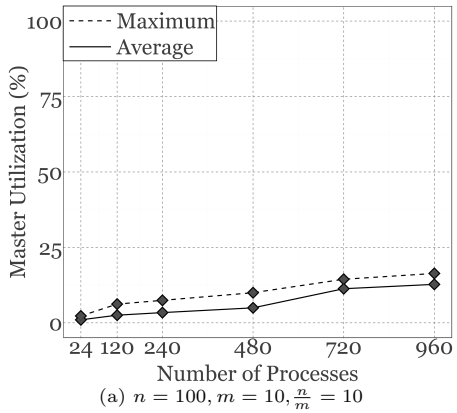
Figure D.7: Parallel efficiencies part 1



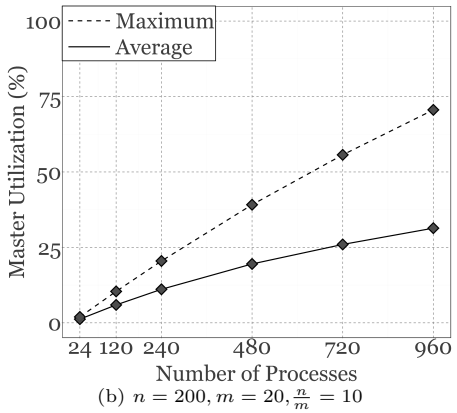
(a) $n = 150, m = 30, \frac{n}{m} = 5$ (b) $n = 200, m = 40, \frac{n}{m} = 5$

(c) $n = 250, m = 50, \frac{n}{m} = 5$ (d) $n = 375, m = 75, \frac{n}{m} = 5$

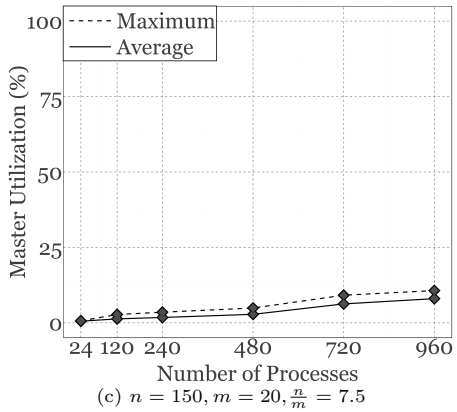
Figure D.8: Parallel efficiencies part 2



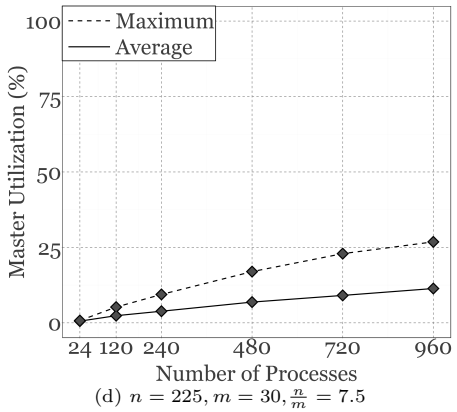
(a) $n = 100, m = 10, \frac{n}{m} = 10$



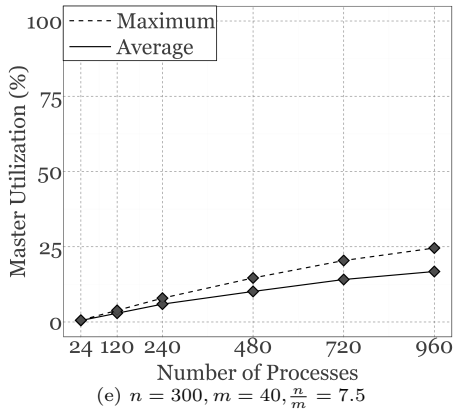
(b) $n = 200, m = 20, \frac{n}{m} = 10$



(c) $n = 150, m = 20, \frac{n}{m} = 7.5$

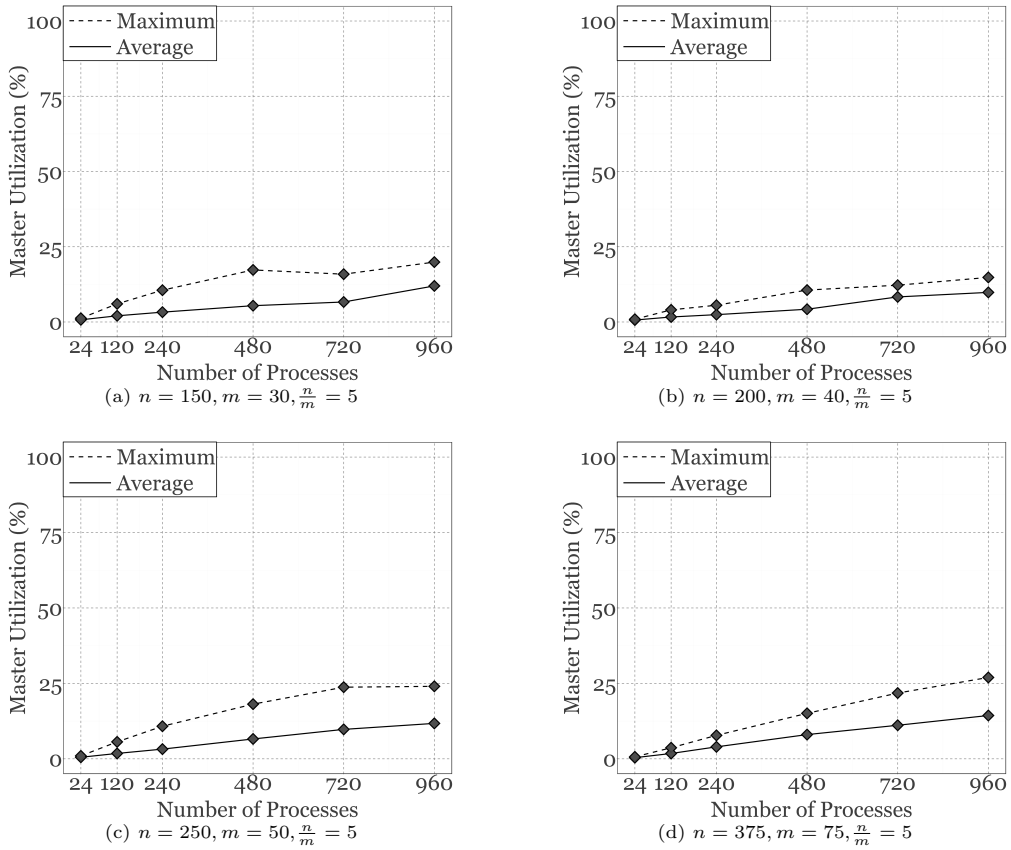


(d) $n = 225, m = 30, \frac{n}{m} = 7.5$



(e) $n = 300, m = 40, \frac{n}{m} = 7.5$

Figure D.9: Master utilizations part 1



(a) $n = 150, m = 30, \frac{n}{m} = 5$ (b) $n = 200, m = 40, \frac{n}{m} = 5$

(c) $n = 250, m = 50, \frac{n}{m} = 5$ (d) $n = 375, m = 75, \frac{n}{m} = 5$

Figure D.10: Master utilizations part 2