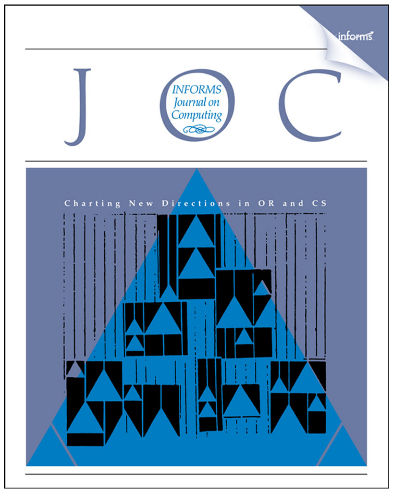

INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Heuristic and Exact Algorithms for the Identical Parallel Machine Scheduling Problem

Mauro Dell'Amico, Manuel Iori, Silvano Martello, Michele Monaci,

INFORMS Journal on Computing
Vol. 20, No. 3, Summer 2008, pp. 333–344
ISSN 1091-9856 | EISSN 1526-5528 | 08 | 2003 | 0333

DOI 10.1287/ijoc.1070.0246
© 2008 INFORMS

Heuristic and Exact Algorithms for the Identical Parallel Machine Scheduling Problem

Mauro Dell'Amico, Manuel Iori
Dipartimento di Scienze e Metodi dell'Ingegneria (DISMI), Università di Modena e Reggio Emilia, 42100 Reggio Emilia, Italy {dellamico@unimore.it, manuel.iori@unimore.it}

Silvano Martello
Dipartimento di Eleltronica, Informatica e Sistemistica (DEIS), Università di Bologna, 40136 Bologna, Italy, silvano.martello@unibo.it

Michele Monaci
Dipartimento di Ingegneria dell'Informazione (DEI), Università di Padova, 40136 Padova, Italy, monaci@dei.unipd.it

Given a set of jobs with associated processing times, and a set of identical machines, each of which can process at most one job at a time, the parallel machine scheduling problem is to assign each job to exactly one machine so as to minimize the maximum completion time of a job. The problem is strongly NP-hard and has been intensively studied since the 1960s. We present a metaheuristic and an exact algorithm and analyze their average behavior on a large set of test instances from the literature. The metaheuristic algorithm, which is based on a scatter search paradigm, computationally proves to be highly effective and capable of solving to optimality a very high percentage of the publicly available test instances. The exact algorithm, which is based on a specialized binary search and a branch-and-price scheme, was able to quickly solve to optimality all remaining instances.

Key words: scheduling; identical parallel machines; bin packing; scatter search; column generation
History: Accepted by Michel Gendreau, Area Editor for Heuristic Search and Learning; received August 2006; revised February 2007; accepted September 2007. Published online in *Articles in Advance* January 25, 2008.

1. Introduction

Given a set $\{1, \ldots, n\}$ of n *jobs*, each having an associated processing time p_j $(j = 1, \ldots, n)$, and a set $\{1, \ldots, m\}$ of m parallel identical *machines*, each of which can process at most one job at a time, the (identical) *parallel machine scheduling problem* is to assign each job to exactly one machine so as to minimize the maximum completion time of a job (*makespan*).

The problem, denoted as $P\|C_{\max}$ in the three-field classification by Graham et al. (1979), is NP-hard in the strong sense (see Garey and Johnson 1979). This is one of the most intensively studied problems in combinatorial optimization because it has considerable theoretical interest and it arises (either directly or as a subproblem) in many real-world applications. Most of the literature concerns the approximate solution of the problem. Among the recent contributions, the heuristic algorithms by França et al. (1994), Frangioni et al. (2004), and Alvim and Ribeiro (2004) are regarded as the most effective ones. The literature on exact solution methods is considerably smaller. We mention here the branch-and-bound algorithm by Dell'Amico and Martello (1995, 2005) and the cutting plane algorithm by Mokotoff (2004). The reader is referred to Brucker (2001) and to Leung (2004) for recent comprehensive volumes on scheduling problems, and to Hoogeveen et al. (1997) for an annotated bibliography.

Problem $P\|C_{\max}$ can be seen as the "dual" of another well-known combinatorial optimization problem. In the *bin packing problem* (BPP), one is given n items, each having an associated size p_j $(j = 1, \ldots, n)$, and an unlimited number of identical bins of capacity c: The objective is to assign each item to one bin without exceeding its capacity so that the number of bins used is minimized. Hence, BPP can be seen as a parallel machine scheduling problem in which one wants to minimize the number of machines needed not to exceed a prefixed makespan c. Making use of this duality, Coffman et al. (1978) proposed an approximation algorithm (known as *multifit*) that solves $P\|C_{\max}$ by finding, through binary search, the smallest value c such that the solution found for BPP by the well-known *first-fit decreasing* (*FFD*) approximation algorithm has a value not greater than m. Hochbaum and Shmoys (1987) obtained a polynomial-time approximation scheme for $P\|C_{\max}$ by replacing *FFD* with a dual approximation algorithm.

In this paper, we make use of the above relationship to exactly solve $P\|C_{\max}$. Our algorithm consists of two phases. In the first phase, lower and upper bounds from the literature are computed (§2). The heuristic solution found is then improved through local search (§2.3) and an effective scatter search metaheuristic (§3), which computationally turns out to be superior to the other metaheuristic algorithms from the literature. If the best lower and upper bounds, say L and U, coincide, an optimal solution is found. Otherwise, the second phase (§4) consists of a binary search that checks whether there exists a solution with makespan at most $c = \lfloor (L+U)/2 \rfloor$, and so on. The latter problem is the decision version of a BPP, which can be formulated as an integer linear program minimizing the number of items (jobs) that cannot be packed into m bins of capacity c (§5). The LP relaxation of such integer linear program (ILP) model is solved by column generation. If the optimal objective value is positive, then L is replaced by $c+1$. Otherwise, if the solution is integral, then U is replaced by c. However, if the objective value is zero and the solution has fractional components, then a branch-and-price method is applied to find the exact solution of the ILP. In §6, extensive computational comparisons with other algorithms from the literature on all publicly available test instances show that: (i) the scatter search algorithm is the most effective heuristic for the problem; and (ii) the overall exact algorithm outperforms the other optimization algorithms and is able to solve to optimality all instances in the test bed, including a number of previously unsolved instances.

In the following, we will assume, without loss of generality, that $1 < m < n$ and that the processing times are positive integers sorted so that

$$p_1 \geq p_2 \geq \cdots \geq p_n. \tag{1}$$

2. Lower and Upper Bounds

Our algorithm for $P\|C_{\max}$ is initialized by the computation of a lower and an upper bound on the optimal solution value. In §§2.1 and 2.2, we briefly review results from the literature on lower and upper bounds whereas in §2.3, we introduce an improvement procedure based on local search.

2.1. Lower Bounds

Immediate lower bounds for $P\|C_{\max}$ are (see Dell'Amico and Martello 1995):

$$L_0 = \left\lceil \sum_{j=1}^{n} p_j \Big/ m \right\rceil, \tag{2}$$

$$L_1 = \max\{L_0, p_1\}, \tag{3}$$

$$L_2 = \max\{L_1, p_m + p_{m+1}\}. \tag{4}$$

It has been proved in Dell'Amico and Martello (1995) that the worst-case performance ratio of L_2 is equal to 2/3.

Lower bounds (2)–(4) can be computed in $O(n)$ time, with no need of sorting the jobs according to Equation (1). Observe indeed that the m-th and $(m+1)$-st largest processing times can be found in linear time (see Blum et al. 1973, Fischetti and Martello 1988).

More complex but tighter bounds are L_{HS} by Hochbaum and Shmoys (1987), and L_θ and L_3 by Dell'Amico and Martello (1995). The former two can be computed in $O(n \log U)$ time while the latter requires $O(n^2 \log U)$ time, where U denotes an upper bound on the optimal makespan.

In the following, we denote by $\bar{L}$ the best lower bound value among these six bounds.

2.2. Upper Bounds

There is a huge literature on the approximate solution of $P\|C_{\max}$. (The reader is referred to the surveys by Mokotoff 2001 and Chen 2004.) We used the following algorithms for determining initial feasible solutions.

Algorithm *LPT* (*longest processing time*) by Graham (1966, 1969) is one of the most famous approximation algorithms in combinatorial optimization. It orders the jobs by nonincreasing processing time and iteratively assigns the next job to the machine whose current completion time is a minimum. The worst-case performance ratio of *LPT* is equal to $4/3 - 1/3m$.

Algorithm *MS* (*multi-subset*) by Dell'Amico and Martello (1995) operates in two phases. First, an attempt is made to obtain a solution of value $\bar{L}$ (hence, optimal) by determining, for each machine in sequence, a subset of the unassigned jobs whose total processing time is closest to, without exceeding, $\bar{L}$. This NP-hard problem (known as *subset sum*) is solved, for each machine, through the greedy algorithms by Martello and Toth (1984). If all jobs are assigned, the solution obtained is optimal. Otherwise, the second phase completes the solution by assigning the remaining jobs through an *LPT* strategy.

Another two-phase approach was proposed by Mokotoff et al. (2001). Jobs are first assigned through *LPT* until a job is found for which the minimum completion time of a machine exceeds a given threshold value. The remaining jobs are then processed according to the following rule. Assign the next job to the machine for which the resulting completion time is closest to, without exceeding, $\bar{L}$ if such a machine exists; otherwise, assign it to the machine with minimum completion time. The solution is obtained by attempting different threshold values.

Note that both two-phase approaches provide a feasible solution also by using, instead of $\bar{L}$, a different tentative value. Hence, in our implementation, we start by computing the *LPT* upper bound and iteratively execute the two-phase approaches for the tentative values in the range $[\bar{L}, U-1]$ through a binary search.

2.3. Local Search

Each solution generated by the heuristic algorithms of the previous section is improved through the following $k-l$ *swap* procedure, which swaps groups of jobs between two machines. Given two machines, say m_1 and m_2, let us denote their current completion times as $C(m_1)$ and $C(m_2)$. A $k-l$ swap consists in exchanging k jobs currently assigned to m_1 with l jobs currently assigned to m_2, provided this decreases the resulting $\max\{C(m_1), C(m_2)\}$ value. The procedure starts by sorting the machines according to nonincreasing completion time and finding the last machine, say $m_{\bar{L}}$, whose completion time is greater than lower bound $\bar{L}$. The iterative part consists in testing, for $m_1 = 1, 2, \ldots, m_{\bar{L}}$ and $m_2 = m, m-1, \ldots, m_1+1$, all possible $k-l$ swaps, for $k \in \{1, 2, \ldots, \bar{k}\}$ and $l \in \{0, 1, \ldots, \bar{l}\}$, where $\bar{k}$ and $\bar{l}$ are prefixed parameters. This procedure generalizes the classical local search improvements known as *move* (for $\bar{k} = 1$ and $\bar{l} = 0$) and *exchange* (for $\bar{k} = \bar{l} = 1$) (see, e.g., Dell'Amico et al. 2004). On the basis of extensive computational experiments, we adopted a "first-improvement" policy: As soon as a feasible $k-l$ swap is found, it is performed, and the procedure is restarted from the new solution.

3. Scatter Search

A scatter search algorithm (see Martí et al. 2006 for a recent survey) starts by generating a set of feasible solutions, called *reference set* ($\mathcal{RS}$), and iteratively creates new solutions by combining subsets of $\mathcal{RS}$. The new solutions are possibly used to periodically update the reference set, and the final outcome is the best solution it contains. The value is not the only criterion for a solution to enter the reference set, because one also wants it to contain solutions differing in structure from each other. We adopted a classical scatter search template (see Glover et al. 2004), consisting of three steps:

1. generate a pool S of solutions,
2. create the initial reference set $\mathcal{RS}$ by selecting the Q solutions with the highest quality in the pool, and adding the D solutions with the highest diversity from them, and
3. iteratively perform the following steps, until a stopping criterion is met:
   (i) generate subsets of $\mathcal{RS}$,
   (ii) for each subset, combine the solutions it contains to obtain new solutions, and
   (iii) improve each new solution through local search and update the reference set.

These three steps were implemented as follows.

Step 1. The pool is initialized with the $|S|/2$ best solutions among all those produced by the constructive heuristics and improved through the subsequent local search of §2.3. Set S is then completed with new solutions obtained by assigning each job to a randomly chosen machine and by improving through $k-l$ swaps.

Step 2. The initial reference set is obtained as follows. We measure the *quality* of a solution s from its makespan $z(s)$, so we start by selecting the Q solutions with smallest $z(s)$ value from the pool. To define the diversity of a solution, let $\mu_i(s)$ denote the machine job i is assigned to in solution s, and define, for two jobs i and j (with $j > i$) and two solutions $s \notin \mathcal{RS}$ and $t \in \mathcal{RS}$, the binary *diversity function*

$$\delta_{ij}(s,t) = \begin{cases} 1 & \text{if } (\mu_i(s) = \mu_j(s) \text{ and } \mu_i(t) \neq \mu_j(t)) \text{ or} \\ & (\mu_i(s) \neq \mu_j(s) \text{ and } \mu_i(t) = \mu_j(t)); \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

which takes the value 1 iff the two jobs are processed together in one solution and separately in the other. Since jobs with "large" processing time are particularly critical, we define the *diversity* $d(s)$ of a solution s with respect to $\mathcal{RS}$, by only considering, for a given parameter $\tilde{n}$, the diversity function (5) of the first $\tilde{n}$ jobs:

$$d(s) = \min_{t \in \mathcal{RS}} \left\{ \sum_{i=1}^{\tilde{n}-1} \sum_{j=i+1}^{\tilde{n}} \delta_{ij}(s,t) \right\}. \tag{6}$$

(Observe that if $\mathcal{RS}$ contains a solution equivalent to s, i.e., that can be obtained from s by permuting the machines, Equation (6) gives $d(s) = 0$.) In the reference set, the Q solutions with highest quality are sorted by increasing $z(s)$ value and the D additional solutions with highest diversity are sorted by decreasing $d(s)$ value. These sorted solutions are denoted in the following as $s_1, s_2, \ldots, s_Q, s_{Q+1}, \ldots, s_{Q+D}$.

Step 3. We generate all 2-element subsets and a number of 3-, 4-, and 5-element subsets. To combine these solutions, we define, for each subset T, an $\tilde{n} \times \tilde{n}$ matrix φ with

$$\varphi_{ij} = \sum_{s \in T:\, \mu_i(s) = \mu_j(s)} \frac{\bar{L}}{z(s) - \bar{L}} \qquad i = 1, \ldots, \tilde{n},\ j = i+1, \ldots, \tilde{n}. \tag{7}$$

For each pair of jobs (i,j), function φ_{ij} sums the inverse relative errors of all solutions where i and j are together in a machine. Hence, φ_{ij} has a relatively high value for those job pairs (i,j) that are assigned to the same machine in many good solutions. We then produce a combined solution by iteratively selecting a pair (i,j) with probability proportional to φ_{ij}, and assigning them to the first machine, if any, for which the resulting completion time does not exceed lower bound $\bar{L}$. If no such machine exists, i and j are assigned to the machine with minimum completion time. After each pair assignment, the entries in rows i and j and in columns i and j of φ are set to zero, and the process is iterated until a complete solution s is obtained. Once s has been improved through $k-l$ swaps, it will replace a solution in the reference set if $z(s) < z(s_Q)$ or $d(s) > d(s_{Q+D})$.

4. An Exact Algorithm for P‖C$_{\max}$

In this section, we describe an exact algorithm for P‖C$_{\max}$, based on the dual relation with BPP discussed in §1, which iteratively solves the following recognition version of BPP:

BPP$_m(c)$: *input*: an instance of P‖C$_{\max}$, and a threshold value c;
question: is there a solution of a BPP with n items of size p_j $(j=1,\ldots,n)$ that requires at most m bins of capacity c?

We assume that, when the *answer* is "yes," the certificate is returned.

```
Algorithm DIMM:
begin
  1. let $L := \bar{L}$ be the best initial lower bound (see §2.1);
     let $U$ be the best initial upper bound
        (see §§2.2, 2.3, and 3);
     store in $x$ the solution of value $U$;
     if $L = U$ then stop;
     if check_BPP$_m(L)$ = "yes" then store in $x$ the
        returned certificate and stop
     else $L := L + 1$;
  2. while $L < U$ do
        $c := \lfloor (L+U)/2 \rfloor$;
        if check_BPP$_m(c)$ = "yes" then $U := c$ and store
           in $x$ the returned certificate
        else $L := c + 1$;
     end while
end.
function check_BPP$_m(c)$:
begin
  1. if $L^f > m$ for one of the dual feasible functions
     then return "no";
  2. let $\overline{\mathrm{BPP}}_m(c)$ be an ILP model to minimize the number $z$ of
        items that cannot be packed into $m$ bins of
        capacity $c$ (see §5);
     solve the LP relaxation of $\overline{\mathrm{BPP}}_m(c)$ by column generation
        (see §5.1), and let $\sigma$ and $z$ be the solution and the
        value obtained;
     if $z > 0$ then return "no";
     if $\sigma$ is integer then return "yes" and the certificate $\sigma$;
  3. solve $\overline{\mathrm{BPP}}_m(c)$ through branch and price (see §5.2) and let
        $\sigma$ and $z$ be the solution and the value obtained;
     if $z > 0$ then return "no"
     else return "yes" and the certificate $\sigma$
end.
```

Figure 1 **Exact Algorithm for P‖C$_{\max}$**

The overall algorithm *DIMM* is outlined in Figure 1. After an attempt to directly solve the problem through the initial lower and upper bounds $\bar{L}$ and U, introduced in the previous sections, we perform a specialized binary search for BPP$_m(c)$ over the c values in the interval $[\bar{L}, U-1]$. At each iteration, the solution of BPP$_m(c)$ is determined by function *check_BPP$_m(c)$* through a sequence of three attempts (described in detail in the next sections):

(i) a check on a sufficient condition,
(ii) a test on the LP relaxation of a specialized ILP model, and
(iii) a branch-and-price algorithm for solving the ILP model.

Note that *DIMM* explicitly handles, at Step 1, the case where the optimal solution has value $\bar{L}$, which most frequently occurs in practice (see §6.3).

4.1. A Sufficient Condition for BPP$_m(c)$

Since BPP$_m(c)$ is the recognition version of BPP, it is natural to obtain sufficient conditions for a negative answer from lower bounds for BPP. Let $L(I)$ be a lower bound value for an instance I of BPP: if $L(I) > m$, then we know that the answer to BPP$_m(c)$ is "no."

Computational experiments with various lower bounds for BPP proposed in the literature showed that good results are obtained, in short computing times, with dual feasible functions.

Dual feasible functions (see Johnson et al. 1974, who first used them for bin packing problems) are functions f: $[0,1] \to [0,1]$ such that, for any finite set F of nonnegative real numbers, the following relation holds:

$$\sum_{i\in F} w_i \le 1 \;\Rightarrow\; \sum_{i\in F} f(w_i) \le 1.$$

In other words, if we consider a bin packing problem with capacity normalized to one, for any subset of items that can be allocated to a bin using the given sizes w_i, the same holds if the transformed sizes $f(w_i)$ are used.

Given a BPP instance, let $\tilde{p}_j = p_j/c$ $(j=1,\ldots,n)$ be the normalized item sizes. Fekete and Schepers (2001) observed that, for any dual feasible function f, the quantity

$$L^f = \sum_{j=1}^{n} f(\tilde{p}_j) \tag{8}$$

is a valid lower bound on the optimal BPP solution. Hence, if $L^f > m$, then we know that the answer to BPP$_m(c)$ is "no."

We implemented Step 1 of function *check_BPP$_m(c)$* (see Figure 1) by testing the three dual feasible functions proposed by Fekete and Schepers (2001). These functions have the advantage of including, as special cases, some effective lower bounds from the BPP literature such as the continuous lower bound and bound L_2 by Martello and Toth (1990a, b).

5. An ILP Model for $\mathrm{BPP}_m(c)$

To perform the binary search of algorithm *DIMM* (see Figure 1), we need a method to exactly solve $\mathrm{BPP}_m(c)$ when the check with the three dual feasible functions fails. This is obtained by solving the optimization version of $\mathrm{BPP}_m(c)$, denoted as $\overline{\mathrm{BPP}}_m(c)$, that arises when one is required to minimize the number of items that cannot be packed into m bins of capacity c. The following model is an adaptation to $\mathrm{BPP}_m(c)$ of the famous ILP formulation proposed by Gilmore and Gomory (1961, 1963) for the cutting stock problem. Let

$$\mathcal{P}(c) = \left\{ P \subseteq \{1, \ldots, n\}\colon \sum_{j \in P} p_j \leq c \right\} \tag{9}$$

denote the family of item sets (*patterns*) that can be assigned to a bin without exceeding its capacity. Moreover, for each item j, let

$$\mathcal{P}_j(c) = \{P \in \mathcal{P}(c)\colon j \in P\} \tag{10}$$

denote the family of those patterns that contain item j.

Let us introduce binary variables

$$x_P = \begin{cases} 1 & \text{if pattern } P \text{ is assigned to a bin;} \\ 0 & \text{otherwise} \end{cases} \qquad P \in \mathcal{P}(c) \tag{11}$$

and

$$y_j = \begin{cases} 1 & \text{if item } j \text{ is not assigned to any bin;} \\ 0 & \text{otherwise.} \end{cases} \qquad j \in \{1, \ldots, n\}. \tag{12}$$

We obtain the ILP model

$$\overline{\mathrm{BPP}}_m(c)\colon \quad \min \sum_{j=1}^{n} y_j \tag{13}$$

$$\sum_{P \in \mathcal{P}(c)} x_P \leq m \tag{14}$$

$$y_j + \sum_{P \in \mathcal{P}_j(c)} x_P \geq 1 \quad j \in \{1, \ldots, n\} \tag{15}$$

$$x_P \in \{0, 1\} \quad P \in \mathcal{P}(c) \tag{16}$$

$$y_j \in \{0, 1\} \quad j \in \{1, \ldots, n\}. \tag{17}$$

Objective function (13) minimizes the number of unassigned items. Constraints (14) impose that at most m bins are used. Constraints (15) impose that, for each item j, we have $y_j = 1$ if j is not assigned. Note that our description of $\overline{\mathrm{BPP}}_m(c)$ would require the "=" sign in constraints (15). However, given any solution with an item assigned to more than one bin, we can remove the item from all such bins but one (arbitrarily chosen), thus obtaining a nonworse solution that satisfies constraints (15) with equality. The answer to $\mathrm{BPP}_m(c)$ is "yes" if and only if the optimal solution to problem (13)–(17) has value zero.

The exponential number of x variables in problem (13)–(17) makes it impossible to directly handle the model when the number of items is large, because the explicit enumeration of all feasible patterns can be computationally too expensive and can lead to very large size problems for which even the solution of the associated LP relaxation is very hard. In their seminal papers, Gilmore and Gomory (1961, 1963) introduced the column generation technique for effectively handling linear problems with a huge number of variables.

In the next section, we describe a column generation approach for solving the LP relaxation of problem (13)–(17), as required by Step 2 of function *check_BPP*$_m(c)$ (see Figure 1). In §5.2, we present a branch-and-price algorithm for the exact solution of problem (13)–(17).

5.1. Column Generation

We obtain the LP relaxation of $\overline{\mathrm{BPP}}_m(c)$ by disregarding the integrality requirements and by dropping the constraints $y_j \leq 1$ and $x_P \leq 1$, which are redundant. Indeed, given a feasible solution with a $y_j > 1$ (resp. an $x_P > 1$), by setting $y_j = 1$ (resp. $x_P = 1$) we obtain a better (resp. equivalent) feasible solution. The resulting linear program is defined by objective function (13), constraints (14)–(15), and

$$x_P \geq 0 \quad P \in \mathcal{P}(c) \tag{18}$$

$$y_j \geq 0 \quad j \in \{1, \ldots, n\}. \tag{19}$$

By multiplying constraint (14) by -1, and associating dual variables α to constraint (14) and β_j $(j = 1, \ldots, n)$ to constraints (15), we obtain the dual problem

$$\max -m\alpha + \sum_{j=1}^{n} \beta_j \tag{20}$$

$$\beta_j \leq 1 \quad j \in \{1, \ldots, n\} \tag{21}$$

$$-\alpha + \sum_{j \in P} \beta_j \leq 0 \quad P \in \mathcal{P}(c) \tag{22}$$

$$\alpha \geq 0 \tag{23}$$

$$\beta_j \geq 0 \quad j \in \{1, \ldots, n\}, \tag{24}$$

where constraints (21) are associated with variables y_j and constraints (22) with variables x_P.

We initially solve to optimality a restricted *master problem* of $\overline{\mathrm{BPP}}_m(c)$ in which only the y variables are kept, and then we iteratively add x variables. At any iteration, if no constraint (22) is violated by the current dual solution, we have an optimal solution to the LP relaxation of $\overline{\text{BPP}}_m(c)$. Otherwise, we add to the master (a subset of) x variables corresponding to violated constraints (22). Each iteration requires solving the *slave problem*: Given the current dual solution (α^*, β^*), find a pattern $P \in \mathcal{P}(c)$, if any, for which

$$\Lambda(P) = -\alpha^* + \sum_{j \in P} \beta_j^* \tag{25}$$

is strictly positive. It is known that the convergence of a column generation algorithm benefits from a slave problem that selects highly violated constraints. To this end, we look for a pattern $P^* \in \mathcal{P}(c)$ that maximizes $\Lambda(P)$ by solving the *0-1 knapsack problem* (KP01) in n items with *profits* β_j^*, *weights* p_j, and *capacity* c:

$$\max \sum_{j=1}^{n} \beta_j^* \xi_j \tag{26}$$

$$\sum_{j=1}^{n} p_j \xi_j \le c \tag{27}$$

$$\xi_j \in \{0, 1\} \quad j \in \{1, \ldots, n\}, \tag{28}$$

and defining $P^* = \{j\colon \xi_j = 1\}$. Note that in Equation (25) we have $\Lambda(P^*) \ge 0$ because the reduced cost of a pattern P is $-\Lambda(P)$ and the patterns of the current LP basis are included in the search. If $\Lambda(P^*) = 0$, we know that the current solution is optimal, and otherwise, x_{P^*} has negative reduced cost and can be added to the restricted master.

Although KP01 is an NP-hard problem, effective codes for its solution can be found in the literature. Our computational experiments for determining the best KP01 code to embed in the algorithm are reported in §6.

At each iteration, before exactly solving problem (26)–(28) through the KP01 algorithm, we heuristically look for patterns $P \in \mathcal{P}(c)$ with $\Lambda(P) > 0$ by iteratively running a variation of the well-known *greedy* algorithm. Items are initially sorted according to nonincreasing value of the ratio β_j^*/p_j. Given a prefixed k $(1 \le k \le n)$, the algorithm considers the items in turn, in cyclic order $k, k+1, \ldots, n, 1, \ldots, k-1$: Each item is selected if its weight does not exceed the current residual capacity, and the pattern corresponding to k consists of the selected items. The algorithm is executed n times, for $k = 1, \ldots, n$, and the patterns with highest $\Lambda(P)$ values are stored (avoiding duplicated patterns) in a queue ordered by nonincreasing $\Lambda(P)$ values. When the heuristic terminates, two cases can occur, according to the value associated with the first pattern, say $\widetilde{P}$, in the queue:

(i) $\Lambda(\widetilde{P}) > 0$: all x_P variables associated with (consecutive) stored patterns having $\Lambda(P) > 0$ are added to the master, and the process is iterated with no need of executing the KP01 algorithm,

(ii) $\Lambda(\widetilde{P}) = 0$: pattern P^* is determined through the KP01 algorithm. If $\Lambda(P^*) > 0$, then x_{P^*} is added to the master and the process is iterated; otherwise, the execution is halted with an optimal LP solution.

This concludes the description of Step 2 of function *check_BPP*$_m(c)$ (Figure 1): If the optimal LP solution has a positive value, the function returns "no." If the solution value is zero and no variable has a fractional value, the function returns "yes" and this solution as a certificate. If instead the solution has value zero but is not integer, we solve $\overline{\text{BPP}}_m(c)$ to optimality through the method described in the next section.

5.2. Branch and Price

To solve $\overline{\text{BPP}}_m(c)$ through model (13)–(17) (see Step 3 of function *check_BPP*$_m(c)$ in Figure 1), we implemented a depth-first branch-and-price algorithm. At each decision node, we solve the LP relaxation of the current problem through the column generation algorithm described in the previous section. Recall that $\overline{\text{BPP}}_m(c)$ minimizes the number of items that cannot be packed into m bins of capacity c. Hence, three cases can occur:

(i) if the solution of the LP relaxation has a positive value, the current node can be immediately killed;

(ii) if the solution value is zero and no variable has a fractional value, the exploration is halted and the function returns "yes" and the current solution as a certificate; and

(iii) if the solution has a value zero but is not integer, we branch on a fractional x_P variable and generate two children nodes by setting it to one and to zero, respectively.

Column generation can suffer from poor convergence when the so-called *tailing off* phenomenon occurs, i.e., when the solution values tend to be very close (but not equal) to the final LP value for a large number of iterations. Hence, in cases (i) and (iii), we do not test the solution value against zero but do check if it is greater than a given value ε. The choice of this value needs a careful evaluation (see §6). High ε values allow one to avoid the computational effort (usually relevant in the last iterations) required to determine the exact LP solution. On the other hand, too high values can prevent the possibility of fathoming useless decision nodes.

In case (iii), our strategy for selecting the branching variable is as follows. Let ϑ $(0 < \vartheta < 1)$ be a prefixed threshold value. We determine the fractional variables x_P^1 and x_P^0 whose value is the closest to one and to zero, respectively. If $x_P^1 \ge \vartheta$, we branch on x_P^1 and continue the search from the child node generated by condition $x_P^1 = 1$. Otherwise, we branch on the variable x_P^a $(a \in \{1, 0\})$, which is the closest to a, and continue the search from the child node generated by condition $x_P^a = a$.

The branching strategy above favors branchings to one. The first reason for this choice is that in this way we increase the probability of quickly obtaining a feasible (integer) solution. The second reason comes from the way the branching affects the structure of the slave problem to be solved at the next node. Observe indeed that when we set $x_P = 1$, all constraints (15) associated with items $j \in P$ become redundant; hence we reduce the size of the slave problem (26)–(28) because the corresponding dual variables β_j^* $(j \in P)$ take the value zero. When instead we set $x_P = 0$, the effect is to prohibit the slave to generate pattern P in the descending nodes. In other words, if X^1 and X^0 denote the sets of variables currently fixed to one and to zero, respectively, the slave problem to be solved is problem (26)–(28) with the additional constraints:

$$\xi_j = 0 \quad j \in P,\, x_P \in X^1, \tag{29}$$

$$\sum_{j\in P} \xi_j \le |P| - 1 \quad x_P \in X^0. \tag{30}$$

For what concerns the solution of the slave problems, we can observe that during the initial series of branchings to one, it just consists of a reduced 0-1 knapsack problem and hence can still be solved through the KP01 algorithm. Once the first branching to zero has been performed, the structure of the slave problem requires the use of a general ILP solver. In our implementation, we adopted Cplex and to obtain a stronger LP model, we lifted constraints (30) as follows. For a given pattern P, let $\bar{p}(P) = \max\{p_j\colon j \in P\}$ and define the total weight of the $|P| - 1$ smallest items of P, $S(P) = \sum_{j\in P} p_j - \bar{p}(P)$. Now let $B(P) = \{j \notin P\colon p_j + S(P) > c\}$ (and observe that, by definition, $p_j > \bar{p}(P)$ for all $j \in B(P)$). It is then clear that constraints (30) can be replaced by the stronger constraints (each definable in linear time):

$$\sum_{j\in P} \xi_j + \sum_{j\in B(P)} \xi_j \le |P| - 1 \quad x_P \in X^0. \tag{31}$$

The execution of the exact algorithm (KP01 or Cplex) is preceded by an heuristic attempt, performed with the greedy algorithm of §5.1 with the obvious modifications needed to take into account constraints (29)–(30).

6. Computational Experiments

The algorithms introduced in the previous sections were coded in C language and experimentally tested on a large set of $P\|C_{\max}$ instances from the literature. The computational experiments were performed on a Pentium IV at 3 GHz running under a Windows OS.

We considered all test problems recently proposed in the literature and publicly available. These problems belong to two groups:

- *uniform* instances, proposed by França et al. (1994), obtained by randomly generating all the processing times from a uniform distribution in a given range $[a, b]$,
- *nonuniform* instances, obtained for a given range $[a, b]$ by randomly generating 98% of the processing times from a uniform distribution in $[0.9(b-a), b]$ and the remaining processing times from a uniform distribution in $[a, 0.2(b-a)]$. This generation was proposed by Frangioni et al. (2004) to produce instances that are particularly difficult for their algorithm.

Each group contains three classes of instances, obtained by generating the processing times in the ranges $[1, 100]$, $[1, 1{,}000]$, and $[1, 10{,}000]$, respectively. Each class consists of 13 pairs (m, n), with $m \in \{5, 10, 25\}$, $n \in \{10, 50, 100, 500, 1{,}000\}$, and $m < n$. For each pair, there are 10 instances so the total number of test instances is 780. All these instances can be downloaded from http://www.inf.puc-rio.br/~alvim/adriana/tese.html or from http://www.or.deis.unibo.it/research.html.

6.1. Scatter Search

In Tables 1 and 2, we compare the metaheuristic of §3 (denoted hereafter as *DIMM-SS*) with the following heuristics and metaheuristics from the literature:

- *FGLM* (França et al. 1994), an heuristic that tries to balance the load between pairs of machines through job exchanges,
- *FNS* (Frangioni et al. 2004), a network flow-based algorithm that explores a very large neighborhood by performing multiple exchanges of jobs among machines (the results refer to variant *1-SPT*, which provides the best solution values), and
- *AR* (Alvim and Ribeiro 2004), a hybrid heuristic based on a binary search in which, at each iteration, a tabu search looks for a feasible solution to a BPP instance.

The experimental tuning of *DIMM-SS* produced the following values for the parameters (see §§2 and 3 for their precise meaning):

- $k-l$ swaps in local search: $\bar{k} = \bar{l} = 2$, i.e., $k \in \{1, 2\}$ and $l \in \{0, 1, 2\}$,
- pool size: $|S| = 40$,
- number of "quality" solutions: $Q = 10$,
- number of "diversified" solutions: $D = 8$, and
- number of jobs with "large" processing time: $\tilde{n} = \min\{n, 4m\}$,

and suggested, as stopping criterion, a time limit equal to 50 seconds for $10 < n \le 50$ and to 120 seconds for $n > 50$, while *DIMM-SS* is not executed at all for $n \le 10$.

Tables 1 and 2 refer to uniform and nonuniform instances, respectively. The results obtained by algorithms *FGLM* and *FNS* are taken from Frangioni et al.

Table 1 Metaheuristic Algorithms: Uniform Instances

Instances			*FGLM*		*AR*			*DIMM-SS*		
Range	*m*	*n*	*%gap*	*sec*	*%gap*	*sec*	*#opt*	*%gap*	*sec*	*#opt*
$[1, 10^2]$	5	10	3.2600	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	50	0.0196	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	100	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	500	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	1,000	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
	10	50	0.2350	0.02	0.0000	0.00	10	0.0000	0.00	10
	10	100	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
	10	500	0.0000	0.03	0.0000	0.00	10	0.0000	0.01	10
	10	1,000	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
	25	50	3.4200	0.02	0.0909	0.01	9	0.0909	3.00	9
	25	100	0.3620	0.02	0.0000	0.00	10	0.0000	0.00	10
	25	500	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
	25	1,000	0.0000	0.02	0.0000	0.00	10	0.0000	0.00	10
Average/Total			0.5613	0.02	0.0070	0.00	129	0.0070	0.23	129
$[1, 10^3]$	5	10	4.0300	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	50	0.0248	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	100	0.0060	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	500	0.0000	0.02	0.0000	0.02	10	0.0000	0.03	10
	5	1,000	0.0000	0.02	0.0000	0.05	10	0.0000	0.11	10
	10	50	0.4430	0.02	0.0077	0.02	8	0.0000	0.01	10
	10	100	0.0343	0.02	0.0000	0.00	10	0.0000	0.00	10
	10	500	0.0000	0.02	0.0000	0.01	10	0.0000	0.02	10
	10	1,000	0.0000	0.03	0.0000	0.04	10	0.0000	0.11	10
	25	50	2.3300	0.02	0.1190	0.02	9	0.1190	3.00	9
	25	100	0.4040	0.03	0.0099	0.04	8	0.0052	20.34	9
	25	500	0.0000	0.03	0.0000	0.00	10	0.0000	0.02	10
	25	1,000	0.0000	0.02	0.0000	0.02	10	0.0000	0.09	10
Average/Total			0.5594	0.02	0.0105	0.02	125	0.0096	1.82	128
$[1, 10^4]$	5	10	0.6780	0.02	0.2040	0.03	8	0.1669	0.00	9
	5	50	0.0544	0.01	0.0000	0.03	10	0.0000	0.00	10
	5	100	0.0078	0.02	0.0000	0.00	10	0.0000	0.00	10
	5	500	0.0000	0.03	0.0000	0.01	10	0.0000	0.03	10
	5	1,000	0.0000	0.03	0.0000	0.14	10	0.0000	0.21	10
	10	50	0.4810	0.02	0.0103	0.25	0	0.0014	18.37	6
	10	100	0.0248	0.02	0.0000	0.00	10	0.0000	0.01	10
	10	500	0.0002	0.02	0.0000	0.01	10	0.0000	0.02	10
	10	1,000	0.0000	0.03	0.0000	0.04	10	0.0000	0.15	10
	25	50	2.0800	0.02	0.1797	0.03	9	0.1797	3.00	9
	25	100	0.4330	0.02	0.0347	0.59	0	0.0119	120.03	0
	25	500	0.0023	0.03	0.0000	0.01	10	0.0000	0.02	10
	25	1,000	0.0004	0.03	0.0000	0.03	10	0.0000	0.09	10
Average/Total			0.2894	0.02	0.0330	0.09	107	0.0277	10.92	114
Overall average/total			0.4700	0.02	0.0168	0.04	361	0.0147	4.32	371

Note. FGLM: Pentium II 400 MHz, *AR*: AMD 2.4 GHz, *DIMM-SS*: Pentium IV 3 GHz.

(2000, 2004), who obtained from the authors the original *FGLM* code used in Franca et al. (1994). The results obtained by algorithm *AR* on an AMD 2.4 GHz are taken from Alvim and Ribeiro (2004). Algorithm *DIMM-SS* was tested on a Pentium IV 3 GHz.

For each algorithm, the entries in the tables give the average percentage gap (*%gap*) and the average CPU time (*sec*) in seconds of the adopted machine over the corresponding 10 instances. The percentage gap of the solution value found, say $\bar{U}$, with respect to the best initial lower bound $\bar{L}$ (see §2.1), was computed as $100(\bar{U} - \bar{L})/\bar{L}$ as in França et al. (1994), Frangioni et al. (2000, 2004), and Alvim and Ribeiro (2004). For *AR* and *DIMM-SS*, we also give the number (*#opt*) of proved optimal solutions found out of ten, i.e., the number of instances for which $\bar{U} = \bar{L}$. (This information was not available for algorithms *FGLM* and *FNS*.) The columns corresponding to algorithm *FNS* are missing in Table 1, since these results were not reported by Frangioni et al. (2000, 2004), who claimed that uniform instances are very easy to solve (although, according to our experiments, this is not always the case). For each considered range, an additional line gives the average/total values over the corresponding 130 instances. The last line of each table reports the overall average/total values over the 390 tested instances.

The tables show that the best heuristics are *AR* and *DIMM-SS*. Algorithm *AR* is faster (0.17 CPU seconds on average over the 780 tested instances versus 2.77 CPU seconds) but has higher percentage gaps (0.0237% versus 0.0073%) and less proved optimal solutions (730 versus 758). For no instance was the solution value found by *DIMM-SS* worse than that found by *AR*. The other algorithms are dominated.

Note that the average CPU time can be quite high when the number of proven optimal solutions is small, reaching a maximum value of 120 seconds.

Table 3 synthesizes the most significative results obtained in the process of tuning the parameters of algorithm *DIMM-SS*. Concerning $k - l$ swaps, it turned out that it is convenient to always have $\bar{k} = \bar{l}$. For different tentative values of parameters $\bar{k}$, $|S|$, Q, D, and $\tilde{n}$, we report the overall average/total values of *%gap*, *sec*, and *#opt* over the 780 tested instances.

The table shows the substantial stability of *DIMM-SS* when a single parameter varies, with a relevant exception for $\bar{k}$. This also suggests that one of the reasons why *DIMM-SS* behaves better than the other metaheuristics could be the use of $k - l$ swaps in local search: The first three lines of the table show indeed that a considerable improvement is obtained if swaps involve pairs of jobs instead of single jobs. Preliminary computational experiments also showed that another relevant ingredient for the effectiveness of the algorithm was the adoption of function φ (see Equation (7)) for combination.

6.2. Branch and Price

Because our overall algorithm *DIMM* (see §4) starts by determining a solution through *DIMM-SS*, there was no point in testing the branch-and-price phase on those instances for which the *DIMM-SS* finds a proved optimal solution of value $\bar{U} = \bar{L}$. Hence, this phase of the computational experiments was performed on the 22 instances that remained unsolved after the execution of *DIMM-SS*.

The tuning of the branch-and-price algorithm concerned three decisions: the choice of the computer

Table 2 Metaheuristic Algorithms: Nonuniform Instances

Instances			*FGLM*		*FNS*		*AR*			*DIMM-SS*		
Range	*m*	*n*	*%gap*	*sec*	*%gap*	*sec*	*%gap*	*sec*	*#opt*	*%gap*	*sec*	*#opt*
$[1, 10^2]$	5	10	0.7520	0.00	0.0000	0.00	0.0000	0.00	10	0.0000	0.00	10
	5	50	1.5400	0.01	0.8580	0.01	0.0000	0.03	10	0.0000	0.00	10
	5	100	0.6050	0.01	0.0053	0.02	0.0000	0.01	10	0.0000	0.00	10
	5	500	0.2030	0.00	0.0000	1.44	0.0000	0.01	10	0.0000	0.04	10
	5	1,000	0.1250	0.01	0.0000	12.01	0.0000	0.02	10	0.0000	0.17	10
	10	50	1.4200	0.01	1.5700	0.00	0.7711	0.31	4	0.0000	0.00	10
	10	100	0.7310	0.01	0.5090	0.04	0.0213	0.22	8	0.0000	0.00	10
	10	500	0.6630	0.01	0.0021	3.26	0.0000	0.00	10	0.0000	0.03	10
	10	1,000	0.2740	0.01	0.0000	20.21	0.0000	0.00	10	0.0000	0.11	10
	25	50	0.5280	0.00	0.0000	0.01	0.0000	0.00	10	0.0000	0.00	10
	25	100	0.7440	0.00	0.9850	0.04	0.1336	0.55	8	0.0000	0.00	10
	25	500	0.6430	0.01	0.0212	5.78	0.0000	0.08	10	0.0000	0.46	10
	25	1,000	0.6160	0.01	0.0080	52.99	0.0000	0.18	10	0.0000	1.02	10
Average/Total			0.6803	0.01	0.3045	7.37	0.0712	0.11	120	0.0000	0.14	130
$[1, 10^3]$	5	10	0.6130	0.00	0.0000	0.00	0.0000	0.00	10	0.0000	0.00	10
	5	50	1.5900	0.00	0.8920	0.01	0.0000	0.03	10	0.0000	0.00	10
	5	100	0.6520	0.01	0.0074	0.06	0.0000	0.02	10	0.0000	0.00	10
	5	500	0.0000	0.01	0.0000	1.87	0.0000	0.00	10	0.0000	0.03	10
	5	1,000	0.0188	0.01	0.0000	18.31	0.0000	0.02	10	0.0000	0.18	10
	10	50	1.4200	0.01	1.4600	0.01	0.0000	0.00	10	0.0000	0.00	10
	10	100	0.7180	0.01	0.4640	0.07	0.0000	0.35	10	0.0000	0.02	10
	10	500	0.6460	0.01	0.0000	5.61	0.0000	0.07	10	0.0000	0.03	10
	10	1,000	0.0453	0.01	0.0000	21.81	0.0000	0.06	10	0.0000	0.18	10
	25	50	0.4030	0.01	0.0000	0.01	0.0000	0.00	10	0.0000	0.00	10
	25	100	0.7500	0.00	0.7930	0.08	0.1338	1.08	8	0.0000	0.02	10
	25	500	0.6590	0.01	0.0043	15.39	0.0000	0.13	10	0.0000	0.72	10
	25	1,000	0.6180	0.01	0.0008	138.21	0.0000	0.43	10	0.0000	0.43	10
Average/Total			0.6256	0.01	0.2786	15.50	0.0103	0.17	128	0.0000	0.12	130
$[1, 10^4]$	5	10	0.6180	0.00	0.0000	0.00	0.0000	0.00	10	0.0000	0.00	10
	5	50	1.5800	0.01	0.8960	0.01	0.0000	0.02	10	0.0000	0.00	10
	5	100	0.6640	0.01	0.0058	0.09	0.0000	0.01	10	0.0000	0.02	10
	5	500	0.0000	0.01	0.0000	1.97	0.0000	0.00	10	0.0000	0.03	10
	5	1,000	0.0000	0.01	0.0000	19.19	0.0000	0.03	10	0.0000	0.18	10
	10	50	1.4200	0.01	1.4600	0.02	0.0004	0.35	8	0.0000	0.01	10
	10	100	0.7250	0.01	0.4630	0.15	0.0000	0.24	10	0.0000	0.03	10
	10	500	0.6550	0.01	0.0002	7.15	0.0000	0.01	10	0.0000	0.05	10
	10	1,000	0.0000	0.01	0.0000	15.57	0.0000	0.02	10	0.0000	0.12	10
	25	50	0.4090	0.01	0.0000	0.01	0.0000	0.00	10	0.0000	0.00	10
	25	100	0.7580	0.01	0.7760	0.14	0.1351	4.29	3	0.0008	39.82	7
	25	500	0.6580	0.01	0.0021	23.20	0.0000	3.13	10	0.0000	1.23	10
	25	1,000	0.6200	0.02	0.0001	195.88	0.0000	0.22	10	0.0000	2.75	10
Average/Total			0.6236	0.01	0.2772	20.26	0.0104	0.64	121	0.0001	3.40	127
Overall average/total			0.6432	0.01	0.2867	14.38	0.0306	0.31	369	0.0000	1.22	387

Note. FGLM and *FNS*: Pentium II 400 MHz, *AR*: AMD 2.4 GHz, *DIMM-SS*: Pentium IV 3 GHz.

code for solving the 0-1 knapsack problems with real profits arising as slave problems during the initial series of branchings to one (see §5.1), and the values of the threshold parameters ε (for tailing off) and ϑ (for branching) discussed in §5.2.

Good threshold values were experimentally determined as

- $\varepsilon = 10^{-4}$ (tested values: 10^{-3}, 10^{-4}, and 10^{-5}),
- $\vartheta = 0.7$ (tested values: 0.6, 0.7, 0.8, and 0.9).

To select the best KP01 code for our application, we tested the most effective available programs. Most of the specialized codes for KP01 require integer data, while in our case the β_j^* values are generally noninteger. We tested both codes capable of handling real data, namely:

- *MT1R* by Martello and Toth (1990b) (to our knowledge, this is the only specialized code for the non-integer KP01),
- the general ILP solver *Cplex 9.0*,

and codes restricted to integer instances, namely:

- *MT1* by Martello and Toth (1990b),
- *Combo* by Martello et al. (1999),

Table 3 Parameters Tuning

$\bar{k}$	$\|S\|$	Q	D	$\tilde{n}$	%gap	sec	#opt
2	40	10	8	$\min\{n, 4m\}$	0.00728	2.78	758
1	40	10	8	$\min\{n, 4m\}$	0.00899	5.40	730
3	40	10	8	$\min\{n, 4m\}$	0.00724	3.18	758
2	30	10	8	$\min\{n, 4m\}$	0.00730	2.73	756
2	50	10	8	$\min\{n, 4m\}$	0.00730	2.76	757
2	40	8	8	$\min\{n, 4m\}$	0.00731	2.73	756
2	40	12	8	$\min\{n, 4m\}$	0.00730	2.78	755
2	40	10	6	$\min\{n, 4m\}$	0.00731	2.77	755
2	40	10	10	$\min\{n, 4m\}$	0.00730	2.79	757
2	40	10	8	$\min\{n, 3m\}$	0.00729	2.66	756
2	40	10	8	$\min\{n, 5m\}$	0.00729	2.86	755

by appropriately scaling the real input data (although at the expenses of possible numerical troubles). In Table 4, we give the following information for each of the 22 "hard" instances:

- class, range, m, n, and progressive instance number N^{o} (as reported in our Web page);
- total CPU time (in seconds of Pentium IV, 3 GHz) required by *DIMM* with the different KP01 codes (with a time limit of 900 seconds, including the time initially required by the scatter search).

The last two lines report the average CPU times and the numbers of proved optimal solutions. All $P\|C_{\max}$ instances were solved to optimality with the exception of those for which the time limit was reached (denoted by *time*). *DIMM* with *MT1* embedded had the lowest average CPU time and solved all instances to optimality.

6.3. Overall Algorithm

In Table 5, the final version of *DIMM* with *MT1* embedded, is compared on a Pentium IV 3 GHz with the branch-and-bound algorithm *DM95* by Dell'Amico and Martello (1995, 2005), which is regarded as the most effective exact algorithm for $P\|C_{\max}$. The table gives the comparison on the 22 "hard" instances, providing, for each instance:

- class, range, m, n, and progressive instance number N^{o} (as reported in our Web page),
- solution value and CPU time for *DM95*, and
- best initial lower bound $\bar{L}$ (see §2.1) and upper bound U (see §§2.2, 2.3, and 3), solution value, and CPU time for *DIMM*.

Both algorithms had a time limit of 900 seconds per instance (*time* indicates a time limit). The last two lines report the average CPU times and the numbers of proved optimal solutions.

The table shows a clear superiority of *DIMM*, which exactly solves all instances within smaller average CPU times. Note that for 18 instances out of 22, the optimal solution value is $\bar{L}$, whereas for the remaining four instances the optimal solution was the one

Table 4 Hard Instances: Comparison of KP01 Solvers (Time Limit $(\textit{time}) = 900''$)

Instance					*MT1R*	*Cplex9*	*MT1*	*Combo*
Class	Range	*m*	*n*	N^{o}	*sec*	*sec*	*sec*	*sec*
Uniform	$[1, 10^2]$	25	50	3	30.03	30.49	30.08	30.02
	$[1, 10^3]$	25	50	3	30.02	30.72	30.02	30.03
	$[1, 10^3]$	25	100	2	124.11	124.63	124.13	123.80
	$[1, 10^4]$	5	10	6	0.02	0.05	0.02	0.00
	$[1, 10^4]$	10	50	1	181.63	186.86	30.63	162.00
	$[1, 10^4]$	10	50	3	30.56	203.58	221.70	44.88
	$[1, 10^4]$	10	50	5	32.81	55.95	34.05	30.55
	$[1, 10^4]$	10	50	8	64.64	109.02	82.13	51.52
	$[1, 10^4]$	25	50	1	30.02	30.78	30.03	30.03
	$[1, 10^4]$	25	100	0	*time*	*time*	124.13	358.88
	$[1, 10^4]$	25	100	1	125.08	157.39	126.06	133.34
	$[1, 10^4]$	25	100	2	142.47	437.89	123.83	329.77
	$[1, 10^4]$	25	100	3	317.59	200.50	123.84	188.13
	$[1, 10^4]$	25	100	4	*time*	*time*	124.22	*time*
	$[1, 10^4]$	25	100	5	123.81	148.19	125.97	129.91
	$[1, 10^4]$	25	100	6	123.58	165.92	123.53	125.80
	$[1, 10^4]$	25	100	7	130.72	148.95	130.13	123.78
	$[1, 10^4]$	25	100	8	127.02	165.81	130.69	126.91
	$[1, 10^4]$	25	100	9	124.13	238.41	154.48	126.84
Nonuniform	$[1, 10^4]$	25	100	1	122.97	*time*	403.98	121.66
	$[1, 10^4]$	25	100	3	202.84	*time*	199.70	121.58
	$[1, 10^4]$	25	100	7	345.95	*time*	743.17	121.95
Average time					191.36	315.23	146.20	155.06
Optimal solutions out of 22					20	17	22	21

already found by *DIMM-SS* (without proving optimality). An analysis of the 18 instances for which *DIMM-SS* was not able to find the optimum seems to indicate that instances of this kind are hard to solve to optimality without adopting considerably larger neighborhoods. Since we tested many variants of job exchanges involving two machines at a time, a possibility could be to generalize $k-l$ swaps to the case of more machines.

Finally, Table 6 summarizes a comparison of the exact algorithms over the entire test bed of 780 instances. For each class, range, and algorithm, we give the average percentage gap and CPU time as well as the number of proven optimal solutions out of 130. The same information is given for the 390 uniform instances and the 390 nonuniform instances and, in the last line, for the entire set of 780 instances. The table confirms the clear superiority of *DIMM* for what concerns both CPU times and the number of optimal solutions found. As to memory requirements, *DIMM* never exceeded the available 512 Mb RAM.

7. Conclusions

We have presented an algorithm for $P\|C_{\max}$ that is composed of a scatter search heuristic followed by an exact algorithm based on a specialized binary search and a branch-and-price scheme. We have tested these algorithms on all the 780 $P\|C_{\max}$ benchmark instances

Table 5 **Hard Instances: Exact P‖C$_{max}$ Algorithms (Time Limit (*time*) = 900″)**

Instance					*DM95*		*DIMM*			
Class	Range	*m*	*n*	Nº	*z*	*sec*	*L̄*	*U*	*z*	*sec*
Uniform	$[1, 10^2]$	25	50	3	111	0.02	110	111	111	30.08
	$[1, 10^3]$	25	50	3	1,105	80.45	1,092	1,105	1,105	30.02
	$[1, 10^3]$	25	100	2	1,944	*time*	1,941	1,942	1,941	124.13
	$[1, 10^4]$	5	10	6	11,575	0.00	11,385	11,575	11,575	0.02
	$[1, 10^4]$	10	50	1	26,241	*time*	26,233	26,234	26,233	30.63
	$[1, 10^4]$	10	50	3	27,784	*time*	27,764	27,765	27,764	221.70
	$[1, 10^4]$	10	50	5	25,300	*time*	25,296	25,297	25,296	34.05
	$[1, 10^4]$	10	50	8	32,280	*time*	32,266	32,267	32,266	82.13
	$[1, 10^4]$	25	50	1	9,688	19.61	9,517	9,688	9,688	30.03
	$[1, 10^4]$	25	100	0	21,231	*time*	21,169	21,172	21,169	124.13
	$[1, 10^4]$	25	100	1	17,264	*time*	17,197	17,199	17,197	126.06
	$[1, 10^4]$	25	100	2	21,674	*time*	21,572	21,575	21,572	123.83
	$[1, 10^4]$	25	100	3	20,934	*time*	20,842	20,844	20,842	123.84
	$[1, 10^4]$	25	100	4	20,655	*time*	20,568	20,571	20,568	124.22
	$[1, 10^4]$	25	100	5	20,748	*time*	20,695	20,697	20,695	125.97
	$[1, 10^4]$	25	100	6	20,174	*time*	20,021	20,023	20,021	123.53
	$[1, 10^4]$	25	100	7	19,306	*time*	19,272	19,274	19,272	130.13
	$[1, 10^4]$	25	100	8	20,701	*time*	20,598	20,600	20,598	130.69
	$[1, 10^4]$	25	100	9	19,197	*time*	19,124	19,127	19,124	154.48
Nonuniform	$[1, 10^4]$	25	100	1	37,899	*time*	37,881	37,882	37,881	403.98
	$[1, 10^4]$	25	100	3	37,979	*time*	37,929	37,930	37,929	199.70
	$[1, 10^4]$	25	100	7	37,969	*time*	37,942	37,943	37,942	743.17
Average time						740.95				146.20
Optimal solutions out of 22						4				22

available on the Internet. The scatter search component is superior to the other metaheuristic algorithms from the literature: Within few seconds on average, it solves to optimality a larger number of instances and finds better approximations for the remaining instances. The branch-and-price component finds the optimal solution for all the instances that are not solved by scatter search. The overall algorithm is superior to the best exact algorithms proposed so far and constitutes the state of the art for the exact solution of the identical parallel machine scheduling problem.

Table 6 **All 780 Instances: Exact P‖C$_{max}$ Algorithms (Time Limit (*time*) = 900″)**

Instances		*DM95*			*DIMM*		
Class	Range	*%gap*	*sec*	*#opt*	*%gap*	*sec*	*#opt*
Uniform	$[1, 10^2]$	0.0000	0.00	130	0.0000	0.23	130
	$[1, 10^3]$	0.0019	21.75	127	0.0000	1.86	130
	$[1, 10^4]$	0.0348	146.98	109	0.0000	13.50	130
Average/Total		0.0122	56.25	366	0.0000	5.19	390
Nonuniform	$[1, 10^2]$	0.0155	131.56	120	0.0000	0.14	130
	$[1, 10^3]$	0.0169	124.66	112	0.0000	0.12	130
	$[1, 10^4]$	0.0190	346.32	80	0.0000	10.99	130
Average/Total		0.0171	200.85	312	0.0000	3.75	390
Overall average/total		0.0147	128.55	678	0.0000	4.47	780

Acknowledgments
The authors thank the Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR), Italy for the support given to this project. The computational experiments were executed at the Laboratory of Operations Research of the University of Bologna (Lab.O.R.). The authors also thank anonymous referees for helpful comments.

References
Alvim, A. C. F., C. C. Ribeiro. 2004. A hybrid bin-packing heuristic to multiprocessor scheduling. C. C. Ribeiro, S. L. Martins, eds. *Lecture Notes in Computer Science*, Vol. 3059. Springer-Verlag, Berlin, 1–13.

Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, R. E. Tarjan. 1973. Time bounds for selection. *J. Comput. System Sci.* **7** 448–461.

Brucker, P. 2001. *Scheduling Algorithms*. Springer-Verlag, New York.

Chen, B. 2004. Parallel scheduling for early completion. J. Y. T. Leung, ed. *Handbook of Scheduling*: *Algorithms, Models, and Performance Analysis*. CRC Press, Boca Raton, FL, 175–184.

Coffman, E. G., M. R. Garey, D. S. Johnson. 1978. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.* **7** 1–17.

Dell'Amico, M., S. Martello. 1995. Optimal scheduling of tasks on identical parallel processors. *ORSA J. Comput.* **7** 191–200.

Dell'Amico, M., S. Martello. 2005. A note on exact algorithms for the identical parallel machine scheduling problem. *Eur. J. Oper. Res.* **160** 576–578.

Dell'Amico, M., M. Iori, S. Martello. 2004. Heuristic algorithms and scatter search for the cardinality constrained P‖C$_{max}$ problem. *J. Heuristics* **10** 169–204.

Fekete, S. P., J. Schepers. 2001. New classes of fast lower bounds for bin packing problems. *Math. Programming* **91** 11–31.

Fischetti, M., S. Martello. 1988. A hybrid algorithm for finding the k-th smallest of n elements in $O(n)$ time. B. Simeone, P. Toth, G. Gallo, F. Maffioli, S. Pallottino, eds. *FORTRAN Codes for Network Optimization*, Vol. 13. *Annals of Operations Research*. J. C. Baltzer AG, Basel, Switzerland, 401–419.

França, P. M., M. Gendreau, G. Laporte, F. M. Müller. 1994. A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Comput. Oper. Res.* **21** 205–210.

Frangioni, A., E. Necciari, M. G. Scutellà. 2000. A multi-exchange neighborhood for minimum makespan machine scheduling problems. Technical Report TR-00-17, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.

Frangioni, A., E. Necciari, M. G. Scutellà. 2004. A multi-exchange neighborhood for minimum makespan machine scheduling problems. *J. Combin. Optim.* **8** 195–220.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco.

Gilmore, P. C., R. E. Gomory. 1961. A linear programming approach to the cutting stock problem. *Oper. Res.* **9** 849–859.

Gilmore, P. C., R. E. Gomory. 1963. A linear programming approach to the cutting stock problem—Part II. *Oper. Res.* **11** 863–888.

Glover, F., M. Laguna, R. Martí. 2004. Scatter search and path relinking: Foundations and advanced designs. G. C. Onwubolu, B. V. Babu, eds. *New Optimization Techniques in Engineering*. Springer-Verlag, Heidelberg, Germany, 87–99.

Graham, R. L. 1966. Bounds for certain multiprocessing anomalies. *Bell System Tech. J.* **45** 1563–1581.

Graham, R. L. 1969. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* **17** 416–429.

Graham, R. L., E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Math.* **5** 287–326.

Hochbaum, D. S., D. B. Shmoys. 1987. Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *J. ACM* **34** 144–162.

Hoogeveen, A., J. K. Lenstra, S. L. van de Velde. 1997. Sequencing and scheduling. M. Dell'Amico, F. Maffioli, S. Martello, eds. *Annotated Bibliographies in Combinatorial Optimization*. John Wiley & Sons, Chichester, UK, 181–197.

Johnson, D. S., A. Demers, J. D. Ullman, M. R. Garey, R. L. Graham. 1974. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.* **3** 299–325.

Leung, J. Y. T., ed. 2004. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Boca Raton, FL.

Martello, S., P. Toth. 1984. Worst-case analysis of greedy algorithms for the subset-sum problem. *Math. Programming* **28** 198–205.

Martello, S., P. Toth. 1990a. Lower bounds and reduction procedures for the bin packing problem. *Discrete Appl. Math.* **28** 59–70.

Martello, S., P. Toth. 1990b. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, UK, http://www.or.deis.unibo.it/knapsack.html.

Martello, S., D. Pisinger, P. Toth. 1999. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Sci.* **45** 414–424.

Martí, R., M. Laguna, F. Glover. 2006. Principles of scatter search. *Eur. J. Oper. Res.* **169** 359–372.

Mokotoff, E. 2001. Parallel machine scheduling problems: A survey. *Asia-Pacific J. Oper. Res.* **18** 193–242.

Mokotoff, E. 2004. An exact algorithm for the identical parallel machine scheduling problem. *Eur. J. Oper. Res.* **152** 758–769.

Mokotoff, E., J. J. Jimeno, I. Gutiérrez. 2001. List scheduling algorithms to minimize the makespan on identical parallel machines. *TOP* **9** 243–269.