

Parallel Branch-and-Price Algorithms for the Single Machine Total Weighted Tardiness Scheduling Problem with Sequence-Dependent Setup Times

Philipp Speckenmeyer^{a,*}, Constanze Hilmer^b, Gerhard Rauchecker^b,
Guido Schryen^a

^aWarburger Str. 100, 33098 Paderborn, Germany

^bUnaffiliated, n/a, Germany

Abstract

Scheduling problems occur in a broad range of real-world application fields and have attracted a huge set of research articles. However, there is only little research on exact algorithms for scheduling problems, many of which are NP-hard in the strong sense. We investigate the problem on a single machine with a total weighted tardiness objective function and sequence-dependent setup times. First, we adopt a serial branch-and-price algorithm from the literature and present a modified branching strategy and a primal heuristic. Second, we use the potential of parallel computing architectures by presenting two parallel versions of the branch-and-price algorithm. Third, we conduct extensive computational experiments to show that our parallelization approaches provide substantial parallel speedups on well-known benchmark instances from the literature. We further observe that the parallel speedups achieved by our parallel algorithms are very robust among all tested instances.

Keywords: single machine scheduling, weighted tardiness, sequence-dependent setup times, branch-and-price algorithm, dynamic programming, shared-memory parallelization

*Corresponding author

1. Introduction

In this study, a single machine scheduling problem with sequence-dependent setup times and a total weighted tardiness objective function is examined. This scheduling problem can be classified as $1 \mid ST_{sd} \mid \sum w_j T_j$ in the established $\alpha \mid \beta \mid \gamma$ notation by Graham et al. (1979). We refer to this problem as the *Weighted Tardiness Scheduling Problem with Sequence-Dependent Setup Times* (WTSDS). The scheduling problem $1 \parallel \sum w_j T_j$ without setup times is already NP-hard in the strong sense (Tasgetiren et al., 2009; Lawler, 1977). Since it is a special case of WTSDS (by setting all setup times to zero), WTSDS is strongly NP-hard as well.

Problem WTSDS can be described as follows: a set of jobs has to be processed without disruption by a single machine, which can only process one job at a time. Each job has a processing time, a due date and a weight. For every pair of jobs, processed in direct succession on the machine, a setup time is incurred. Since these setup times depend on both jobs, they are called sequence-dependent. When the completion time of a job is later than its due date, it is called tardy, with the tardiness set to the time elapsed between the due date and the time of completion. The goal of WTSDS is to find a feasible schedule with minimum total weighted tardiness.

There already exist many studies addressing the WTSDS, see Allahverdi et al. (1999, 2008), and Allahverdi (2015) for an overview. However, only a few articles present exact algorithms for WTSDS or one of its generalizations, see Section 2 for details. Since none of these papers report optimal solutions to large (e.g., more than 100 jobs) WTSDS instances, such instances are still computationally intractable.

In our paper, we adopt a serial branch-and-price (b&p) algorithm – a special kind of a branch-and-bound (b&b) algorithm where linear relaxations are solved by column generation – which was originally proposed by Lopes and de Carvalho (2007) for a generalization of WTSDS – and present a modified branching strategy which exploits problem specific sequencing information obtained from

the column generation process. We also develop a primal heuristic in order to derive feasible schedules from the fractional solutions of the linear relaxations. Using this sequential b&p algorithm, we present two parallelization approaches to utilize the computing power of modern parallel hardware (Hager and Wellein,
35 2010). The scalability of our parallel algorithms (and limits thereof) is tested using established benchmark instances introduced by Cicirello (2003), Gagné et al. (2002) as well as Rubin and Ragatz (1995). Our computational experiments show that the proposed parallelization strategies provide substantial speedups with very little variations for the instances mentioned above.

40 Unlike most studies that parallelize b&b algorithms by concurrently solving nodes in the b&b tree, our study focuses on using CPU cores to parallelize the column generation process in individual b&b nodes, which has not been studied for the underlying problem. Gaining insight into the computational behavior of this type of algorithmic parallelization is valuable for a joint application of both
45 parallelization approaches in future research, not only for the studied problem WTSDS, but also for its extensions.

The remainder of this paper is structured as follows. In Section 2, we provide a comprehensive overview of literature on machine scheduling as well as parallel branch-and-x (b&x) and dynamic programming algorithms.¹ The mathematical
50 formulation of WTSDS is proposed in Section 3. In Section 4, a serial b&p algorithm for the WTSDS is presented. Then, we introduce two parallelization strategies for the b&p algorithm in Section 5. Detailed computational experiments of the proposed parallel algorithms are presented and discussed in Section 6. Finally, Section 7 concludes.

¹The latter is used to solve the so called *pricing problem* during column generation in our b&p algorithms.

55 2. Literature Review

2.1. Machine Scheduling

Various types of machine scheduling problems have been studied by numerous authors. Comprehensive reviews covering machine scheduling problems with setup times were composed by Allahverdi et al. (1999, 2008) and Allahverdi
60 (2015). Using the three-field notation $\alpha | \beta | \gamma$ by Graham et al. (1979), we focus on scheduling problems on single machines ($\alpha = 1$) and parallel machines ($\alpha = P$ for identical machines, $\alpha = Q$ for uniform machines and $\alpha = R$ for unrelated machines) with sequence-dependent setup times ($\beta = ST_{sd}$), and objective functions γ that are at least as general as the total weighted tardiness.

There exist a variety of (meta-)heuristic approaches for these problems, among them Ant Colony Optimization (Liao and Juan, 2007; Anghinolfi and Paolucci, 2008), Evolutionary Algorithms (Xu et al., 2014b), Iterated Local Search (Subramanian et al., 2014; Xu et al., 2014a; Subramanian and Farias, 2017), Scatter Search (Bożejko, 2010; Guo and Tang, 2015) and Variable Neighbourhood Search (Chen, 2019; Nogueira et al., 2022). Kramer and Subramanian
70 (2019) provide an overview of solution approaches for various earliness-tardiness scheduling problems, including WTSDS. Although (meta-)heuristic approaches are often able to produce high-quality solutions, they generally don't come with any guarantee of finding optimal solutions – let alone proving their optimality. Especially since proving optimality often times takes up most of the running
75 time for b&b algorithms (*tailing-off*), a comparison regarding running-times would be skewed. Hence, we focus on exact algorithms in the following.

Balakrishnan et al. (1999), Zhu and Heady (2000), and Akyol and Bayhan (2008) considered the total weighted earliness-tardiness objective function
80 $(\sum w'_j E_j + \sum w''_j T_j)$, which coincides with the total weighted tardiness when all earliness weights w'_j are set to zero. Balakrishnan et al. (1999) modeled the problem $Q | ST_{sd} | \sum w'_j E_j + \sum w''_j T_j$, while Zhu and Heady (2000) and Akyol and Bayhan (2008) addressed the problem $R | ST_{sd} | \sum w'_j E_j + \sum w''_j T_j$ by *Mixed-Integer Programs* (MIPs). The former two were able to solve instances

85 with up to 12 (9 respectively) jobs and 3 machines using a commercial solver. The latter were also not able to solve larger instances with their exact algorithm based on artificial neural networks.

Lin and Hsieh (2014) present a MIP model for $R \mid ST_{sd} \mid \sum w_j T_j$ and use a commercial solver to solve instances with up to 12 jobs on 3 machines. 90 Tavakkoli-Moghaddam and Aramon-Bajestani (2009), Lopes and de Carvalho (2007), and Lopes et al. (2014) solved the same scheduling problem by presenting b&b and b&p algorithms based on MIP formulations. The former were capable of solving instances with up to 10 jobs and 4 machines, while the latter two were able to solve larger instances with up to 150 (180 respectively) jobs and 95 50 machines within slightly more than one hour (four hours respectively).

Regarding WTSDS, Tanaka and Araki (2013) implemented an exact algorithm capable of solving instances with up to 85 jobs within a maximum of 34 days, while Nogueira et al. (2019) proposed several MIP formulations which were used to solve instances with up to 20 jobs within one hour. Sewell et al. 100 (2012) considered the problem without weights, $1 \mid ST_{sd} \mid \sum T_j$, and developed an exact algorithm able to solve instances with up to 45 jobs within half an hour.

In addition to the paucity of research on exact algorithms for machine scheduling problems, none of the exact approaches cited in this Section exploits the opportunities of parallel computation. In our paper, we address both 105 of these issues by adopting the algorithm of Lopes and de Carvalho (2007) for WTSDS, modifying its branching strategy, developing a primal heuristic to obtain upper bounds more frequently, and presenting and computationally evaluating parallelized versions of the adapted algorithm.

110 2.2. Parallel Branch-and-X Algorithms

Parallel b&x algorithms have been applied to various types of problems. The list of considered problem types comprises assignment problems (Galea and Le Cun (2011)), graph theory problems (e.g., Christou and Vassilaras (2013)), knapsack problems (e.g., Ismail et al. (2014)), mixed integer linear programs

115 (e.g., Carvajal et al. (2014)), stochastic optimization problems (e.g., Aldasoro et al. (2017)), and traveling salesman problems (e.g., Ozden et al. (2017)). Furthermore, flow shop scheduling problems (e.g., Chakroun et al. (2013a,b)), job shop scheduling problems (e.g., Adel et al. (2016); AitZai and Boudhar (2013)), parallel machine scheduling problems (Rauchecker and Schryen (2015, 2018))
120 and the problem $1 \parallel \sum T_i$ (Wodecki, 2008) are considered; for a comprehensive overview of the application of parallel b&x algorithms to optimization problems, see Schryen (2020).

According to the framework for parallel optimization in operations research introduced by Schryen (2020), there are three types of parallelization strategies
125 for b&x algorithms. These are based on the taxonomy of Crainic and Toulouse (2003) and Gendron and Crainic (1994). In the first strategy, which is based on domain decomposition, the solution space is split, and the partitions are explored concurrently, which for b&x algorithms means that multiple active nodes of the b&b tree are explored simultaneously. For this concept, Schryen (2020) introduced the term *coarse-grained intra-algorithm* parallelism. In coarse-grained
130 inter-algorithm parallelism, the second strategy, the solution space is not decomposed, but multiple b&x procedures are executed concurrently on the same solution space. In the third strategy, which Schryen (2020) calls *fine-grained intra-algorithm* parallelism, only a small, predefined part of the algorithm is
135 parallelized when exploring a local region. Hence, this is also called a low-level strategy. This entails that the computation of an operation at a single b&b node is executed in parallel.

Rauchecker and Schryen (2015) implemented a fine-grained intra-algorithm parallelization strategy in which solving the pricing problem and the branching
140 decision is parallelized. Chakroun et al. (2013b) chose the same parallelization strategy by computing the lower bound of each node of the tree in parallel on graphics processing units (GPUs). Carvajal et al. (2014) chose a coarse-grained inter-algorithm parallelization strategy. They execute several configurations of a solver in parallel, each applying a b&b procedure on the same mixed-integer
145 linear programming problem while sharing information with each other.

In all other studies on parallel b&x algorithms (based on the review by Schryen (2020)), a coarse-grained intra-algorithm parallelization strategy is realized (e.g., Aldasoro et al. (2017), Christou and Vassilaras (2013), Ismail et al. (2014), Ozden et al. (2017), and Rauchecker and Schryen (2018)). This is the most natural, since most straightforward, way to parallelize b&x algorithms (Schryen, 2020). Galea and Le Cun (2011) combined the coarse-grained with a fine-grained intra-algorithm parallelization strategy in a hybrid approach. In their article, the computation of the lower bound is vectorized and then parallelized by means of single instruction, multiple data (SIMD) instructions. Adel et al. (2016) also implemented a two-level parallelization on a GPU. On the one hand, the computation of the lower bound at each node is executed in parallel, on the other hand, multiple nodes are explored in parallel. These examples also show that fine-grained and coarse-grained intra-algorithm parallelism are not mutually exclusive and can be combined in hybrid approaches.

Regarding scheduling problems, there appears a high diversity of parallelization strategies. Chakroun et al. (2013b) and Rauchecker and Schryen (2015) use a fine-grained intra-algorithm parallelization strategy for a flow shop scheduling problem and a parallel machine scheduling problem, respectively. Chakroun et al. (2013a), Chakroun and Melab (2015), Gmys et al. (2016, 2017), Mez maz et al. (2014), and Vu and Derbel (2016) apply coarse-grained intra-algorithm parallelization to flow shop scheduling problems. AitZai and Boudhar (2013) and Rauchecker and Schryen (2018) also present coarse-grained intra-algorithm parallelization strategies for a job shop scheduling problem and a parallel machine scheduling problem, respectively. Finally, Adel et al. (2016) introduce a hybrid approach of coarse-grained and fine-grained intra-algorithm parallelization for a job shop scheduling problem.

In our paper, we apply a fine-grained intra-algorithm parallelization to the (single) machine scheduling problem WTSDDS by parallelizing the processing of linear relaxations at the b&p tree nodes. In particular, we parallelize the *dynamic programming* algorithm, which is the most time-consuming part of the column generation procedure used to solve the linear relaxations. As this en-

tails solving a kind of shortest path problem, this approach is not limited to the problem WTSDS, but could be adapted to a variety of other algorithms for optimization problems that involve solving a similar dynamic programming
180 problem. Acknowledging that parallelizing b&x algorithms through a coarse-grained intra-algorithm parallelization by solving nodes of the b&b tree concurrently is the most studied approach, we focus in this work on fine-grained intra-algorithm parallelization. Once the individual effects of the fine-grained parallelization are analyzed, without interfering effects such as early pruning of
185 b&b nodes due to processing them in parallel, a hybridization can be approached in future research.

2.3. Parallel Dynamic Programming Algorithms

Parallel dynamic programming (DP) algorithms have also been applied to numerous types of problem. Kumar et al. (2011), Stivala et al. (2010), and
190 Tran (2010) use parallel DP to solve graph theory problems while Boyer et al. (2012), Rashid et al. (2010), and again Stivala et al. (2010) apply parallel DP to knapsack problems. Aldasoro et al. (2015) and Rauchecker and Schryen (2015) present parallel DP algorithms for a stochastic optimization problem and a parallel machine scheduling problem, respectively. Applications of parallel DP
195 to several other problem types are presented by Boschetti et al. (2016), Dias et al. (2013), Maleki et al. (2016), and Tan et al. (2009).

The only approach for using parallel DP as part of a column generation procedure is presented by Rauchecker and Schryen (2015) for a parallel machine scheduling problem. In their parallelization, the solution space is divided into
200 independent subproblems, each corresponding to exactly one of the parallel machines, which are solved in parallel. This strategy is not applicable to WTSDS since there is only a single machine in WTSDS. Consequently, we present a new parallel DP approach for executing the column generation procedure in our b&p algorithm in parallel.

3. Decision Support Model

To present a decision support model for WTSDS, we rely on Lopes and de Carvalho (2007), who investigate a generalization of WTSDS. Let $\{1, \dots, n\}$ be a set of jobs which must be processed by the single machine. For each job $j \in \{1, \dots, n\}$, the due date is denoted by d_j , the processing time by p_j , and its weight by w_j . We denote the setup times between two jobs i and j by s_{ij} . A feasible 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 and represents the order in which the jobs are processed on the machine. We denote the set of all feasible schedules by Ω . The parameter $a_{j\omega} \in \{0, 1\}$ indicates whether job j is contained in schedule ω . Furthermore, the weighted tardiness of a schedule ω , which is formalized at the end of this section in equation (5), is denoted by T_ω . For each schedule $\omega \in \Omega$ we introduce a binary decision variable x_ω which equals 1 if ω is operated and 0 otherwise. Consequently, WTSDS can be formulated as a binary linear program:

$$\min \sum_{\omega \in \Omega} T_\omega \cdot x_\omega \quad (1)$$

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

$$\sum_{\omega \in \Omega} x_\omega \leq 1 \quad (3)$$

$$x_\omega \in \{0, 1\} \quad \omega \in \Omega \quad (4)$$

The objective function (1) represents the total weighted tardiness. Constraints (2) ensure that each job is processed exactly once, while constraint (3) guarantees that at most one schedule $\omega \in \Omega$ is allowed on the machine.

The weighted tardiness T_ω of a schedule $\omega = (j_1, \dots, j_h) \in \Omega$ is defined as

$$T_\omega = \sum_{\ell=1}^h \max\{C_{j_\ell} - d_{j_\ell}, 0\} \cdot w_{j_\ell} \quad , \quad (5)$$

where C_{j_ℓ} denotes the completion time of job j_ℓ . Then $C_{j_1} = s_{0j_1} + p_{j_1}$, where s_{0j} represents the initial setup time to process job j first, and $C_{j_r} = C_{j_{r-1}} + s_{j_{r-1}j_r} + p_{j_r}$ for all $2 \leq r \leq h$.

4. A Serial Branch-And-Price Algorithm

The model presented in the previous section can be solved by a b&p algorithm. A generic description of this type of algorithm, first conceptualized by Barnhart et al. (1998), is presented in Algorithm 1.

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

- 1: Solve linear relaxation of root node using column generation
 - 2: Initialize set of active nodes
 - 3: **repeat**
 - 4: Select an active node
 - 5: Branch on selected node
 - 6: Solve new nodes' relaxations using column generation
 - 7: Update set of active nodes
 - 8: **until** set of active nodes is empty
-

In the first step of the algorithm (line 1), the linear relaxation of the root node is solved by column generation, which is described in Section 4.2. Regarding WTSDS instances, the root node of the b&b tree corresponds to the problem modeled in (1)–(4), where constraints (4) are relaxed to $0 \leq x_\omega \leq 1$ for all $\omega \in \Omega$ (actually, $x_\omega \geq 0$ suffices due to constraint (3)). If the optimal solution of the root node relaxation is integer, an optimal solution for WTSDS has been found. Otherwise, the set of active nodes is initialized with the root node (line 2).

After initializing the set of active nodes, lines 4–7 are repeated until there are no more active nodes left. First, one of the active nodes is selected to be branched on. Branching creates two child nodes that are added to the set of active nodes while the parent node is removed. The node selection (line 4) and the corresponding branching strategy (line 5) are specified in Section 4.1. The next step is to solve the child nodes, again using column generation (line 6). Based on their optimal solutions, the set of active nodes of the b&b tree is updated (line 7). In the course of the update, an active node is declared as

inactive in three cases. First, if the selected node’s relaxation is not feasible, or second, if it has an integer optimal solution. The first case cannot happen for our algorithm, since all columns are inherited from the parent node, but those
 235 which become infeasible due to branching are heavily penalized in the objective function. Regarding the third case, note that in a minimization problem, the optimal solution of a node’s relaxation constitutes a lower bound on its optimal integer solution. Therefore, third, a node is declared as inactive if the solution value of the node’s relaxation is greater than the value of the best known integer
 240 solution found so far in the tree, which is called *pruning* (or *fathoming*).

Once there are no active nodes left, the current best integer solution is also the optimal solution for the WTSDS instance.

4.1. Node Selection and Branching Strategy

For node selection, we apply *best-first search*, where the active node with the lowest lower bound is selected to be explored next. For branching on a selected node, we follow an established approach, see for example Chen and Powell (1999) or Akker et al. (1999), and introduce branching decision variables X_{ij} , which are defined as

$$X_{ij} = \sum_{\omega \in \Omega} \delta_{ij\omega} \cdot x_{\omega} \leq 1 \quad (6)$$

for every $i = 0, \dots, n$ and $j = 1, \dots, n$, where $\delta_{ij\omega} \in \{0, 1\}$ indicates whether
 245 job i is processed immediately before job j in schedule ω while $(x_{\omega})_{\omega \in \Omega}$ is an optimal solution of the selected node’s relaxation. Hereby, $\delta_{0j\omega}$ is defined as 1 if j has no predecessor i in schedule ω because j is the first job processed in ω , and 0 otherwise. In case x_{ω} is binary for all schedules $\omega \in \Omega$, X_{ij} indicates whether job i is processed immediately before job j or not. Accordingly, X_{0j}
 250 indicates whether job j is processed first or not. Note that $a_{j\omega} \in \{0, 1\}$ in (2) and $\delta_{ij\omega} \in \{0, 1\}$ in (6) will be relaxed to $a_{j\omega}, \delta_{ij\omega} \in \mathbb{N}$ to significantly reduce the state space of the dynamic programming procedure during the pricing phase (Lopes and de Carvalho, 2007).

The branching strategy we use is a modified version of *most-fractional branching* (abbreviated by MF). While in MF a variable with a value closest to 0.5 is selected, we also use problem-specific information. Our strategy for selecting a branching edge therefore consists of two steps. First, we determine a set A of the $\frac{n \cdot (n+1)}{100}$ edges (i, j) with the lowest values for $|X_{ij} - 0.5|$. In other words, these are the edges with the 1% most fractional values of X_{ij} . Second, from this set, we select the edge (i^*, j^*) with the lowest average weighted position over all feasible schedules ω , calculated by

$$(i^*, j^*) = \arg \min_{(i,j) \in A} \sum_{\omega \in \tilde{\Omega}} x_{\omega} \cdot \text{position}_{ij\omega} \quad , \quad (7)$$

as we want to branch on an edge which is scheduled preferably early in the most promising schedules. Note that for $\text{position}_{ij\omega}$ we only consider the first occurrence of an edge (i, j) in a schedule ω ; a discussion of why there may be more than one occurrence is provided in Section 4.2. The variable $X_{i^*j^*}$ is the one that we branch on.

By branching on $X_{i^*j^*}$, job ordering restrictions are set for the two child nodes of the node that we branch on. In the first child node, $X_{i^*j^*}$ is set to 1. This implies that only schedules in which job i^* is processed immediately before j^* are permitted in this node. In the second child node, $X_{i^*j^*}$ is set to 0, which means that only schedules with job i^* not being processed directly before j^* are feasible for this node. For both child nodes, the set of allowed schedules is modified according to the imposed restrictions. Therefore, each node of the b&b tree represents a problem following the structure of model (1)-(4), only each with its node-specific set of schedules. In the following, we denote these sets by $\tilde{\Omega} \subseteq \Omega$, where each $\omega \in \tilde{\Omega}$ complies with every branching decision that led to the selected node.

4.2. Column Generation Process

In this Section, we detail the column generation procedure used to solve the linear relaxations of the b&b nodes (lines 1 and 6 in Algorithm 1). All algorithms in this Section are adopted unchanged from Lopes and de Carvalho (2007). As

noted above, b&b node relaxations all have the same structure – only differing
 275 by node-specific sets of feasible schedules $\tilde{\Omega}$ – and can therefore be solved by a
 single column generation procedure, which we present in Algorithm 2.

Algorithm 2 Column Generation Procedure

- 1: **repeat**
- 2: Solve a restricted form of the original LP called the *restricted LP* considering only a (typically small) subset of variables
- 3: Solve the *pricing problem* for the restricted LP:
 - 3.1: Let (π, σ) denote the optimal dual solution of the restricted LP, i.e., π_j is the dual variable corresponding to job j in constraints (2) and σ is the dual variable corresponding to constraint (3).
 - 3.2: Determine if there is any variable, i.e., column, x_ω in the original LP that has a negative reduced cost

$$r_\omega = T_\omega - \sum_{j=1}^n a_{j\omega} \pi_j - \sigma \quad (8)$$

- 4: Add the variable x_ω with the least reduced cost to the restricted LP if the least reduced cost

$$r^* = \min_{\omega \in \tilde{\Omega}} r_\omega = \min_{\omega \in \tilde{\Omega}} \left\{ T_\omega - \sum_{j=1}^n a_{j\omega} \pi_j - \sigma \right\} \quad (9)$$

is negative, i.e., $r^* < 0$

- 5: **until** there are no more variables with negative reduced cost, i.e., an optimal solution is found
-

The linear relaxation of a b&b node is referred to as the *original LP*. By considering only a small subset of variables, the original LP is initially scaled down to a so-called *restricted LP* (line 2). To build the initial restricted LP
 280 of a node in the first execution of line 2, an initial set of variables is required. For the root node, this initial set can be generated by any WTSDS heuristic. We use the solution heuristic suggested by Lee et al. (1997) for our algorithm. For all other nodes, the initial set is inherited from the parent node. Note

that branching on an edge $(i^*.j^*)$ creates two child nodes, in one of which the
 285 job i^* is prohibited from directly preceding job j^* . The way the edge was
 calculated in (7), this leads to at least one infeasible schedule ω , where $x_\omega > 0$
 holds in the solution of the parent node. To ensure a feasible linear program,
 schedules which become infeasible must still be accessible in the next iteration
 when the child node is processed. Otherwise, the dual variables needed in the
 290 pricing problem (see line 3 in Algorithm 2) cannot be computed. Hence, these
 schedules must not be removed from the problem immediately, but are penalized
 in the objective function by a sufficiently large value instead. Note that the
 alternative of introducing an artificial schedule $\bar{\omega}$ containing all jobs also adds
 the corresponding coefficients $a_{j\bar{\omega}}$ to the model, which affects the computation
 295 of the dual variables π_j and could ultimately have an impact on the column
 generation process. Since infeasible columns are only added to model in one
 more iteration, the size of the LP is not an issue.

After that, in line 3, the so-called *pricing problem* is solved, which corre-
 sponds to identifying the variable (i.e., column) x_ω with the least reduced cost.
 300 The pricing problem is solved by dynamic programming, which is described later
 in Algorithm 3. In line 4, the variable x_ω with the least reduced cost (if nega-
 tive) is added to the restricted LP. If there are no more columns with negative
 reduced costs, the optimal solution obtained for the restricted LP also serves as
 an optimal solution for the original LP and thus for the linear relaxation of the
 305 processed b&b node.

To solve the pricing problem in line 3 of Algorithm 2, a dynamic program-
 ming algorithm, which is described in Algorithm 3, is used. For this algorithm
 to be viable, we have to allow for cyclic schedules, where jobs may be processed
 more than once in the same schedule $\omega \in \Omega$. This reduces the state space of the
 310 dynamic programming algorithm for the pricing problem from exponential to
 pseudo-polynomial size, since only the immediate predecessor of each job needs
 to be remembered. This entails that $a_{j\omega}$ and $\delta_{ij\omega}$ for jobs i and j in a schedule
 ω are no longer necessarily binary for linear relaxations of the WTSDS. Note
 that this does not affect the optimal solution of an WTSDS instance, since the

Algorithm 3 Solving the Pricing Problem

- 1: Initialize $f(t, j) = \infty$ for each time $t \leq 0$, and job j .
- 2: Initialize $f(0, 0) := -\sigma$ and $f(t, 0) = \infty$ for each time $0 \neq t \leq T$.
- 3: For each time $1 \leq t \leq T$, and job j , set

$$f(t, j) = \min_{i \in \mathcal{P}_j} f(t - s_{ij} - p_j, i) + \max\{t - d_j, 0\}w_j - \pi_j. \quad (10)$$

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

$$r_T^* = \min_{t=0, \dots, T} \min_{j=0, \dots, n} f(t, j). \quad (11)$$

315 exactly-once processing of each job is ensured by constraints (2).

The set of all possible predecessors of job j , denoted by \mathcal{P}_j , differs for each node as it depends on node-specific ordering restrictions imposed by the branching strategy. We define $f(t, j)$ for a time $t \in \mathbb{Z}$ and a job j as the minimum reduced cost of a variable x_ω , where job j is completed at time t as the last
320 job being processed in schedule ω . For each variable x_ω , reduced costs $f(t, j)$ for all jobs j at all times $t \leq T$ (with an upper bound T on the makespan of an integer optimal solution of the current node) are calculated recursively to identify the minimum reduced cost. Based on that, the corresponding schedule with minimum reduced cost under the time limit T can be constructed reversely.

325 Furthermore, we implement two methods for enhancing the efficiency. On the one hand, we start with a low value of T and successively adjust the time limit until there are no more variables with negative reduced costs for two consecutive values of T . On the other hand, we significantly reduce the solution space of the pricing problem by considering only so-called *decreasing reduced cost schedules* under certain conditions and by preventing generated schedules
330 from containing sequences of the form (i, j, i) (so called *2-cycles*). For details, see Lopes and de Carvalho (2007).

4.3. Primal Heuristic

After the column generation process terminates, the resulting solution of the
 335 *restricted problem* $(x_\omega)_{\omega \in \Omega}$ is fractional for most nodes of the b&b tree. Additionally, since we allow cyclic schedules to be generated, there will be infeasible
 schedules ω in the solution of the LP; i.e., $x_\omega > 0$ holds for such schedules. To illustrate that this phenomenon, consider an instance with jobs 1, 2, 3, 4 and
 suppose the schedules $\omega_1 = (1, 2, 1, 2)$ and $\omega_2 = (3, 4, 3, 4)$ have been generated.
 340 We can confirm that $x_{\omega_1} = x_{\omega_2} = \frac{1}{2}$ is feasible for the relaxation of the set
 partitioning model (1)–(4). Although both of these schedules are infeasible for
 the original problem, they still contain sequencing information because their
 columns were generated based on their corresponding reduced costs. Thus, ω_1
 and ω_2 indicate that it’s favorable to have jobs 2 and 4 directly succeed jobs 1
 345 and 3.

This sequence information has already been used in Section 4.1 to select the
 branching edge and stored in the variables X_{ij} from Equation (6). We construct
 a feasible schedule by iteratively using the largest values of X_{ij} ; i.e., we start
 with the (artificial) job, set $i = 0$, schedule

$$j = \arg \max_{k \text{ unscheduled}} X_{ik} \quad (12)$$

next and continue this way with j as the next i until all jobs are sequenced or
 the maximum value of X_{ij} is zero. In the latter case, there are jobs left which
 have not been scheduled yet (as would be the case in the example above), and
 the remaining jobs are appended to the sequence in ascending order of their
 350 (*weighted*) *due dates* (EDD rule).

Since the X_{ij} values have already been computed prior to the primal heuristic,
 the above procedure is very efficient in terms of additional computation
 time. Thus, we follow the idea of Atakan et al. (2017) to extend the primal
 heuristic by a simple local search procedure. We start by iteratively considering
 355 all possible swaps of two jobs in our sequence until the best swap among them
 does not lead to any improvement of the current schedule’s objective value (2-
 opt). We then apply the same procedure with three jobs at a time (3-opt). This

extension is applied whenever the primal heuristic described above generates a solution which is at most 10% worse than the upper bound found so far by the b&b algorithm, or whenever an improving integer solution is found. With these
 360 restrictions, preliminary experiments showed that only a few improvements are made each time, and thus there is no need to set a tight iteration limit for the two extensions in order to reduce computation time.

When abbreviations are used to refer to an algorithm, if the primal heuristic
 365 is applied, it is indicated by the suffix “PH”.

5. Parallelization of the Serial Branch-And-Price Algorithm

In this section, we present our parallelization strategies for the serial b&p algorithm presented in Section 4. We use fine-grained intra-algorithm parallelization, see Section 2, by solving the linear relaxations of individual b&p tree
 370 nodes in parallel (lines 1 and 6 in Algorithm 1). This corresponds to parallelizing the column generation procedure (Algorithm 2). We identified the dynamic programming (Algorithm 3) as the most time-consuming part of the column generation procedure, and therefore this part is chosen for parallelization.²

The crucial part of dynamic programming is to calculate $f(t, j)$ for all time
 375 slots $1 \leq t \leq T$ and for all jobs $0 \leq j \leq n$. Fixing $1 \leq t_0 \leq T$ and using parallel threads to calculate $f(t_0, j)$ simultaneously for all jobs j proved to be inefficient in our pretests. Therefore, we use each parallel thread for calculating all values $f(t, 0), f(t, 1), \dots, f(t, n)$ for several $1 \leq t \leq T$. However, there are data dependencies between the values $f(t, j)$ and $f(t', j')$ for different time slots t, t' and
 380 jobs j, j' , as we can see from the definition in equation (10). Hence, parallel dynamic programming may be obstructed by waiting times. Therefore, reducing

²The other non-trivial part of the column generation procedure is the repeated solving of restricted LPs. This can be easily parallelized by off-the-shelf solvers. However, we did not observe any substantial benefit from using solver parallelization (probably because the restricted LPs are too small for effective parallelization), and therefore we do not follow this approach in our paper.

waiting times to a minimum constitutes the prime challenge in parallelizing our dynamic programming algorithm.

Also, these dependencies lead to an instance-specific number of threads
385 where we can expect to observe little positive (or even negative) effects from using additional threads. For an instance of WTSDS, let $\bar{T} = \max_{i,j} s_{ij} + p_j$ be the maximum amount of time that can elapse before another job can be processed, and let us assume that we use at least $\bar{T} + 1$ threads. In this scenario, the thread processing the latest time slot is exclusively accessing time
390 slots which are all still being processed by other threads and therefore contain missing entries. This phenomenon is evaluated and discussed in Section 6.

We introduce two versions of parallel dynamic programming, which we explain in below. Both versions are implemented using the OpenMP shared-memory programming paradigm (OpenMP, 2015), which allows code to be
395 executed in parallel on multiple shared-memory threads using the statement `#pragma omp parallel for`. We use this pragma to parallelize the outer loop $t = 0, \dots, T$ of dynamic programming on multiple OpenMP threads.

The first version is shown in Algorithm 4 and performs what we refer to as *strict parallel dynamic programming (s-pdp)*. Whenever the reduced cost
400 $f(t - s_{ij} - p_j, i)$ for any possible predecessor i of job j is not available at the time when the calculations for job j at the time t are trying to access the value, *s-pdp* waits for it to become available before continuing, as shown in lines 5–7 of Algorithm 4.³

³Due to potential synchronization delays, simply waiting for the values to become available is not economical. In OpenMP, a thread uses a cache where it temporarily holds data from the shared memory. For efficiency reasons, consistency between this temporary cache and the shared memory is not always given (Hoffmann and Lienhart, 2008, p.109). Therefore, in many cases, the required values may be already calculated, but may just not yet be accessible due to a lack of synchronization. Consequently, if a thread determines that a shared variable value required for its calculations is not yet available, it first updates its cache to ensure that its cached values are up-to-date. An update of the required data can be initiated by a so-called *flush* directive, which synchronizes a thread’s cache and the shared memory.

Algorithm 4 Strict Parallel Dynamic Programming (s-pdp)

```
1: #pragma omp parallel for
2: for  $t = 0, \dots, T$  do
3:   for  $j = 1, \dots, N$  do
4:     for  $i \in \mathcal{P}_j$  do
5:       while  $f(t - s_{ij} - p_j, i)$  is not available do
6:         flush  $f(t - s_{ij} - p_j, i)$ 
7:         read  $f(t - s_{ij} - p_j, i)$ 
8:       write  $f(t, j) = \min_{i \in \mathcal{P}_j} f(t - s_{ij} - p_j, i) + \max\{t - d_j, 0\}w_j - \pi_j$ 
```

In our second version, dynamic programming is executed on parallel threads without verifying the values of shared variables. We refer to this as *loose parallel dynamic programming (l-pdp)*, which is described in Algorithm 5. In the case that the reduced cost $f(t - s_{ij} - p_j, i)$ is not yet available, calculations are still continued using the default value for $f(t - s_{ij} - p_j, i)$ set in line 1. Consequently, the values for $f(t, j)$ returned by *l-pdp* may be incorrect.

Algorithm 5 Loose Parallel Dynamic Programming (l-pdp)

```
1:  $f(t, j) = \infty \quad \forall 0 \leq t \leq T$  and  $\forall 1 \leq j \leq N$ 
2: #pragma omp parallel for
3: for  $t = 0, \dots, T$  do
4:   for  $j = 1, \dots, N$  do
5:     for  $i \in \mathcal{P}_j$  do
6:       read  $f(t - s_{ij} - p_j, i)$ 
7:       write  $f(t, j) = \min_{i \in \mathcal{P}_j} f(t - s_{ij} - p_j, i) + \max\{t - d_j, 0\}w_j - \pi_j$ 
```

Based on these two versions of parallel dynamic programming, we implemented two exact parallel b&p algorithms, which are explained below:

- *Strict Parallel Branch-and-Price (SPBP)*: Use *s-pdp* to solve the pricing problem during column generation (line 3 of Algorithm 2).
- *Hybrid Parallel Branch-and-Price (HPBP)*: Use *l-pdp* to solve the pricing

415 problem during column generation (line 3 of Algorithm 2). If more than
one thread is used and when there is no more column with negative reduced
cost according to *l-dpd*, permanently switch to *s-dpd* to solve the pricing
problem during column generation (line 3 of Algorithm 2).

Since *s-dpd* and *l-dpd* coincide when using a single OpenMP thread, both
420 SPBP and HPBP coincide with the serial b&p algorithm from Section 4 when
executed on a single OpenMP thread (serial execution).

6. Computational Experiments

In this section, we specify the design of our experiments and discuss the
results and findings from their execution. First, Section 6.1 describes the com-
425 putational environment in which the experiments were conducted and introduces
the benchmark instances used to evaluate our algorithms. Next, an initial set
of experiments to analyze our serial algorithm and its features is discussed in
Section 6.2. An important goal of this section is to identify the best perform-
ing combination of features in our algorithms, for which the computationally
430 expensive parallel execution will be evaluated next. The following Section 6.3.1
details the in-depth experiments conducted to evaluate the performance bene-
fits obtained from our two parallelization approaches. Finally, we discuss and
compare the scalability of our two parallelization strategies in 6.3.2, analyzing
which and how the different facets of our algorithm affect it.

435 6.1. HPC Environment and Benchmark Instances

The experiments are conducted on the Linux-based HPC cluster *Noctua 2* of
the *Paderborn Center for Parallel Computing* (PC²) at Paderborn University.
We used a two-socket AMD Milan 7763 shared-memory system with 64 cores per
socket, a clock speed of 2.45 GHz per core, and a total of 256 GB main memory.
440 The upper bound on the running time (wall time) for a single compute task
executed on the cluster is 21 days (or 504 hours). The b&p algorithms are
coded in C++ and compiled by the g++ (v12.2.0) compiler with optimization flag

-03. To solve the restricted linear programs during column generation (line 2 of Algorithm 2), we use the *Gurobi* 10.0.3 API. The parallelization on shared
445 memory is based on OpenMP 4.5.

Three established benchmark sets from the literature are used to evaluate the proposed parallel algorithms: First, the algorithms are tested on the 120 benchmark instances from Cicirello (2003) and Cicirello (2009)⁴, which are of problem type 1 | ST_{sd} | $\sum w_j T_j$. Each instance consists of 60 jobs with processing times p_j and weights w_j generated from the integer uniform distribution
450 between 50 and 150, and 0 and 10, respectively. The due dates d_j and setup times s_{ij} are characterized by the three parameters τ , R and η , which define the tightness and the range of the due dates, as well as the size of the average setup time with respect to the size of the average processing time. Cicirello
455 (2003) and Cicirello (2009) created twelve combinations of parameter settings and generated 10 instances for each combination. In the following, we refer to the benchmark set of Cicirello (2003) and Cicirello (2009) as the *Cicirello set*.

Second and third, we apply our algorithms to the benchmark sets from Rubin and Ragatz (1995) and Gagné et al. (2002).⁵ These benchmark instances are of
460 type 1 | ST_{sd} | $\sum T_j$. Our algorithms are applicable to this kind of problem by defining the weights for all jobs as one. Rubin and Ragatz (1995)'s set comprises 32 instances with 15, 25, 35 and 45 jobs while Gagné et al. (2002)'s set comprises 32 instances with 55, 65, 75 and 85 jobs. In both sets, processing times are normally distributed with a mean of 100, and setup times are uniformly
465 distributed between 0 and 20. Furthermore, the instances are characterized by the parameters processing time variance, tardiness factor, and due dates range, which build eight possible parameter settings. We refer to the benchmark sets of Rubin and Ragatz (1995) and Gagné et al. (2002) as the *Rubin set* and the *Gagné set*, respectively.

⁴<https://www.cicirello.org/datasets/wtsds/> (last accessed 07/15/24)

⁵http://www.uqac.ca/portfolio/carolinegagne/recherche/ordonnancement_n_travaux/ (last accessed 07/15/24)

470 6.2. Serial Performance

In this section, we evaluate the performance of our serial algorithms, in particular, we investigate which benefits regarding running times are gained from the modified branching strategy and the primal heuristic introduced in Sections 4.1 and 4.3. For this series of experiments, we consider all four combinations
 475 of the two features in the algorithm. The versions examined are summarized in Table 1.

Table 1: Tested versions of our serial algorithms

Acronym	Primal Heuristic	Branching Strategy
SPBP-PH	✓	modified
SPBP		modified
MF-SPBP-PH	✓	most-fractional
MF-SPBP		most-fractional

^{SPBP} Strict Parallel Branch-and-Price

^{PH} Primal Heuristic

^{MF} Most-Fractional Branching

We apply the algorithms to all instances from the three instance sets and set a time limit of one hour. Table 2 shows the number of instances solved within the time limit for SPBP-PH, SPBP, MF-SPBP-PH and MF-SPBP. Detailed
 480 running times for all instances that were solved within one hour by at least one version of the algorithm are given in Appendix A. From the Rubin set, both versions using the modified branching strategy, SPBP-PH and SPBP, solved 29 instances, while MF-SPBP-PH and MF-SPBP, using *most-fractional branching*, both solved 27 of the instances. For the Gagné set, all four versions of the
 485 algorithm solved the same 14 instances within the time limit. As for the Cicirello instances, the differences are more distinctive, with 53, 45, 27, and 25 instances solved by the algorithms, respectively.

From these numbers, we can see that each of the two features individually

Table 2: Solved instances within the first hour for SPBP-PH, SPBP, MF-SPBP-PH and MF-SPBP

#solved	SPBP-PH	SPBP	MF-SPBP-PH	MF-SPBP	(total)
Rubin	29	29	27	27	(32)
Gagné	14	14	14	14	(32)
Cicirello	53	45	27	25	(120)
total	96	88	69	66	(184)

has a positive effect on the total number of instances solved within the time
490 limit, and thus both features enhance the performance of the serial algorithm.
In addition, the branching strategy affects the benefits gained from the primal
heuristic, with 8 more instances solved with the modified strategy compared
to two instance when *most-fractional branching* is applied. Since both features
are from the serial portion of our algorithm, all further experiments will be
495 performed with both the modified branching strategy and the primal heuristic.

Before proceeding to the parallelization of the two identified algorithms,
another parameter we analyzed is the number of columns per iteration (i.e., per
call of the dynamic programming algorithm) to be added to the *restricted LP*.
In Line 9 of Algorithm 2, only a single column with minimal reduced costs is
500 selected, whereas in fact often more than one *negative reduced cost schedule* may
be computed in a single call to the DP algorithm. Especially in the early stages
of the b&b procedure, this can cause the same column to be generated again in
a subsequent iteration if it was not added to the *restricted LP* because another
column was generated at the time with even lower reduced costs. Preliminary
505 tests with up to ten columns per iteration showed that (up to) four columns per
iteration resulted in a robust trade-off between saving calls to the DP algorithm
without inflating the size of the *restricted LP*. This number was used in all
experiments, including those discussed above.

6.3. Parallel Performance

510 The focus of this section is on the parallelization of our algorithms; i.e., we study the impact of the number of threads used during column generation for both the strict and hybrid approaches. In Section 6.3.1, we discuss the results of the experiments in terms of solved instances, before analyzing our algorithms in terms of speedup; i.e., the benefit gained from using additional computational
515 resources, in Section 6.3.2.

Due to the results discussed in Section 6.2, we consider the algorithm using both the modified branching strategy and the primal heuristic, that is, algorithms SPBP-PH and HPBP-PH. The former algorithm uses the strict strategy in dynamic programming, where threads wait for missing values to become
520 accessible before continuing with the next calculation. The latter uses the hybrid strategy, where missing values are ignored for as long as possible before switching to the strict mode. Both algorithms are executed on all instances of the three benchmark sets, using $th \in \{1, 2, 4, 8, 16, 32, 64, 128\}$ threads for up to 504 hours (21 days).

525 To compare our results with other exact methods, we use as reference data the algorithm used of Tanaka and Araki (2013). [Another exact algorithm worth mentioning is from a study by Sewell et al. \(2012\).](#) The authors addressed the problem without tardiness weights in the objective function. Using their Branch-and-Bound-and-Remember (BB&R) algorithm, they were able to solve most
530 instances of the Rubin set and some of the Gagné set within 30 minutes. Since they did not consider weights, the authors also did not apply their algorithm to the Cicirello set, and the reported results were not superior to those of the former study, we will only report results from the former study for comparison. Since the ~~their~~ [aforementioned](#) study is the only one in the literature to report
535 (provably) optimal solutions for most larger instances (up to 85 jobs), their algorithm can be considered the state-of-the-art in exact solution methods for WTSDS. Due to improvements in hardware performance since the release of their study, we re-ran their program with the same settings as in their study, but on the same hardware as we used to run our experiments (remark: we did

Table 3: Solved instances over time using 1 and 64 threads in HPBP-PH

Set	#instances	#threads	Timelimit (h)				(Tanaka*)
			1	12	48	504	(336 h)
Rubin	32	1	29	29	31	32	(32)
		64	29	31	32	32	
Gagné	32	1	14	21	26	30	(29)
		64	21	27	30	31	
Cicirello	120	1	53	99	107	116	(118)
		64	97	111	117	117	

^{HPBP-PH} Hybrid Parallel Branch-and-Price with Primal Heuristic

540 not use the preprocessed instances Nos. 1–40 of the Cicirello set to run our two algorithms). Although, we could not run the very long jobs (30 days and longer), since all running times are down by up to more than 50% (with very few exceptions), we use these results as reference in all following tables (in this Section, marked with a “*”). For details on the re-enacted experiments and the
545 individual running times, see Appendix C.

6.3.1. Impact of Parallelization

The performance results for our two algorithms HPBP-PH and SPBP-PH in terms of solved instances are presented in Tables 3 and 4, respectively. For each set of benchmark instances, the tables show the total number of instances in the
550 set and the number of instances that our algorithms were able to solve within one hour, twelve hours, two days (48 hours) and 21 days (504 hours). These numbers are shown for the algorithms running in serial mode (one thread) and using 64 parallel threads. The last column in both tables shows the number of instances solved by the algorithm from Tanaka and Araki (2013) for comparison.⁶ We

⁶Note, that for instance 751 from the Gagné set and instance No. 7 from the Cicirello set, HPBP-PH on a single thread took more than 14 days (336 hours) to prove optimality. The

Table 4: Solved instances over time using 1 and 64 threads in SPBP-PH

Set	#instances	#threads	Timelimit (h)				(Tanaka*)
			1	12	48	504	(336 h)
Rubin	32	1	29	29	31	32	(32)
		64	29	31	32	32	
Gagné	32	1	14	21	26	30	(29)
		64	20	26	30	31	
Cicirello	120	1	53	98	107	116	(118)
		64	95	111	116	117	

^{SPBP-PH} Strict Parallel Branch-and-Price with Primal Heuristic

555 choose the 64-thread version as a representative because this version of the algorithms achieves the best overall performance. The complete data for all numbers of threads considered, as well as detailed running times for all instances (including the state-of-the-art running times from Tanaka and Araki (2013) for comparison) are provided in Appendix A. Since the one-hour runs already
560 showed a massive performance degradation, runs with 128 threads were excluded for all further experiments to reduce the computational quota (just the final execution of our experiments alone consumed about 3 million CPU hours).

Using 64 threads, HPBP-PH solved all 32 instances of the Rubin set within 48 hours, all instances but problem 855 of the 32 instances in the Gagné set, and
565 117 of the 120 instances in the Cicirello set within 21 days to proven optimality. Absolute results for SPBP-PH are comparable. For a closer comparison of the two versions, another measure of performance is discussed in the following section.

For the remaining three instances from the Cicirello set, Nos. 8, 18, and 24,
570 we used 128 GB of RAM which didn't suffice. Since this limit was reached in

same is true for instance 751 from the Gagné set when SPBP-PH was run on a single thread.

less than a week of computing time, we didn't attempt to run them again, even with the available maximum of 256 GB.

We see that for both algorithms, as the number of threads increases, more instances can be solved as well. While using 64 threads results in the same
575 number of solved instances as using 32 threads, running the algorithms with 128 threads actually shows a performance degradation with 10 fewer solved instances from the Cicirello set within the first hour (87 compared to 97 using 64 threads). As mentioned in Section 5, this effect is to be expected, with the values of setup plus processing time ($s_{ij} + p_j$) averaging at 148 for the Cicirello,
580 108 for the Gagné and 104 for the Rubin instances. The same effects are reflected in the following speedup analysis.

For the three instance sets, Tanaka and Araki (2013) solved all instances to proven optimality within 34 days, except for the instances designated as 851 and 855 from the Gagné set. We were able to solve instance 851 to proven optimality
585 within seven hours using 64 threads (3.5 days in serial execution), showing that the aforementioned authors had found the optimal solution with an objective value of 360. While we can see that their algorithm still has considerably lower overall running times (especially for the Cicirello instances), this shows how difficult it is to compare two b&b algorithms that use different approaches to
590 compute bounds. As for instance 855, we were not able to prove optimality within 21 days, but found a solution with an objective value of 256 (current lower bound of 253.263), which was also found by Sewell et al. (2012) and Xu et al. (2014b) using a BB&R and hybrid evolutionary algorithm, respectively.

In comparison to Tanaka and Araki (2013) and our exact algorithms, Chen
595 (2019) was able to find the optimal solution for 119 out of the 120 instances from the Cicirello set within 1500 seconds using their *Iterated Population Based VND* (IPBVND) heuristic. Within a time limit of 100 seconds, IPBVND found all 32 optimal solutions from the Rubin set and 27 solutions from the Gagné set. Using a relaxed time limit of 80.000 seconds, IPBVND also found three of
600 the remaining five best known solutions from the latter set. This shows that heuristic methods can be effective in computing excellent solutions that exact

methods take much longer to obtain. On the other hand, heuristics rely on the results from exact methods to perform such analyses, and in many cases include randomized features. For example, this is also the case for IPBVND, where
605 the reported solution value is the best one obtained from multiple runs of the algorithm. Although Chen (2019) evaluates the robustness of computed solution values among multiple executions of the algorithm, the fact that a single run can produce suboptimal solutions shows that exact methods are still necessary and cannot be discarded.

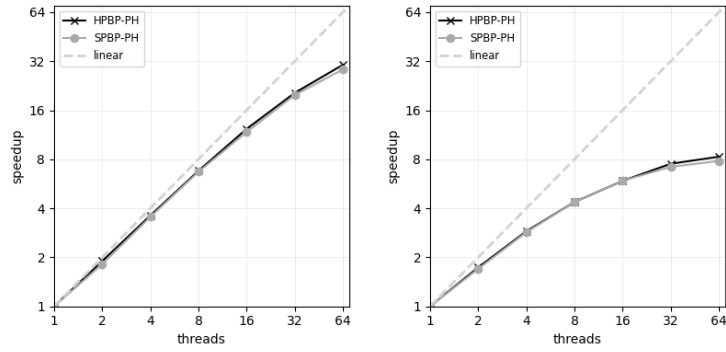
610 6.3.2. Speedup Analysis

To compare our two parallelization strategies, in this section we analyze the scalability of both approaches. We first introduce an established metric used for this task and discuss the results, before examining the various aspects that lead to the observed effects.

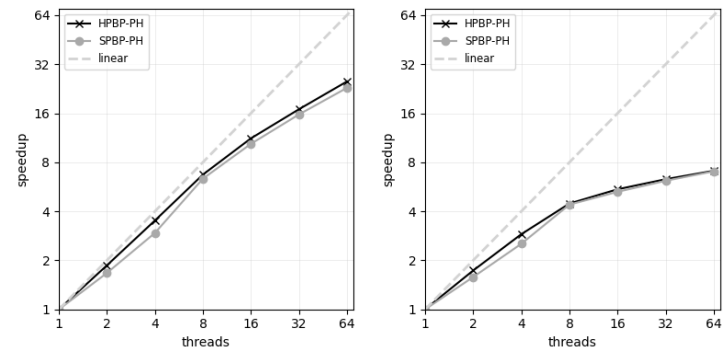
615 To evaluate the potential from parallelization for algorithms SPBP-PH and HPBP-PH, we use established scalability metrics, namely the parallel speedup and the parallel efficiency (Hager and Wellein, 2010, p.120ff). Parallel speedup on R threads is defined as the ratio of the time to execute the parallel algorithm on one thread to the time to execute the parallel algorithm on R threads, which
620 is called a *relative speedup* (Barr and Hickman, 1993). Parallel efficiency on R threads is defined as the parallel speedup on R threads normalized to the number of threads R .

The median parallel speedups for the three benchmark sets are shown in Figure 1. The left subfigures (a), (c) and (e) show the speedups of dynamic
625 programming (purely parallel part), while the right subfigures (b), (d) and (f) show the overall speedups of the parallel b&p algorithms. The dotted lines represent the so-called *linear speedup*; i.e., a speedup of R on R threads, which corresponds to an efficiency of 100%. Appendix B contains figures for the corresponding parallel efficiencies, as well as the underlying median and average
630 parallel speedup data and their respective standard deviations, also visualized as box plots.

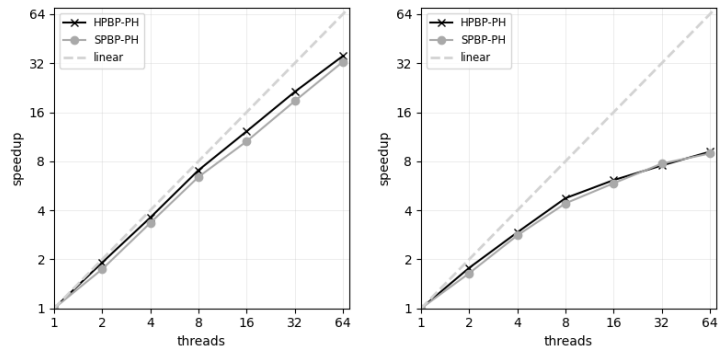
Figure 1: Median parallel speedups



(a) Parallel speedup DP (Cicirello) (b) Overall parallel speedup (Cicirello)



(c) Parallel speedup DP (Rubin) (d) Overall parallel speedup (Rubin)



(e) Parallel speedup DP (Gagné) (f) Overall parallel speedup (Gagné)

As can be seen in Figures 1 (a), (c) and (e), the median parallel speedup of the dynamic programming part in HPBP-PH is almost linear for all benchmark sets up to 16 threads. For 32 and 64 threads, the speedup still increases but
635 not nearly as linear as before. Although the speedup values for the dynamic programming parallelization of SPBP-PH are similar to those of HPBP-PH, they are consistently lower. This is due to the high number of calls to the expensive *flush* directive and the time spent waiting for shared variable values to become available. As shown in a corresponding table in Appendix B, for
640 our three benchmark sets, the waiting condition is entered on average 76, 182 and 107 times more often in SPBP-PH than in HPBP-PH, respectively. On the other hand, if enough threads are used, too many values will still be missing when accessed, and HPBP-PH will quickly switch to *s-pdp* mode when no more negative reduced cost columns are found. For the three instance sets considered,
645 this is not yet the case for 128 threads, but the increase in running times for almost all instances compared to 32 and 64 threads clearly indicates that this is the limit for the scalability of our parallelization. The observed running times suggest that peak performance is achieved in the range of 16–32 threads for most instances from the Rubin and Gagné sets, and 32–64 threads for the Cicirello
650 set.

The overall parallel speedups shown in Figures 1 (b), (d) and (f) are significantly lower than the parallel speedups of the dynamic programming part in both SPBP-PH and HPBP-PH. This is due to a non-vanishing serial part, which comprises anything but dynamic programming, and in particular the solution
655 of restricted LPs by Gurobi during column generation (Algorithm 2). Note that using Amdahl’s law to provide upper bounds on possible speedups based on the fraction of the serial part (Amdahl, 1967) is not meaningful for our algorithms, since we have observed that the shares in total running time of both the serial and the parallel parts are not constant when different numbers of threads are used.⁷For the Gagné and Rubin instances, the curve of the overall speedup for
660

⁷The dynamic programming algorithm (i.e., the parallel part of the b&p algorithms) is

both versions becomes almost stagnant using 64 threads (see Figures 1 (d) and (f)); i.e., using more computing resources not only brings no benefit, but may even become detrimental.

Further, we observe that, for the Rubin and Gagné sets, the speedups vary
665 only slightly over all instances considered when 16 or fewer threads are used, while for HPBP-PH, the numbers for the Cicirello set can vary significantly when 16 or more threads are used. The difference in this case can also be seen in a much higher average speedup compared to the median (however, this indicates, that outliers are mostly positive, even though they are the product
670 of coincidentally generating good schedules in a few cases). With the exception of the Cicirello set running HPBP-PH with 8 or more threads, the observed standard deviations are consistently below 1.14 even for 16 threads. We see a different picture for 32 and 64 threads. Here, we even observe a standard deviation of up to 41.33 from the average (overall) speedup of 12.57 for HPBP-
675 PH applied to the Cicirello set, which is considerably high. This also indicates that the scalability of the algorithm is highly dependent on the solved instance, and using too many threads leads to unpredictable behavior (see Footnote 7). These effects are noticeably stronger for HPBP-PH compared to SPBP-PH. The detailed numbers can also be derived from Appendix B. Also, for SPBP-
680 PH, when the number of threads is held constant, the average and median speedup values are similar for the Gagné and Cicirello instances, and slightly lower for the Rubin set, which consists of small instances (45 jobs and fewer). For instances with fewer jobs, threads quickly finish computing all entries from the timeslots in the DP matrix to which they were initially assigned. Thus, during
685 dynamic programming, new timeslots are assigned to threads more frequently to

executed several times during the b&p algorithms. In addition, it cannot be guaranteed that the same set of columns is generated by the b&p algorithms if different numbers of threads are used. Therefore, the serial part of the b&p algorithms (i.e., everything except dynamic programming) is also affected when, for example, more/less columns are generated on a different number of threads.

process the corresponding entries, so the overhead of parallelization is greater, which affects the measured speedup. Regarding HPBP-PH, similar values can be observed across the three sets for up to 8 threads, but variation increases for the Cicirello set. when 16 or more parallel threads are used.

690 In summary, the application of our two proposed parallel b&p algorithms (HPBP-PH and SPBP-PH) yields substantial speedups that are robust over all sets of benchmark instances up to an instance-specific number of threads. The observed speedup values of the dynamic programming algorithms are similar when the number of threads used is below the limit mentioned above, although
695 they are consistently larger for HPBP-PH compared to SPBP-PH. When using more threads than the aforementioned limit of about 32, the speedups for SPBP-PH are significantly lower compared to HPBP-PH when applied to larger instances, but the standard deviation also increases for the latter version of the algorithm. Since these speedups (for dynamic programming) are reflected in
700 the observed overall speedups (and outliers are mostly on the positive side), HPBP-PH is superior to SPBP-PH with respect to all benchmark sets tested.

7. Conclusion

In this study, we introduce, implement, and evaluate two exact parallel b&p algorithms (referred to as HPBP-PH and SPBP-PH) for the $1 \mid ST_{sd} \mid \sum w_j T_j$
705 scheduling problem, developed from a serial b&p algorithm we adopted from Lopes and de Carvalho (2007) with a modified branching strategy and an added primal heuristic. For both algorithms, we applied a fine-grained intra-algorithm parallelization strategy by executing the dynamic programming part, which is used to solve the pricing problem during column generation, on up to 128 parallel
710 threads. Before conducting extensive experiments regarding the parallelized part of our algorithms, we verified that the branching strategy and the primal heuristic, both designed to improve the performance of the serial part of the algorithm, allowed us to solve additional instances even within tight time limits. Our computational experiments on established benchmark instances from

715 Cicirello (2003), Gagné et al. (2002) and Rubin and Ragatz (1995) allowed us to
identify an instance-specific limit of threads for which both parallel algorithms
achieve substantial speedups with very little variation between instances within
the aforementioned benchmark sets. For experiments where this limit is not
exceeded, we observe an almost linear speedup for the dynamic programming
720 algorithm in both versions of the algorithm, while the corresponding speedup in
SPBP-PH is slightly lower. Also, speedups for SPBP-PH degraded noticeably
faster when the number of threads used exceeded the limit. This results in a
higher parallelization potential for HPBP-PH compared to SPBP-PH, and thus
the former outperforms the latter on all of our benchmark sets tested.

725 There are limitations to our approach that can be addressed in future work.
First, the fine-grained intra-algorithm strategy of our parallelization efforts al-
lows parallelizing the processing of individual nodes of the b&p tree, but still re-
quires sequential processing of the nodes. Parallelizing this processing through
coarse-grained intra-algorithm parallelization would allow exploiting this ad-
730 ditional speedup potential. When both parallelization strategies are applied
jointly, a hybrid parallelization strategy emerges. One possible design of such
a hybrid approach is to solve different nodes of the b&p tree simultaneously
on different processes, while multiple threads on each process are used to solve
the respective b&p node in parallel (e.g., by our parallel dynamic programming
735 algorithms). Second, the observed lower bounds develop more slowly compared
to the upper bounds during the execution of the algorithms for most instances.
Some acceleration techniques could be implemented and tested, such as dom-
inance rules. While we discarded this idea due to the massive impact on the
running time for each call to the dynamic programming algorithm, there may
740 be a strategy that is worth the overhead. Third, the primal heuristic used in our
algorithm accounts for less than 0.1% of the total running time. Using a more
sophisticated procedure could lead to a higher frequency of finding improving
upper bounds, even when a tight time limit is invoked.

Acknowledgements

745 The authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing (PC2). We also thank the authors of Tanaka and Araki (2013) for providing us with information and their algorithm.

References

- 750 Adel, D., Bendjoudi, A., El Baz, D., Abdelhakim, A.Z., 2016. GPU-based two level parallel B&B for the blocking job shop scheduling problem. *Applied Soft Computing* , 747–755.
- 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.
- 755 Akker, J.M.V.D., Hoogeveen, J.A., Velde, S.L.V.D., 1999. Parallel machine scheduling by column generation. *Operations Research* 47, 862–872. URL: <http://dx.doi.org/10.1287/opre.47.6.862>, doi:10.1287/opre.47.6.862.
- 760 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. URL: <https://doi.org/10.1007/s00170-007-0993-0>, doi:10.1007/s00170-007-0993-0.
- 765 Aldasoro, U., Escudero, L.F., Merino, M., Monge, J.F., Perez, G., 2015. On parallelization of a stochastic dynamic programming algorithm for solving large-scale mixed 0-1 problems under uncertainty. *Top* 23, 703–742. doi:10.1007/s11750-014-0359-3.

- 770 Aldasoro, U., Escudero, L.F., Merino, M., Perez, G., 2017. A parallel branch-
and-fix coordination based matheuristic algorithm for solving large sized mul-
tistage stochastic mixed 0-1 problems. *European Journal of Operational Re-
search* 258, 590–606. doi:10.1016/j.ejor.2016.08.072.
- Allahverdi, A., 2015. The third comprehensive survey on scheduling prob-
lems with setup times/costs. *European Journal of Operational Research* 246,
775 345 – 378. URL: <http://www.sciencedirect.com/science/article/pii/S0377221715002763>, doi:<https://doi.org/10.1016/j.ejor.2015.04.004>.
- Allahverdi, A., Gupta, J.N., Aldowaisan, T., 1999. A review
of scheduling research involving setup considerations. *Omega* 27,
219 – 239. URL: <http://www.sciencedirect.com/science/article/pii/S0305048398000425>, doi:[https://doi.org/10.1016/S0305-0483\(98\)00042-5](https://doi.org/10.1016/S0305-0483(98)00042-5).
780
- Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M.Y., 2008. A sur-
vey of scheduling problems with setup times or costs. *European
Journal of Operational Research* 187, 985 – 1032. URL: <http://www.sciencedirect.com/science/article/pii/S0377221706008174>,
785 doi:<https://doi.org/10.1016/j.ejor.2006.06.060>.
- Amdahl, G.M., 1967. Validity of the single processor approach to achieving
large scale computing capabilities, in: *Proceedings of the April 18-20, 1967,
spring joint computer conference*, pp. 483–485.
- 790 Anghinolfi, D., Paolucci, M., 2008. A new ant colony optimization approach for
the single machine total weighted tardiness scheduling problem. *International
Journal of Operations Research* 5, 44–60.
- Atakan, S., Bülbül, K., Noyan, N., 2017. Minimizing value-at-risk in single-
machine scheduling. *Annals of Operations Research* 248, 25–73.
- 795 Balakrishnan, N., Kanet, J.J., Sridharan, V., 1999. Early/tardy schedul-
ing with sequence dependent setups on uniform parallel machines.

Computers & Operations Research 26, 127 – 141. URL: <http://www.sciencedirect.com/science/article/pii/S0305054898000513>,
doi:[https://doi.org/10.1016/S0305-0548\(98\)00051-3](https://doi.org/10.1016/S0305-0548(98)00051-3).

800 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. *Oper. Res.* 46, 316–329. URL: [http://dx.doi.org/10.1287/
opre.46.3.316](http://dx.doi.org/10.1287/opre.46.3.316), doi:10.1287/opre.46.3.316.

Barr, R.S., Hickman, B.L., 1993. Feature article - reporting computational
805 experiments with parallel algorithms: Issues, measures, and experts' opinions.
INFORMS Journal on Computing 5, 2–18.

Boschetti, M.A., Maniezzo, V., Strappaveccia, F., 2016. Using GPU comput-
ing for solving the two-dimensional guillotine cutting problem. *INFORMS
Journal on Computing* 28, 540–552. doi:10.1287/ijoc.2016.0693.

810 Boyer, V., El Baz, D., Elkihel, M., 2012. Solving knapsack problems on GPU.
Computers & Operations Research 39, 42–47.

Bożejko, W., 2010. Parallel path relinking method for the single machine to-
tal weighted tardiness problem with sequence-dependent setups. *Journal of
Intelligent Manufacturing* 21, 777–785. URL: [https://doi.org/10.1007/
s10845-009-0253-2](https://doi.org/10.1007/s10845-009-0253-2), doi:10.1007/s10845-009-0253-2.
815

Carvajal, R., Ahmed, S., Nemhauser, G., Furman, K., Goel, V., Shao, Y., 2014.
Using diversification, communication and parallelism to solve mixed-integer
linear programs. *Operations Research Letters* 42, 186–189. doi:10.1016/j.
orl.2013.12.012.

820 Chakroun, I., Melab, N., 2015. Towards a heterogeneous and adaptive parallel
branch-and-bound algorithm. *Journal of Computer and System Sciences* 81,
72–84.

- Chakroun, I., Melab, N., Mezmaz, M., Tuyttens, D., 2013a. Combining multi-core and gpu computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing* 73, 1563–1577. 825
- Chakroun, I., Mezmaz, M., Melab, N., Bendjoudi, A., 2013b. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurrency and Computation-practice & Experience* 25, 1121–1136. doi:10.1002/cpe.2931.
- Chen, C.L., 2019. Iterated population-based vnd algorithms for single-machine scheduling with sequence-dependent setup times. *Soft Computing* 23, 3627–3641. 830
- Chen, Z.L., Powell, W.B., 1999. Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing* 11, 78–94. URL: 835 <http://dx.doi.org/10.1287/ijoc.11.1.78>, doi:10.1287/ijoc.11.1.78.
- Christou, I.T., Vassilaras, S., 2013. A parallel hybrid greedy branch and bound scheme for the maximum distance-2 matching problem. *Computers & Operations Research* 40, 2387–2397. doi:10.1016/j.cor.2013.04.009.
- Cicirello, V.A., 2003. Weighted Tardiness Scheduling with Sequence-Dependent 840 Setups: A Benchmark Library. Technical Report. Intelligent Coordination and Logistics Laboratory, Robotics Institute, Carnegie Mellon University. Pittsburgh, PA. URL: <https://www.cicirello.org/publications/wtsbenchmarks.pdf>.
- Cicirello, V.A., 2009. Weighted tardiness scheduling with sequence-845 dependent setups: A benchmark problem for soft computing, in: *Applications of Soft Computing: Updating the State of the Art*, Springer. pp. 189–198. URL: <https://www.cicirello.org/publications/ApplicationsOfSoftComputing.pdf>.
- Crainic, T.G., Toulouse, M., 2003. Parallel strategies for meta-heuristics, in: 850 *Handbook of metaheuristics*. Springer, Boston, MA, pp. 475–513.

- Dias, B.H., Tomim, M.A., Marques Marcato, A.L., Ramos, T.P., Brandi, R.B.S., Da Silva Junior, I.C., Passos Filho, J.A., 2013. Parallel computing applied to the stochastic dynamic programming for long term operation planning of hydrothermal power systems. *European Journal of Operational Research* 229, 212–222. doi:10.1016/j.ejor.2013.02.024.
- 855
- Gagné, C., Price, W.L., Gravel, M., 2002. Comparing an aco algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times. *Journal of the Operational Research Society* 53, 895–906. URL: <https://doi.org/10.1057/palgrave.jors.2601390>, doi:10.1057/palgrave.jors.2601390.
- 860
- Galea, F., Le Cun, B., 2011. A parallel exact solver for the three-index quadratic assignment problem, in: *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pp. 1940–1949.
- 865
- Gendron, B., Crainic, T.G., 1994. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research* 42, 1042–1066.
- Gmys, J., Mezmaz, M., Melab, N., Tuyttens, D., 2016. A gpu-based branch-and-bound algorithm using integer–vector–matrix data structure. *Parallel Computing* 59, 119–139.
- 870
- Gmys, J., Mezmaz, M., Melab, N., Tuyttens, D., 2017. Ivm-based parallel branch-and-bound using hierarchical work stealing on multi-gpu systems. *Concurrency and Computation: Practice and Experience* 29, e4019.
- Graham, R., Lawler, E., Lenstra, J., Kan, A., 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey, in: Hammer, P., Johnson, E., Korte, B. (Eds.), *Discrete Optimization II*. Elsevier. volume 5 of *Annals of Discrete Mathematics*, pp. 287 – 326. URL: <http://www.sciencedirect.com/science/article/pii/S016750600870356X>, doi:[https://doi.org/10.1016/S0167-5060\(08\)70356-X](https://doi.org/10.1016/S0167-5060(08)70356-X).

- 880 Guo, Q., Tang, L., 2015. An improved scatter search algorithm for the single machine total weighted tardiness scheduling problem with sequence-dependent setup times. *Applied Soft Computing* 29, 184 – 195. URL: <http://www.sciencedirect.com/science/article/pii/S1568494614006747>, doi:<https://doi.org/10.1016/j.asoc.2014.12.030>.
- 885 Hager, G., Wellein, G., 2010. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press.
- Hoffmann, S., Lienhart, R., 2008. *OpenMP - Eine Einführung in die parallele Programmierung mit C/C++*. Springer-Verlag Berlin Heidelberg. doi:10.1007/978-3-540-73123-8.
- 890 Ismail, M.M., Abd El-Raouf, O., Abd El-Wahed, W.F., 2014. A parallel branch and bound algorithm for solving large scale integer programming problems. *Applied Mathematics & Information Sciences* 8, 1691–1698. doi:10.12785/amis/080425.
- Kramer, A., Subramanian, A., 2019. A unified heuristic and an annotated
895 bibliography for a large class of earliness–tardiness scheduling problems.
- Kumar, S., Misra, A., Tomar, R.S., 2011. A modified parallel approach to single source shortest path problem for massively dense graphs using CUDA, in: *Proceedings of the 2nd International Conference on Computer and Communication Technology (ICCCT)*, pp. 635–639.
- 900 Lawler, E.L., 1977. A “pseudopolynomial” algorithm for sequencing jobs to minimize total tardiness, in: Hammer, P., Johnson, E., Korte, B., Nemhauser, G. (Eds.), *Studies in Integer Programming*. Elsevier. volume 1 of *Annals of Discrete Mathematics*, pp. 331 – 342. URL: <http://www.sciencedirect.com/science/article/pii/S0167506008707428>, doi:[https://doi.org/10.1016/S0167-5060\(08\)70742-8](https://doi.org/10.1016/S0167-5060(08)70742-8).
905

- Lee, Y.H., Bhaskaran, K., Pinedo, M., 1997. A heuristic to minimize the total weighted tardiness with sequence-dependent setups. *IIE Transactions* 29, 45–52.
- 910 Liao, C.J., Juan, H.C., 2007. An ant colony optimization for single-machine tardiness scheduling with sequence-dependent setups. *Computers & Operations Research* 34, 1899 – 1909. URL: <http://www.sciencedirect.com/science/article/pii/S0305054805002467>, doi:<https://doi.org/10.1016/j.cor.2005.07.020>.
- 915 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.
- Lopes, M., Alvelos, F., Lopes, H., 2014. Improving branch-and-price for parallel machine scheduling, in: Murgante, B., Misra, S., Rocha, A.M.A.C., Torre, C., Rocha, J.G., Falcão, M.I., Tanian, D., Aduhan, B.O., Gervasi, O. (Eds.),
920 *Computational Science and Its Applications – ICCSA 2014*, Springer International Publishing, Cham. pp. 290–300.
- Lopes, M.J.P., de Carvalho, J.V., 2007. A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup
925 times. *European Journal of Operational Research* 176, 1508 – 1527. URL: <http://www.sciencedirect.com/science/article/pii/S0377221705008738>, doi:<https://doi.org/10.1016/j.ejor.2005.11.001>.
- Maleki, S., Musuvathi, M., Mytkowicz, T., 2016. Efficient parallelization using rank convergence in dynamic programming algorithms. *Communications of*
930 *the ACM* 59, 85–92. doi:10.1145/2983553.
- Mezmaz, M., Leroy, R., Melab, N., Tuyttens, D., 2014. A multi-core parallel branch-and-bound algorithm using factorial number system, in: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE*. pp. 1203–1212.

- 935 Nogueira, T.H., Carvalho, C.R.V.d., Ravetti, M.G., Souza, M.C.d., 2019. Analysis of mixed integer programming formulations for single machine scheduling problems with sequence dependent setup times and release dates. *Pesquisa Operacional* 39, 109–154.
- Nogueira, T.H., Ramalhinho, H.L., de Carvalho, C.R., Gomez Ravetti, M.,
940 2022. A hybrid vns-lagrangean heuristic framework applied on single machine scheduling problem with sequence-dependent setup times, release dates and due dates. *Optimization Letters* , 1–20.
- OpenMP, 2015. Openmp application program interface version 4.5. retrieved september 12, 2017. URL: [http://www.openmp.org/wp-content/uploads/](http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf)
945 [openmp-4.5.pdf](http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf).
- Ozden, S.G., Smith, A.E., Gue, K.R., 2017. Solving large batches of traveling salesman problems with parallel and distributed computing. *Computers & Operations Research* 85, 87–96. doi:10.1016/j.cor.2017.04.001.
- Rashid, H., Novoa, C., Qasem, A., 2010. An evaluation of parallel knapsack
950 algorithms on multicore architectures., in: *Proceedings of the 2010 International Conference on Scientific Computing*, pp. 230–235.
- 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 (Adelaide, Australia)*, p. 1–13.
955
- Rauchecker, G., Schryen, G., 2018. 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. *Computers & Operations Research* (published online) doi:<https://doi.org/10.1016/j.cor.2018.12.020>.
960
- Rubin, P.A., Ragatz, G.L., 1995. Scheduling in a sequence dependent setup environment with genetic search. *Computers & Oper-*

- ations Research 22, 85 – 99. URL: <http://www.sciencedirect.com/science/article/pii/0305054893E0021K>, doi:[https://doi.org/10.1016/0305-0548\(93\)E0021-K](https://doi.org/10.1016/0305-0548(93)E0021-K).
- 965
- Schryen, G., 2020. Parallel computational optimization in operations research: A new integrative framework, literature review and research directions. *European Journal of Operational Research* 287, 1–18.
- Sewell, E.C., Sauppe, J.J., Morrison, D.R., Jacobson, S.H., Kao, G.K., 2012. A bb&r algorithm for minimizing total tardiness on a single machine with sequence dependent setup times. *Journal of Global Optimization* 54, 791–812.
- 970
- Stivala, A., Stuckey, P.J., Garcia De La Banda, M., Hermenegildo, M., Wirth, A., 2010. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing* 70, 839–848. doi:10.1016/j.jpdc.2010.01.004.
- 975
- Subramanian, A., Battarra, M., Potts, C.N., 2014. An iterated local search heuristic for the single machine total weighted tardiness scheduling problem with sequence-dependent setup times. *International Journal of Production Research* 52, 2729–2742.
- 980
- Subramanian, A., Farias, K., 2017. Efficient local search limitation strategy for single machine total weighted tardiness scheduling with sequence-dependent setup times. *Computers & Operations Research* 79, 190 – 206. URL: <http://www.sciencedirect.com/science/article/pii/S0305054816302568>.
- Tan, G., Sun, N., Gao, G.R., 2009. Improving performance of dynamic programming via parallelism and locality on multicore architectures. *IEEE Transactions on Parallel and Distributed Systems* 20, 261–274. doi:10.1109/TPDS.2008.78.
- 985
- Tanaka, S., Araki, M., 2013. An exact algorithm for the single-machine total weighted tardiness problem with sequence-dependent setup times. *Computers & Operations Research* 40, 344 – 352. URL: <http://www.sciencedirect.com/science/article/pii/S0305054812200000>.
- 990

[//www.sciencedirect.com/science/article/pii/S0305054812001499](http://www.sciencedirect.com/science/article/pii/S0305054812001499),
doi:<https://doi.org/10.1016/j.cor.2012.07.004>.

Tasgetiren, M.F., Pan, Q.K., Liang, Y.C., 2009. A discrete differential evolution algorithm for the single machine total weighted tardiness problem with
995 sequence dependent setup times. *Computers & Operations Research* 36, 1900
– 1915. URL: <http://www.sciencedirect.com/science/article/pii/S0305054808001135>, doi:<https://doi.org/10.1016/j.cor.2008.06.007>.

Tavakkoli-Moghaddam, R., Aramon-Bajestani, M., 2009. A novel b and b algorithm for a unrelated parallel machine scheduling problem to minimize the
1000 total weighted tardiness. *International Journal of Engineering* 22(3), 269–286.

Tran, Q.N., 2010. Designing efficient many-core parallel algorithms for all-pairs shortest-paths using CUDA, in: *Proceedings of the 7th International Conference on Information Technology: New Generations (ITNG)*, pp. 7–12.

Vu, T.T., Derbel, B., 2016. Parallel branch-and-bound in multi-core multi-
1005 cpu multi-gpu heterogeneous environments. *Future Generation Computer Systems* 56, 95–109.

Wodecki, M., 2008. A branch-and-bound parallel algorithm for single-machine total weighted tardiness problem. *The International Journal of Advanced Manufacturing Technology* 37, 996–1004.

1010 Xu, H., Lü, Z., Cheng, T., 2014a. Iterated local search for single-machine scheduling with sequence-dependent setup times to minimize total weighted tardiness. *Journal of Scheduling* 17, 271–287. doi:10.1007/s10951-013-0351-z.

1015 Xu, H., Lü, Z., Yin, A., Shen, L., Buscher, U., 2014b. A study of hybrid evolutionary algorithms for single machine scheduling problem with sequence-dependent setup times. *Computers & Operations Research* 50, 47 – 60. URL: <http://www.sciencedirect.com/science/article/pii/S0305054814001002>, doi:<https://doi.org/10.1016/j.cor.2014.04.009>.

Zhu, Z., Heady, R.B., 2000. Minimizing the sum of earliness/tardiness
1020 in multi-machine scheduling: a mixed integer programming approach.
Computers & Industrial Engineering 38, 297 – 305. URL: <http://www.sciencedirect.com/science/article/pii/S0360835200000486>,
doi:[https://doi.org/10.1016/S0360-8352\(00\)00048-6](https://doi.org/10.1016/S0360-8352(00)00048-6).

Appendix A. Detailed Running times

1025 This section of the appendix provides detailed results of the experiments
discussed in Section 6. Appendix A.1 contains data for HPBP-PH. For the
parallelization experiments in 6.3.1, Table A.1 shows a summary of the number
of solved instances over increasing time limits for different numbers of threads,
while Tables A.2, A.4 and A.6 give the detailed running times of the algo-
1030 rithm using up to 64 threads for the instances from the Rubin, Gagné, and
Cicirello sets, respectively. For each instance (“ins” column), the second and
third columns of the tables show the objective value found by Tanaka and Araki
(2013) and the computation time to find that value. The fourth column shows
the objective value found by our algorithm, followed by seven columns (the “t_i”
1035 columns) with the computation times needed to solve the instance when using
up to 64 threads. Whenever the time limit was exceeded and optimality was
not proved, it is indicated by a dash.

For the sake of completeness, Tables A.3, A.5, and A.7 list the instances and
running times of each benchmark set using 128 threads, only if they were solved
1040 within one hour. As discussed in 6, no further experiments were run with 128
threads due to a significant performance degradation in this setting.

Appendix A.2 contains the same tables as Appendix A.1, only for algorithm
SPBP-PH.

Finally, Appendix A.3 shows additional statistics for the experiments con-
1045 sidering the different serial versions of the algorithms compared in Section 6.2.
Tables A.15 – A.17 summarize statistics for each benchmark set, such as the
number of instances solved within one hour for all versions considered, with
additional information on the averages of the number of b&b-nodes used, the
time taken to solve a single b&b-node, the number of columns generated, and
1050 the number of columns generated per node. The specific values used to calcu-
late the above averages are given in the following tables A.18 – A.20. For each
instance solved to optimality within the time limit by at least one serial version
of the algorithm, they show the running times, the number of b&b nodes, and

the number of columns generated for all versions.

1055 *Appendix A.1. Hybrid Parallel Branch-and-Price with primal heuristic*

Table A.1: Solved instances over time in HPBP-PH

Set	#instances	#threads	Timelimit (h)			
			1	12	48	504
Rubin	32	1	29	29	31	32
		2	29	31	31	32
		4	29	31	32	32
		8	29	31	32	32
		16	29	31	32	32
		32	29	31	32	32
		64	29	31	32	32
Gagné	32	1	14	21	26	30
		2	15	22	27	31
		4	16	23	27	31
		8	19	24	27	31
		16	20	23	28	31
		32	20	26	30	31
		64	21	27	30	31
Cicirello	120	1	53	99	107	116
		2	65	103	110	117
		4	73	105	111	117
		8	84	109	111	117
		16	92	111	115	117
		32	93	111	116	117
		64	97	111	117	117

Table A.2: Running times HPBP-PH over threads for Rubin instances

ins	Tanaka		HPBP-PH run with #threads											
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64				
401	90	0.31	90	0.76	0.49	0.33	3.07	0.27	0.38	0.23				
402	0	0.00	0	0.31	0.20	0.13	0.58	0.20	0.13	0.26				
403	3,418	0.53	3,418	0.46	0.26	0.20	1.69	0.16	0.14	0.28				
404	1,067	0.44	1,067	0.42	0.36	0.60	0.19	0.19	0.85	1.73				
405	0	0.00	0	0.07	0.06	0.05	0.08	0.05	0.11	0.06				
406	0	0.00	0	0.31	0.19	0.12	0.32	0.10	0.21	0.10				
407	1,861	0.48	1,861	5.49	3.61	3.16	1.90	1.74	1.53	1.10				
408	5,660	0.85	5,660	0.56	0.35	0.33	0.37	0.32	0.13	0.18				
501	261	2.09	261	13.56	8.00	5.09	4.79	2.78	2.23	1.46				
502	0	0.01	0	0.01	0.02	0.02	0.04	0.04	0.03	0.06				
503	3,497	3.04	3,497	15.09	9.09	6.61	4.05	3.01	2.61	2.99				
504	0	0.03	0	0.02	0.01	0.01	0.02	0.02	0.06	0.05				
505	0	0.02	0	0.79	0.46	0.30	0.46	0.64	0.20	0.39				

Table A.2 (cont. - Running times HPBP-PH Rubin)

Tanaka		HPBP-PH run with #threads											
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
506	0	0.02	0	4.94	3.21	2.39	1.59	1.16	1.10	0.99			
507	7,225	4.22	7,225	13.24	8.46	7.16	4.05	2.93	2.42	1.43			
508	1,915	3.84	1,915	4.18	2.52	2.25	1.19	0.85	0.70	0.96			
601	12	7.31	12	59.46	30.79	23.80	12.63	9.74	7.27	5.52			
602	0	0.05	0	19.25	10.77	11.69	5.20	3.76	3.19	3.07			
603	17,587	18.37	17,587	326.75	181.07	109.03	65.84	52.11	44.34	34.88			
604	19,092	25.73	19,092	276.97	159.85	89.82	59.08	47.27	40.14	39.03			
605	228	13.57	228	63,921.60	40,444.90	33,674.60	19,114.10	13,954.80	18,682.40	12,130.40			
606	0	0.05	0	10.16	6.11	8.17	2.20	1.81	1.82	3.60			
607	12,969	17.60	12,969	216.74	121.80	74.87	48.53	39.69	33.48	30.40			
608	4,732	32.92	4,732	166.86	97.24	57.37	38.30	31.15	26.50	22.03			
701	97	49.55	97	59,237.60	35,380.40	24,331.40	14,977.20	10,276.90	11,626.20	10,374.20			
702	0	0.09	0	0.03	0.02	0.01	0.02	0.02	0.02	0.04			
703	26,506	52.36	26,506	2,232.51	1,246.19	700.21	409.36	409.46	339.51	279.18			
704	15,206	71.40	15,206	401.06	236.39	141.19	90.47	70.20	62.52	76.28			

Table A.2 (cont. - Running times HPBP-PH Rubin)

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
705	200	599.48	200	276,520.00	154,790.00	97,719.10	74,613.50	60,567.70	55,560.40	40,603.30		
706	0	0.10	0	0.02	0.02	0.02	0.01	0.02	0.07	0.04		
707	23,789	52.32	23,789	2,718.81	1,543.32	912.34	565.42	444.44	503.30	307.31		
708	22,807	99.77	22,807	537.58	314.38	214.78	120.08	105.89	92.32	76.78		

Table A.3: Running times HPBP-PH with 128 threads for Rubin instances solved within 1 hour (29 of 32)

ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128
401	0.61	402	0.20	403	0.21	404	0.88	405	0.12	406	0.27
407	3.61	408	0.32	501	3.43	502	0.05	503	4.43	504	0.05
505	0.28	506	2.13	507	3.64	508	1.56	601	7.57	602	2.21
603	70.18	604	66.93	606	5.70	607	57.20	608	53.22	702	0.03
703	383.42	704	93.65	706	0.05	707	529.21	708	153.24		

Table A.4: Running times HPBP-PH over threads for Gagné instances

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
551	183	116.23	183	581.69	322.88	180.51	104.69	103.53	62.97	72.54		
552	0	0.17	0	0.02	0.02	0.07	0.01	0.04	0.15	0.03		
553	40,498	112.04	40,498	4,489.52	2,528.38	1,559.80	961.26	750.93	462.75	396.06		
554	14,653	277.87	14,653	537.38	309.23	207.74	131.28	83.53	68.37	53.77		
555	0	0.27	0	366.56	211.41	122.63	72.71	59.62	49.79	35.56		
556	0	0.18	0	0.04	0.02	0.02	0.02	0.03	0.07	0.03		
557	35,813	140.91	35,813	81,157.90	48,404.80	28,310.70	20,094.60	13,835.20	11,550.10	10,318.10		
558	19,871	228.58	19,871	2,008.25	1,067.57	1,229.77	421.65	332.80	279.99	333.73		
651	247	2,298.27	247	7,894.18	4,260.64	2,369.51	1,340.28	903.11	702.39	785.19		
652	0	0.25	0	0.03	0.02	0.03	0.02	0.03	0.07	0.03		
653	57,500	262.80	57,500	56,300.40	35,414.90	19,252.00	12,887.30	11,855.50	10,637.20	7,110.25		
654	34,301	381.56	34,301	13,822.30	7,794.42	4,637.62	2,919.35	2,152.15	1,792.97	1,690.33		
655	0	355.72	0	1,546.44	840.77	493.00	264.19	188.54	137.11	98.23		

Table A.4 (cont. - Running times HPBP-PH Gagné)

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
656	0	0.29	0	0.03	0.02	0.02	0.04	0.04	0.05	0.05		
657	54,895	253.71	54,895	20,946.10	11,325.60	6,368.04	4,861.22	2,695.80	2,043.43	2,184.69		
658	27,114	466.63	27,114	14,618.20	8,348.02	5,070.36	3,433.72	2,262.53	2,250.13	2,183.71		
751	225	-	225	1,734,710.00	799,259.00	451,773.00	260,670.00	199,572.00	140,400.00	125,786.00		
752	0	0.44	0	0.03	0.03	0.02	0.03	0.03	0.04	0.06		
753	77,544	565.95	77,544	38,272.60	20,265.80	13,039.50	7,209.60	6,459.31	5,973.60	3,584.04		
754	35,200	887.85	35,200	161,347.00	96,021.10	59,851.40	39,158.40	34,203.60	33,641.60	22,041.10		
755	0	0.41	0	80.58	42.80	46.41	12.76	8.60	3.51	4.05		
756	0	0.52	0	0.02	0.02	0.03	0.03	0.04	0.06	0.07		
757	59,635	741.33	59,635	992,190.00	548,745.00	316,801.00	205,914.00	171,721.00	138,385.00	142,180.00		
758	38,339	987.59	38,339	13,087.00	7,384.84	4,563.21	3,036.63	2,265.90	1,911.95	1,631.00		
851	363	-	360	293,887.00	170,756.00	67,051.40	54,795.80	46,168.30	34,727.00	20,216.70		
852	0	0.61	0	0.03	0.03	0.05	0.03	0.06	0.03	0.07		
853	97,497	2,219.22	97,497	150,326.00	112,859.00	55,120.20	78,984.20	37,061.30	34,812.60	26,296.30		
854	79,042	1,177.04	79,042	170,598.00	98,652.60	59,548.70	37,920.80	27,911.20	23,744.10	19,536.00		

Table A.4 (cont. - Running times HPBP-PH Gagné)

Tanaka		HPBP-PH run with #threads									
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64	
855	260	—	256	—	—	—	—	—	—	—	
856	0	0.63	0	0.02	0.03	0.04	0.02	0.07	0.07	0.12	
857	87,011	4,594.41	87,011	953,237.00	500,272.00	275,904.00	177,832.00	139,693.00	109,228.00	96,656.30	
858	74,739	1,817.68	74,739	—	1,297,410.00	759,494.00	486,510.00	508,457.00	396,232.00	386,687.00	

Table A.5: Running times HPBP-PH with 128 threads for Gagné instances solved within 1 hour (18 of 32)

ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128		
551	157.78	552	0.06	553	802.33	554	126.40	555	95.41	556	0.06
558	698.23	652	0.10	654	1,860.93	655	289.42	656	0.07	657	2,674.07
752	0.12	755	23.73	756	0.13	758	2,201.67	852	0.13	856	0.13

Table A.6: Running times HPBP-PH over threads for Cicirello instances

ins	Tanaka		HPBP-PH run with #threads											
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64				
1	453	93.58	453	738,050.00	418,241.00	253,589.00	175,362.00	165,861.00	166,537.00	134,565.00				
2	4,794	1,474.67	4,794	46,882.30	27,721.00	17,304.70	9,753.96	7,223.23	4,892.97	10,447.30				
3	1,390	589.07	1,390	162,174.00	96,175.60	54,025.70	32,396.90	24,020.50	19,340.60	17,311.10				
4	5,866	129.80	5,866	15,539.40	9,158.78	6,497.61	4,052.13	3,376.39	3,014.43	3,478.35				
5	4,054	2,350.37	4,054	186,962.00	113,541.00	70,061.60	39,090.40	29,617.60	23,942.80	22,598.90				
6	6,592	166.86	6,592	7,027.44	4,180.15	2,746.27	2,149.91	1,600.41	1,340.05	1,943.94				
7	3,267	3,522.47	3,267	1,235,430.00	685,638.00	376,841.00	227,258.00	155,839.00	159,457.00	119,181.00				
8	100	105.54	100	—	—	—	—	—	—	—				
9	5,660	124.33	5,660	58,244.40	34,481.50	19,301.90	11,641.50	8,595.91	6,884.01	6,520.25				
10	1,740	8,883.74	1,740	674,219.00	389,384.00	248,377.00	162,748.00	157,431.00	131,881.00	129,288.00				
11	2,785	42,921.26	2,785	657,229.00	351,074.00	195,740.00	136,593.00	96,718.40	89,086.30	67,863.90				
12	0	0.41	0	239.88	144.75	80.13	55.92	34.84	29.15	27.74				
13	3,904	8,722.61	3,904	804,494.00	443,670.00	257,297.00	169,091.00	144,871.00	116,698.00	100,173.00				

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads											
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
14	2,075	3,555.71	2,075	207,036.00	120,214.00	65,288.70	37,976.80	26,492.30	20,102.00	16,601.40			
15	724	496.78	724	20,272.50	12,010.30	6,622.83	4,022.50	3,016.53	2,448.73	2,126.71			
16	3,285	504.49	3,285	57,173.00	35,182.10	18,470.40	10,460.50	7,395.82	5,836.84	4,729.33			
17	0	1,189.80	0	675.48	444.96	212.60	125.09	93.92	71.24	42.03			
18	773	-	767	-	-	-	-	-	-	-			
19	0	3.41	0	12,962.30	8,117.18	4,350.02	2,765.81	2,527.24	89.64	145.83			
20	1,757	316.22	1,757	35,709.50	21,574.40	11,607.30	7,434.90	5,604.76	4,709.62	3,248.46			
21	0	0.20	0	284.65	206.25	95.74	64.14	47.63	37.09	33.17			
22	0	0.23	0	272.59	154.79	91.59	61.03	47.53	35.41	36.87			
23	0	0.17	0	0.02	0.02	0.02	0.03	0.04	0.03	0.12			
24	761	-	782	-	-	-	-	-	-	-			
25	0	0.31	0	385.12	216.28	122.49	182.92	79.71	59.74	43.18			
26	0	0.23	0	293.13	156.42	94.28	63.66	45.87	37.01	33.47			
27	0	0.34	0	563.74	315.37	185.31	125.56	104.73	75.78	75.15			
28	0	0.40	0	561.77	312.44	228.90	141.56	81.11	67.75	138.11			

Table A.6 (cont. - Running times HPBP-PH Cicirello)

ins	Tanaka		HPBP-PH run with #threads											
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64				
29	0	0.23	0	291.79	166.80	129.50	79.50	40.90	38.88	37.84				
30	0	7.85	0	1,884.77	1,072.74	304.79	245.48	360.31	120.33	88.34				
31	0	0.45	0	326.00	182.51	110.70	70.37	44.37	33.33	68.36				
32	0	0.48	0	360.44	204.78	115.74	85.51	54.30	106.53	40.41				
33	0	0.48	0	424.56	233.91	176.16	80.57	79.14	42.57	41.29				
34	0	0.40	0	0.18	0.21	0.22	0.09	0.06	0.04	0.09				
35	0	0.47	0	0.16	0.19	0.13	0.08	1.23	0.04	0.08				
36	0	0.44	0	343.26	188.19	124.45	72.72	59.07	36.75	30.26				
37	0	595.89	0	2,106.18	1,228.23	679.50	441.75	275.39	211.30	3,461.21				
38	0	0.38	0	0.18	0.18	0.12	0.10	0.15	0.04	0.04				
39	0	0.48	0	370.15	208.21	114.28	106.08	49.59	43.65	34.12				
40	0	0.40	0	377.01	211.19	116.31	84.91	48.67	38.32	29.77				
41	69,102	32.06	69,102	189.20	105.95	62.07	45.10	30.41	24.82	19.27				
42	57,487	45.69	57,487	5,034.76	3,055.21	1,959.84	1,695.58	1,208.39	1,080.91	1,114.24				
43	145,310	80.55	145,310	3,196.19	1,778.61	1,070.34	845.56	542.22	457.19	380.69				

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
44	35,166	58.20	35,166	2,230.09	1,337.65	898.33	737.66	575.48	521.28	490.85		
45	58,935	80.98	58,935	8,437.05	5,508.25	3,077.07	2,280.30	1,771.53	1,457.56	1,234.94		
46	34,764	54.08	34,764	11,237.50	7,193.17	4,240.14	3,499.23	2,567.75	2,286.80	2,295.43		
47	72,853	61.35	72,853	8,055.50	4,981.94	2,807.94	2,331.52	1,661.37	1,414.07	1,439.80		
48	64,612	105.70	64,612	27,606.70	16,597.30	8,896.76	5,258.33	3,938.23	3,195.50	2,984.14		
49	77,449	59.38	77,449	1,040.95	620.55	388.81	311.07	234.69	207.59	217.79		
50	31,092	59.55	31,092	4,045.10	2,421.64	1,744.35	1,260.79	1,069.90	957.95	908.71		
51	49,208	91.84	49,208	9,507.31	5,275.98	3,127.90	1,962.65	1,490.49	1,252.84	1,211.09		
52	93,045	117.65	93,045	12,190.00	7,300.46	3,897.61	2,422.85	1,599.22	1,256.31	1,106.22		
53	84,841	116.94	84,841	4,997.73	2,769.89	1,631.14	970.62	717.64	571.50	683.33		
54	118,809	104.63	118,809	22,261.00	12,330.30	7,691.73	4,441.04	3,268.75	2,662.15	2,603.50		
55	64,315	108.07	64,315	400.54	206.92	119.05	67.19	47.43	41.49	31.98		
56	74,889	122.09	74,889	32,306.40	17,852.90	10,031.90	5,820.59	4,151.05	3,199.50	2,862.95		
57	63,514	97.11	63,514	14,493.10	8,307.46	5,248.61	3,262.54	2,549.75	2,117.78	1,849.47		
58	45,322	135.31	45,322	110,454.00	68,034.20	37,564.40	28,286.10	23,133.60	18,533.60	17,144.30		

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
59	50,999	92.49	50,999	14,893.40	8,187.56	5,157.30	3,384.20	2,362.15	2,007.60	1,791.37		
60	60,765	137.23	60,765	42,527.60	23,444.30	13,851.00	8,584.94	6,486.70	5,410.47	5,097.46		
61	75,916	41.42	75,916	1,725.33	933.65	622.60	384.91	338.03	200.58	195.20		
62	44,769	37.31	44,769	116,427.00	76,237.50	51,724.80	39,064.90	32,300.50	28,560.20	29,490.60		
63	75,317	33.11	75,317	2,923.13	1,625.64	975.13	804.83	508.59	444.08	551.67		
64	92,572	39.94	92,572	3,753.18	2,264.02	1,451.79	1,108.59	981.70	1,110.75	1,077.76		
65	126,696	41.08	126,696	1,082.44	623.41	395.96	292.13	229.03	197.68	185.59		
66	59,685	17.81	59,685	2,576.15	1,504.15	1,513.20	1,010.95	617.18	559.00	1,136.13		
67	29,390	26.93	29,390	1,490.87	905.97	572.25	425.74	317.30	277.16	315.53		
68	22,120	25.99	22,120	2,187.11	1,784.17	891.40	630.20	531.23	301.93	364.13		
69	71,118	45.03	71,118	28,413.90	17,885.90	12,009.30	8,506.78	8,250.05	3,783.55	3,641.40		
70	75,102	36.69	75,102	921.91	531.74	339.07	275.20	207.19	173.76	176.53		
71	145,007	96.11	145,007	11,206.50	6,735.04	3,477.83	2,179.62	1,489.48	1,173.23	1,076.41		
72	43,286	68.28	43,286	17,043.40	9,079.24	5,349.35	3,355.41	2,117.99	1,599.73	1,382.86		
73	28,785	91.49	28,785	57,335.70	35,520.40	19,603.20	13,374.10	10,153.90	9,563.56	7,622.22		

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads											
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
74	29,777	73.14	29,777	20,701.10	12,401.80	7,629.03	4,918.86	3,912.37	3,299.08	3,205.99			
75	21,602	77.57	21,602	2,450.75	1,376.08	834.02	613.10	459.10	387.49	351.15			
76	53,555	75.75	53,555	19,937.00	11,442.30	6,913.72	4,387.27	3,514.32	2,931.73	2,705.13			
77	31,817	84.48	31,817	11,867.70	6,597.99	4,644.14	2,840.16	1,788.79	1,419.43	1,286.52			
78	19,462	79.56	19,462	3,451.97	2,270.12	1,115.59	879.09	552.23	454.31	410.29			
79	114,999	70.12	114,999	2,165.96	1,222.21	677.12	475.34	331.05	279.01	253.07			
80	18,157	76.73	18,157	15,437.50	8,722.04	5,361.19	2,666.83	2,116.22	1,667.86	1,067.12			
81	383,485	30.13	383,485	814.12	528.33	327.69	163.59	221.38	146.37	60.88			
82	409,479	66.12	409,479	10,963.40	7,216.17	2,655.10	2,962.72	1,573.76	925.90	1,091.57			
83	458,752	42.51	458,752	16,303.20	8,614.88	7,929.49	3,397.98	3,085.43	2,628.61	1,733.58			
84	329,670	46.59	329,670	6,016.53	4,606.30	2,666.53	1,635.41	1,576.74	1,151.87	1,179.52			
85	554,766	74.38	554,766	33,941.70	19,181.60	10,791.50	6,503.43	4,952.05	3,908.13	3,166.27			
86	361,417	68.14	361,417	1,151.19	667.94	374.46	247.38	185.95	153.58	138.19			
87	398,551	47.96	398,551	77,025.50	51,584.40	13,899.00	28,421.10	21,428.10	990.97	6,282.45			
88	433,186	60.54	433,186	16,027.90	9,100.59	6,516.63	3,191.08	2,428.39	1,961.72	2,630.78			

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
89	410,092	46.85	410,092	10,594.30	6,282.13	4,190.29	3,229.05	1,645.60	2,546.36	1,221.00		
90	401,653	56.97	401,653	24,829.90	14,178.30	7,511.98	6,872.51	5,959.55	5,506.50	3,937.28		
91	339,933	68.98	339,933	5,219.29	2,770.79	1,424.70	2,030.58	955.96	1,326.17	1,296.74		
92	361,152	103.36	361,152	6,524.69	3,565.67	2,341.25	1,549.56	1,364.40	970.49	960.60		
93	403,423	125.52	403,423	9,903.85	5,250.08	8,970.99	2,105.23	849.66	1,223.02	3,472.39		
94	332,941	98.30	332,941	—	1,544,020.00	13,493.80	4,846.76	5,709.92	509,442.00	3,231.40		
95	516,926	101.20	516,926	11,017.00	6,287.79	4,058.48	2,745.49	2,165.11	1,866.63	995.33		
96	455,448	79.50	455,448	1,212.43	682.69	480.06	251.63	189.35	153.76	120.76		
97	407,590	92.92	407,590	2,507.43	1,436.89	932.41	532.35	437.81	358.92	249.43		
98	520,582	90.23	520,582	1,123.88	768.73	378.60	267.64	155.63	118.82	167.13		
99	363,518	106.21	363,518	955.67	518.80	373.85	220.00	150.26	122.40	109.43		
100	431,736	80.98	431,736	2,368.53	1,359.36	851.75	687.37	407.81	340.19	471.17		
101	352,990	58.17	352,990	2,238.86	1,287.26	932.22	649.28	512.70	448.70	367.86		
102	492,572	55.05	492,572	2,639.89	1,792.91	1,329.65	1,070.20	644.65	594.28	577.61		
103	378,602	46.02	378,602	2,249.77	2,881.25	1,033.49	889.21	584.65	549.73	477.26		

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads											
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
104	357,963	56.07	357,963	6,127.37	3,612.43	2,653.60	1,832.78	1,534.37	1,345.85	1,150.95			
105	450,806	41.61	450,806	3,503.15	2,107.07	1,648.34	898.80	802.73	709.48	830.75			
106	454,379	56.18	454,379	3,545.67	2,231.19	1,515.70	1,034.37	805.25	769.93	746.07			
107	352,766	41.31	352,766	1,445.77	851.96	690.28	514.57	385.58	352.43	357.19			
108	460,793	39.30	460,793	2,147.61	1,271.65	999.40	540.53	345.77	295.77	293.42			
109	413,004	54.97	413,004	432,592.00	273,070.00	186,337.00	62,381.50	1,323.62	37,023.60	39,682.00			
110	418,769	60.62	418,769	5,894.66	3,588.86	2,798.92	1,372.20	792.82	1,341.13	409.84			
111	342,752	101.73	342,752	5,486.56	3,043.86	1,963.67	967.39	709.30	543.10	596.02			
112	367,110	114.06	367,110	3,138.09	1,710.08	1,013.18	563.55	410.97	326.76	272.92			
113	259,649	107.98	259,649	5,509.58	3,124.46	1,631.67	1,273.87	1,011.24	871.78	969.94			
114	463,474	107.27	463,474	3,200.65	1,803.71	1,088.43	711.25	561.96	469.64	520.57			
115	456,890	115.58	456,890	10,342.50	5,521.51	3,128.51	2,014.67	1,492.26	1,158.62	1,018.82			
116	530,601	107.89	530,601	3,774.90	2,299.98	1,333.40	906.58	712.92	639.94	542.20			
117	502,840	117.72	502,840	8,434.68	2,915.53	2,787.13	1,818.96	1,431.50	1,117.85	790.67			
118	349,749	62.23	349,749	231,655.00	141,930.00	84,103.60	10,465.40	747.87	16,316.70	529.76			

Table A.6 (cont. - Running times HPBP-PH Cicirello)

Tanaka		HPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
119	573,046	121.78	573,046	8,244.46	4,460.24	2,533.63	1,573.38	1,163.44	772.67	707.33		
120	396,183	84.73	396,183	1,247.44	778.44	398.49	247.11	200.97	150.38	146.27		

Table A.7: Running times HBPB-PH with 128 threads for Cicirello instances solved within 1 hour (87 of 120)

ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128
6	2,465.02	12	219.80	17	110.32	19	94.57	21	69.75	22	101.36		
23	0.10	25	130.79	26	110.56	27	135.25	28	160.17	29	80.02		
30	244.66	31	90.65	32	84.06	33	73.25	34	0.24	35	0.33		
36	64.63	37	2,350.57	38	0.20	39	71.30	40	57.12	41	36.67		
42	2,241.53	43	671.37	44	851.74	45	2,322.29	46	3,131.06	47	2,338.79		
49	313.52	50	1,521.66	51	1,535.99	52	2,013.67	53	1,098.85	54	3,392.79		
55	646.05	57	3,125.18	59	2,597.17	61	359.02	63	799.99	64	1,377.25		
65	341.30	66	363.12	67	597.97	68	612.05	70	282.63	71	1,545.29		
72	2,682.94	75	534.11	77	1,793.13	78	797.75	79	348.64	80	1,592.17		
81	91.48	82	2,053.63	83	2,555.91	84	1,248.69	86	361.17	89	1,636.34		
91	1,246.35	92	2,707.17	93	1,747.27	95	1,706.37	96	217.23	97	485.50		
98	197.65	99	180.32	100	601.23	101	742.75	102	953.37	103	1,164.07		
104	1,845.02	105	1,015.08	106	1,065.19	107	334.27	108	427.85	109	1,480.51		
110	1,651.42	111	1,031.93	112	522.71	113	1,236.49	114	1,350.06	115	1,511.84		
116	919.43	117	1,136.17	119	1,052.29	120	226.64						

Appendix A.2. *Strict Parallel Branch-and-Price with primal heuristic*

Table A.8: Solved instances over time in SPBP-PH

Set	#instances	#threads	Timelimit (h)			
			1	12	48	504
Rubin	32	1	29	29	31	32
		2	29	30	31	32
		4	29	31	32	32
		8	29	31	32	32
		16	29	31	32	32
		32	29	31	32	32
		64	29	31	32	32
Gagné	32	1	14	21	26	30
		2	15	22	26	31
		4	16	23	27	31
		8	19	25	27	31
		16	20	24	28	31
		32	20	26	30	31
Cicirello	120	1	53	98	107	116
		2	61	103	110	117
		4	74	106	110	117
		8	84	108	111	117
		16	91	110	115	117
		32	94	110	116	117
		64	95	111	116	117

Table A.9: Running times SPBP-PH over threads for Rubin instances

ins	Tanaka		SPBP-PH run with #threads											
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64				
401	90	0.31	90	0.87	0.61	0.37	0.60	0.43	0.24	0.18				
402	0	0.00	0	0.32	0.29	0.16	0.18	0.10	0.97	0.24				
403	3,418	0.53	3,418	0.45	0.36	0.27	0.25	0.14	0.11	0.09				
404	1,067	0.44	1,067	0.43	0.42	0.26	0.20	0.13	0.10	0.16				
405	0	0.00	0	0.08	0.06	0.11	0.08	0.03	0.12	0.15				
406	0	0.00	0	0.29	0.21	0.17	0.16	0.13	0.08	0.19				
407	1,861	0.48	1,861	5.47	3.78	2.44	1.87	1.62	1.51	1.70				
408	5,660	0.85	5,660	0.55	0.44	0.26	0.16	0.16	0.17	0.23				
501	261	2.09	261	13.34	8.93	5.18	3.29	3.18	2.32	2.17				
502	0	0.01	0	0.01	0.06	0.01	0.02	0.04	0.02	0.07				
503	3,497	3.04	3,497	15.06	10.07	6.00	4.49	4.45	2.62	2.59				
504	0	0.03	0	0.01	0.01	0.02	0.01	0.02	0.03	0.07				
505	0	0.02	0	0.79	0.58	0.28	0.27	0.18	0.16	0.24				

Table A.9 (cont. - Running times SPBP-PH Rubin)

Tanaka		SPBP-PH run with #threads											
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
506	0	0.02	0	4.95	3.25	2.51	1.57	1.17	1.02	1.21			
507	7,225	4.22	7,225	12.94	8.48	5.02	3.36	3.09	3.38	3.02			
508	1,915	3.84	1,915	4.16	2.68	2.35	1.07	0.85	1.18	0.70			
601	12	7.31	12	53.79	34.52	18.71	11.83	9.69	15.93	6.61			
602	0	0.05	0	17.36	11.31	8.08	4.25	3.78	12.18	3.12			
603	17,587	18.37	17,587	311.55	197.19	130.03	68.58	54.96	48.24	41.69			
604	19,092	25.73	19,092	279.03	176.47	116.00	61.61	51.31	46.38	37.12			
605	228	13.57	228	65,980.90	42,210.70	37,096.20	20,076.80	15,847.80	12,353.60	11,162.70			
606	0	0.05	0	10.00	6.19	4.89	2.37	3.52	1.97	1.80			
607	12,969	17.60	12,969	217.88	138.50	94.49	50.29	41.73	44.44	32.11			
608	4,732	32.92	4,732	173.77	108.48	63.07	39.51	34.04	28.21	26.15			
701	97	49.55	97	60,233.10	37,306.70	22,409.60	16,112.80	11,678.50	8,894.87	7,776.05			
702	0	0.09	0	0.01	0.01	0.01	0.06	0.02	0.04	0.09			
703	26,506	52.36	26,506	2,162.98	1,377.65	943.06	531.22	456.79	388.37	328.33			
704	15,206	71.40	15,206	417.36	266.46	152.12	94.81	75.13	62.41	57.59			

Table A.9 (cont. - Running times SPBP-PH Rubin)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
705	200	599.48	200	275,068.00	187,097.00	105,455.00	75,797.30	52,198.50	49,144.60	64,945.50		
706	0	0.10	0	0.01	0.01	0.01	0.01	0.02	0.02	0.07		
707	23,789	52.32	23,789	2,771.11	1,679.34	1,096.34	596.03	469.54	376.43	339.57		
708	22,807	99.77	22,807	556.75	349.40	199.31	125.19	99.00	82.75	79.05		

Table A.10: Running times SPBP-PH with 128 threads for Rubin instances solved within 1 hour (29 of 32)

ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128
401	0.44	402	0.22	403	0.32	404	0.44	405	0.29	406	0.27
407	5.02	408	0.53	501	4.28	502	0.10	503	4.75	504	0.10
505	0.62	506	3.22	507	6.88	508	1.48	601	12.39	602	6.17
603	69.35	604	66.55	606	4.87	607	57.29	608	47.93	702	0.10
703	570.83	704	102.57	706	0.10	707	722.90	708	149.49		

Table A.11: Running times SPBP-PH over threads for Gagné instances

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
551	183	116.23	183	594.41	328.63	226.17	113.16	80.42	59.23	49.07		
552	0	0.17	0	0.01	0.01	0.01	0.01	0.02	0.03	0.08		
553	40,498	112.04	40,498	5,064.50	2,735.47	1,622.80	1,036.18	890.20	645.95	571.87		
554	14,653	277.87	14,653	556.77	339.43	202.30	118.86	95.12	71.60	61.72		
555	0	0.27	0	389.21	226.76	131.68	75.62	60.35	48.08	43.58		
556	0	0.18	0	0.01	0.22	0.02	0.02	0.03	0.04	0.08		
557	35,813	140.91	35,813	81,481.50	53,681.70	30,241.20	19,426.20	14,346.00	12,473.80	10,127.80		
558	19,871	228.58	19,871	1,851.68	1,141.82	689.12	441.38	344.59	301.87	463.18		
651	247	2,298.27	247	8,463.07	4,956.35	2,573.08	1,483.96	1,194.38	744.95	606.51		
652	0	0.25	0	0.02	0.22	0.01	0.02	0.11	0.02	0.08		
653	57,500	262.80	57,500	55,896.70	36,206.80	22,319.30	14,583.00	10,567.40	10,166.30	8,332.93		
654	34,301	381.56	34,301	14,403.10	8,704.86	5,167.69	3,079.38	2,452.78	1,870.45	1,588.65		
655	0	355.72	0	1,559.29	925.41	507.17	282.26	222.87	152.73	121.96		

Table A.11 (cont. - Running times SPBP-PH Gagné)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
656	0	0.29	0	0.02	0.22	0.07	0.03	0.04	0.02	0.08		
657	54,895	253.71	54,895	21,221.10	12,452.80	7,072.64	3,987.54	2,968.20	2,157.51	1,810.51		
658	27,114	466.63	27,114	15,103.70	9,172.86	5,351.47	3,581.59	2,974.71	2,444.16	2,340.26		
751	225	-	225	1,460,150.00	883,918.00	481,676.00	320,124.00	223,806.00	158,629.00	130,147.00		
752	0	0.44	0	0.03	0.20	0.02	0.06	0.08	0.04	0.09		
753	77,544	565.95	77,544	38,980.30	24,225.90	14,014.00	9,214.73	6,728.23	5,608.75	4,941.36		
754	35,200	887.85	35,200	157,443.00	122,138.00	69,312.60	46,871.40	36,234.70	27,899.20	26,245.70		
755	0	0.41	0	87.90	48.26	25.23	13.41	9.31	7.64	8.72		
756	0	0.52	0	0.02	0.24	0.02	0.48	0.27	0.03	0.09		
757	59,635	741.33	59,635	984,865.00	606,760.00	343,578.00	241,954.00	181,176.00	142,062.00	126,754.00		
758	38,339	987.59	38,339	13,718.60	8,082.19	4,650.96	3,195.28	2,312.76	1,855.87	1,651.76		
851	363	-	360	292,313.00	178,364.00	113,623.00	66,935.70	42,273.60	36,027.60	29,402.40		
852	0	0.61	0	0.02	0.25	0.02	0.09	0.04	0.06	0.07		
853	97,497	2,219.22	97,497	148,568.00	99,633.40	60,884.60	37,689.40	35,306.40	25,434.40	37,185.70		
854	79,042	1,177.04	79,042	171,476.00	111,018.00	72,559.50	41,472.90	32,311.30	26,243.80	19,195.70		

Table A.11 (cont. - Running times SPBP-PH Gagné)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
855	260	—	256	—	—	—	—	—	—	—	—	—
856	0	0.63	0	0.02	0.23	0.02	0.05	0.04	0.03	0.08		
857	87,011	4,594.41	87,011	934,332.00	569,395.00	310,244.00	209,474.00	147,052.00	114,258.00	99,223.10		
858	74,739	1,817.68	74,739	—	1,682,130.00	943,703.00	675,113.00	623,068.00	515,538.00	366,282.00		

Table A.12: Running times SPBP-PH with 128 threads for Gagné instances solved within 1 hour (19 of 32)

ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128
551	114.27	552	0.11	553	905.16	554	117.75	555	120.51	556	0.11
558	518.86	651	1,366.39	652	0.11	654	2,525.70	655	393.90	656	0.10
657	3,610.62	752	0.11	755	15.94	756	0.11	758	2,697.11	852	0.09
856	0.05										

Table A.13: Running times SPBP-PH over threads for Cicirello instances

ins	Tanaka		SPBP-PH run with #threads											
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64				
1	453	93.58	453	764,819.00	434,062.00	270,093.00	203,251.00	172,724.00	141,066.00	135,057.00				
2	4,794	1,474.67	4,794	46,323.20	30,260.50	17,238.30	10,447.10	7,822.45	6,134.58	5,541.13				
3	1,390	589.07	1,390	161,807.00	100,924.00	61,477.60	34,926.80	25,338.70	19,887.40	18,058.70				
4	5,866	129.80	5,866	18,020.00	9,794.37	6,321.59	4,188.10	3,523.34	3,326.81	2,936.05				
5	4,054	2,350.37	4,054	187,346.00	118,072.00	69,500.50	39,463.40	31,651.10	25,945.20	22,176.40				
6	6,592	166.86	6,592	7,407.97	4,366.65	2,994.94	1,987.68	1,569.61	1,393.82	1,288.26				
7	3,267	3,522.47	3,267	1,197,540.00	687,397.00	388,091.00	264,245.00	191,244.00	150,127.00	133,418.00				
8	100	105.54	100	—	—	—	—	—	—	—				
9	5,660	124.33	5,660	58,397.30	36,511.80	21,067.60	13,806.90	9,095.70	7,542.26	9,549.96				
10	1,740	8,883.74	1,740	659,084.00	404,800.00	252,465.00	185,270.00	159,510.00	133,952.00	123,176.00				
11	2,785	42,921.26	2,785	642,293.00	364,362.00	203,490.00	143,124.00	95,066.20	73,474.30	63,757.20				
12	0	0.41	0	254.60	149.37	78.17	45.97	45.12	29.51	25.73				
13	3,904	8,722.61	3,904	777,434.00	451,972.00	268,627.00	188,024.00	149,776.00	123,711.00	112,130.00				

Table A.13 (cont. - Running times SPBP-PH Cicirello)

Tanaka		SPBP-PH run with #threads											
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
14	2,075	3,555.71	2,075	206,692.00	136,137.00	68,711.20	37,087.30	28,131.20	20,603.70	17,610.80			
15	724	496.78	724	21,172.30	12,036.30	6,661.74	5,157.82	3,109.24	2,538.09	2,276.94			
16	3,285	504.49	3,285	57,225.20	38,362.20	18,852.10	15,148.80	7,636.27	5,743.31	4,963.54			
17	0	1,189.80	0	703.79	406.06	223.21	127.60	104.80	75.80	67.42			
18	773	-	767	-	-	-	-	-	-	-			
19	0	3.41	0	13,813.50	8,215.91	4,502.77	3,545.32	2,448.71	2,064.86	1,869.15			
20	1,757	316.22	1,757	37,225.80	21,856.10	13,374.20	8,261.42	5,765.41	4,935.41	5,201.19			
21	0	0.20	0	302.59	170.28	97.75	69.86	46.33	44.41	34.67			
22	0	0.23	0	291.89	164.07	96.89	65.76	44.60	35.92	33.52			
23	0	0.17	0	0.01	0.02	0.01	0.01	0.03	0.05	0.02			
24	761	-	878	-	-	-	-	-	-	-			
25	0	0.31	0	411.21	233.44	131.46	78.01	69.24	70.39	51.61			
26	0	0.23	0	308.29	167.75	93.59	56.50	45.90	40.71	33.91			
27	0	0.34	0	576.63	333.37	190.33	116.76	93.91	100.45	62.20			
28	0	0.40	0	569.51	329.56	183.32	111.35	86.91	69.51	62.66			

Table A.13 (cont. - Running times SPBP-PH Cicirello)

ins	Tanaka			SPBP-PH run with #threads										
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64				
29	0	0.23	0	300.54	171.65	96.60	59.38	48.45	40.06	37.82				
30	0	7.85	0	1,869.82	1,146.47	676.72	445.89	452.35	422.75	291.57				
31	0	0.45	0	333.14	187.00	98.67	57.00	47.27	37.69	28.88				
32	0	0.48	0	366.70	210.39	112.66	65.97	56.83	41.90	36.48				
33	0	0.48	0	431.63	243.12	129.59	76.41	55.34	47.29	37.75				
34	0	0.40	0	0.18	0.11	0.06	0.07	0.09	0.06	0.06				
35	0	0.47	0	0.16	0.11	0.05	0.12	0.08	0.68	0.08				
36	0	0.44	0	350.21	195.32	103.71	59.54	45.57	40.63	32.46				
37	0	595.89	0	2,137.46	1,223.43	664.85	386.00	325.01	254.98	193.28				
38	0	0.38	0	0.16	0.11	0.05	0.06	0.27	0.15	0.38				
39	0	0.48	0	388.50	211.10	115.05	65.85	55.07	38.97	33.02				
40	0	0.40	0	391.47	214.76	117.45	68.34	52.74	49.54	37.32				
41	69,102	32.06	69,102	197.93	115.55	65.94	40.64	31.55	25.81	24.85				
42	57,487	45.69	57,487	5,708.65	3,182.31	2,037.80	1,442.22	1,234.92	1,219.92	1,042.50				
43	145,310	80.55	145,310	3,148.87	1,903.58	1,121.68	710.50	561.53	466.88	427.15				

Table A.13 (cont. - Running times SPBP-PH Cicirello)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
44	35,166	58.20	35,166	2,176.77	1,374.20	933.72	675.59	592.81	516.00	579.17		
45	58,935	80.98	58,935	9,115.60	5,673.01	3,231.14	2,215.64	1,855.13	1,774.86	1,535.40		
46	34,764	54.08	34,764	11,654.50	7,312.72	4,565.23	3,113.49	2,678.60	2,351.15	2,240.55		
47	72,853	61.35	72,853	8,360.93	5,166.25	2,950.39	2,001.94	1,601.44	1,447.86	1,351.23		
48	64,612	105.70	64,612	29,552.60	17,667.10	10,637.20	6,203.00	4,192.46	3,330.12	2,885.62		
49	77,449	59.38	77,449	1,024.81	687.51	409.37	285.77	239.32	210.97	199.08		
50	31,092	59.55	31,092	4,645.45	2,614.82	1,696.37	1,234.75	1,199.18	958.51	997.95		
51	49,208	91.84	49,208	10,404.60	6,843.21	3,276.75	2,041.59	1,568.81	1,257.59	1,117.60		
52	93,045	117.65	93,045	13,518.60	7,431.77	4,048.03	2,323.00	1,783.39	1,300.97	1,126.97		
53	84,841	116.94	84,841	5,696.02	2,969.06	1,661.31	1,025.81	837.19	584.72	519.53		
54	118,809	104.63	118,809	24,431.00	13,077.00	8,027.45	5,243.56	3,372.13	2,694.49	2,440.91		
55	64,315	108.07	64,315	400.08	238.91	122.19	72.49	49.71	37.02	31.34		
56	74,889	122.09	74,889	34,401.30	19,068.50	10,518.20	6,760.98	4,424.26	3,294.30	2,794.30		
57	63,514	97.11	63,514	15,633.50	8,818.79	5,728.87	3,389.40	2,621.27	2,173.27	2,006.54		
58	45,322	135.31	45,322	110,531.00	70,389.60	40,906.30	27,333.70	22,196.60	17,502.90	16,484.40		

Table A.13 (cont. - Running times SPBP-PH Cicirello)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
59	50,999	92.49	50,999	15,788.40	8,742.06	5,530.64	3,211.87	2,498.78	1,987.60	1,783.85		
60	60,765	137.23	60,765	39,347.20	24,586.50	14,230.00	11,114.30	7,228.24	5,587.36	6,971.11		
61	75,916	41.42	75,916	1,692.68	1,052.40	569.44	343.96	259.24	212.42	183.56		
62	44,769	37.31	44,769	117,872.00	79,972.30	40,298.40	39,387.70	32,359.30	29,798.60	27,559.00		
63	75,317	33.11	75,317	2,861.67	1,776.20	1,023.50	656.18	515.71	428.89	403.22		
64	92,572	39.94	92,572	4,469.63	2,433.89	1,506.70	1,069.80	891.23	939.31	736.52		
65	126,696	41.08	126,696	1,068.20	708.34	406.38	280.79	232.44	200.41	203.80		
66	59,685	17.81	59,685	2,458.32	1,630.56	1,018.29	733.08	629.57	557.44	559.03		
67	29,390	26.93	29,390	1,442.07	947.00	564.25	386.56	329.40	290.74	278.69		
68	22,120	25.99	22,120	2,081.19	1,311.07	732.45	536.98	371.00	310.96	334.88		
69	71,118	45.03	71,118	29,807.70	18,495.10	12,786.80	11,640.20	8,192.51	4,625.74	8,863.59		
70	75,102	36.69	75,102	893.90	596.55	350.36	288.62	202.10	189.42	168.82		
71	145,007	96.11	145,007	12,093.30	6,554.68	3,605.16	2,341.81	1,567.67	1,228.40	1,047.04		
72	43,286	68.28	43,286	17,621.80	9,595.61	5,271.37	3,057.71	2,223.00	1,655.54	1,400.90		
73	28,785	91.49	28,785	57,803.30	36,634.60	22,441.20	16,254.00	10,742.80	8,980.61	8,319.94		

Table A.13 (cont. - Running times SPBP-PH Cicirello)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
74	29,777	73.14	29,777	22,520.70	12,914.50	7,690.24	6,755.54	3,982.13	3,340.68	3,074.77		
75	21,602	77.57	21,602	2,398.51	1,510.01	801.81	590.24	466.27	393.63	376.10		
76	53,555	75.75	53,555	20,577.20	11,774.10	6,993.58	6,133.12	3,560.96	3,040.99	2,728.77		
77	31,817	84.48	31,817	12,426.50	6,900.87	3,913.04	2,397.30	1,807.87	1,449.07	1,287.47		
78	19,462	79.56	19,462	3,378.59	2,048.43	1,364.95	740.87	574.15	463.65	416.21		
79	114,999	70.12	114,999	2,127.41	1,284.75	842.19	460.03	346.41	279.04	245.41		
80	18,157	76.73	18,157	15,984.20	9,011.19	5,009.50	3,425.87	2,195.39	1,791.84	1,500.78		
81	383,485	30.13	383,485	737.00	497.72	381.36	250.84	221.77	199.36	194.00		
82	409,479	66.12	409,479	11,705.20	7,040.22	4,430.60	3,069.58	2,570.08	1,412.15	2,155.13		
83	458,752	42.51	458,752	16,796.90	9,873.81	6,323.81	5,364.14	3,159.35	2,772.44	2,508.05		
84	329,670	46.59	329,670	6,681.81	3,916.26	2,661.25	1,862.87	1,590.37	1,357.31	1,390.78		
85	554,766	74.38	554,766	35,768.70	20,588.00	11,284.10	7,727.14	5,188.70	4,220.73	3,586.68		
86	361,417	68.14	361,417	1,151.50	694.49	429.55	279.86	202.62	163.51	144.16		
87	398,551	47.96	398,551	77,762.80	53,796.90	40,367.70	27,922.90	22,651.60	20,076.20	19,119.90		
88	433,186	60.54	433,186	16,727.80	10,158.40	6,698.28	3,853.34	2,541.64	2,064.79	1,869.23		

Table A.13 (cont. - Running times SPBP-PH Cicirello)

ins	Tanaka		SPBP-PH run with #threads										
	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64			
89	410,092	46.85	410,092	10,520.50	7,130.25	5,799.47	3,371.02	2,778.55	2,510.19	2,438.73			
90	401,653	56.97	401,653	23,878.20	15,318.40	11,447.40	7,681.06	6,374.23	5,467.48	5,200.92			
91	339,933	68.98	339,933	5,497.26	2,937.65	1,791.85	1,220.80	1,023.28	856.59	791.81			
92	361,152	103.36	361,152	7,121.18	4,073.99	2,233.87	1,477.01	1,210.78	987.69	912.02			
93	403,423	125.52	403,423	10,374.90	5,487.01	3,189.32	1,987.97	1,536.54	1,262.64	1,122.01			
94	332,941	98.30	332,941	—	1,417,060.00	931,276.00	679,787.00	619,479.00	631,159.00	574,826.00			
95	516,926	101.20	516,926	12,002.90	7,118.30	5,266.95	2,855.92	2,220.89	1,903.90	1,884.07			
96	455,448	79.50	455,448	1,221.29	709.73	418.42	281.62	199.89	156.14	141.32			
97	407,590	92.92	407,590	2,525.89	1,484.59	884.63	570.12	442.96	366.21	338.23			
98	520,582	90.23	520,582	1,113.84	634.85	362.12	232.87	157.18	122.98	110.48			
99	363,518	106.21	363,518	922.87	544.00	323.13	239.28	152.94	125.64	118.13			
100	431,736	80.98	431,736	2,302.93	1,374.24	813.16	526.28	415.47	346.50	314.13			
101	352,990	58.17	352,990	2,122.52	1,353.44	879.36	617.20	526.36	464.54	447.15			
102	492,572	55.05	492,572	2,604.45	1,676.38	1,096.58	836.58	663.98	589.48	563.65			
103	378,602	46.02	378,602	2,225.61	1,437.94	959.55	756.20	620.95	534.99	506.67			

Table A.13 (cont. - Running times SPBP-PH Cicirello)

Tanaka		SPBP-PH run with #threads										
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64		
104	357,963	56.07	357,963	6,875.76	3,871.10	2,504.74	1,930.98	1,569.30	1,403.32	1,347.61		
105	450,806	41.61	450,806	3,456.39	2,178.43	1,400.60	1,056.95	755.82	723.12	709.46		
106	454,379	56.18	454,379	3,500.56	2,423.99	1,417.11	1,055.06	828.61	745.94	684.06		
107	352,766	41.31	352,766	1,422.14	972.31	609.92	450.59	384.54	345.22	362.39		
108	460,793	39.30	460,793	2,143.26	1,323.81	748.53	465.91	358.88	294.13	261.22		
109	413,004	54.97	413,004	417,437.00	288,744.00	207,514.00	165,728.00	155,953.00	60,405.10	1,816.81		
110	418,769	60.62	418,769	6,599.02	4,224.25	2,671.24	2,046.09	1,783.98	1,624.00	1,453.60		
111	342,752	101.73	342,752	6,193.81	3,101.55	1,708.98	1,013.39	746.70	558.42	482.78		
112	367,110	114.06	367,110	3,219.53	1,791.80	1,005.38	592.69	433.92	328.10	291.18		
113	259,649	107.98	259,649	6,373.45	3,470.44	1,976.84	1,313.70	1,044.19	882.36	839.95		
114	463,474	107.27	463,474	3,165.91	1,920.08	1,132.87	735.49	579.46	474.33	448.63		
115	456,890	115.58	456,890	10,847.60	5,815.56	3,251.93	1,981.15	1,461.02	1,142.67	1,008.80		
116	530,601	107.89	530,601	4,471.09	2,274.17	1,374.35	925.51	731.15	621.91	570.08		
117	502,840	117.72	502,840	9,005.66	4,938.52	2,915.08	1,900.23	1,482.87	1,222.67	1,112.71		
118	349,749	62.23	349,749	231,111.00	147,701.00	92,251.70	55,174.00	42,446.50	35,042.10	34,875.10		

Table A.13 (cont. - Running times SPBP-PH Cicirello)

Tanaka		SPBP-PH run with #threads									
ins	best	time	our_best	t_1	t_2	t_4	t_8	t_16	t_32	t_64	
119	573,046	121.78	573,046	8,851.38	5,157.44	2,665.85	1,640.46	1,213.36	956.61	845.22	
120	396,183	84.73	396,183	1,248.65	731.60	414.50	253.81	192.57	153.05	136.06	

Table A.14: Running times SPBP-PH with 128 threads for Cicirello instances solved within one hour (84 of 120)

ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128	ins	t_128
6	2,179.07	12	55.79	17	145.75	21	123.07	22	88.92	23	0.10		
25	121.74	26	106.49	27	197.25	28	176.19	29	93.40	30	620.97		
31	54.02	32	78.64	33	81.74	34	0.38	35	0.20	36	65.87		
37	448.37	38	0.44	39	73.87	40	94.31	41	50.46	42	1,749.57		
43	806.17	44	1,072.35	45	2,149.37	46	3,109.11	47	2,145.51	49	592.72		
50	1,598.03	51	1,673.90	52	2,002.01	53	1,019.30	55	51.15	57	3,303.80		
59	2,802.97	61	337.92	63	788.86	64	1,070.95	65	283.15	66	842.37		
67	415.32	68	772.89	70	248.10	71	1,675.24	72	2,137.06	75	560.18		
77	2,002.14	78	626.54	79	422.99	80	2,532.78	81	260.89	84	2,420.44		
86	284.93	88	3,504.64	89	3,408.24	91	848.44	92	1,214.10	93	1,984.76		
95	2,361.51	96	200.92	97	532.40	98	270.60	99	200.13	100	414.10		
101	912.15	102	738.74	103	1,016.57	104	1,738.29	105	1,527.29	106	926.61		
107	427.84	108	640.01	110	2,189.81	111	1,046.99	112	437.69	113	1,035.14		
114	548.76	115	1,777.65	116	715.74	117	2,075.06	119	1,296.12	120	204.48		

Appendix A.3. Comparison of serial versions of the algorithm

Table A.15: Statistics for the four algorithms on Rubin instances solved within one hour in serial execution (means taken over instances solved by all four versions)

Algorithm	SPHP+PH	SPHP	MF-SPHP-PH	MF-SPHP
#solved	29	29	27	27
#b&b-nodes (\emptyset)	13	20	19	24
time per node (\emptyset)	3.20	3.06	2.96	2.89
#columns (\emptyset)	1905	2084	2586	2714
#columns per node (\emptyset)	51	59	52	56

Table A.16: Statistics for the four algorithms on Gagné instances solved within one hour in serial execution (means taken over instances solved by all four versions)

Algorithm	SPHP+PH	SPHP	MF-SPHP-PH	MF-SPHP
#solved	14	14	14	14
#b&b-nodes (\emptyset)	11	21	13	22
time per node (\emptyset)	18.26	17.92	18.70	18.46
#columns (\emptyset)	2813	3293	3463	3856
#columns per node (\emptyset)	94	100	102	107

Table A.17: Statistics for the four algorithms on Cicirello instances solved within one hour in serial execution (means taken over instances solved by all three versions)

Algorithm	SPHP+PH	SPHP	MF-SPHP-PH	MF-SPHP
#solved	53	45	27	25
#b&b-nodes (\emptyset)	63	70	79	79
time per node (\emptyset)	19.35	22.05	12.12	14.27
#columns (\emptyset)	4524	5343	4806	4916
#columns per node (\emptyset)	98	99	39	10

Table A.18: Comparison of serial execution results for the algorithm using new branching strategy and primal heuristic and using most-fractional branching on Rubín instances

ins	Tanaka		SPHP+PH			SPHP			MF-SPHP-PH			MF-SPHP		
	val	time	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols
401	90	0.31	0.87	3	230	0.81	3	230	0.74	3	220	0.75	3	220
402	0	0.00	0.32	3	98	1.16	15	439	0.94	9	282	1.41	23	659
403	3,418	0.53	0.45	1	116	0.46	1	116	0.48	1	116	0.43	1	116
404	1,067	0.44	0.43	1	129	0.42	1	129	0.47	1	129	0.47	1	129
405	0	0.00	0.08	1	31	0.40	25	227	0.13	1	31	0.40	29	255
406	0	0.00	0.29	1	92	1.41	23	631	0.34	1	92	1.51	23	664
407	1,861	0.48	5.47	27	1,778	5.48	27	1,778	3.91	17	1,260	4.06	17	1,260
408	5,660	0.85	0.55	1	141	0.55	1	141	0.67	1	141	0.70	1	141
501	261	2.09	13.34	7	1,079	13.27	7	1,079	10.46	5	835	10.60	5	835
502	0	0.01	0.01	1	2	0.01	1	2	0.01	1	2	0.01	1	2
503	3,497	3.04	15.06	9	1,132	17.91	11	1,344	17.27	9	1,260	17.33	9	1,260
504	0	0.03	0.01	1	2	0.03	1	2	0.04	1	2	0.03	1	2
505	0	0.02	0.79	1	72	4.74	41	676	0.91	1	72	5.68	37	803
506	0	0.02	4.95	23	540	6.48	35	840	4.94	29	598	6.24	47	1,004
507	7,225	4.22	12.94	7	1,028	12.79	7	1,028	13.47	7	1,075	12.39	7	1,075
508	1,915	3.84	4.16	1	323	4.19	1	323	4.45	1	323	4.01	1	323

Table A.18 (cont. - Serial comparison Rubin)

ins	Tanaka		SPHP+PH			SPHP			MF-SPHP-PH			MF-SPHP		
	val	time	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols
601	12	7.31	53.79	5	1,740	95.18	11	3,231	64.74	7	2,039	61.23	7	2,039
602	0	0.05	17.36	41	982	20.05	61	1,409	21.96	55	1,414	22.21	61	1,617
603	17,587	18.37	311.55	55	8,984	294.75	55	8,984	489.64	89	14,731	482.09	89	14,731
604	19,092	25.73	279.03	61	7,994	264.50	61	7,994	497.28	119	14,689	489.41	119	14,689
606	0	0.05	10.00	25	411	19.44	69	1,135	21.67	61	1,251	22.29	69	1,521
607	12,969	17.60	217.88	23	6,445	204.94	23	6,445	174.67	21	5,283	166.24	21	5,283
608	4,732	32.92	173.77	15	4,652	163.21	15	4,652	119.18	9	3,276	135.41	11	3,961
702	0	0.09	0.01	1	2	0.03	1	2	0.03	1	2	0.01	1	2
703	26,506	52.36	2,162.98	133	32,778	2,121.90	133	32,778	—	—	—	—	—	—
704	15,206	71.40	417.36	21	5,598	399.61	21	5,598	809.69	43	11,223	788.49	43	11,223
706	0	0.10	0.01	1	2	0.01	1	2	0.01	1	2	0.01	1	2
707	23,789	52.32	2,771.11	191	42,262	2,734.51	191	42,262	—	—	—	—	—	—
708	22,807	99.77	556.75	21	7,837	537.53	21	7,837	662.10	31	9,465	662.92	31	9,465

Table A.19: Comparison of serial execution results for the algorithm using new branching strategy with (SPBP-PH)/without (SPBP) primal heuristic and using most-fractional branching (MF-SPBP) on Gagné instances

ins	Tanaka		SPHP+PH			SPHP			MF-SPHP-PH			MF-SPHP		
	val	time	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols
551	183	116.23	594.41	11	5,454	574.69	11	5,454	640.22	13	5,766	638.62	13	5,766
552	0	0.17	0.01	1	2	0.02	1	2	0.05	1	2	0.02	1	2
554	14,653	277.87	556.77	11	4,337	535.48	11	4,337	687.37	15	5,469	691.85	15	5,469
555	0	0.27	389.21	69	6,369	369.24	69	6,369	323.06	81	6,025	319.49	81	6,025
556	0	0.18	0.01	1	2	0.02	1	2	0.02	1	2	0.02	1	2
558	19,871	228.58	1,851.68	21	13,523	1,827.66	21	13,523	1,582.40	19	11,769	1,584.37	19	11,769
652	0	0.25	0.02	1	2	0.02	1	2	0.05	1	2	0.02	1	2
655	0	355.72	1,559.29	19	9,320	1,542.74	19	9,320	3,138.50	31	19,066	2,987.73	31	19,066
656	0	0.29	0.02	1	2	0.04	1	2	0.03	1	2	0.02	1	2
752	0	0.44	0.03	1	2	0.02	1	2	0.02	1	2	0.04	1	2
755	0	0.41	87.90	9	366	604.20	149	7,080	80.05	11	371	569.77	145	5,875
756	0	0.52	0.02	1	2	0.07	1	2	0.02	1	2	0.02	1	2
852	0	0.61	0.02	1	2	0.03	1	2	0.03	1	2	0.02	1	2
856	0	0.63	0.02	1	2	0.05	1	2	0.07	1	2	0.02	1	2

Table A.20: Comparison of serial execution results for the algorithm using new branching strategy with (SPBP-PH)/without (SPBP) primal heuristic and using most-fractional branching (MF-SPBP) on Cicirello instances

ins	Tanaka		SPHP+PH			SPHP			MF-SPHP-PH			MF-SPHP		
	val	time	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols
12	0	0.41	254.60	99	2,934	249.24	105	3,219	242.71	115	3,359	243.28	115	3,359
17	0	1,189.80	703.79	39	4,486	1,108.22	57	8,937	72.24	3	326	—	—	—
19	0	3.41	—	—	—	—	—	—	1,071.94	167	22,001	1,089.16	167	22,001
21	0	0.20	302.59	93	3,885	307.15	107	4,701	378.74	119	6,066	389.45	121	6,371
22	0	0.23	291.89	99	3,667	289.48	111	4,272	389.13	131	6,129	389.44	131	6,129
23	0	0.17	0.01	1	2	0.01	1	2	0.01	1	2	0.01	1	2
25	0	0.31	411.21	97	5,573	578.08	173	12,489	369.05	113	6,158	372.32	113	6,158
26	0	0.23	308.29	105	3,829	306.94	117	4,860	300.64	135	4,815	302.50	135	4,815
27	0	0.34	576.63	67	6,006	603.84	77	6,954	536.34	97	7,513	555.46	99	7,835
28	0	0.40	569.51	77	7,187	566.05	79	7,604	564.59	103	9,270	578.91	103	9,270
29	0	0.23	300.54	111	4,278	288.73	113	4,429	343.84	115	5,871	346.54	115	5,871
30	0	7.85	1,869.82	89	20,204	2,037.94	101	22,877	670.88	73	8,863	700.50	73	8,863
31	0	0.45	333.14	89	2,841	338.91	101	3,395	291.00	129	3,371	293.99	129	3,371
32	0	0.48	366.70	101	4,062	350.35	101	4,062	326.30	123	3,831	329.31	123	3,831
33	0	0.48	431.63	93	3,467	418.37	95	3,542	346.31	131	3,834	339.72	131	3,834
34	0	0.40	0.18	1	2	0.18	1	2	0.20	1	2	0.18	1	2
35	0	0.47	0.16	1	2	0.18	1	2	0.16	1	2	0.16	1	2

Table A.20 (cont. - Serial comparison Cicirello)

ins	val	Tanaka		SHPH+PH			SHPH			MF-SHPH-PH			MF-SHPH				
		time	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols
36	0	0.44	350.21	101	3,134	340.27	107	3,514	339.23	133	4,349	342.12	135	4,458	—	—	—
37	0	595.89	2,137.46	59	15,549	2,134.75	59	15,549	—	—	—	—	—	—	—	—	—
38	0	0.38	0.16	1	2	0.17	1	2	0.18	1	2	0.16	1	2	—	—	—
39	0	0.48	388.50	87	3,165	383.19	91	3,508	323.90	123	4,328	322.84	123	4,328	—	—	—
40	0	0.40	391.47	101	3,820	371.01	101	3,820	300.14	119	3,288	303.18	119	3,288	—	—	—
41	69,102	32.06	197.93	1	1,459	470.63	5	3,496	196.28	1	1,459	292.04	3	2,115	—	—	—
43	145,310	80.55	3,148.87	71	20,705	3,273.58	75	21,942	—	—	—	—	—	—	—	—	—
44	35,166	58.20	2,176.77	33	16,003	3,143.68	55	24,709	3,464.29	51	25,982	—	—	—	—	—	—
49	77,449	59.38	1,024.81	15	6,826	2,547.87	29	18,276	—	—	—	—	—	—	—	—	—
55	64,315	108.07	400.08	5	1,791	533.10	7	2,518	288.85	3	1,314	468.87	5	2,120	—	—	—
61	75,916	41.42	1,692.68	29	12,331	1,911.25	31	14,109	—	—	—	—	—	—	—	—	—
63	75,317	33.11	2,861.67	41	21,118	2,834.91	41	21,118	—	—	—	—	—	—	—	—	—
65	126,696	41.08	1,068.20	13	6,756	1,433.85	19	9,304	—	—	—	—	—	—	—	—	—
66	59,685	17.81	2,458.32	11	16,647	2,437.09	11	16,647	2,899.82	33	21,122	2,893.09	33	21,122	—	—	—
67	29,390	26.93	1,442.07	19	10,260	2,592.62	31	19,721	—	—	—	—	—	—	—	—	—
68	22,120	25.99	2,081.19	39	17,164	2,061.16	39	17,164	—	—	—	—	—	—	—	—	—
70	75,102	36.69	893.90	11	6,105	993.39	13	6,981	810.99	11	5,594	897.91	13	5,914	—	—	—
75	21,602	77.57	2,398.51	25	10,738	3,235.81	35	14,864	—	—	—	—	—	—	—	—	—
78	19,462	79.56	3,378.59	41	13,955	—	—	—	—	—	—	—	—	—	—	—	—
79	114,999	70.12	2,127.41	21	8,811	2,959.92	29	12,824	—	—	—	—	—	—	—	—	—

Table A.20 (cont. - Serial comparison Cicirello)

ins	val	Tanaka		SPHP+PH			SPHP			MF-SPHP-PH			MF-SPHP				
		time	time	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols	time	#nodes	#cols		
81	383,485	30.13	737.00	1	4,974	—	—	—	—	—	—	743.40	1	4,974	2,662.26	13	16,216
86	361,417	68.14	1,151.50	19	8,984	1,364.22	23	10,619	—	—	—	—	—	—	—	—	—
96	455,448	79.50	1,221.29	13	5,434	1,353.00	15	5,941	—	—	—	—	—	—	—	—	—
97	407,590	92.92	2,525.89	29	11,851	—	—	—	—	—	—	—	—	—	—	—	—
98	520,582	90.23	1,113.84	19	5,070	1,110.57	19	5,070	—	—	—	—	—	—	—	—	—
99	363,518	106.21	922.87	9	4,081	2,155.09	23	10,613	—	—	—	—	—	—	—	—	—
100	431,736	80.98	2,302.93	19	9,966	—	—	—	—	—	—	—	—	—	—	—	—
101	352,990	58.17	2,122.52	25	14,025	—	—	—	—	—	—	—	—	—	—	—	—
102	492,572	55.05	2,604.45	29	17,775	3,126.29	37	21,535	—	—	—	—	—	—	—	—	—
103	378,602	46.02	2,225.61	27	15,748	3,198.39	43	24,234	—	—	—	—	—	—	—	—	—
105	450,806	41.61	3,456.39	43	24,070	—	—	—	—	—	—	—	—	—	—	—	—
106	454,379	56.18	3,500.56	49	22,629	—	—	—	—	—	—	—	—	—	—	—	—
107	352,766	41.31	1,422.14	11	9,953	2,980.28	29	21,909	—	—	—	—	—	—	—	—	—
108	460,793	39.30	2,143.26	21	14,973	2,162.82	21	14,973	—	—	—	—	—	—	—	—	—
112	367,110	114.06	3,219.53	57	14,379	3,558.68	63	16,420	—	—	—	—	—	—	—	—	—
114	463,474	107.27	3,165.91	37	13,676	—	—	—	—	—	—	—	—	—	—	—	—
120	396,183	84.73	1,248.65	13	5,495	2,692.26	23	12,362	—	—	—	—	—	—	—	—	—

Appendix B. Detailed Speedup Results of Computational Experiments

1060 This section of the appendix presents detailed results on the speedup of
the parallelization approaches (SPBP-PH and HPBP-PH). The following tables
B.21–B.26 show summary speedup statistics for both versions of the algorithm
and for the three benchmark sets. The first column contains the algorithm
acronym, the second column contains the abbreviated name of the statistical
1065 metric, followed by the metric values over the number of threads used in the
algorithm. The indicators include medians (Med), averages (Avg), standard de-
viations (SD), and the coefficients of variation (CV; which is the ratio of the SD
and Avg). They were taken from instances in the respective benchmark set that
took at least 60 seconds to solve, to account for cases where no parallelization
1070 (or even column generation) was used, e.g. when the algorithm terminated with
the initial schedule found by the heuristic.

This is followed by visualizations of the same data as box plots in figures
B.2–B.4. Finally, plots of the parallel efficiency for the algorithms are shown in
figures B.5, and table B.27 shows the comparison of waiting conditions during
1075 the execution of the two algorithms (see Section 5).

Table B.21: Average overall parallel speedup (Rubin)

#threads		1	2	4	8	16	32	64
	Med	1.00	1.58	2.53	4.40	5.27	6.16	7.04
SPBP	Avg	1.00	1.58	2.48	4.18	5.26	6.14	6.85
-PH	(SD)	(0.00)	(0.04)	(0.30)	(0.44)	(0.48)	(0.74)	(1.07)
	(CV)	(0.00)	(0.03)	(0.12)	(0.11)	(0.09)	(0.12)	(0.16)
	Med	1.00	1.73	2.89	4.47	5.46	6.30	7.10
HPBP	Avg	1.00	1.73	2.78	4.42	5.47	5.89	7.10
-PH	(SD)	(0.00)	(0.07)	(0.37)	(0.59)	(0.56)	(1.11)	(1.34)
	(CV)	(0.00)	(0.04)	(0.13)	(0.13)	(0.10)	(0.19)	(0.19)

Table B.22: Average overall parallel speedup (Gagné)

#threads		1	2	4	8	16	32	64
	Med	1.00	1.64	2.80	4.41	5.86	7.74	8.93
SPBP	Avg	1.00	1.64	2.82	4.62	6.10	7.89	8.87
-PH	(SD)	(0.00)	(0.12)	(0.30)	(0.73)	(1.14)	(1.79)	(2.56)
	(CV)	(0.00)	(0.07)	(0.11)	(0.16)	(0.19)	(0.23)	(0.29)
	Med	1.00	1.77	2.93	4.75	6.13	7.54	9.16
HPBP	Avg	1.00	1.78	2.97	4.83	6.46	8.63	9.88
-PH	(SD)	(0.00)	(0.15)	(0.58)	(0.99)	(1.35)	(3.81)	(3.46)
	(CV)	(0.00)	(0.09)	(0.19)	(0.20)	(0.21)	(0.44)	(0.35)

Table B.23: Average overall parallel speedup (Cicirello)

#threads		1	2	4	8	16	32	64
	Med	1.00	1.70	2.86	4.37	5.90	7.19	7.81
SPBP	Avg	1.00	1.68	2.86	4.36	5.73	7.04	9.78
-PH	(SD)	(0.00)	(0.12)	(0.37)	(0.86)	(1.28)	(1.85)	(21.11)
	(CV)	(0.00)	(0.07)	(0.13)	(0.20)	(0.22)	(0.26)	(2.16)
	Med	1.00	1.74	2.89	4.37	5.89	7.51	8.29
HPBP	Avg	1.00	1.71	2.86	4.46	11.44	9.17	12.57
-PH	(SD)	(0.00)	(0.19)	(0.58)	(1.92)	(41.60)	(14.69)	(41.33)
	(CV)	(0.00)	(0.11)	(0.20)	(0.43)	(3.64)	(1.60)	(3.29)

Table B.24: Average parallel DP speedup (Rubin)

#threads		1	2	4	8	16	32	64
	Med	1.00	1.67	2.95	6.35	10.38	15.72	22.89
SPBP	Avg	1.00	1.67	2.96	6.07	10.15	15.33	22.54
-PH	(SD)	(0.00)	(0.04)	(0.34)	(0.60)	(1.01)	(2.45)	(5.35)
	(CV)	(0.00)	(0.02)	(0.12)	(0.10)	(0.10)	(0.16)	(0.24)
	Med	1.00	1.86	3.51	6.72	11.22	16.92	25.05
HPBP	Avg	1.00	1.85	3.39	6.57	10.96	15.45	25.93
-PH	(SD)	(0.00)	(0.06)	(0.41)	(0.62)	(0.97)	(4.01)	(5.55)
	(CV)	(0.00)	(0.03)	(0.12)	(0.09)	(0.09)	(0.26)	(0.21)

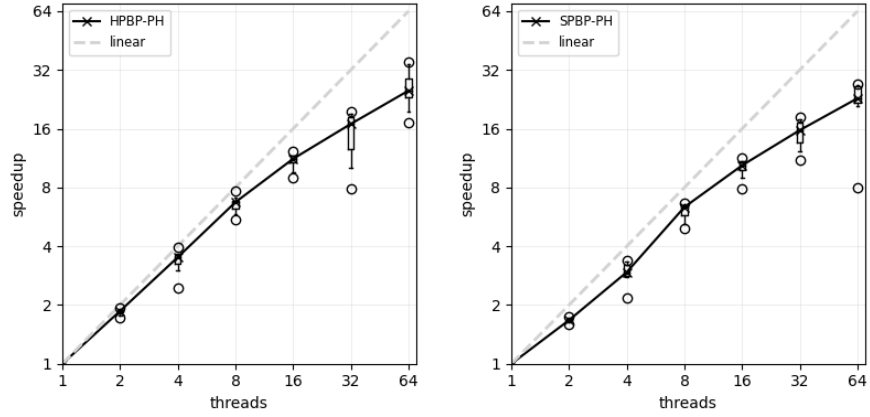
Table B.25: Average parallel DP speedup (Gagné)

#threads		1	2	4	8	16	32	64
	Med	1.00	1.74	3.35	6.41	10.60	18.74	32.63
SPBP	Avg	1.00	1.73	3.30	6.36	10.78	18.51	29.39
-PH	(SD)	(0.00)	(0.10)	(0.27)	(0.49)	(0.86)	(2.31)	(6.36)
	(CV)	(0.00)	(0.06)	(0.08)	(0.08)	(0.08)	(0.12)	(0.22)
	Med	1.00	1.91	3.61	7.05	12.21	21.35	35.44
HPBP	Avg	1.00	1.90	3.54	6.89	12.11	21.52	34.42
-PH	(SD)	(0.00)	(0.10)	(0.61)	(0.79)	(1.12)	(3.95)	(6.78)
	(CV)	(0.00)	(0.05)	(0.17)	(0.12)	(0.09)	(0.18)	(0.20)

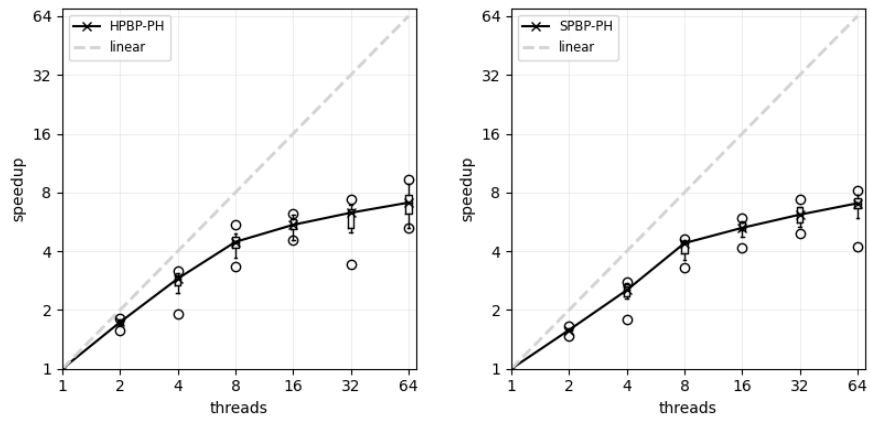
Table B.26: Average parallel DP speedup (Cicirello)

#threads		1	2	4	8	16	32	64
	Med	1.00	1.82	3.54	6.70	11.76	19.84	28.46
SPBP	Avg	1.00	1.83	3.56	6.69	11.75	19.93	35.52
-PH	(SD)	(0.00)	(0.11)	(0.29)	(0.68)	(1.14)	(3.66)	(76.87)
	(CV)	(0.00)	(0.06)	(0.08)	(0.10)	(0.10)	(0.18)	(2.16)
	Med	1.00	1.90	3.61	6.79	12.28	20.44	30.40
HPBP	Avg	1.00	1.87	3.62	7.10	27.95	27.25	50.93
-PH	(SD)	(0.00)	(0.19)	(0.67)	(3.14)	(116.36)	(45.94)	(190.33)
	(CV)	(0.00)	(0.10)	(0.19)	(0.44)	(4.16)	(1.69)	(3.74)

Figure B.2: Box plots for parallel Speedups (Rubin)

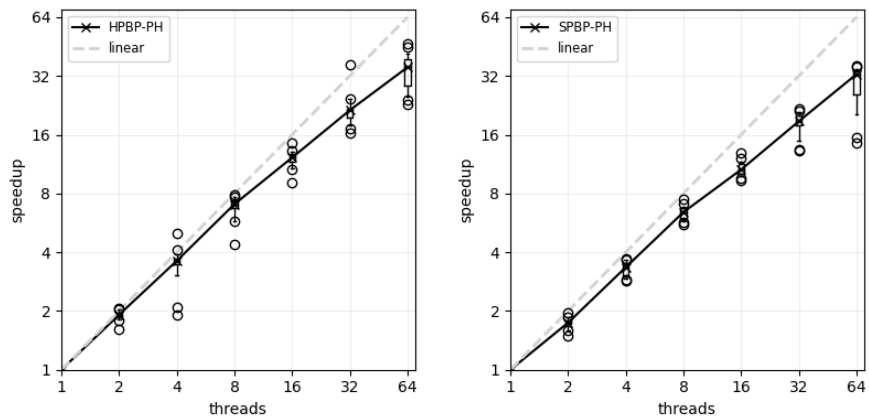


(a) Parallel Speedup DP (Rubin, HPBP-PH) (b) Parallel Speedup DP (Rubin, SPBP-PH)

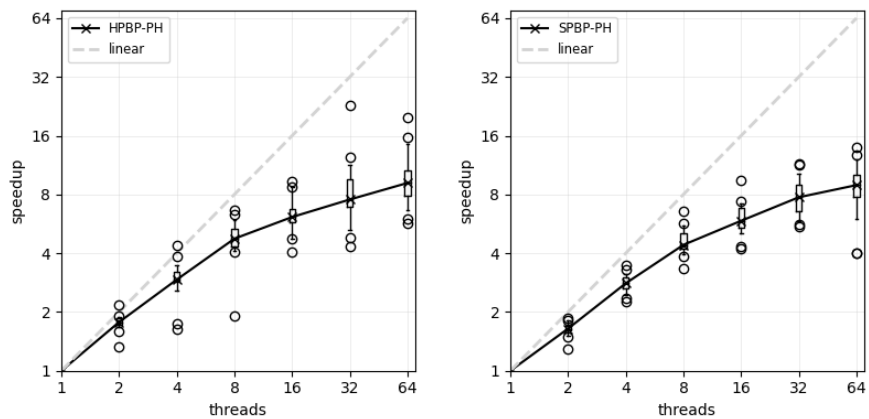


(c) Overall parallel Speedup (Rubin, HPBP-PH) (d) Overall parallel Speedup (Rubin, SPBP-PH)

Figure B.3: Box plots for parallel Speedups (Gagné)

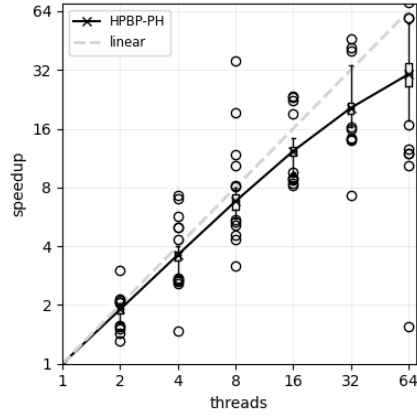


(a) Parallel Speedup DP (Gagné, HPBP-PH) (b) Parallel Speedup DP (Gagné, SPBP-PH)

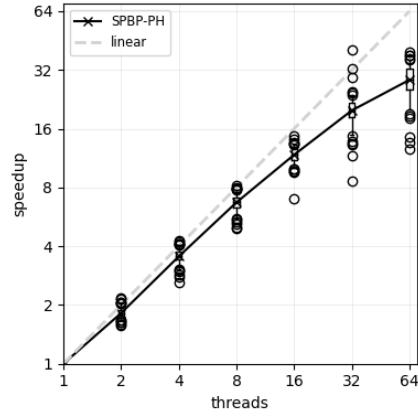


(c) Overall parallel Speedup (Gagné, HPBP-PH) (d) Overall parallel Speedup (Gagné, SPBP-PH)

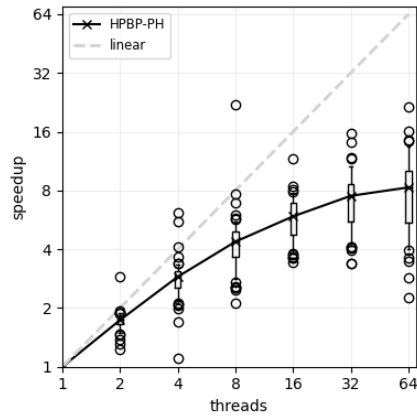
Figure B.4: Box plots for parallel Speedups (Cicirello)



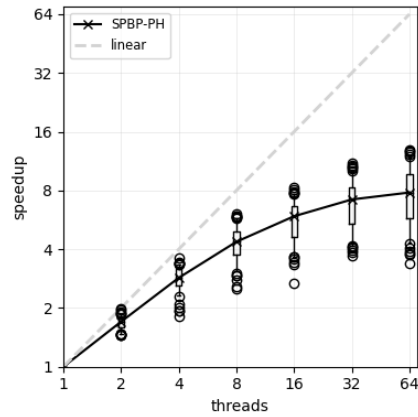
(a) Parallel Speedup DP (Cicirello, HPBP-PH)



(b) Parallel Speedup DP (Cicirello, SPBP-PH)

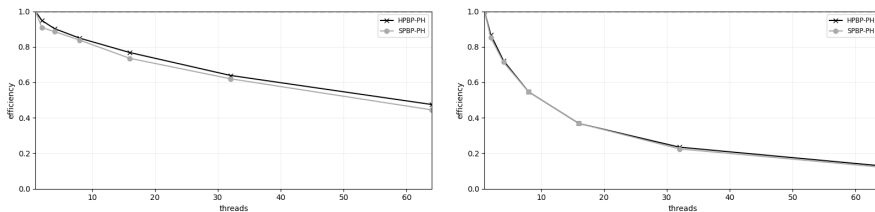


(c) Overall parallel Speedup (Cicirello, HPBP-PH)



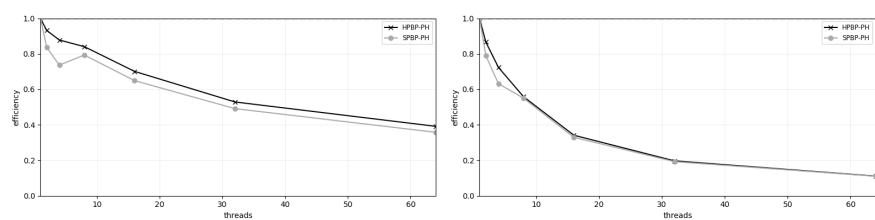
(d) Overall parallel Speedup (Cicirello, SPBP-PH)

Figure B.5: Average parallel efficiencies



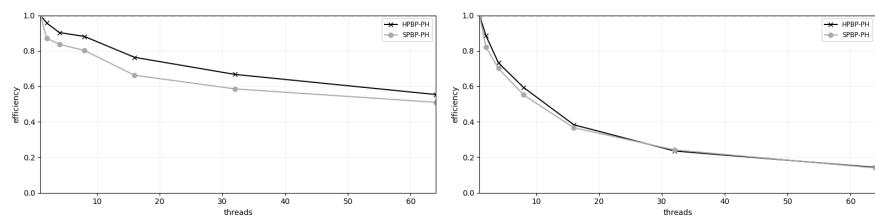
(a) Parallel efficiency DP (Cicirello)

(b) Overall parallel efficiency (Cicirello)



(c) Parallel efficiency DP (Rubin)

(d) Overall parallel efficiency (Rubin)



(e) Parallel efficiency DP (Gagné)

(f) Overall parallel efficiency (Gagné)

Table B.27: Ratio of average number of times entering the waiting condition in SPBP-PH and HPBP-PH for 2 to 64 threads (no waiting in single-core execution)

$\frac{\emptyset \text{ \#wait SPBP-PH}}{\emptyset \text{ \#wait HPBP-PH}}$	2	4	8	16	32	64	\emptyset
Rubin	126	317	86	1	5	0	76
Gagné	541	219	92	137	287	2	182
Cicirello	47	24	588	5	28	54	107

Appendix C. Recomputed Results of Tanaka and Araki’s Experiments

We re-enacted all the experiments in Tanaka and Araki (2013) on the same hardware as the experiments in section 6. In their study, the authors used different settings for the algorithm depending on the instances to which it was applied to. The first setting was applied to instances Nos. 41–120 of the Cicirello set and the second one to the Rubin, Gagné and the rest of the Cicirello set, using different amounts of memory to store their network structure for what the authors call the “second stage” (denoted by “512MB”, “2GB” and “20GB” in the tables below). For the specifics, see Tanaka and Araki (2013).

To replicate the computational environment as closely as possible, we conducted the experiments with the implementation unchanged, applied preprocessing to instances Nos. 1–40 of the Cicirello set (removing zero-weight jobs under certain conditions), and used the same settings reported by the authors.

The two main differences from the originally reported data are hardware and the fact that the cluster didn’t allow us to run programs for more than 21 days. Since the instances in question (except for Cicirello No. 18 with 14 days) took the authors more than 30 days to solve (using “20GB” memory), we ran them for 14 days. The timelimit affects the results for the following instances (with the running times from Tanaka and Araki (2013) in parentheses):

- 751 (34 days), 851 (> 30 days; unsolved) and 855 (> 30 days; unsolved) from the Gagné set; and
- Nos. 18 (2 weeks) and 24 (30 days) from the Cicirello set.

Note, that we set a hard time limit of 14 days, so instance No. 18 may have taken only another hour to solve and prove optimality.

The detailed running times from the re-run experiments are given in Tables C.28–C.32. The first three columns in Tables C.28 and C.29 give the name of the instance from the Rubin and Gagné set, the number of jobs in the instance, and the optimal or best known objective value for the instance. The following

1105 three columns show the running times for the instance from the re-enacted experiments on the cluster, using different amounts of memory for the “second stage” of the algorithm as mentioned above. In case the re-run experiment did not terminate within the timelimit, the second column also shows the original objective value (from the original experiments in their study) in parentheses 1110 where applicable. Also, provably optimal objective values are highlighted as bold numbers.

Table C.30 gives the same information for Cicirello instances Nos. 1–40 as the previous two tables, only omitting the column with the number of jobs, since all instances from this set contain the same number of jobs (60).

1115 Finally, Tables C.31 and C.32 address instances Nos. 41–80 and Nos. 81–120 of the Cicirello set, respectively. Both tables contain columns for the name of the instance, the optimal value, and a single column for the running time, since no increased amount of memory was required for these instances.

Table C.28: Runningtimes Tanaka and Araki on Rubin instances

Name	jobs	Opt	512MB	2GB	20GB
401	15	90	0.34	0.34	0.31
402	15	0	0.01	0.00	0.01
403	15	3,418	0.56	0.54	0.53
404	15	1,067	0.46	0.48	0.44
405	15	0	0.00	0.00	0.00
406	15	0	0.00	0.01	0.01
407	15	1,861	0.48	0.48	0.48
408	15	5,660	0.87	0.85	0.85
501	25	261	2.18	2.14	2.09
502	25	0	0.01	0.01	0.01
503	25	3,497	3.10	3.08	3.04
504	25	0	0.03	0.03	0.03
505	25	0	0.02	0.02	0.02

Table C.28: cont. - Runningtimes Tanaka and Araki on Rubin instances

Name	jobs	Opt	512MB	2GB	20GB
506	25	0	0.02	0.02	0.02
507	25	7,225	4.27	4.28	4.22
508	25	1,915	3.86	3.84	3.88
601	35	12	7.48	7.36	7.31
602	35	0	0.05	0.05	0.05
603	35	17,587	18.37	20.33	19.73
604	35	19,092	25.73	25.74	27.32
605	35	228	13.84	13.57	14.90
606	35	0	0.05	0.05	0.05
607	35	12,969	18.32	17.95	17.60
608	35	4,732	33.33	33.09	32.92
701	45	97	49.55	60.15	55.34
702	45	0	0.10	0.09	0.09
703	45	26,506	53.58	53.71	52.36
704	45	15,206	72.01	72.31	71.40
705	45	200	866.47	599.48	862.37
706	45	0	0.11	0.10	0.11
707	45	23,789	53.35	53.63	52.32
708	45	22,807	100.73	100.58	99.77

Table C.29: Runningtimes Tanaka and Araki on Gagné instances

Name	jobs	Opt	512MB	2GB	20GB
551	55	183	116.23	150.63	282.44
552	55	0	0.18	0.17	0.17

Table C.29: cont. - Runningtimes Tanaka and Araki on Gagné instances

Name	jobs	Opt	512MB	2GB	20GB
553	55	40,498	116.16	114.99	112.04
554	55	14,653	277.87	281.62	278.99
555	55	0	0.30	0.27	0.28
556	55	0	0.19	0.18	0.18
557	55	35,813	146.85	146.07	140.91
558	55	19,871	229.22	230.05	228.58
651	65	247	26,575.61	4,611.69	2,298.27
652	65	0	0.25	0.25	0.26
653	65	57,500	343.74	304.17	262.80
654	65	34,301	390.76	381.56	389.30
655	65	0	355.72	430.00	1,816.32
656	65	0	0.30	0.29	0.30
657	65	54,895	263.95	261.66	253.71
658	65	27,114	471.35	469.43	466.63
751	75	225 (225)	—	—	—
752	75	0	0.45	0.44	0.45
753	75	77,544	839.59	598.87	565.95
754	75	35,200	907.44	888.11	887.85
755	75	0	0.42	0.41	0.41
756	75	0	0.54	0.54	0.52
757	75	59,635	1,197.81	844.79	741.33
758	75	38,339	998.00	991.69	987.59
851	85	363 (360)	—	—	—
852	85	0	0.62	0.63	0.61
853	85	97,497	2,519.42	2,219.22	6,188.71
854	85	79,042	1,451.49	1,222.28	1,177.04

Table C.29: cont. - Runningtimes Tanaka and Araki on Gagné instances

Name	jobs	Opt	512MB	2GB	20GB
855	85	260 (260)	—	—	—
856	85	0	0.63	0.64	0.63
857	85	87,011	6,130.77	4,594.41	4,897.54
858	85	74,739	1,867.57	1,850.23	1,817.68

Table C.30: Runningtimes Tanaka and Araki on Cicirello instances 1–40

Name	Opt	512MB	2GB	20GB
1	453	120.06	101.78	93.58
2	4,794	4,841.34	1,499.98	1,474.67
3	1,390	2,173.60	1,000.58	589.07
4	5,866	206.60	169.31	129.80
5	4,054	3,563.57	2,350.37	2,768.09
6	6,592	166.86	243.35	171.90
7	3,267	8,640.60	3,522.47	6,680.84
8	100	122.22	109.55	105.54
9	5,660	139.24	130.39	124.33
10	1,740	16,455.70	9,783.92	8,883.74
11	2,785	—	120,893.73	42,921.26
12	0	0.42	0.43	0.41
13	3,904	15,674.95	8,722.61	9,907.07
14	2,075	20,705.47	8,764.03	3,555.71
15	724	2,802.18	841.15	496.78
16	3,285	840.15	802.80	504.49

Table C.30: cont. - Runningtimes Tanaka and Araki on Cicirello instances 1–40

Name	Opt	512MB	2GB	20GB
17	0	1,189.80	2,767.39	5,696.84
18	773 (767)	—	—	—
19	0	3.41	3.47	3.42
20	1,757	762.29	334.69	316.22
21	0	0.22	0.20	0.21
22	0	0.23	0.25	0.23
23	0	0.17	0.18	0.17
24	761 (761)	—	—	—
25	0	0.32	0.33	0.31
26	0	0.23	0.25	0.24
27	0	0.35	0.36	0.34
28	0	0.40	0.40	0.46
29	0	0.24	0.23	0.23
30	0	8.00	8.14	7.85
31	0	0.47	0.48	0.45
32	0	0.51	0.51	0.48
33	0	0.50	0.51	0.48
34	0	0.40	0.43	0.44
35	0	0.47	0.47	0.50
36	0	0.44	0.46	0.48
37	0	2,890.32	595.89	2,542.07
38	0	0.38	0.38	0.40
39	0	0.48	0.48	0.53
40	0	0.41	0.40	0.43

Table C.31: Runningtimes Tanaka and Araki on Cicirello instances 41-80

Name	Opt	Time	Name	Opt	Time
41	69,102	32.06	61	75,916	41.42
42	57,487	45.69	62	44,769	37.31
43	145,310	80.55	63	75,317	33.11
44	35,166	58.20	64	92,572	39.94
45	58,935	80.98	65	126,696	41.08
46	34,764	54.08	66	59,685	17.81
47	72,853	61.35	67	29,390	26.93
48	64,612	105.70	68	22,120	25.99
49	77,449	59.38	69	71,118	45.03
50	31,092	59.55	70	75,102	36.69
51	49,208	91.84	71	145,007	96.11
52	93,045	117.65	72	43,286	68.28
53	84,841	116.94	73	28,785	91.49
54	118,809	104.63	74	29,777	73.14
55	64,315	108.07	75	21,602	77.57
56	74,889	122.09	76	53,555	75.75
57	63,514	97.11	77	31,817	84.48
58	45,322	135.31	78	19,462	79.56
59	50,999	92.49	79	114,999	70.12
60	60,765	137.23	80	18,157	76.73

Table C.32: Runningtimes Tanaka and Araki on Cicirello instances 81–120

Name	Opt	Time	Name	Opt	Time
81	383,485	30.13	101	352,990	58.17
82	409,479	66.12	102	492,572	55.05
83	458,752	42.51	103	378,602	46.02
84	329,670	46.59	104	357,963	56.07
85	554,766	74.38	105	450,806	41.61
86	361,417	68.14	106	454,379	56.18
87	398,551	47.96	107	352,766	41.31
88	433,186	60.54	108	460,793	39.30
89	410,092	46.85	109	413,004	54.97
90	401,653	56.97	110	418,769	60.62
91	339,933	68.98	111	342,752	101.73
92	361,152	103.36	112	367,110	114.06
93	403,423	125.52	113	259,649	107.98
94	332,941	98.30	114	463,474	107.27
95	516,926	101.20	115	456,890	115.58
96	455,448	79.50	116	530,601	107.89
97	407,590	92.92	117	502,840	117.72
98	520,582	90.23	118	349,749	62.23
99	363,518	106.21	119	573,046	121.78
100	431,736	80.98	120	396,183	84.73