Branching List Scheduling Algorithms for the Identical Parallel Machine Scheduling Problem

Hakim Hadj-Djilani, Julien Bernard, Louis-Claude Canon, Laurent Philippe

femto-st SCIENCES & TECHNOLOGIES

INSTITUT FEMTO-ST

UMR CNRS 6174

Branching List Scheduling Algorithms for the Identical Parallel Machine Scheduling Problem

Version 1

Hakim Hadj-Djilani — Julien Bernard — Louis-Claude Canon — Laurent Philippe

Research report n° RR-FEMTO-ST-2919

DÉPARTEMENT DISC – April 30, 2024

cnrs

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

ensmm École Nationale Supérieure de Mécanique et des Microtechniques

utbm université de technologie Belfort-Montbéliard

Branching List Scheduling Algorithms for the Identical Parallel Machine Scheduling Problem

Version 1

Hakim Hadj-Djilani , Julien Bernard , Louis-Claude Canon , Laurent Philippe

Département DISC

DEODIS

Research report no RR-FEMTO-ST-2919 April 30, 2024 (pages)

Abstract: This paper proposes new heuristics to the classic non-preemptive scheduling problem of assigning n jobs on m identical parallel machines with the objective to minimize the makespan. Starting from the List Scheduling method (LS), used for example in the LPT [Graham, 1969] or SLACK [Della Croce & Scatamacchia, 2020] heuristics, we derive a branching strategy that also considers assigning a job to the second best machine, in addition the first one, the only one usually dealt with in literature. Along the exploration of solutions, we keep only the best solution. This strategy leads to two heuristics, thought in an effort to speed up the solution search. The first, named Branch & Parallelize LS (BPLS), parallelizes the two alternatives considered on each job. The second, named BBLS, applies a Branch & Bound method to widely prune the solution tree. As a trade-off between computation time and solution quality, these heuristics are parameterized in order to assign only a subset of jobs according to the branching strategy. Out of this subset, the assignment is made by the classic LS, that is, each time on the first available machine. We show on literature instances that our heuristics outperform many of well-known algorithms on the vast majority of the considered instances. We also investigate the subspace of instances for which our heuristics are not able to beat all of these algorithms. Finally, we propose an ad hoc heuristic named MULTI-BBLS (MBBLS) which consists in multiple calls to BBLS that permit to rank first even on this instance subspace.

Key-words: identical parallel machines scheduling, heuristics, branch and bound, list scheduling

FEMTO-ST Institute, DISC research department
UFR Sciences - Route de Gray - F-25030 BESANÇON cedex FRANCE
Tel: (33 3) 81 66 65 15 – Fax: (33 3) 81 66 64 50 – e-mail: fvernill@femto-st.fr

Branching List Scheduling Algorithms for the Identical Parallel Machine Scheduling Problem

Version 1

Résumé : Cet article propose de nouvelles heuristiques pour le problème classique de l'ordonnancement non préemptif qui consiste à assigner n tâches à m machines parallèles identiques, avec pour objectif de minimiser le temps de terminaison. En partant de la méthode du List Scheduling (LS), utilisée par exemple dans LPT [Graham, 1969] ou SLACK [Della Croce & Scatamacchia, 2020], nous dérivons une stratégie d'exploration qui considère également l'assignation d'une tâche à la deuxième meilleure machine, en plus de la première, la seule habituellement traitée dans la littérature. Tout au long de l'exploration des solutions, nous ne conservons que la meilleure solution. Cette stratégie conduit à deux heuristiques, pensées dans le but d'accélérer la recherche de solutions. La première, nommée Branch & Parallelize LS (BPLS), parallélise les deux alternatives considérées pour chaque tâche. La seconde, appelée BBLS, applique une méthode Branch & Bound pour élaguer largement l'arbre des solutions. Dans le but d'obtenir un compromis entre le temps de calcul et la qualité de la solution, ces heuristiques sont paramétrées de manière à n'assigner qu'un sous-ensemble de tâches selon la stratégie de parcours. Hors de ce sous-ensemble, l'assignation est faite par LS classique, c'est-à-dire à chaque fois sur la première machine disponible. Nous montrons sur des exemples de la littérature que notre heuristique surpasse de nombreux algorithmes bien connus sur la grande majorité des instances considérées. Nous étudions également le sous-espace des instances pour lesquelles notre heuristique n'est pas capable de battre tous ces algorithmes. Enfin, nous proposons une heuristique ad hoc appelée MULTI-BBLS (MBBLS) qui consiste en de multiples appels à BBLS permettant de se classer premier même sur ce sous-espace d'instances.

Mots-clés : ordonnancement, machines parallèles identiques, heuristique, branch and bound, ordonnancement de liste

FEMTO-ST Institute, DISC research department
UFR Sciences - Route de Gray - F-25030 BESANÇON cedex FRANCE
Tel: (33 3) 81 66 65 15 – Fax: (33 3) 81 66 64 50 – e-mail: fvernill@femto-st.fr

Branching List Scheduling Algorithms for the Identical Parallel Machine Scheduling Problem

Hakim Hadj-Djilani , Julien Bernard , Louis-Claude Canon , Laurent Philippe

April 30, 2024

1 Introduction

Let us consider the identical parallel machine scheduling problem, denoted $P||C_{\max}$ according to Graham's three-field notation [Graham et al., 1979]: given m identical parallel machines $\{1 \leq i \leq m\}$ and a larger number of jobs $\{1 \leq j \leq n\}$ whose processing times are denoted $\{p_j\}_{j\in\{1,2,\ldots,n\}}$, we have to find a non-preemptive schedule such that the makespan C_{max} is minimum. More formally, the makespan is defined by $C_{\max} = \max_i\{C_i, 1 \leq i \leq m\}$, $C_i = \sum_{k_i} p_{k_i}$ being the i-th machine completion time for its assigned jobs k_i. $C_{\max}$ is hence the completion time for the execution of all jobs. Most of the time, as we do in this paper, the p_j values are considered as a set of positive integers[1]. Finding the optimal solution, denoted C^*_{max}, constitutes a strongly NP-hard problem in combinatorial optimization as described in [Garey & Johnson, 1979]. This problem has practical utility in industry and engineering and has been the focus of extensive research for more than half a century. Several approximation algorithms have been proposed in the literature.

A first family of algorithms is based on the List Scheduling algorithm (LS). These algorithms all work in two stages: the jobs are first sorted under a certain order then the LS algorithm is applied. The LS heuristic rule is to assign each job to the current least loaded machine, that is the first one available. The most famous of these algorithms is the Longest Processing Time First rule (LPT) due to Graham (see [Graham, 1969]), which sorts the jobs in non-increasing order of p_j's. More recently, another LS algorithm, named SLACK, was published in [Della Croce & Scatamacchia, 2020]. SLACK proceeds in a bit more complicated way than LPT for the sorting. Firstly, it sorts also the jobs in non-increasing order of p_j's. But then it splits the list in tuples of size m, completing the last tuple, if necessary, with null processing time jobs. Next, the tuples are sorted in non-increasing order of their slack, which is the absolute difference between the smallest and the largest p_j's of the tuple. Finally, LS is applied to the job list formed by concatenation of the sorted tuples.

Another family of heuristics developed to tackle the $P||C_{\max}$ problem is based on the bin-packing problem (BPP). In this problem, n objects of different finite sizes have to fit into bins of finite capacity or length L. The objective is to minimize the number of bins used. The First Fit Decreasing algorithm (FFD) [Coffman et al., 1978] gives a bin-packing solution in two steps. First, it sorts the objects in non-increasing order of their size. Then, starting with only one open bin, it assigns the objects according to this order, one by one to the first bin it can fit into. If there is not enough space in all bins then a new bin is opened. A form of duality exists between BPP and $P||C_{\max}$. Indeed, given a solution of BPP, as a number of m' bins, the capacity L is an upper bound to C^*_{max} for the $P||C_{\max}$ instance composed of as many jobs as objects in the BPP instance, with processing times p_j equal to the object sizes and with an assignment on $m \geq m'$ machines. This duality is the base of the MULTIFIT algorithm [Coffman et al., 1978]: a binary search of the $P||C_{\max}$ instance makespan, identified to the BPP capacity

[1] But note that this is made without loss of generality because floating/fixed-point numbers can be bijectively converted to integers, the $P||C_{\max}$ problem solved and then the resulting schedule converted back to floating/fixed-point numbers

Table 1: Upper bounds on approximation ratio & Time complexity

Algorithm	Upper bounds	Time complexity
LPT [Graham, 1969]	$r^{LPT} \leq \frac{4}{3} - \frac{1}{3m}$	$O(n \log n)$
SLACK [Della Croce & Scatamacchia, 2020]	N.D.	$O(n \log n)$
MULTIFIT [Coffman et al., 1978]	$r^{MULTIFIT} \leq 1.22 + 2^{-k}$	$O(n \log n + k\, n \log m)$
COMBINE [Lee & Massey, 1988]	$r^{COMBINE} \leq r^{MULTIFIT}, r^{LPT}$	$O(n \log n + k\, n \log m)$
LISTFIT [Gupta & Ruiz-Torres, 2001]	$r^{LISTFIT} \leq 13/11 + 2^{-k}$	$O(n^2 \log n + n^2\, k \log m)$
Hochbaum & Shmoys PTAS [Hochbaum & Shmoys, 1987]	$r^{PTAS} \leq 1 + \varepsilon$	$O((n/\varepsilon)^{\lceil 1/\varepsilon^2 \rceil})$
LDM [Michiels et al., 2003]	$\frac{4}{3} - \frac{1}{3(m-1)} \leq r^{LDM} \leq \frac{4}{3} - \frac{1}{3m}$ for $m \geq 3$ and $r^{LDM} = 7/6$ for $m = 2$	$O(n\, log\, n)$

L, by iterating on the FFD algorithm to find out if it is an upper bound, i.e. $m \geq m'$. The COMBINE algorithm [Lee & Massey, 1988] came next as a combination of LPT and MULTIFIT. It first uses the former and, if its solution is not guaranteed to be optimal, it tries MULTIFIT with a search upper bound set to the makespan found by LPT. About a decade later, Gupta and Ruiz-Torres proposed LISTFIT in [Gupta & Ruiz-Torres, 2001]. Their approach is also based on FFD but tries $4n$ different specially forged orders of p_j's as input of FFD. Thanks to its wider solution exploration, LISTFIT provides very often better solutions than MULTIFIT.

Of course a myriad of other algorithms exists for $P||C_{\max}$. We can, for instance, cite the Polynomial Time Approximation Scheme proposed in [Hochbaum & Shmoys, 1987], which has some similarities with MULTIFIT in that it also uses the duality with BPP and a binary search, but turns to dynamic programming for the problem solving. The Largest Differencing Method (LDM) is another method that allows to solve a $P||C_{\max}$ equivalent partitioning problem ([Michiels et al., 2003], [Karmarkar & Karp, 1982]). Many metaheuristics as genetic algorithms [Min & Cheng, 1999], tabu search, simulated annealing [Glass et al., 1994], swarm optimization [Kashan & Karimi, 2009], harmony search [Chen et al., 2012] or ant colony optimization [Yibao et al., 2002], have also largely been used to search for local optima of $P||C_{\max}$ solutions.

Because they produce suboptimal solutions, $P||C_{\max}$ heuristics, are evaluated according to their approximation ratio: $r^A = \frac{C^A_{\max}}{C^*_{\max}}$, with A an algorithm to be evaluated, $C^A_{\max}$ the algorithm makespan and $C^*_{\max}$ the optimal solution. When possible, a worst-case analysis is made to define upper bounds, the more tight as possible, for an algorithm approximation ratio. Table 1 lists known upper bounds and algorithm computational complexities of several well-known algorithms. Alternatively, evaluating and comparing the performance of algorithms is made through empirical protocols using pseudo-random instances defined in the literature.

The main contribution of this paper is the Branch & Bound List Scheduling algorithm (BBLS). Because LS-based heuristics concentrate on the job list order and restricts the assignment of jobs to the first available machine, we propose with BBLS to consider assigning jobs not only to the first available machine but also to the next ones. The Branch & Bound part of BBLS is highly linked to the research introduced in [Dell'Amico & Martello, 1995] which aims at producing exact/optimal $P||C_{\max}$ solutions considering basically all the m machines for a job assignment. On the other hand BBLS is more about searching approximate solutions in a smaller amount of time and hence limits the number of considered machines. In this goal it leverages many elements of optimization that are developed and assessed in this paper.

This article is organized in five sections. In section 2, we examine the LS heuristic and describe the branching strategy of a first LS variant, called BLS for Branch LS. BLS is a first step toward BBLS. It represents the alternative job assignments on the first available machines as a tree and then searches for the best solution. We show how this strategy can easily be parallelized into an algorithm named BPLS (for Branch & Parallelize LS). Then, in section 3, pursuing an effort to speed up these variants, we derive BBLS that is able to prune the tree of solutions by testing their makespan lower bounds as in [Dell'Amico & Martello, 1995]. Besides, we introduce several optimization properties and parameters to speed up the tree pruning. Next, in the section 4 we define MULTI-BBLS (or MBBLS) methods that mainly consist in calling BBLS several times in parallel in order to enhance the quality of solutions. Finally, in section 5, we set up an empirical protocol based on literature instances [Gupta & Ruiz-Torres, 2001], [Della Croce & Scatamacchia, 2020] to show how BPLS and BBLS can outperform the renowned algorithms presented before. The section 6 comes as a complement to see how the MBBLS approaches can be combined together to beat all the tested algorithms for subgroups of instances on which BBLS and BPLS were not able to dominate.

2 The BLS and BPLS algorithms

As previously explained, the LS algorithm consists in assigning each job to the current least loaded machine. In many cases however it can be observed that some of the machines are not much more loaded than the first one. Hence it is meaningful to assess the impact of also considering several least loaded machines instead of only the first one. On the other hand it is worth noticing that the exploration with several machines at each step of the assignment quickly leads to a combinatorial explosion. Furthermore, tests made by considering the third least loaded machine showed to be less efficient when adding the pruning process[2]. We hence only explore alternative of the second least loaded machine in the remainder of the paper.

Let us start with a very simple example, before explaining how an algorithm based on this idea works. Consider the following instance $I = \{P = (91,90,71,59,56,27,16,16,16,7), m = 3\}$ as input of LPT. The sorting stage of P is already done in non-increasing order. Then, with LS, comes the assignment of the jobs whose processing times are listed in P to obtain the schedule shown in Figure 1. If we introduce an exception to the LS rule, by allowing once to assign a job to the second least loaded machine, we obtain the assignment illustrated in Figure 2, which happens to be an optimal solution for I. This short example shows that exploring solutions with the two least loaded machines has a potential to find better results than LS or even optimal solutions for certain instances.

We can now define the Branch LS algorithm (BLS). Basically, it is a recursive algorithm that concurrently tries, for each job, an assignment on the two least loaded machines. Once the last job is assigned, a backtracking process takes place to compare, at each level of recursion, the two job assignment alternatives according to their makespans in order to keep the best solution. This approach is simply a solution search structured in a binary tree. The left child node represents an assignment on the first least loaded machine, while the right one is an assignment on the second least loaded machine. Notice that there is no interest to assess the two alternatives for the m first jobs that are always assigned one on each machine. The last job of the list is likewise assigned according to LS because this is always the best choice.

[2]We give more details about this question and our related tests in 5.3.1 and also in discussion about complexity equation (1) in the following of this section. The pruning process is discussed in 3.2

Figure 1

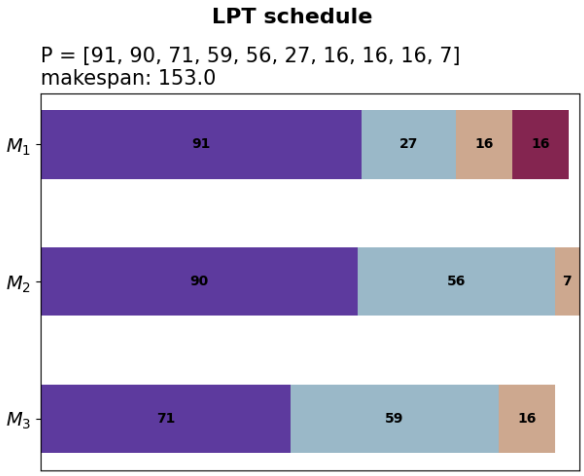
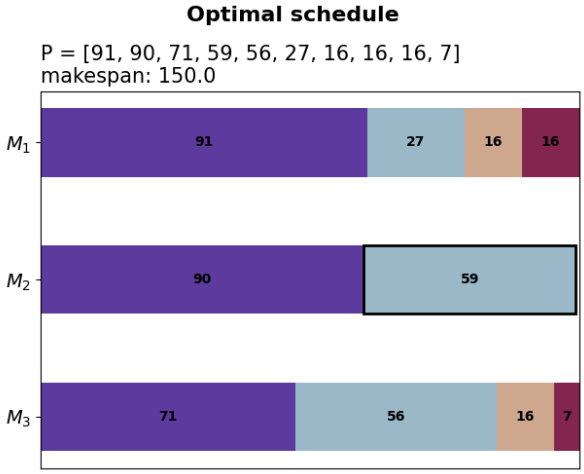


Figure 2: *The black framed job represents an assignment on the second least loaded machine, other assignments are all made on the first least loaded one according to LS*



Unfortunately, this approach has an exponential computational cost of $C(n,m) = \Theta(m2^{n-m})$. A first step for reducing this cost is to consider that not only the m first jobs can be excluded from the binary search but also several other jobs since, as can be seen on the example of Figure 2, several jobs are assigned to the first least loaded machine in the optimal solution. Knowing which jobs to exclude seems complicated but limiting to a parameter, denoted N, the number of consecutive jobs included in the binary search can achieve a linear asymptotic complexity since 2^N is a constant. The BLS complexity is then defined in (1).

$$C(n,m) = O(2^N m) + O(n-N) = O(n) \tag{1}$$

Note that, even if 2^N is just a multiplicative constant, it should not be underestimated as it remains an exponential term. By the way, if our strategy has considered the three first available machines instead of only the two first ones, the basis of this exponential would have been 3, increasing again the computational cost. On the other hand, it is possible to reduce the value of N to achieve a comparable cost whatever is the basis of the exponential. Precisely, if $N = N_2$ in basis 2 and $N = N_3$ in basis 3 and we choose an arbitrary N_2, we can adjust the value of N_3 to obtain in the induced ternary tree an equal or

lower number of nodes than in the binary tree induced by N_2. The formula of this constraint is simply: $N_3 = \lfloor log_3(2^{N_2+2}-1)-1 \rfloor$. However, as explained in 5.3.1, using a ternary tree does not give good results with the tested instances.

Algorithm 1: BLS(P, m, σ, N)

Input:
m : *integer*
// $\sigma[j][k]$: job k put on machine j
$\sigma \leftarrow \{()_{i\in\{1,\dots,m\}}\}$: *schedule*
// $p_{\sigma[j][k]}$: job k processing time
$P \leftarrow \{p_j\}_{j\in\{1,\dots,n\}}$
// default complexity limit
$Optionalparameter : N \leftarrow \infty$: *integer*
Data:
i_1, i_2: integer

1 **begin**
2 **if** *empty*(σ) **then**
3 $\sigma \leftarrow assign(\sigma, 1, p_1)$
4 $\sigma \leftarrow assign(\sigma, 2, p_2)$
5 ...
6 $\sigma \leftarrow assign(\sigma, m, p_m)$
7 $\sigma' \leftarrow \text{BLS}(P - \{p_1, \dots, p_m\}, m, \sigma, N)$
8 **return** σ'
9 **end if**
10 $i_1 \leftarrow \arg\min_{i\in\{1,\dots,m\}} \sum_{1\le k\le size(\sigma[i])}(p_{\sigma[i][k]})$
11 $i_2 \leftarrow \arg\min_{i\in\{1,\dots,m\}-\{i_1\}} \sum_{1\le k\le size(\sigma[i])}(p_{\sigma[i][k]})$
12 **if** $size(P) > 1$ **then**
13 **if** $N > 0$ **then**
14 $\sigma'_1 \leftarrow assign(\sigma, i_1, p_1)$
15 $\sigma'_2 \leftarrow assign(\sigma, i_2, p_1)$
16 $\sigma'_1 \leftarrow \text{BLS}(P - \{p_1\}, m, \sigma'_1, N-1)$
17 $\sigma'_2 \leftarrow \text{BLS}(P - \{p_1\}, m, \sigma'_2, N-1)$
18 **if** $makespan(\sigma'_2) < makespan(\sigma'_1)$ **then return** σ'_2
19 **else return** σ'_1
20 **end if**
21 $\sigma' \leftarrow assign(\sigma, i_1, p_1)$
22 $\sigma' \leftarrow \text{BLS}(P - \{p_1\}, m, \sigma', 0)$
23 **return** σ'
24 **end if**
25 $\sigma' \leftarrow assign(\sigma, i_1, p_1)$
26 **return** σ'
27 **end**

The BLS heuristic, given as Algorithm 1, recursively builds the schedule, denoted σ for an input instance defined by $(P = \{p_j\}_{1\le j\le n}, m)$. It starts (line 3) from an empty σ, assigns the first m jobs $j \in \{1, \dots, m\}$ according to LS and makes a recursive call (line 7) to enter in the branching part of the

algorithm. Once σ is initialized the two least loaded machines, i_1 and i_2[3], are identified (lines 10, 11). The two following conditions (lines 12, 13) assert that it remains at least two jobs to assign (remember that the machines are not challenged for the last jobs) and less than $N+m$ jobs are already assigned into σ. The algorithm then takes the first of the remaining jobs, whose processing time is p_1, assigns it alternatively on i_1 and i_2 by updating respective schedules σ'_1 and σ'_2 (lines 14, 15). Then (lines 16, 17) it makes two BLS recursive calls responsible for assigning the remaining jobs into σ'_1 and σ'_2 according to the same recursive process. On each recursive call N is decremented and P loses its first processing time. Going ahead, N becomes zero or P becomes a singleton. In both cases BLS switches back to LS algorithm for assigning the remaining jobs: jobs $m+1+N$ to $n-1$, lines 21-23 and last job, line 25. Afterward a backtracking takes place on lines 18 and 19, where the makespans of the two alternative schedules σ'_1 and σ'_2 are compared, keeping the best makespan solution. The backtracking continues toward parent calls until the recursion level of the first job assigned is reached back to finally return the BLS overall best solution found.

It is worth noticing that, to keep it simple, the branching strategy is here applied starting from the job $m+1$ to the job $m+N$. For more generality, the N jobs could have been picked in the remainder of the job list [4]. Anyway, the parameter N cannot be greater than $(n-m-1)$ because, as already mentioned, the m first jobs and the last one are assigned according to LS.

2.1 BPLS: a parallelization of BLS

Due to the BLS algorithmic complexity it is worth proposing solutions to improve its running time on large instances. Since BLS is considering many alternative solutions, it is possible to parallelize the search. Note that this is not the case for LPT and SLACK that produce a single solution for which there is no possible parallelization regarding the dependency between the algorithm steps. A basic parallelization can be done on the two alternatives (i_1, i_2 choices) using two processes, each one computing one of the assignments made according to the branching strategy. We name BPLS the parallelization variant of BLS (for Branch & Parallelize LS). An example is given in Figure 3 for a parallelization using a total of four processes $\text{Proc}_{k\in\{1,\dots,4\}}$ to produce a schedule when $N=2$. The number of processes must be limited to at most 2^N, which is the number of leaves of the binary tree, above what there is no interest to parallelize. There is likewise and obviously no point to parallelize using more processes than the number of threads the CPU used is capable to run in parallel. A basic experiment has, for example, shown that running BPLS with at most 2^7 processes on instances defined as $n=2^{12}$, $m=2,4,8,16$ and $N=10$ can generate a median speedup of 28 (from a time of approximately 12.5 sec for BLS to only 0.44 sec for BPLS[5]). More elements about both the computation time and performance of the makespan minimization are presented in section 5. About the performance, we can already state that BLS and BPLS cannot produce worse solutions than LS because the LS solution is included in the solutions evaluated by the branching strategy and kept as final solution if it is the best.

2.2 A branching strategy for all LS-based algorithms

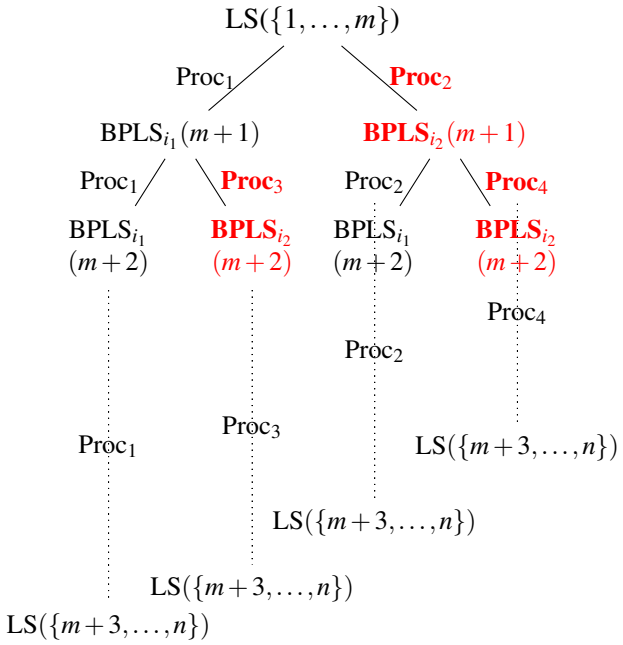
A primary interest of this branching strategy used in BLS and BPLS algorithms is that it can be used in any LS algorithm. Thereby we can easily write a B-SLACK or a BP-SLACK algorithm or likewise a B-LPT or a BP-LPT algorithm. The job assignment stage, in the LS algorithm, has only to be replaced by one of the branching LS algorithms. This is also true for the BBLS algorithm presented in the next section which has been used to write a BB-SLACK variant.

[3] Notice that we should respectively name $i_{1,j}$ and $i_{2,j}$ the two least loaded machines at the assignment time of the j-th job but, when there is no ambiguity, we only denote them i_1 and i_2, without any mention of the next job to assign.

[4] We work toward this variation with BBLS and its parameter S in 3.2.1

[5] On an Intel(R) Xeon(R) Platinum 9242 equipped of 48 cores, that is 96 threads

Figure 3: *Parallelization tree of BPLS (N = 2) using 4 processes $Proc_k$. Left and right BPLS nodes represent respectively i_1 and i_2 alternative machine assignments performed each time on two separate processors. The classic LS heuristic is used for the first jobs indexed from 1 to m and also for the last jobs indexed from $m+3$ to n. The two jobs in between are assigned using BPLS.*



3 A Branch & Bound LS algorithm

As previously mentioned, BLS and BPLS algorithms explore the whole binary tree for assigning N jobs, each time on the two first available machines. On the other hand, there is no interest to continue exploring toward a node if the best makespan found so far in the exploration is a lower bound for the subset of solutions this node is encoding. That remark leads us straight right to branch-and-bound (BB) algorithms. In fact, BLS is half way to be a BB algorithm because it already provides the branching part but not the bounding one. The principle for a Branch and Bound LS algorithm (BBLS) is, as for BLS, that any node in the tree represents a partial schedule and child nodes encode the solution subsets of the parent node. One job is added in the partial schedule at each level of the tree whether on machine i_1 or on machine i_2. In addition to that process, a lower bound of the subset of solutions represented by a node can be computed on the fly to find out if this solution has a chance to give a better makespan than the current best one. So the algorithm starts by exploring the tree with the LS solution as initial best makespan and, for each node, it computes a lower bound for the corresponding solution. If this lower bound is greater than the current best makespan, then there is no interest to go further on this branch, since all child solutions are bounded with the same lower limit, and the branch is pruned out of the tree. Otherwise the algorithm continues exploring and, at each time it reaches a leaf, it compares the makespan of this particular schedule to the current best makespan. If the new solution has a lower makespan it becomes the new best solution for the next of the exploration. Such a method can really speed up the tree exploration by widely pruning the tree.

3.1 $P||C_{\max}$ Lower bounds

To implement BBLS, especially the pruning strategy discussed above, we need lower bounds as tight as possible to the optimal makespan. Simple lower bounds can be established on any $P||C_{\max}$ instance,

whatever algorithm is used. We present here the lower bounds already described at least in [Dell'Amico & Martello, 1995 The simplest one, L_0, is obtained by a preemptive relaxation of the $P||C_{\max}$ instance:

$$L_0 = \frac{1}{m}\sum_{j=1}^{n} p_j$$

L_0 formula means that this is not possible to obtain a smaller makespan than the one obtained by considering that the jobs can be preempted. Indeed, if the jobs can be split, the resulting fragments can always be spread equitably to the machines, such that the completion time is the same on all machines: $C_i = C_{i+1}, \forall i < m$. This completion time is the optimal makespan.

A second lower bound comes from the non-preemptive constraint of the problem which implies that the makespan of any $P||C_{\max}$ instance cannot be lower than any p_j. This lower bound is

$$L_1 = \max(L_0, \max\{p_j\})$$

A third bound, more precise when not equal to L_1 or L_0, is obtained by another problem relaxation which consists in reducing the instance to the $(m+1)$ jobs of largest processing times. Assume that the processing times $(p_j)_{j\in 1,\ldots,n}$ are sorted in non-increasing order, then the jobs of processing times p_m and p_{m+1} must be assigned to the same machine to get an optimal solution for the relaxed instance. Indeed, the machine running the job m will be the first available machine at the moment of assigning job $(m+1)$. Therefore we have the following bound:

$$L_2 = \max(L_1, p_m + p_{m+1})$$

Another lower bound, denoted L_3, is proposed in [Dell'Amico & Martello, 1995]. To introduce this bound we recall that the bin-packing problem (BPP) is dual to $P||C_{\max}$. Dell'Amico and Martello limit the number of bins of the BPP obtained by a dual reformulation of a $P||C_{\max}$ instance I into an instance I' of BPP. We denote $m'(L, \{p_j\}_{1\leq j\leq n})$, a solution of the bin packing problem, i.e. a number of bins for I' with L the bin capacity and p_j the item sizes, that come directly from I. If an optimal solution $m'^*(L, \{p_j\})$ is such that $m'^*(L, \{p_j\}) > m$, m being the number of machines in I, then L is obviously a lower bound of the optimal makespan of I. Since the p_j's are integers, it is also clear that $L+1$ is also a valid lower bound for I. That is what Dell'Amico and Martello did using two possible lower bounds of the optimal solution of I' with a certain capacity L to prove that $L+1$ is a lower bound for I. The two lower bounds are denoted B_α and B_β. The first one was introduced in [Martello & Toth, 1990] and the second one in [Dell'Amico & Martello, 1995]. Their definitions are based on a subset of processing times $\{\bar{p}\} = \{p_j \leq \frac{L}{2}\}$ and other subsets denoted J_1, J_2 and J_3. These three subsets form together a partition of all jobs whose processing times are greater or equal to a given $\bar{p}$. They are defined by:

$$J_1 = \{j | L - \bar{p} < p_j\} \tag{2}$$
$$J_2 = \{j | L/2 < p_j \leq L - \bar{p}\} \tag{3}$$
$$J_3 = \{j | \bar{p} \leq p_j \leq L/2\} \tag{4}$$

The two bounds are then defined as follows for a given $\bar{p}$ and a given bin capacity L:

$$B_\alpha(L, \bar{p}) = |J_1| + |J_2| + \max(0, \left\lceil \frac{\sum_{j\in J_3} p_j - (L|J_2| - \sum_{i\in J_2} p_j)}{L} \right\rceil) \tag{5}$$

$$B_\beta(L, \bar{p}) = |J_1| + |J_2| + \max(0, \left\lceil \frac{|J_3| - \sum_{j\in J_2} \left\lfloor \frac{L-p_j}{\bar{p}} \right\rfloor}{\left\lfloor \frac{L}{\bar{p}} \right\rfloor} \right\rceil) \tag{6}$$

The idea of these bounds is that no jobs in J_1 or J_2 can be in the same bin because they all exceed the processing time $\frac{L}{2}$. So one bin is used for each one of these jobs. The remaining J_3 jobs cannot fit in

bins that already contain a J_1 job. Since this type of job has a p_j greater than $L - \bar{p}$, this does not leave enough space to let a J_3 job fit into the bin. So the bounds B_α and B_β, in their right 'max' term, take both account of the minimum number of new bins that are necessary to make all the J_3 jobs fit into the bins and form a full bin packing. Precisely, some of the J_3 jobs might fit into bins that already contain one J_2 job, if there is such. But if it remains other jobs of J_3 then the opening of at least one new bin is needed. Finally, it is stated that, consistently with the dual relationship between $P||C_{\max}$ and BPP, if

$$B_\alpha(L,\bar{p}) > m \text{ or } B_\beta(L,\bar{p}) > m \tag{7}$$

then $(L+1)$ is a lower bound for I. In [Dell'Amico & Martello, 1995] the bound named L_3 is the maximum capacity that verifies the property (7) but in this paper we denote L_3 any bound that verifies this property because, as detailed in the next subsection, it is enough to justify a pruning.

For a single value L, it costs up to $O(n^2)$ to compute if $L+1$ is a lower bound for a $P||C_{\max}$ instance. This is a considerable cost but, hoping that the tree of solutions will be massively pruned in BBLS, the lower bound computation could be worth it. This is empirically verified in section 5. Besides, we give optimization properties for the calculation of these bounds in 3.2.2.

3.2 The BBLS algorithm and its pruning process

Algorithm 2: Function *bb_node_instance* Computes a tree node transformed instance.
- P is the p_j's of jobs yet to assign,
- σ is the node partial schedule.

```
1 Function bb_node_instance(σ, P, m)
2 begin
      // P does not contain any p_j of job already assigned in σ
3     for j ← 1,...,m do
4         p'_j ← Σ_{1≤k≤size(σ[j])} (p_{σ[j][k]})
5     end for
6     for j ← 1,...,size(P) do p'_{m+j} ← P[j]
7     return (p'_j)_{1≤j≤m+size(P)}
8 end
```

The condition (7) now allows to complete the bounding part of the BBLS algorithm envisioned in the beginning of this section, i.e. pruning nodes for which the best makespan found so far, denoted $C^b_{\max}$, is larger than the node lower bound. Considering a given node, we can proceed, as in [Dell'Amico & Martello, 1995], by transforming the problem instance I to another instance I' whose solutions are precisely the subset of solutions encoded by this node. The *bb_node_instance* function, given as Algorithm 2, builds the transformed instance from a BBLS tree node.

The jobs previously assigned to the m machines are replaced by m jobs where p'_j $(1 \leq j \leq m)$, the processing time of job j in the transformed instance I', is equal to the sum of the processing times of the jobs already assigned to the j-th machine. These jobs are those assigned by the parent nodes that led to the considered node. From the way they are created, we name these m jobs the *aggregated jobs*. These jobs processing times are calculated in line 4 of *bb_node_instance*. The jobs that were not assigned yet to any machine are simply added as is in the transformed instance (line 6).

Next, property (7) has to be tested to verify that the current best bound $C^b_{\max}$ is a lower bound of the transformed instance. The function *is_L3_bound*, given as Algorithm 3, is used in this purpose. If this is the case, then no better solution can be found by considering this instance node and further child nodes. The node is hence pruned out. On the contrary, if $C^b_{\max}$ is not a lower bound for the transformed instance

Algorithm 3: Function *is_L3_bound*
Tests if L is a L_3 lower bound for $I' = (P,m)$

Function *is_L3_bound*(L,P,m)
begin
  $L \leftarrow L-1$
  `// use (7) on` $I' = (P,m)$ `for all` $0 < \bar{p} \leq L/2$
  for $\bar{p} \in \{p_j | j \in \{1,\ldots,size(P)\}, p_j \leq L/2\}$ **do**
    $J_1 \leftarrow \{j | L - \bar{p} < p_j\}$
    $J_2 \leftarrow \{j | L/2 < p_j \leq L - \bar{p}\}$
    $J_3 \leftarrow \{j | \bar{p} \leq p_j \leq L/2\}$
    $B_\alpha(L,\bar{p}) \leftarrow |J_1| + |J_2| + \max(0, \left\lceil \frac{\sum_{j\in J_3} p_j - (L|J_2| - \sum_{j\in J_2} p_j)}{L} \right\rceil)$
    $B_\beta(L,\bar{p}) \leftarrow |J_1| + |J_2| + \max(0, \left\lceil \frac{|J_3| - \sum_{j\in J_2} \left\lfloor \frac{L-p_j}{\bar{p}} \right\rfloor}{\left\lfloor \frac{L}{\bar{p}} \right\rfloor} \right\rceil)$
    if $B_\alpha(L,\bar{p}) > m \vee B_\beta(L,\bar{p}) > m$ **then return** *true*
  end for
  return *false*
end

then the exploration continues on the child nodes. Note that $C^b_{\max}$ can be tested first against L_2 which is less expensive to compute than L_3. If $C^b_{\max} > L_2$ then *is_L3_bound* is used to test if $C^b_{\max}$ is a L_3 lower bound. Besides, since L_2 was tested before, the set of $\bar{p}$ tested in *is_L3_bound* can be limited to the jobs that verify the condition $\bar{p} \leq p_{m+2}$ as proved in [Dell'Amico & Martello, 1995]. It implies however to sort the p_j's in non-increasing order.

At this point BBLS, given as Algorithm 4, is almost a complete Branch & Bound algorithm. As mentioned before, the classic LS solution, that assigns each task to the first available machine, is used for the initialization. In the exploration tree it means that, for building the first complete solution, the left child node is always chosen until the leftest leaf is reached. This is made very naturally because the solution search is a depth-first search. For this reason, in Algorithm 4, line 13, the left child node is never ignored as long as $C^b_{\max} = \infty$ (its initial value). Notice that Algorithm 4 is very similar to Algorithm 1. Indeed, only a few parts change compared to BLS, mainly for the pruning implementation integrated in lines 12 to 30. In this block, precisely lines 13 and 19, $C^b_{\max}$ is tested as a lower bound of the transformed instance corresponding to each alternative assignment of the current job (to machine i_1 or i_2). Hence, if $C^b_{\max}$ is confirmed to be a lower bound, the node is pruned out, that is no further recursive call is made for the corresponding partial schedule (σ'_1 or σ'_2) which is marked as pruned out using the empty set $\emptyset$. As in BLS the backtracking process keeps the best solution between each two alternatives except that for BBLS the empty schedules, resulting from the pruning, are ignored. In the following subsection is described the parameter S which is an optimization of BBLS that is not present in BLS.

3.2.1 A shifting parameter S indexing the first job to start BBLS on

In Algorithm 4, the N parameter defines the number of jobs on which the branching is performed, other jobs being assigned according to the classic LS. This parameter however does not indicate the job of the list where to start the branching. The only constraints are that the job start index is strictly greater than m and smaller than or equal to $(n-N)$. For this reason, we introduce another BBLS parameter, named S, a positive integer that defines a shifting to start the branching strategy, i.e. the S parameter allows to start the branching on job of index $(m+1+S)$. There is a computational interest to pick up a larger value of S as the following properties show.

Algorithm 4: BBLS(P, m, σ, N, S)

Input:
$P \leftarrow \{p_j\}_{j\in\{1,\ldots,n\}}$
m : *integer*
$\sigma \leftarrow \{()_{i\in\{1,\ldots,m\}}\}$: *schedule*
Optional parameter : $N \leftarrow \infty$: *integer* `// default value (complexity limit)`
Optional parameter : $S \leftarrow 0$: *integer* `// job index shift for branching strategy start`

Data:
$C^b_{\max} \leftarrow \infty$ `// global variable`
i_1, i_2: integer

begin
  if *empty*(σ) **then**
    $\sigma \leftarrow LS(\{p_1, \ldots, p_{m+S}\}, m)$
    $\sigma' \leftarrow \text{BBLS}(P - \{p_1, \ldots, p_{m+S}\}, m, \sigma, N)$
    return σ'
  else
    $i_1 \leftarrow \arg\min_{i\in\{1,\ldots,m\}} \sum_{1\le k\le size(\sigma[i])} (p_{\sigma[i][k]})$
    $i_2 \leftarrow \arg\min_{i\in\{1,\ldots,m\}-\{i_1\}} \sum_{1\le k\le size(\sigma[i])} (p_{\sigma[i][k]})$
  end if
  if $size(P) > 1$ **then**
    if $N > 0$ **then**
      $\sigma'_1 \leftarrow assign(\sigma, i_1, p_1)$
      if $C^b_{\max} = \infty \vee \neg$ *is_L3_bound*($C^b_{\max}$, *bb_node_instance*(σ'_1, *P, m*), *m*) **then**
        $\sigma'_1 \leftarrow \text{BBLS}(P - \{p_1\}, m, \sigma'_1, N-1)$
      else
        $\sigma'_1 \leftarrow \emptyset$ `// elimination of the` i_1 `node`
      end if
      $\sigma'_2 \leftarrow assign(\sigma, i_2, p_1)$
      if $\neg is\_L3\_bound(C^b_{\max}, bb\_node\_instance(\sigma'_2, P, m), m)$ **then**
        $\sigma'_2 \leftarrow \text{BBLS}(P - \{p_1\}, m, \sigma'_2, N-1)$
      else
        $\sigma'_2 \leftarrow \emptyset$ `// elimination of the` i_2 `node`
      end if
      `// Keep the best solution between` σ'_1 `and` σ'_2 `or` $\emptyset$
      `// if the two nodes were pruned out`
      if $\sigma'_1 = \emptyset \wedge \sigma'_2 = \emptyset$ **then return** $\emptyset$
      else if $\sigma'_1 = \emptyset$ **then return** σ'_2
      else if $\sigma'_2 = \emptyset$ **then return** σ'_1
      else
        if $makespan(\sigma'_2) < makespan(\sigma'_1)$ **then return** σ'_2
        else return σ'_1
      end if
    else
      $\sigma' \leftarrow assign(\sigma, i_1, p_1)$
      $\sigma' \leftarrow \text{BBLS}(P - \{p_1\}, m, \sigma', 0)$
      return σ'
    end if
  else
    $\sigma' \leftarrow assign(\sigma, i_1, p_1)$
    if $makespan(\sigma') < C^b_{\max}$ **then** $C^b_{\max} \leftarrow makespan(\sigma')$
    return σ'
  end if
end

Theorem 3.1. *It exists a $S > 0$ large enough to reduce the practical computational cost of BBLS.*

Proof. First, note that taking a value of S greater than zero does not change the asymptotic complexity of BBLS. The dominant term of the BBLS complexity, implied by *is_L3_bound*, is $O(n^2)$ which does not depend on S but only on the size of the transformed instance produced by *bb_node_instance*. So for a same size of tree and a same pruning efficiency taking a greater S cannot be slower. On the contrary, recall that *bb_node_instance* produces a transformed instance I' which includes m aggregated jobs formed by grouping the jobs already assigned to each machine in parent nodes. So the greater is S, the lower is n', the number of jobs to compose I'. Hence running *is_L3_bound* is faster if S is larger. For the root node of the tree $n' = n - m - S$ for a positive S, while it is $n' = n - m$ if $S = 0$. So with respect to the quadratic growth of the complexity it is obviously faster to take a larger S. Another reason for a larger S to accelerate BBLS is that the subset of jobs $\{p'_j \leq L/2 | L = C^b_{\max} - 1\}$ in the instance I' can only shrink along the depth of the BB tree whatever is the order of the job list. Again, assigned jobs are aggregated into m larger ones in I', so looking at *is_L3_bound* we see that if S is larger enough the number of $\bar{p}$ candidates diminishes and so the time to compute the function. □

Theorem 3.2. *A large enough value of S allows a wider pruning of the BBLS tree.*

Proof. Suppose that the m aggregated jobs of a transformed instance I' have processing times $p'_{j\in\{1,\ldots,m\}} > L/2$ and reconsider B_α and B_β bounds in *is_L3_bound*. The aggregated jobs all belong to J_1 or J_2 whatever is the $\bar{p}$ considered. It gives that $|J_1| + |J_2| \geq m$ hence, if it remains enough jobs in J_3, it is most likely that B_α or B_β will be greater than m and so the node associated to I' in the BBLS tree will be pruned out because the condition (7) is fulfilled. If it exists a $j \in \{1,\ldots,m\}$ such that $p'_j \leq L/2$ then continue to increase S until there is no such j. It should happen because the processing times of aggregated jobs grow with S (the greater is S the more jobs have been assigned before). If it cannot happen consider anyway two values of shifting S_1 and S_2 such that $S_1 < S_2$ and respectively I'_{S_1}, I'_{S_2} the corresponding transformed instances for the root node in the BB tree. Then it is always true that $|J_{1S_1}| + |J_{2S_1}| \leq |J_{1S_2}| + |J_{2S_2}|$ for a same pair $(L, \bar{p})$ because again the aggregated jobs processing times can only be larger if more jobs have been assigned before. So the pruning can only be as or more efficient if the shifting of the branching is made according to a larger S. □

Therefore, BBLS (Algorithm 4) uses the job index $m + S + 1$ to start the branching strategy for N jobs (line 3, jobs $j = 1$ to $j = m + S$ are assigned according the classic LS). Note, that a meaningful S must be such that $0 \leq S \leq n - N - m - 1$. To test our assertions about the speedup and the wider pruning obtained with a greater S used in BBLS we have defined a protocol introduced in 5.2.

3.2.2 Other properties for optimization of the pruning process

According to our tests, using the parameter S, along with the parameter N, is the most efficient optimization of the pruning process we experienced. We present nevertheless several other properties to optimize BBLS and in particular *is_L3_bound*. The simple Theorem 3.3 was already known in [Martello & Toth, 1990] and results related to Theorems 3.4 and 3.5 also but were limited to B_α and decreasingly sorted p_j's. Here we extend the results to B_β and arbitrary order of p_j's. Besides, we introduce two other Theorems (3.6, 3.7) and explain how they might optimize a BBLS implementation for particular cases. To the extent of our knowledge these theorems are new.

Theorem 3.3. *For a given L, $J_1 \cup J_2$ and the sum $|J_1| + |J_2|$ are the same for all $\bar{p}$.*

Proof. $J_1 \cup J_2 = \{p_j | p_j > \frac{L}{2}\}$, this set does not depend on $\bar{p}$. As $\bar{p}$ increases J_2 shrinks and J_1 widens but their union does not change. Besides, by definition $J_1 \cap J_2 = \emptyset$. □

The previous property allows to precompute the sum $|J_1|+|J_2|$ once and for all $\bar{p}$ without having to build J_1 for each $\bar{p}$ in the loop that computes B_α and B_β. Still J_2 has to be built separately on each iteration as it depends on $\bar{p}$. The next theorem mitigates this need.

Theorem 3.4. *For a given L and any pair $(\bar{p}_{j_1}, \bar{p}_{j_2})$ such that $0 < \bar{p}_{j_1} \leq \bar{p}_{j_2} \leq \frac{L}{2}$, we have $J_2(L, \bar{p}_{j_2}) \subseteq J_2(L, \bar{p}_{j_1})$. Likewise, $J_3(L, \bar{p}_{j_2}) \subseteq J_3(L, \bar{p}_{j_1})$.*

Proof. It is obvious by definition of J_2 and J_3. □

Corollary 3.5. *Let $\bar{p}_{sup}$ be the greatest $\bar{p} \leq \frac{L}{2}$, it comes that $\forall \bar{p} \leq \frac{L}{2}, J_2(L, \bar{p}_{sup}) \subseteq J_2(L, \bar{p})$ and $J_3(L, \bar{p}_{sup}) \subseteq J_3(L, \bar{p})$.*

Proof. For any $\bar{p}$, take $\bar{p}_{j_2} = \bar{p}_{\sup}$ and $\bar{p}_{j_1} = \bar{p}$ then refer to Theorem 3.4. □

If the $\bar{p}$'s are considered in non-increasing order in the loop of *is_L3_bound* then the bounds B_α and B_β can be calculated iteratively from a greater $\bar{p}$ to a smaller $\bar{p}$ because according to Theorem 3.4:

$$\sum_{j \in J_2(L, \bar{p}_{j_1})} p_j = \sum_{j \in J_2(L, \bar{p}_{j_2})} p_j + \sum_{j \in (J_2(L, \bar{p}_{j_1}) \setminus J_2(L, \bar{p}_{j_2}))} p_j$$

$$\sum_{j \in J_3(L, \bar{p}_{j_1})} p_j = \sum_{j \in J_3(L, \bar{p}_{j_2})} p_j + \sum_{j \in (J_3(L, \bar{p}_{j_1}) \setminus J_3(L, \bar{p}_{j_2}))} p_j$$

Hence, along iterations on $\bar{p}$'s, keeping track of sums corresponding to $J_2(L, \bar{p}), J_3(L, \bar{p})$ it is possible to compute $B_\alpha(L, \bar{p})$ and $B_\beta(L, \bar{p})$ by incrementing the previous sums instead of computing the whole sums for each $\bar{p}$. As argued in [Martello & Toth, 1990] it allows to move from a $O(n^2)$ to a $O(n)$ complexity for the calculation of B_α for all $\bar{p}$ and as well for B_β.

On the other hand, if the $\bar{p}$'s are sorted in arbitrary order, corollary 3.5 implies that for computing $B_\alpha(L, \bar{p})$ and $B_\beta(L, \bar{p})$ it is possible to precompute the formula sums that correspond to the subsets $J_2(L, \bar{p}_{\sup})$, $J_3(L, \bar{p}_{\sup})$ and increment these sums by the sums corresponding to the specific parts of the $J2(L, \bar{p}), J3(L, \bar{p})$ because:

$$\sum_{j \in J_2(L, \bar{p})} p_j = \sum_{j \in J_2(L, \bar{p}_{\sup})} p_j + \sum_{j \in (J_2(L, \bar{p}) \setminus J_2(L, \bar{p}_{\sup}))} p_j$$

$$\sum_{j \in J_3(L, \bar{p})} p_j = \sum_{j \in J_3(L, \bar{p}_{\sup})} p_j + \sum_{j \in (J_3(L, \bar{p}) \setminus J_3(L, \bar{p}_{\sup}))} p_j$$

Next, we describe another theorem that is able, in certain particular cases, to infer at constant cost $O(1)$ the result of *is_L3_bound*. The idea is to deduce the result of the function for a BBLS tree node knowing the result it gave for the parent node (which is always 'false' when a child node is considered otherwise it would have been pruned out).

We first introduce a bit of context and notations for the next theorem. Let I'_k be the transformed instance of the considered node k in the BBLS tree, i_k the index of the machine to try on the assignment of the next job, whose processing time is p_a. The load of this machine before assignment is denoted C_{i_k} and $p_{i_k} = p_a + C_{i_k}$ is the processing time of the new aggregated job, added to form the transformed instance I'_k. The transformed instance of the parent node $(k-1)$ is denoted I'_{k-1}, the corresponding bounds are denoted $B_{\alpha,k-1}(\bar{p})$ and $B_{\beta,k-1}(\bar{p})$ for the current $C^b_{\max}$ and a given $\bar{p}$.

Theorem 3.6. *For a parent node $(k-1)$ of transformed instance I'_{k-1} and its child node k of transformed instance I'_k corresponding to the assignment of the next job p_a on machine i_k giving $p_{i_k} = p_a + C_{i_k}$, the processing time of the new aggregated job of I'_k, if $p_{i_k} \leq \frac{(C^b_{\max}-1)}{2}$ and $\forall \bar{p}, B_{\alpha,k-1}(\bar{p}) < m$ and $B_{\beta,k-1}(\bar{p}) < m$ then $C^b_{\max}$ is not a L_3 lower bound for I'_k and the node k must not be pruned out.*

Proof. See appendix A. □

Let us discuss about the practical optimization of the pruning process this theorem implies. For any child node k, the following conditions about the parent node are always true: $\forall \bar{p}\ B_{\alpha,k-1} \leq m$ and $B_{\beta,k-1} \leq m$. If they were not, the parent node would have been pruned out and the child node k ignored automatically, which is not the case. The theorem conditions are however slightly different, because it is asked that the bounds are not only lower but strictly lower than m and this is not guaranteed. We mitigate this point noting that, in all the bounds considered in cases (8) to (12), only four out of ten can make the equality of child bounds $B_{\alpha,k}, B_{\beta,k}$ to m possible and there is no reason for this to happen every time. The risk is to explore one child node that would have been pruned out otherwise but allowing to speed up the lower bound testing for many other nodes. Anyway it cannot for sure take BBLS to a worse solution that the one obtained without this optimization. In fact, it gives the exact same solution. Besides, we verified experimentally that it is worth it when $S = 0$.

To sum up the consequences of Theorem 3.6, the process of pruning in BBLS can be accelerated by considering that, if the new aggregated job p_{i_k} verifies the condition $p_{i_k} \leq \frac{(C^b_{\max}-1)}{2}$, then C^b_{max} is most likely not a L_3 lower bound for this node. It avoids to compute the precise bounds B_α and B_β and spare some calculation time.

In a close idea than that of Theorem 3.6, the lower bound testing of a node k might be optimized based on the result obtained on a sibling left node ℓ. The idea is to infer in particular cases that $C^b_{\max}$ is a lower bound of the instance I'_k without any use of the function *is_L3_bound*. In these particular cases, we suppose that the left sibling, whose instance is denoted I'_ℓ, was pruned out, that is $C^b_{\max}$ was tested to be a lower bound for I'_ℓ.

Theorem 3.7. *Consider a node k and its left sibling ℓ in a BBLS tree and their respective transformed instances I'_k, I'_ℓ. The node ℓ represents an assignment of the next job to the machine i_ℓ giving the aggregated job of processing time p_{i_ℓ} to be included in I'_ℓ. Likewise, the node k corresponds to an assignment to the machine i_k and to the I'_k's aggregated job of processing time p_{i_k}. Note that, because ℓ is a left sibling, the related machine i_ℓ is less loaded than the right sibling machine i_k, therefore $p_{i_\ell} \leq p_{i_k}$. Now assume that $C^b_{\max}$ is a L_3 lower bound of I'_ℓ then, if $p_{i_k} \leq \frac{(C^b_{\max}-1)}{2}$, the node k has to be pruned out.*

Proof. First, note that ℓ has been pruned out ($C^b_{\max}$ is the same for nodes ℓ and k), then for any valid $\bar{p}$:

- if $p_{i_k} < \bar{p}, J_{s,k} = J_{s,\ell}, \forall s \in \{1,2,3\}$ and so $B_{\alpha,k} = B_{\alpha,\ell}, B_{\beta,k} = B_{\beta,\ell}$.
- if $\bar{p} \leq p_{i_k} \leq \frac{(C^b_{\max}-1)}{2}$ notice that the only subset of I'_k's jobs with a potential change compared to I'_ℓ's subsets is $J_{3,k}$. Either $p_{i_\ell} \in J_{3,\ell}$ and $J_{3,k} = (J_{3,\ell} \cup \{p_{i_k}\}) \setminus \{p_{i_\ell}\}$ or $p_{i_\ell} \notin J_{3,\ell}$ and we simply have $J_{3,k} = J_{3,\ell} \cup \{p_{i_k}\}$. Recall that $p_{i_\ell} \leq p_{i_k}$ and that $J_{s,k} = J_{s,\ell}, s \in \{1,2\}$ then conclude that $B_{\alpha,k} \geq B_{\alpha,\ell}, B_{\beta,k} \geq B_{\beta,\ell}$. So according to (7) if ℓ node was pruned out, k node can only be pruned out because its bounds are greater or equal to ℓ's.

□

4 MULTI-BBLS: parallel runs of BBLS

In section 2 we present BLS and its parallel version BPLS. Then, in section 3, we introduce BBLS, an improvement of BLS that uses pruning to reduce the solution search tree. Doing so we freed computational power that we can now use to enhance the BBLS solutions at a small additional cost. We name this basic idea MULTI-BBLS (or MBBLS) which consists generally in running BBLS multiple times, mostly in parallel.

We devise three specific strategies in this purpose. They are described in 4.1, 4.2, 4.3. Each one of these strategies is not necessarily more efficient than a simple BBLS for all instances, it depends.

Interesting use cases are presented in section 6 where it is shown that combining these multiple calls of BBLS allows to outperform other heuristics on instances for which all our tested configurations of a single BBLS were unable to do so (see section 5).

4.1 MBBLS1: trying different orders of the job list

As mentioned previously, BBLS, as any other LS algorithm, assigns jobs in a specific order. However, trying different orders does not give the same result not only for BBLS but also for a classic LS like LPT or SLACK. LISTFIT solutions are also more efficient than MULTIFIT's because LISTFIT tries much more orders of the n jobs than MULTIFIT, which only tries one. The idea for MBBLS1 is then to leverage the power of a multi-core CPU to run BBLS on different job list orders. Because these runs are parallel it should not take more time than only one BBLS run, except if we add more than one supplementary order per CPU core. If needed one can decide to try more orders to pursue better solutions but it goes obviously with an increase of the computational cost.

It remains a question about what orders should be tried. We denote $\mathscr{O}$ the number of orders we want to try. We already know that the LPT and SLACK orders produce good solutions for many instances. So, if $\mathscr{O} \geq 2$, we always pick both of them. if $\mathscr{O} > 2$ we simply try to pick additional orders that are as different as possible because we do not have any information about which order is more efficient for a given instance. Indeed, it does not seem very wise to try very similar orders. Following this idea we try permutations of the LPT order taken in lexicographic order. More precisely, if $\mathscr{P}_{LPT}$ is the whole set of permutations of the LPT order of jobs j, we denote $d = \frac{|\mathscr{P}_{LPT}|}{\mathscr{O}-1}$ the offset between two permutations we want to try as input of BBLS (including the LPT order). We denote $\pi(k)$ the function that gives the k-th permutation of $\mathscr{P}_{LPT}$ sorted in lexicographic order. The LPT order is the permutation $\pi(1)$, defined by: $\pi(1) = 1,2,\ldots,n$, assuming that $p_1 \geq p_2 \geq \ldots \geq n$. The other permutations picked are $\pi(\lfloor d \rfloor), \pi(\lfloor 2 * d \rfloor), \ldots, \pi((\mathscr{O}-1)d = |\mathscr{P}_{LPT}|)$. Because of the lexicographic order, $\pi((\mathscr{O}-1)d)$, the last permutation picked is the so called SPT order (for Shortest Processing Time first) $\pi((\mathscr{O}-1)d) = n, n-1, \ldots, 1$. Note that it does not need to build the entire set $\mathscr{P}_{LPT}$ to extract a number of $(\mathscr{O}-1)$ permutations. The permutation extraction complexity is considered $O(1)$.

This approach is written in algorithm 5, in which three functions must be explained.

- $P \mapsto P_{order} = order(P, order)$ is the function that takes a list of processing times P and returns P_{order}, the list ordered in a specific *order* (e.g. 'LPT', 'SLACK' or 'SPT') .
- $parallel(function(args))$ is a non-blocking function that performs the parallel execution, as a thread or a process, of a $function$ configured with arguments *args*.
- $waitall()$ blocks the current program until all $parallel$ calls made before has ended their execution.

4.2 MBBLS2: one BBLS per subgroup of jobs

Another way to run multiple BBLS for a $P||C_{\max}$ instance $I = (P = \{p_j\}_{j\in\{1,\ldots,n\}}, m)$ is to divide the set of n jobs into a partition of K job subsets of sizes $n_1, \ldots, n_K$ such that $n_1 + \cdots + n_K = n$. Then BBLS is performed on each subset so that we obtain K different sub-schedules on m machines. These BBLS executions can be totally parallel if K is lower or equal to the number of CPU cores. Finally, these sub-schedules are grouped together to obtain a full schedule for the original instance I. This approach is presented as algorithm 6. For the grouping part (lines 16-18), we consider I', another $P||C_{\max}$ instance, formed from the K sub-schedules. Each sub-schedule gives m jobs. One job $p'_{k,i}$ is obtained by summing the processing times p_j of the jobs assigned to a machine i. Hence I' has a number of $K \times m$ jobs. A last BBLS execution is performed on I' (line 21) then the resulting schedule is transformed back to the original p_j's of I by splitting the $p'_{k,i}$ accordingly (lines 23-28) in order to reverse the summing

Algorithm 5: MBBLS1($P, m, N, S, \mathscr{O}$)

Input:
$P \leftarrow \{p_j\}_{j\in\{1,\ldots,n\}}$
$m : integer$
$Optional parameter : N \leftarrow \infty : integer$
$Optional parameter : S \leftarrow 0 : integer$
$Optional parameter : \mathscr{O} \leftarrow 1 : integer$
Data:
$C_{max}[\mathscr{O}] : integer$
$\sigma[\mathscr{O}] \leftarrow \{()_{i\in\{1,\ldots,m\}}\} : schedule$
$k \leftarrow 2 : integer$

1 **begin**
2 $P_{LPT} \leftarrow order(P, `LPT')$
3 $\sigma[1], C_{max}[1] \leftarrow parallel(BBLS(P_{LPT}, m, \sigma[1], N, S))$
4 $d \leftarrow \frac{|\mathscr{P}_{LPT}|}{\mathscr{O}-1}$
5 **while** $k < \mathscr{O}$ **do**
6 **if** $k = \mathscr{O} - 1$ **then** $\pi_k \leftarrow \pi(|\mathscr{P}_{LPT}|)$
7
8 **else** $\pi_k \leftarrow \pi(\lfloor d * (k-1) \rfloor)$
9
10 $P_k \leftarrow P_{LPT}[\pi_k[1]], \ldots, P_{LPT}[\pi_k[n]]$
11 $\sigma[k], C_{\max}[k] \leftarrow parallel(BBLS(P_k, m, \sigma[k], N, S))$
12 $k \leftarrow k + 1$
13 **end while**
14 $P_{SLACK} \leftarrow order(P, `SLACK')$
15 $\sigma[\mathscr{O}], C_{max}[\mathscr{O}] \leftarrow parallel(BBLS(P_{SLACK}, m, \sigma[\mathscr{O}], N, S))$
16 $waitall()$
17 $k_{best} \leftarrow \arg\min_k(C_{max})$
18 return $\sigma[k_{best}]$
19 **end**

Algorithm 6: MBBLS2(P,m,N,S,K)

Input:
$P \leftarrow \{p_j\}_{j\in\{1,\dots,n\}}$
$m : integer$
$Optional : N \leftarrow \infty, S \leftarrow 0, K \leftarrow 1 : integer$
Data:
$\sigma_K[K], \sigma', \sigma \leftarrow \{()_{i\in\{1,\dots,m\}}\} : schedule$
$S_k, N_k, S', N', n' \leftarrow K \times m : integer$
$n_1, \dots, n_K, q, r, k \leftarrow 0, j \leftarrow 1 : integer$

begin
  $q \leftarrow \lfloor n/K \rfloor$
  $r \leftarrow n - Kq$
  $n_1, n_2, \dots, n_K \leftarrow q$
  for $j \leftarrow 1$ *to* r **do** $n_j \leftarrow n_j + 1$
  $j \leftarrow 1$
  while $j \leq n$ **do**
    $P_k \leftarrow \{P[j], P[j+1], \dots, P[j+n_k-1]\}$
    $j \leftarrow j + n_k$
    $k \leftarrow k + 1$
    $N_k \leftarrow \max(\min(n_k - m - 1, N), 0)$
    $S_k \leftarrow \max(\min(S, n_k - N_k - m - 1), 0)$
    $\sigma_K[k] \leftarrow parallel(BBLS(P_k, m, \sigma_K[k], N_k, S_k))$
  end while
  $waitall()$
  for $k \leftarrow 1, \dots, K$ **do**
    for $i \leftarrow 1, \dots, m$ **do** $p'_{k,i} \leftarrow p_{\sigma_K[k][i][1]} + p_{\sigma_K[k][i][2]} + \cdots + p_{\sigma_K[k][i][size(\sigma_K[k][i])]}$
  end for
  $N' \leftarrow \max(\min(N, n' - m - 1), 0)$
  $S' \leftarrow \max(0, min(S, n' - N' - m - 1))$
  $\sigma' \leftarrow BBLS((p'_{1,1}, \dots, p'_{1,m}, p'_{2,1}, \dots, p'_{2,m},$
    $\dots, p'_{K,1}, \dots, p'_{K,m}), m, \sigma', N', S')$
  for $k \leftarrow 1, \dots, K$ **do**
    for $i \leftarrow 1, \dots, m$ **do**
      $\sigma \leftarrow replace(\sigma', (k,i), \sigma_K[k], (i,1),$
        $\dots, (i, size(\sigma_K[k])))$
    end for
  end for
  return σ
end

process made to form I'. This conversion is done by the *replace* function. It replaces the merger job of index (k,i) of instance I' set in schedule σ' by the corresponding jobs of original instance I whose p_j were summed to obtain $p'_{k,i}$. The schedule σ thus built is the $P||C_{\max}$ solution of I. The functions $parallel, waitall$ are the same as in 4.1.

Using MBBLS2, makes sense particularly on instances where n is very large. Indeed, it explores combinatorial possibilities on different subsets of the jobs instead of just one subset (defined by N and S in BBLS). The branching strategy is working at different levels of the job list and it is done in parallel, so

the computation time does not have to be more important to get more different solutions and potentially a better schedule. Finally, when the sub-schedules are added together to form one full schedule, we know it has been more refined than what BBLS alone is capable of. The same idea lays behind MBBLS3, described in the following, but for this heuristic the sub-problems are defined on subsets of machines instead of subsets of jobs.

4.3 MBBLS3: one BBLS per subgroup of machines

Yet another way to run multiple BBLS is to pose sub-problems that reduce the number of machines used. If m is the number of machines of the original instance I, we defined a number of K sub-instances I_k in which the numbers of machines are $m_1, m_2, \ldots, m_K$. That is only interesting if m is large enough (in our tests we considered it is worth it if $m > 5$). Indeed, the larger m is, the less binary BBLS is exploring the field of possibilities, because the algorithm is not allowed to try assignments on the third up to the m-th least loaded machines. On the other hand, increasing arity was not found to be an efficient way to proceed (as mentioned earlier and empirically elaborated in 5.3.1).

The strategy of splitting the m machines by groups is an attempt to enlarge the spanning of the exploration. The algorithm 7 follows this strategy. It works in three times:

1. Split the n jobs into K subgroups as balanced as possible. This partition problem is equivalent to a $P||C_{\max}$ problem on K machines so we can use BBLS in that purpose too (line 8). Of course K must stay small to be consistent with the idea to really reduce the number of machines. In our experiments we limit m to at most 25 machines. Another constraint is that all the m_k must be equal to each other to avoid unbalanced partition of jobs.

2. Solve corresponding K sub-instances of $P||C_{\max}$ whose numbers of machines are $m_1, \ldots, m_K$ (line 15).

3. Concatenate together the K (line 18) sub-schedules obtained to solve $P||C_{\max}$ on the initial instance I.

An additional refinement is possible before the last step, by proceeding to job interchanges between sub-schedules in order to reduce the possible unbalance between two sub-schedules. A simple rule would be: let I_{k_1}, I_{k_2} the sub-instances for which the makespans obtained are $C_{\max,k_1}, C_{\max,k_2}$, such that they are respectively the greatest and the smallest of all sub-schedules. Then if $\delta = C_{\max,k_1} - C_{\max,k_2}$ is such that it exists a pair of jobs $(p_1, p_2) \in I_{k_1} \times I_{k_2}$, with p_1 being assigned to the machine of greatest completion time $C_{\max,k_1}$, and $0 < p_1 - p_2 < \delta$ then swapping these two jobs between sub-schedules of I_{k_1} and I_{k_2} is lowering the overall $C_{\max}$ for I. We do not have to assess all pairs (p_1, p_2), only taking the two first largest can be enough. This interchanging process can hence be repeated in a time complexity of at most $O(n)$. It is not included in algorithm 7 but could easily be inserted on the end.

5 Experimental evaluation of BBLS and BPLS

In this section we mainly provide a global evaluation of BBLS and BPLS. They are compared to other algorithms mentioned in the introduction. More precisely, we present benchmarks for BB-SLACK and BP-SLACK, variants of SLACK [Della Croce & Scatamacchia, 2020], already introduced in 2.2. We did not integrate BB-LPT and BP-LPT in our tests because LPT has shown, in [Della Croce & Scatamacchia, 2020], to be less effective than SLACK on most of the instances considered. For reproducibility, all the source

Algorithm 7: MBBLS3(P,m,N,S,K)

Input:
$P \leftarrow \{p_j\}_{j\in\{1,\dots,n\}}$
$m : integer$
$Optional parameter : N \leftarrow \infty : integer$
$Optional parameter : S \leftarrow 0 : integer$
$Optional parameter : K \leftarrow 1 : integer$
Data:
$\sigma_K[K] \leftarrow \{()_{i\in\{1,\dots,m\}}\} : schedule$
$\sigma', \sigma \leftarrow \{()_{i\in\{1,\dots,m\}}\} : schedule$
$k \leftarrow 0 : integer$
$S_k, N_k : integer$
$q, r : integer$
$n_1, \dots, n_K : integer$
$m_1, \dots, m_K : integer$
begin
  $q \leftarrow \lfloor m/K \rfloor$
  $r \leftarrow m - Kq$
  $n_1, n_2, \dots, n_K \leftarrow q$
  if $r > 0$ **then**
    $error$('m must be evenly divided by K')
  end if
  $\sigma' \leftarrow BBLS(P,K,\sigma',N,S)$
  for $k \leftarrow 1,\dots,K$ **do**
    $m_k \leftarrow q$
    $P_k \leftarrow (p_{\sigma'[k][1]}, \dots, p_{\sigma'[k][m_k]})$
    $n_k \leftarrow size(P_k)$
    $N_k \leftarrow \max(\min(n_k - m_k - 1, N), 0)$
    $S_k \leftarrow \max(n_k - N_k - m_k - 1, 0)$
    $\sigma_K[k] \leftarrow parallel(BBLS(P_k, m_k, \sigma_K[k], N_k, S_k))$
  end for
  $waitall()$
  $\sigma \leftarrow (\sigma_K[1][1], \dots, \sigma_K[1][m_1], \dots, \sigma_K[K][1],$
  $\dots, \sigma_K[K][m_k])$
  return σ
end

code on which the experiments are based is available online [Hadj-Djilani, 2023a]. Before the benchmarks we take a little detour for testing the shifting parameter S studied in subsection 3.2.1.

But first we describe the sets of instances used in our experiments. They are all taken as references from the literature.

[6] Exact instances used in 5.3 are available online [Hadj-Djilani, 2023b]

[7] Although the value $m = 5$ is not used in the original E2 experiment ([Gupta & Ruiz-Torres, 2001]), it is used in GR2 instance set.

Table 2: Experiment instances

Instances[6]	Distribution	m	n	[a,b]
Della Croche and Scatamacchia Instances				
DCU	Uniform	5, 10, 25	10, 50, 100, 500, 1000	[1, 100], [1, 1000] [1, 10000]
DCNU	Non-uniform	5, 10, 25	10, 50, 100, 500, 1000	[1, 100], [1, 1000] [1, 10000]
Gupta et Ruiz-Torres Instances				
GR1	Uniform	3, 4, 5	2m, 3m, 5m	[1, 20], [20, 50]
GR2[7]	Uniform	2, 3, 4, 5, 6, 8, 10	10, 30, 50, 100	[100, 800]
GR3	Uniform	3, 5, 8, 10	3m+1, 3m+2, 4m+1, 4m+2, 5m+1, 5m+2	[1, 100], [100, 200]

5.1 The instances

We selected different sets of instances from the literature, they are listed in table 2. The first group of instances comes from [Della Croce & Scatamacchia, 2020] but originally from [França et al., 1994] for uniform instances (DCU) and from [Frangioni et al., 2004] for non-uniform instances (DCNU). The second group of instances is composed of three subgroups denoted GR1, GR2 and GR3 in reference to experiences E1, E2 and E3 made in [Gupta & Ruiz-Torres, 2001]. We chose those instances: (i) firstly because, as variants of SLACK, BB-SLACK and BP-SLACK should be evaluated according to the same protocol used in [Della Croce & Scatamacchia, 2020], (ii) secondly because, in our experiments, LISTFIT delivers good performance but comparing SLACK and LISTFIT using one or another set of instances did not give the same results, hence the need to use several set of instances like the one used to evaluate LISTFIT in [Gupta & Ruiz-Torres, 2001]. The instances are generated as follows: for each combination of m,n and interval $[a,b]$ in the lines of the table 2, such that $m < n$, 20 instances are drawn with processing times picked randomly in interval $[a,b]$ according to the instance group distribution (uniform or non-uniform). The non-uniform distribution is such that 98% of the processing times are integers uniformly distributed in $[0.9(b-a),\ b]$ while the remaining ones are uniformly distributed in $[a,\ 0.2(b-a)]$. All these instances sum up to 780 DCU instances and as many DCNU instances, 360 GR1 instances, 540 GR2 instances and 960 GR3 instances for a total of 3420 instances.

5.2 Testing the parameter S of BBLS

The goal of the experiment described here is to verify empirically the effect of Theorems 3.1, 3.2.

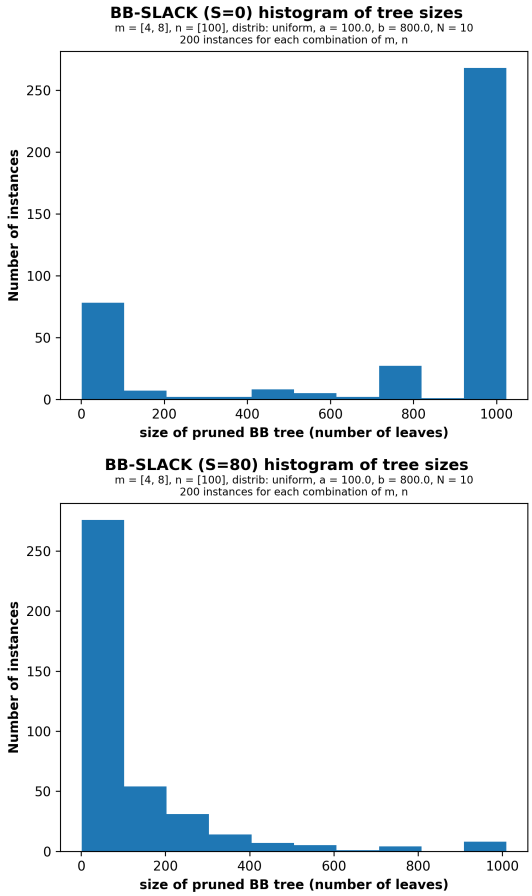
The instance family tested is a subset of GR2 instances (see table 2). Note that other instance subsets of table 2 would allow to make similar observations on computation times and BBLS tree pruning. So we generate 200 pseudo-random instances for each pair (m,n) with $m \in \{4,8\}$ and $n = 100$, that is a total of 400 instances. The processing times are picked in $\{100,\ldots,800\}$ according to an uniform law.

For our tests, we use BB-SLACK and BP-SLACK algorithms, that is BBLS and BPLS. In these runs N is always set to 10 but the value of S varies. BB-SLACK is ran two times, the first one with $S = 0$ and the second one with $S = 80$. For BP-SLACK, S is not used so it is equivalent to the configuration $S = 0$. The algorithm is here parallelized with at most 8 processes [8]. We intend to show that a larger value of S is interesting not only by speeding up the computation of BPP bounds (5), (6) but also by pruning widely the BB trees. BP-SLACK plays the role of the baseline to evaluate BB-SLACK performance in both cases $S = 0$ and $S = 80$. Indeed, remember that BBLS first objective remains to work faster than BPLS, without risking to produce poorer solutions. This is only possible when $S > 0$, otherwise

[8] Precisely, 8 cores of a AMD Ryzen 5 2500U CPU

BBLS (or BB-SLACK) produces the exact same solutions as BPLS (or BP-SLACK) as we verify in the following. We also verify that increasing S for a same N does not necessarily imply a loss of quality of the solutions.

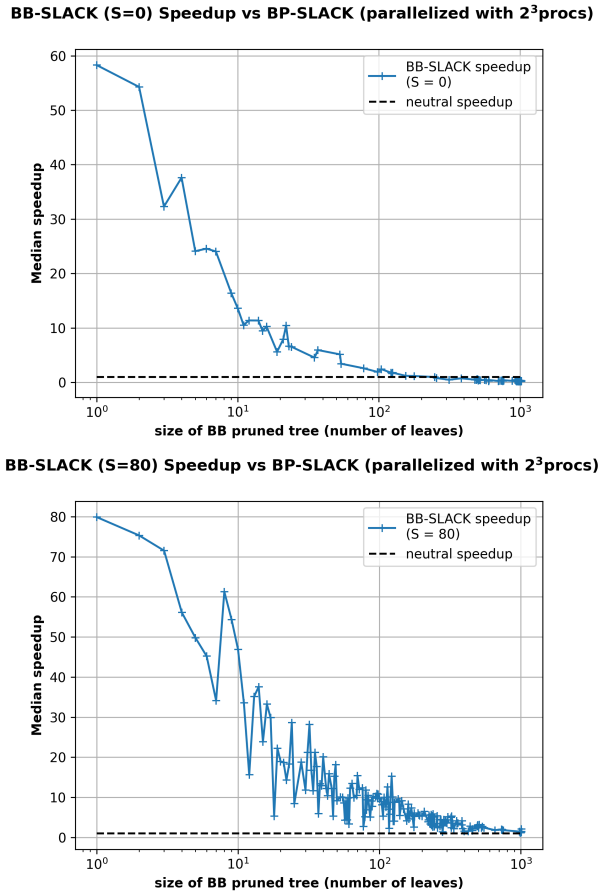
Figure 4: Histograms for the comparison of BB-SLACK tree sizes for $S = 0$ (on the top) and $S = 80$ (on the bottom)



We present the results of this experiment in Figures 4, 5 and 6, where the top subplot is the case $S = 0$ for BB-SLACK while the bottom one is the case $S = 80$. In Figure 4 the histogram shows that $S = 80$ results in a wider pruning of the trees. Indeed, for most of the instances (more than 250 out of 400), the pruning is such that the number of leaves is lower than 100, while it can go up to $2^N = 1024$ with no pruning. Besides, no tree of BB-SLACK with $S = 80$ is full, with 1024 leaves, the greatest tree size being 1010 leaves. The histogram for $S = 0$ shows, on the contrary, that for most of the instances (about 300 over 400) there is almost no pruning in BB-SLACK run (35 trees are full with 1024 leaves). We can therefore verify that increasing S is efficient to achieve a wider pruning for many instances as stated in Theorem 3.2.

The speedup of BB-SLACK, compared to BP-SLACK, according to the size of the pruned tree is shown in Figure 5. We can observe that, even for a same size of tree, BB-SLACK is faster when $S = 80$. This experimentally confirms Theorem 3.1. For example, for a size of 10 leaves, BB-SLACK with $S = 0$ reached a speedup about 13, while BB-SLACK with $S = 80$ allows a speedup about 45.

Figure 5: BB-SLACK vs BP-SLACK speedups obtained w.r.t. to the tree size for $S = 0$ (on the top) and $S = 80$ (on the bottom)



With Figure 6 we can evaluate how S can affect the quality of the $P||C_{\max}$ solutions for the considered instance family. In the two subplots shown in this figure, the results are partitioned in 3 groups:

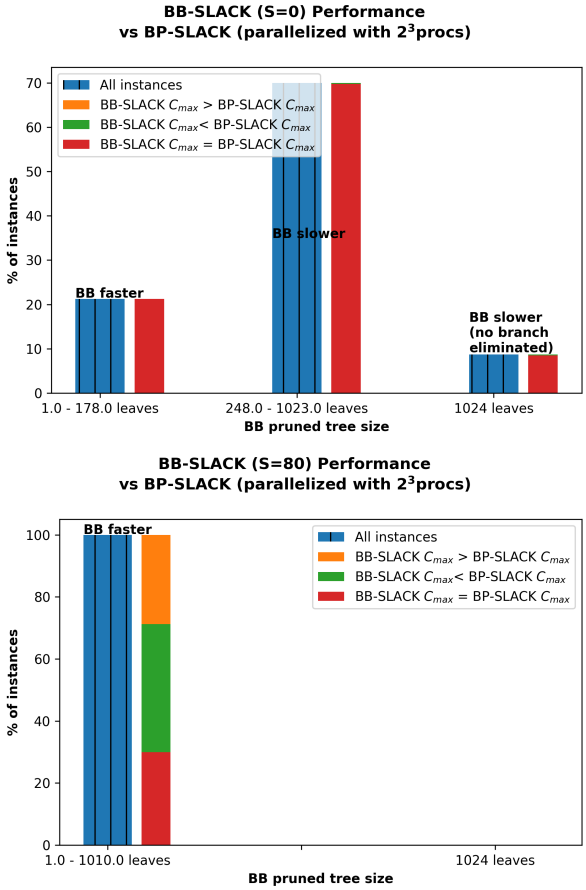
1. The left bars count for instances for which BB-SLACK pruned the tree and was faster than BP-SLACK.
2. The middle bars count the instances for which BB-SLACK pruned the tree but was slower than BP-SLACK .
3. The right bars count cases for which BB-SLACK was not able to prune the tree at all and hence was slower than BP-SLACK.

Note that cases 2 and 3 dit not happen for $S = 80$, so there is no bar at all.

In the top subplot, $S = 0$, so that the same pair (N,S) is used for BB-SLACK and BP-SLACK. Hence their solutions are exactly the same. Indeed, the sizes of the red bars are equal to these of the blue bars, which means that, for all the instances, BB-SLACK (BBLS) and BP-SLACK (BPLS) returned the same makespans. We can also see that, for $S = 0$, on the most part of the instances BB-SLACK is slower than BP-SLACK.

In the bottom subplot, where $S = 80$, the instances for which the makespans of BB-SLACK and BP-SLACK are equal are accounted in red, the instances for which BB-SLACK is better than BP-SLACK

Figure 6: Performance of BB-SLACK compared to BP-SLACK, in case $S = 0$ (on the top) and $S = 80$ (on the bottom)



are in green and the ones for which BP-SLACK does better are in orange. The green part is the largest for this instance family, which means that increasing S does not lead to poorer performance in that case. It is noteworthy in this figure that again, for all of the instances, BB-SLACK set with $S = 80$ is faster than BP-SLACK running on the same instances, even if the tree is not largely pruned. Indeed, up to to a tree of 1010 leaves BB-SLACK does faster than BP-SLACK while when $S = 0$ the tree needs to be smaller than 178 leaves for BB-SLACK to do faster. We see in the next of this section that S can nevertheless impact the performance of BB-SLACK, in a sense that for a same N, the smaller S the better the solutions even if that is not always true for specific subgroups of instances. This tendency is globally observed on the considered instances.

In the remainder of the section we establish a richer protocol with more instances to compare BB-SLACK and BP-SLACK not only to each other but also to well-known heuristics designed to tackle the $P||C_{\max}$ problem.

5.3 Benchmarking BBLS and BPLS against well-known heuristics

5.3.1 Protocol and algorithms

We present now an experiment established in order to evaluate the performance of BBLS and BPLS. This evaluation takes into account not only the quality of the solutions but also the computational time. The tests are performed on BB-SLACK (BBLS) and BP-SLACK (BPLS). We compare those algorithms to other well-known algorithms:

- LS-based algorithms with LPT and SLACK,
- Bin-packing based algorithms like MULTIFIT, LISTFIT for which the number of iterations is always $k = 7$. This is the configuration advised for MULTIFIT in [Coffman et al., 1978] and also applied in [Gupta & Ruiz-Torres, 2001].
- COMBINE which is a mixed strategy between the two previous kinds (MULTIFIT being called with the same number of iterations, $k = 7$),
- We also chose to evaluate LDM because it can be very efficient. Besides, it implements yet another strategy [Michiels et al., 2003].

BP-SLACK and BB-SLACK were both configured with $N = 10$ and $N = 15$ in two separate runs. However the shifting parameter S was always zero for BP-SLACK while a value of S near to the maximum $(n-N-m-1)$ was used for BB-SLACK. Recall that S matters only in a pruning scenario, that is why BP-SLACK (BPLS) does not use it. We executed these algorithms on all the instances described in table 2. The exact instances used are available online [Hadj-Djilani, 2023b].

Note that two sets of experiments, done to widen the scope of our investigations, are not presented here. First, we ran the algorithms on perfect matching instances (PMI's, instances where the optimal solution is known) but it did not give any major difference in the evaluation of our algorithms. Please refer to appendix B for more details on these results and how we built PMI's. Second, we did not include our experiments about BBLS configured with ternary trees because it turns out to be less efficient for the same computational cost as binary tree BBLS. Indeed, we tested a ternary tree based BBLS and limited the number of nodes to get the same cost than that obtained with binary binary BBLS (commentary of (1) in section 2 details how to do that). We hence had to use a smaller value for the parameter N of ternary BBLS than for binary BBLS. This implies that less jobs were assigned by the branching strategy for ternary BBLS than for binary BBLS. Precisely, we used $N_2 \in \{10,15\}$ for binary BBLS and $N_3 = \{6,9\}$ for ternary BBLS. We noticed that for more than 90% of instances listed in table 2, binary BBLS produced equal or better solutions than ternary BBLS while the latter conversely produced better or equal solutions to binary BBLS for only 70% on the instances. Furthermore, ternary BBLS was twice more computationally expensive than binary BBLS even if the full ternary tree of the former was smaller, in number of nodes, than that of binary BBLS. It suggests that the tree pruning is far less efficient with arity 3.

The algorithm implementations were all written in Python using Numpy (`https://numpy.org`). For the BBLS implementation, to optimize the computation of the L_3 bound, we used Numba (`https://numba.pydata.org`). Optimization properties 3.3, 3.4, 3.5 were enabled for BBLS but not Theorem 3.6 because it was tested efficient only when $S = 0$ (in which case it allows an acceleration of one to several orders of magnitude, approximately as much as Numba just-in-time compiling permits). The Theorem 3.7 was not used either because it did not show a significant speedup on the pruning (which obviously depends on the instances used).

The LISTFIT implementation has been parallelized. Indeed, the $4n$ MULTIFIT calls needed by LISTFIT can easily be spread on several processors/cores as threads or processes. A quick experiment allowed to verify that the parallelization of LISTFIT is worth it on the CPU used[9] if the number of jobs is greater or equal to 100 (limit under what we used a sequential LISTFIT). These details matter in the computation time measurement made further in this section.

5.3.2 The results

[9] An Intel(R) Xeon(R) Platinum 9242 equipped of 48 cores, that is 96 threads.

Figure 7: Performance of the branching heuristics relatively to well-known algorithms. All instances of Table 2 (DCU, DCNU, GR1, GR2, GR3) are grouped in the figures.

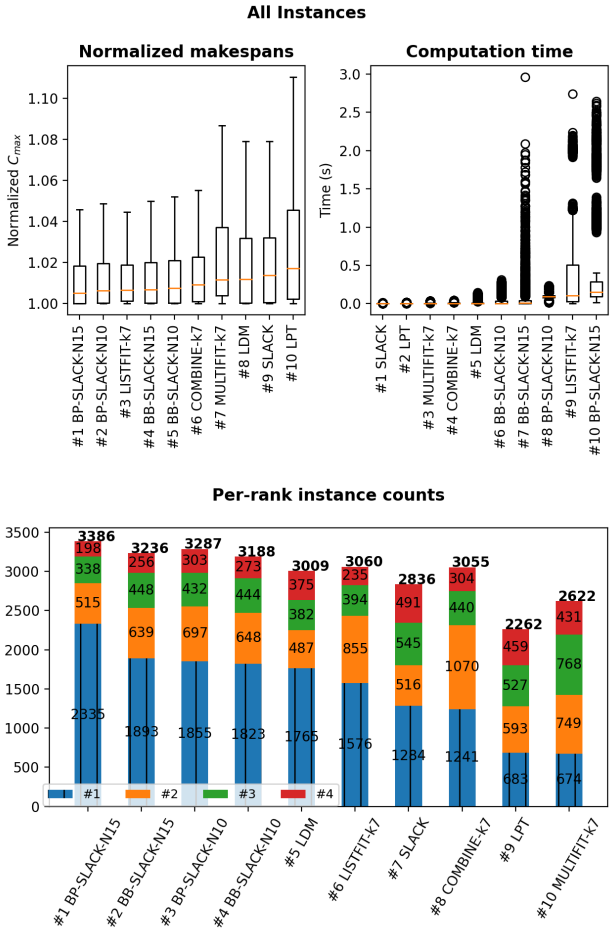


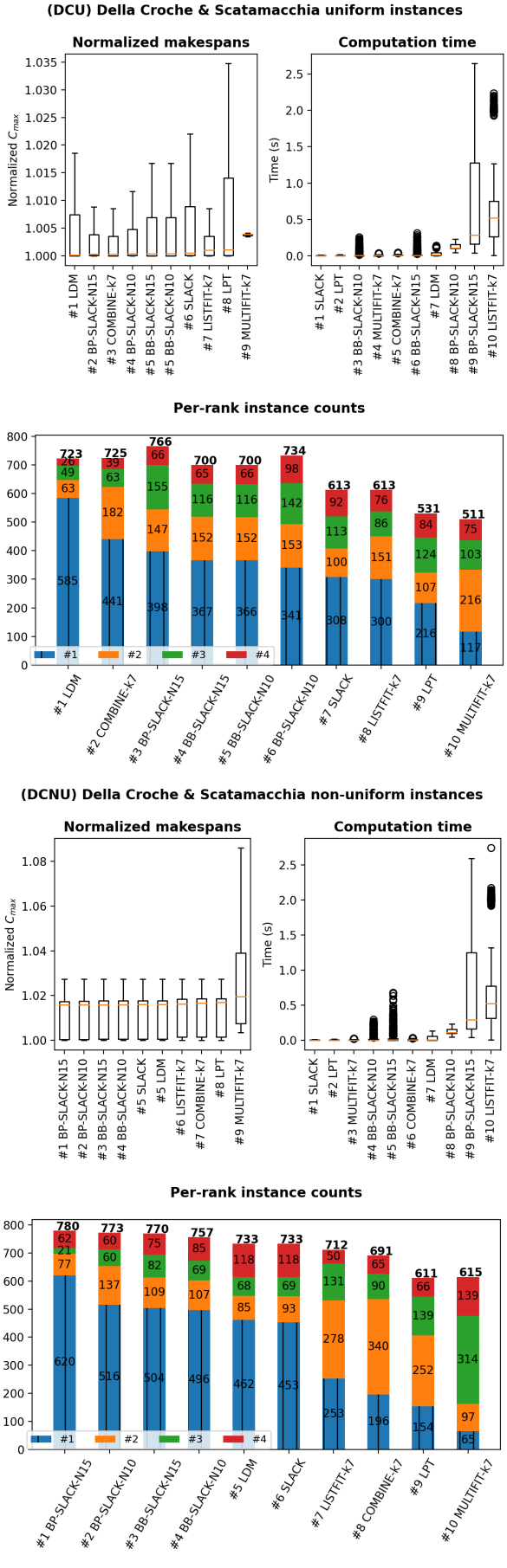
Figure 7 presents the makespans obtained with each algorithm as well as their execution times. In order to compare the results for different instances in the same figure the makespans are normalized using the lower bound L_0 (see section 3.1). Hence the ranks according to the median normalized makespans are indicated on the figure along the algorithm names which are sorted according to their rank from left to right. Two algorithms might have the same rank in case of tie. The makespan boxplot is useful as it gives an insight of the algorithm result distribution over the full set of instances. However, medians should be read carefully because it is totally possible, for two algorithms A_1 and A_2, to obtain makespan medians m_1 and m_2 such that $m_1 \leq m_2$ but with a number of better results far more important for A_2. For that reason, the stacked bar chart at the bottom of Figure 7, displays the number of instances for which an algorithm ranks first, with the best makespan, or second and so on (with possible ties here too). The ranks span from #1 to #4 since the last ranks (#5 to #10) were not shown in the figure for readability reasons. Besides, we are more interested in first ranks even if looking more carefully almost all instances are accounted in the figures. Note that on the top of each bar is printed the total number of instances for all accounted ranks. Likewise, over each sub-bar is displayed the number of instances counted for the corresponding algorithm rank listed in the legend.

All instances grouped: Figure 7 shows the results obtained with instances from all different groups (DC and GR) combined. A first general remark is that all heuristics perform very well with

median normalized makespans very close to one, that is median makespans close to the lower bound L_0. For any tested algorithm the difference between the median makespan and L_0 is smaller than 2% of L_0. The algorithms are very competitive to each other. Indeed they all have been already proven to be effective in the literature. However it is clear that the spanning of the makespans over the instances can vary significantly from one algorithm to another (e.g. about 5% of L_0 for LISTFIT to more than 10% for LPT [10]). Besides, depending on the jobs processing times, even 1% of the makespan can be interesting for the performance of a schedule. Furthermore, even if the makespan changes just a bit from an algorithm to another for a given instance, we are still interested in enhancing the performance toward the optimal solution. Anyway, out of makespan value considerations, counting the number of #1 ranks, we see significant variations among algorithms. We can see in the "per-rank instance counts" bar chart of Figure 7 that BP-SLACK and BB-SLACK give the best makespan for a number of instances significantly greater than all other algorithms. A larger value of N giving obviously the best results. A value of $S = 0$ for BP-SLACK runs shows also a significant advantage over BB-SLACK runs that almost maximize S. This advantage however comes with a larger computational cost. This can be seen in the "Computation time" boxplots, where BP-SLACK ranks 9 and 10, while BB-SLACK ranks 6 and 7, even if there are outliers with larger times for BB-SLACK configured with $N = 15$. These outliers are most likely due to a poor pruning of the corresponding solution trees by BBLS but we note that, for the most part, they do not go as high as BP-SLACK set also to $N = 15$.

[10] The figures for the normalized makespans do not show the outliers for the sake of readability but we noticed outliers were not making a difference between one algorithm to each other. The greatest outlier makespan was about 30% above L_0 for all algorithms.

Figure 8: Benchmark on DCU and DCNU instances (see table 2)



Looking at the "Normalized makespans" boxplots, we see again that BP-SLACK and BB-SLACK algorithms are top ranked, from 1 to 5. LISTFIT ranks 3, just after the BP-SLACK algorithm and before the BB-SLACK algorithm. From the bar chart we can nevertheless verify that LISTFIT has less instances than BB-SLACK ($N = 15$) for which it obtains the best makespan. On the other hand, considering the makespans of all instances, the median for LISTFIT is better than that of BB-SLACK ($N = 10, 15$).

In the following, we assess the heuristics over the different subgroups of instances. In particular we show for which instance subgroups LISTFIT manages to be better than the BB/BP-SLACK algorithms.

DC instances: The results obtained with DC instances are presented in Figure 8. Note that we also ran the experiment on equivalent instances from University of Bologna and obtained similar results (see appendix C, Figure 20).

Starting with the DCNU instances, that is the non-uniform instances, it is pretty obvious that BB-SLACK, and even more BP-SLACK, outperform all other algorithms. Whether we look at the bar chart or at the boxplots, they accumulate a clearly larger number of instances on which they ranked at the top, and reach smaller median makespans than all other algorithms.

Regarding the computation time, LISTFIT occupies the last position (rank 10) just behind BP-SLACK (ranks 8 and 9) and surprisingly BB-SLACK (N=10) managed to do faster than COMBINE and LDM. It is likely that the tree pruning was very efficient on this configuration and set of instances. According to this experiment, if we were to restrain ourselves to only BB-SLACK $N = 10$ against all other algorithms (excluding those of our contributions), we could, for the DCNU instances, have the best solutions to $P||C_{\max}$ and be ranked 4 out of 7 regarding the computation time. In this figure we also notice, as in [Della Croce & Scatamacchia, 2020], that SLACK is very efficient for that kind of instances compared to COMBINE, MULTIFIT or LISTFIT (bin-packing algorithms in general).

For uniform instances (DCU), we observe that BB-SLACK and BP-SLACK do quite good with a median normalized makespan, at rank 2 for BP-SLACK $N = 15$ and rank 3 for its number of best makespan instances. This configuration of BP-SLACK ranks just after LDM (rank 1) and just before COMBINE (rank 3) for the median makespan, while LDM is first, COMBINE second and BP-SLACK $N = 15$ third for the number of best makespan instances. Note however that the latter manages to accumulate the most important number of instances for which it is ranked in the top 4. BP-SLACK $N = 10$ produces less good results than LDM and COMBINE too and is ranked 4 regarding the median makespan and 6 for the number of best makespan instances, behind BB-SLACK $N = 15$. This one ranks better than BB-SLACK $N = 10$ but with almost the same performance because the former is the best for 367 instances, while the latter is for 366 instances. It is however worth noting that, on this set of DCU instances, BB-SLACK $N = 10$ median computation time is smaller than that of MULTIFIT, COMBINE, LDM and LISTFIT. According to the median time BB-SLACK $N = 15$ is faster than LDM and LISTFIT. However, BP-SLACK configurations, $N = 15$ and $N = 10$ (with $S = 0$), are slower than all other algorithms except LISTFIT which is the slowest of all.

In Figure 9, we see that LDM becomes more and more dominant on DCU instances, when b grows from 100 to 10000. It is also noticeably true when n grows as shown in table 3. An explanation for these two facts is to find in the LDM strategy. First, LDM subdivides the jobs into smaller packets of jobs, the partial schedules. When new assignments of jobs are made by merging two partial schedules, the whole set of n jobs is taken into account (by comparing the differences of machine loads among partial schedules). On the other side, BBLS and BPLS only consider the loads of the two least loaded machines when assigning the next job with no consideration at all to other jobs yet to assign. Going blindly on the remaining jobs to assign, is a risk that is more and more important as n grows. Similarly, when b grows, the difference between two p_j's has a greater chance to be larger so that a wrong choice of assignment might be more penalizing on the machine load balancing. Besides, for a same n, the availability of smaller jobs becomes less probably when b is larger (remember that here we consider

Figure 9

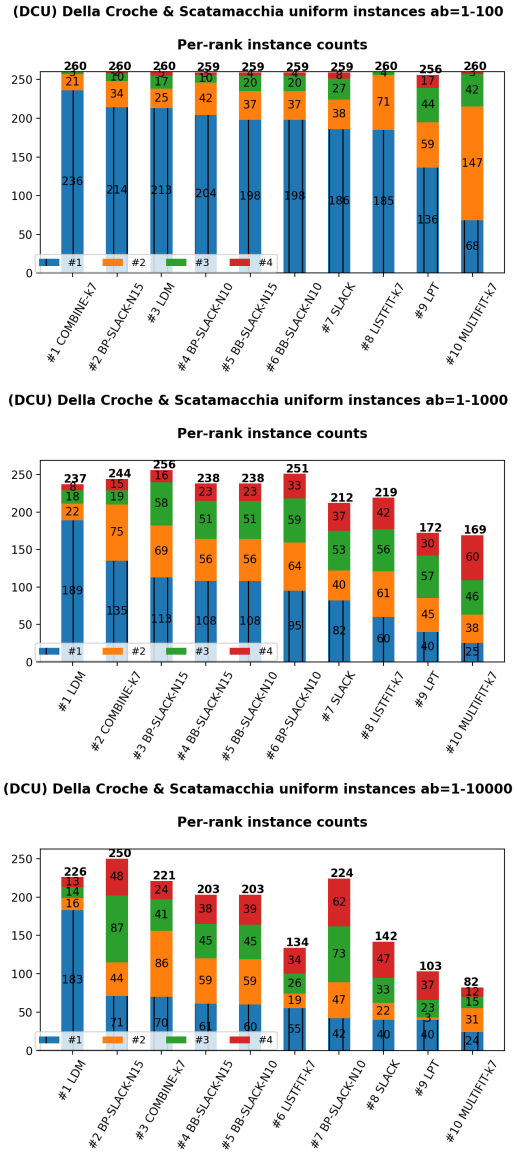
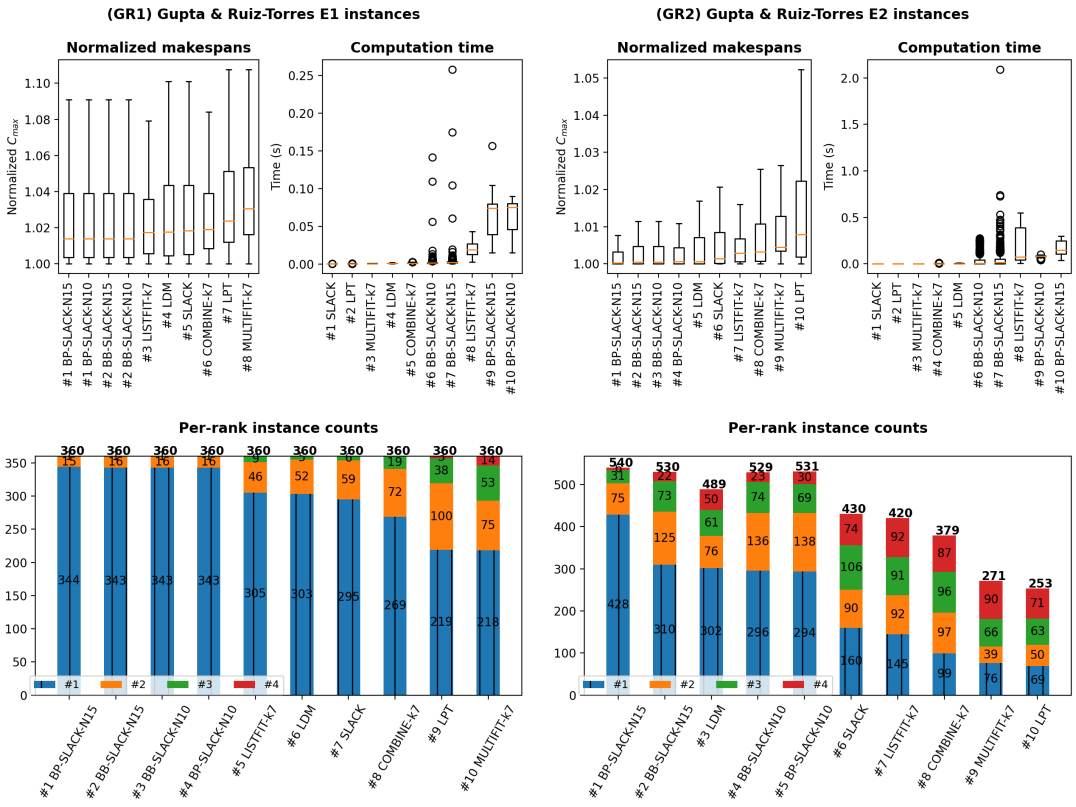


Table 3: LDM versus BP/BB-SLACK wins w.r.t. n

n	number of instances	LDM	BP-SLACK $N = 15$	BP-SLACK $N = 10$	BB-SLACK $N = 15$	BB-SLACK $N = 10$
10	60	54	55	55	55	55
50	180	76	86	131	84	84
100	180	100	35	47	43	44
500	180	175	70	70	78	78
1000	180	180	95	95	106	106

uniform instances). But smaller jobs are precisely what is needed to balance differences of machine loads implied by larger jobs.

Figure 10: Benchmark on GR1 and GR2 instances (see table 2).

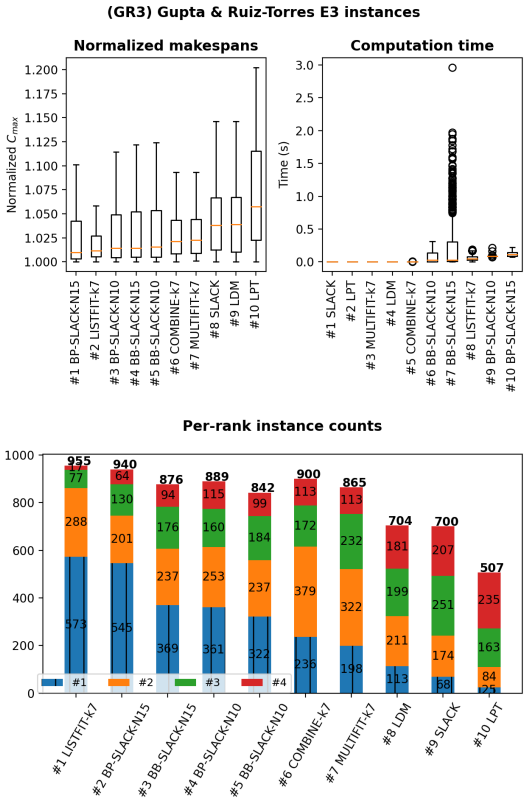


In section 6 we show how MBBLS2, introduced in 4.2, can mitigate those penalties on BBLS, by working on smaller packets of jobs first, as LDM does, and then grouping the sub-schedules obtained into a complete one.

GR instances: For GR instances the rankings are also subtly different. First we consider GR1 and GR2 on Figure 10. It is clear that both BP-SLACK and BB-SLACK outperform all other algorithms for all our metrics, except for the computation time. However BB-SLACK ($N = 10, 15$) is faster than LISTFIT which produced significantly less good solutions, especially for GR2 instances.

For GR3 instances, in Figure 11, the interesting point is that LISTFIT outperforms most of our algorithms on median makespan and all of them on number of best makespan instances. However its median execution time is larger than BB-SLACK's. We zoomed on specific subgroups of GR3 instances to understand where exactly LISTFIT achieves to be the best algorithm.

Figure 11: Benchmark on GR3 instances (see table 2).



In Figures 12 to 15 we clearly see that LISTFIT gets its advantage when the interval of processing times is $[a = 100,\ b = 200]$ (see bottom subplots), while the interval $[a = 1,\ b = 100]$ (see top subplots) fits better to BP-SLACK and BB-SLACK best performance. When $m \geq 8$, there is a slight difference because BP/BB algorithms do less well when m becomes larger. Indeed, with $m = 10$, LISTFIT ranks first, even if $a = 1$ (with 84 instances for which LISTFIT is the best and 80 for BP-SLACK $N = 15$).

Figure 12: Focus on GR3 instances, $m = 3$ (see table 2)

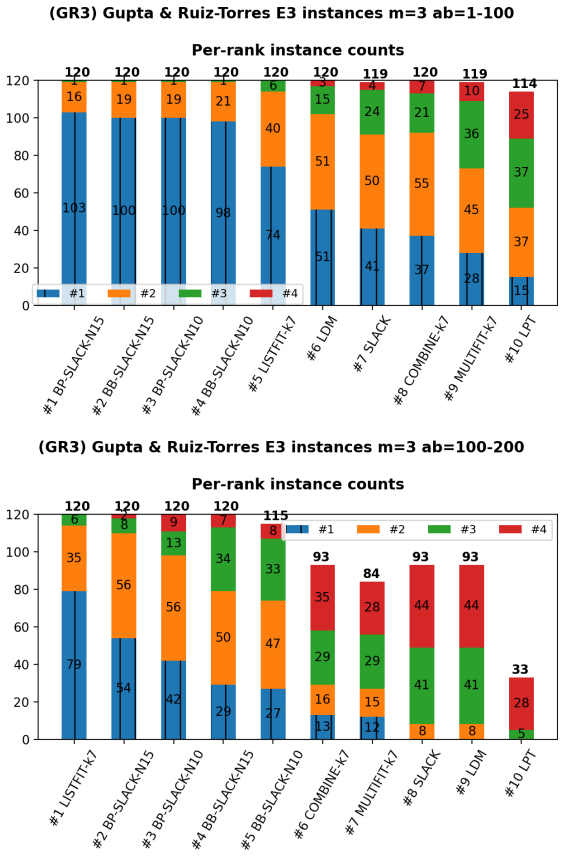


Figure 14: Focus on GR3 instances, $m = 8$ (see table 2)

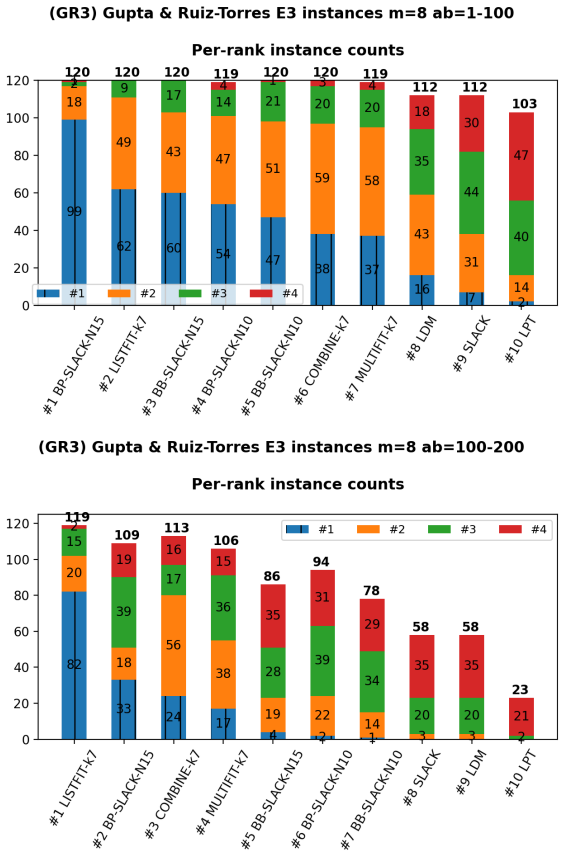


Figure 13: Focus on GR3 instances, $m = 5$ (see table 2)

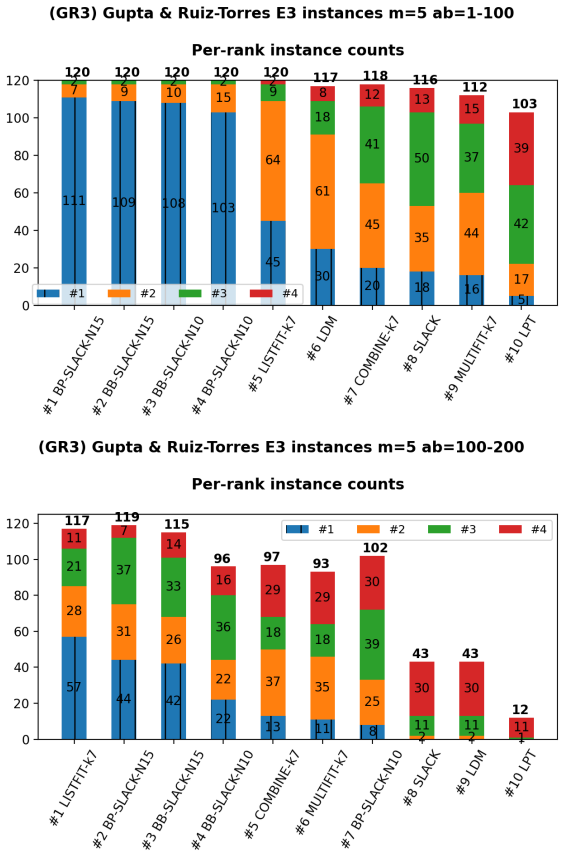
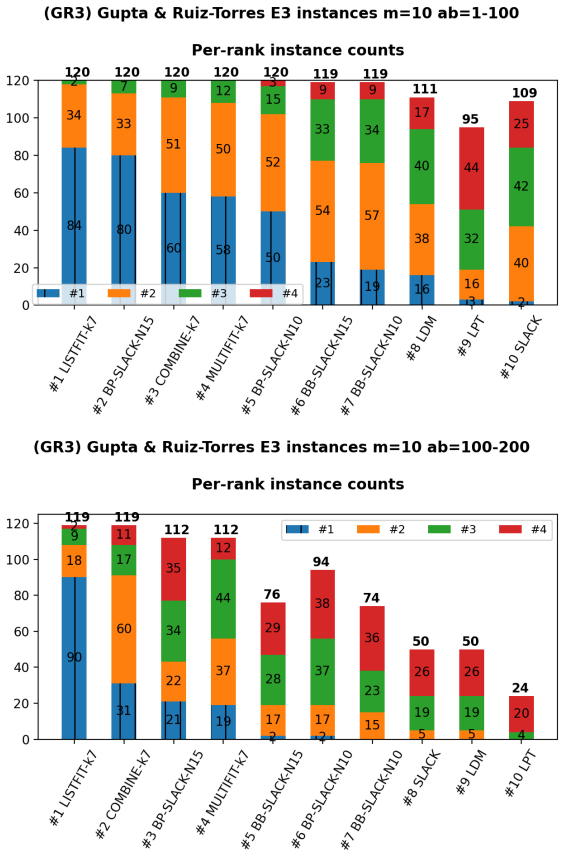


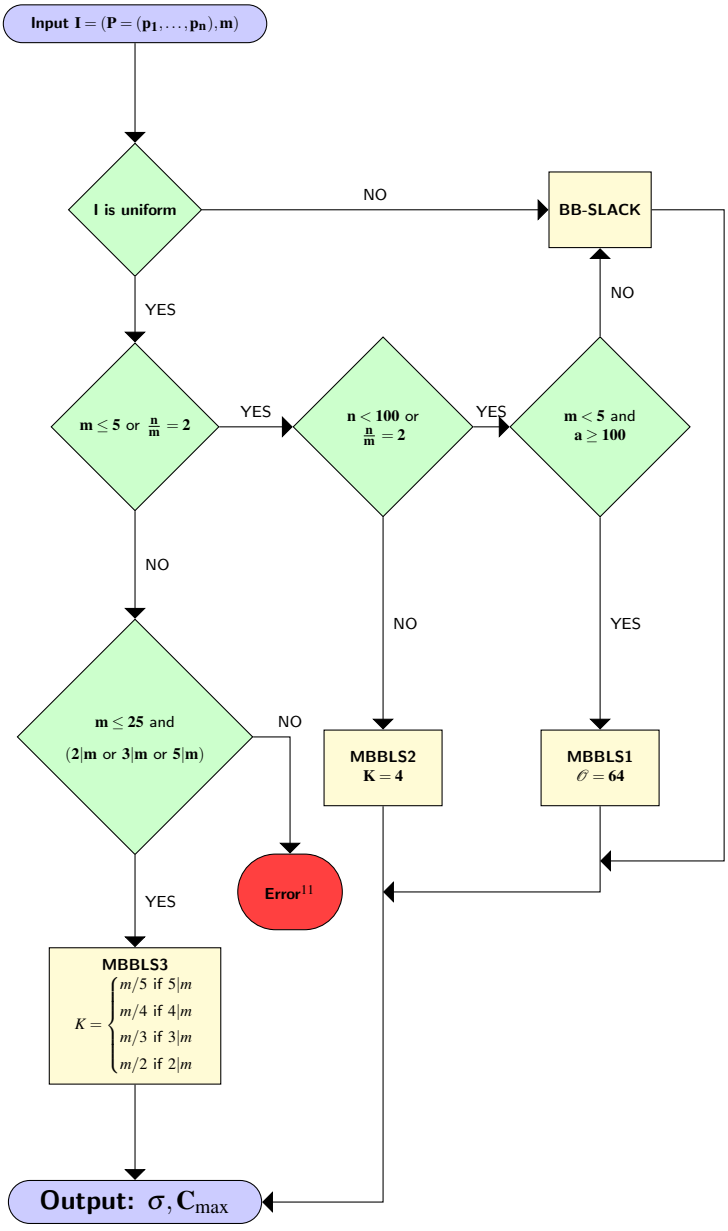
Figure 15: Focus on GR3 instances, $m = 10$ (see table 2)



The fact that our algorithms are less efficient when $a = 100$ is to find in the other LS-based algorithms that seem to all follow this tendency, contrary to FFD based algorithms that behave well in this configuration, LISTFIT being at the top. Preventing any smaller p_j puts LS-based algorithms in difficult situation to balance machines loads because, as mentioned before in LDM performance comparison, the smaller jobs are necessary in this purpose. About the effect of the m value, it is clear that, when m is greater, there is much of a chance that the optimal choice of assignment for each job is another machine than the first or second available ones. Therefore the likelihood to get less efficient solutions for BP-SLACK and BB-SLACK is larger when m grows. MBBLS1 and MBBLS2 introduced in 4.1 and 4.2 are used in section 6 to overcome these LS-based algorithm disadvantages.

6 An ad hoc MBBLS

Figure 16: Ad hoc MBBLS flowchart



Starting from the results of the experiment presented in section 5.3, we propose an ad hoc MBBLS heuristic using the three heuristics introduced in 4.1, 4.2, 4.3. The flowchart, presented in Figure 16, shows how this heuristic works. Basically, it calls MBBLS1, MBBLS2, MBBLS3 or just BB-SLACK in order to maximize the quality of solutions for all the tested instances. Indeed, BB-SLACK performs very well on non-uniform instances (DCNU), so it was maintained for those instances, but has clear weaknesses on other instances. On GR3 instances BB-SLACK does not perform as well as LISTFIT when m grows so, for $m > 5$, MBBLS calls MBBLS3 which divides the $P||C_{\max}$ problem into sub-problems that use a smaller number of machines. On the same subset of instances, we noticed that LISTFIT is also better if the lower bound a, chosen to draw the p_j's, is greater or equal to 100, hence, in that case, we call MBBLS1. Two exceptions should be noticed in the flowchart. First, when $m = 5$, even if $a \geq 100$, rather to call MBBLS1, MBBLS calls BB-SLACK. The reason for this choice is simply that BB-SLACK does better on this subset. For the same reason, a second exception is made about MBBLS3. It is not called when $\frac{n}{m} = 2$, even if $m > 5$. Among tested instances the only precise case that verifies this condition is when $m = 25$ and $n = 50$. Indeed, on these instances, we observed that BB-SLACK is more efficient than MBBLS3. We also use MBBLS2 on cases where $n \geq 100$ since that is the point of this heuristic to run on instances that contain a large number of jobs. That choice was made to manage outperforming LDM on DCU instances.

By running MBBLS as defined by the flowchart, we were able to outperform LDM and LISTFIT on instances for which BB-SLACK alone was not able to do so (as shown in 5.3.2). Figures 17 and 18 [12] show the results respectively for the GR3 and DCU instances. In Figure 17, apart from our heuristics, only LISTFIT is considered, because that is the challenging algorithm identified in 5.3.2 for GR3 instances (see Figure 11). LDM is likewise included in Figure 18 regarding the results obtained for DCU instances (see Figure 8). Another point that requires attention, is the increase of N up to 16 for MBBLS. The reason for this configuration is the greater number of wins for LDM compared to MBBLS $N = 15$. Note however that, even if MBBLS $N = 15$ has less wins than LDM, the former more often ranks in the two first algorithms than the latter. Anyway, MBBLS $N = 16$ is enough to produce more wins than LDM. Finally, note that although LDM is ranked first according to its median normalized makespan (Figure 18) the spanning of its makespans, is quite larger than that obtained with any configuration of MBBLS (even $N = 10$). MBBLS $N = 17$, while not present in the figure, was also ran and gave a median normalized $C_{\max}$ lower than that of LDM. This allows us to conclude that the MBBLS approach, with increasing values of N can outperform LDM.

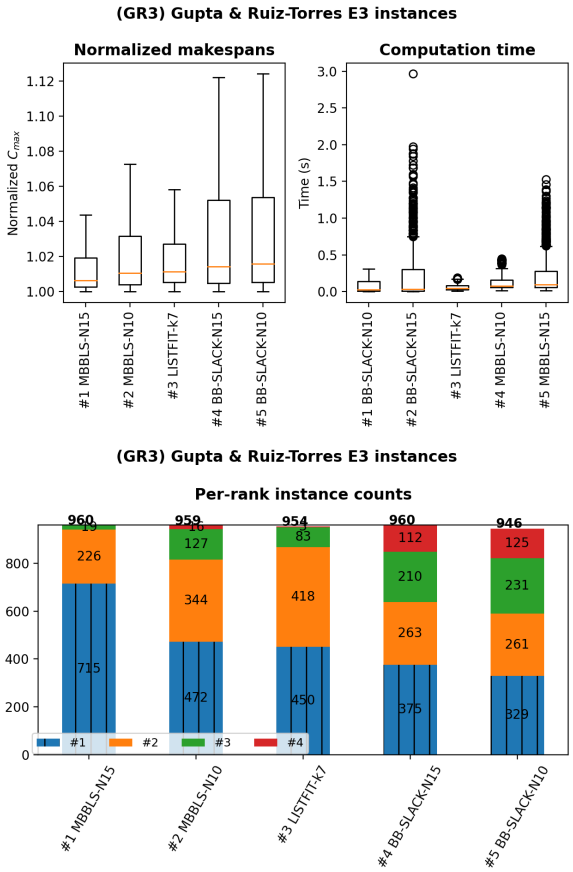
Conclusion

In this paper we show how a modification in the LS algorithm, considering an assignment on other machines than the first available one (actually only the second one), can improve the makespan obtained for a $P||C_{\max}$ problem. We show that branching strategies can easily be derived, as in BLS algorithm, although the cost is exponential in the number of jobs n. A parameter N, that limits the number of jobs treated by BLS, the others being handled by the classic LS, is our first step to speed up the method. This parameter N is used without losing the algorithm interest regarding the quality of solutions. Then we show, with BPLS, that a basic parallelization of the alternatives encoded in the solution search tree allows to speed up significantly the branching process. Besides, it makes possible to scale up to the available computational power of a multi-core CPU or a computation grid, adjusting the quality of solutions through the parameter N. Next, we develop a Branch & Bound algorithm, named BBLS, on the same principle introduced in [Dell'Amico & Martello, 1995] by allowing to prune out from the tree, subsets of

[11] The error case might happen only if MBBLS receives an instance that is not from the tested instances presented in table 2. It is however totally envisionable to extend MBBLS3 to any $m > 25$ with no divisibility constraint for other families of instances.

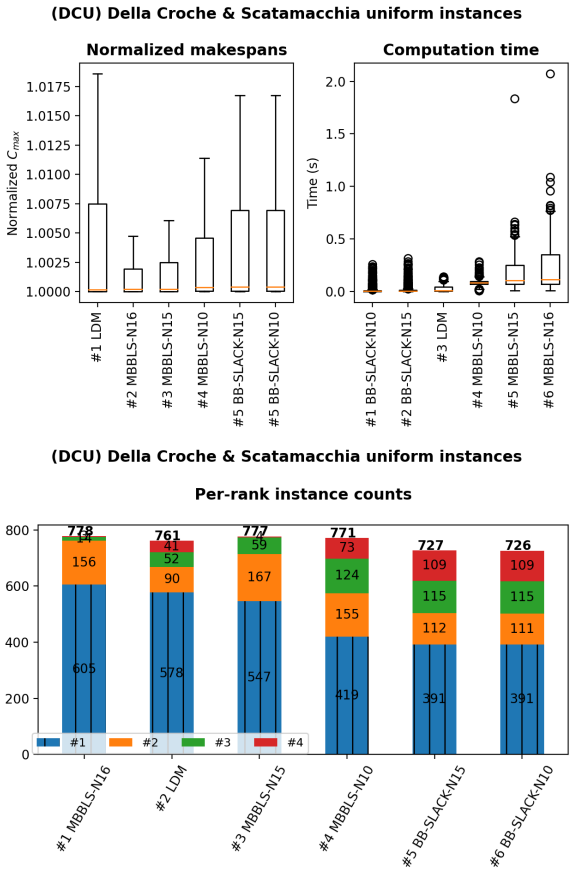
[12] We also ran the experiment on equivalent instances from University of Bologna website and obtained similar results (see appendix C, Figure 21)

Figure 17: MBBLS versus LISTFIT on GR3 instances (see table 2)



solutions for which equivalent transformed instances indicate lower bounds that are not worthy relatively to the best solution so far. A shifting parameter S, added to BBLS, allows to speed up the computation of lower bounds and enhance the pruning effect of the algorithm. We propose additional theorems in the goal to optimize BBLS. To assess the performance of the branching heuristics, benchmarks include many known algorithms as challengers. They show clearly, on literature instances, that BB-SLACK and BP-SLACK can really outperform the other algorithms on the vast majority of these instances. On the other hand, we also identify where they are not so efficient compared to LISTFIT (on instances whose job times start from 100 and above or when m grows) or LDM (on instances for which n or job times are larger). Making multiple calls to BBLS, varying the order of the jobs, or relying on sub-problems with a smaller number of machines or jobs, we have been able to define an ad hoc heuristic, named MBBLS, purposed to beat LDM and LISTFIT on the problematic subsets of instances. Appendices are joined to the paper in order to confirm or complete the results shown in the body of the paper on other instances, as perfect matching instances (appendix B), instances found online (appendix C) or instances created through independent generators (appendices, D). Future works are envisioned as benchmarking our heuristics on other instance families, as bin-packing instances. Evaluating these heuristics against the metaheuristics mentioned in introduction should also be interesting.

Figure 18: MBBLS versus LDM on DCU instances (see table 2)



Acknowledgements

We gratefully acknowledge the support of the Centre Blaise Pascal's IT test platform at ENS de Lyon (Lyon, France) for computing facilities. The platform operates the SIDUS solution [Emmanuel Quemener, 2014].

References

[Chen et al., 2012] Chen, J., Pan, Q.-K., Wang, L., & Li, J.-Q. (2012). A hybrid dynamic harmony search algorithm for identical parallel machines scheduling. *Engineering Optimization*, 44(2), 209–224.

[Coffman et al., 1978] Coffman, Jr, E. G., Garey, M. R., & Johnson, D. S. (1978). An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1), 1–17.

[Della Croce & Scatamacchia, 2020] Della Croce, F. & Scatamacchia, R. (2020). The longest processing time rule for identical parallel machines revisited. *Journal of Scheduling*, 23(2), 163–176.

[Dell'Amico & Martello, 1995] Dell'Amico, M. & Martello, S. (1995). Optimal scheduling of tasks on identical parallel processors. *ORSA Journal on Computing*, 7(2), 191–200.

[Emmanuel Quemener, 2014] Emmanuel Quemener, M. C. (2014). Sidus, the solution for extreme deduplication of an operating system. *The Linux Journal*.

[França et al., 1994] França, P. M., Gendreau, M., Laporte, G., & Müller, F. M. (1994). A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Computers & operations research*, 21(2), 205–210.

[Frangioni et al., 2004] Frangioni, A., Necciari, E., & Scutella, M. G. (2004). A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. *Journal of Combinatorial Optimization*, 8, 195–220.

[Garey & Johnson, 1979] Garey, M. R. & Johnson, D. S. (1979). Computers and intractability wh freeman and company. *New York*, (pp. 209–210).

[Glass et al., 1994] Glass, C. A., Potts, C., & Shade, P. (1994). Unrelated parallel machine scheduling using local search. *Mathematical and Computer Modelling*, 20(2), 41–52.

[Graham, 1969] Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2), 416–429.

[Graham et al., 1979] Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. 5, 287–326.

[Gupta & Ruiz-Torres, 2001] Gupta, J. N. & Ruiz-Torres, A. J. (2001). A listfit heuristic for minimizing makespan on identical parallel machines. *Production Planning & Control*, 12(1), 28–36.

[Hadj-Djilani, 2023a] Hadj-Djilani, H. (2023a). B*ls exp code. Software on Zenodo, `https://doi.org/10.5281/zenodo.10409092`.

[Hadj-Djilani, 2023b] Hadj-Djilani, H. (2023b). Dc & gr instances. Dataset on Zenodo, `https://doi.org/10.5281/zenodo.10203754`.

[Hadj-Djilani, 2023c] Hadj-Djilani, H. (2023c). Gera instances. Dataset on Zenodo, `https://doi.org/10.5281/zenodo.10203980`.

[Hadj-Djilani, 2023d] Hadj-Djilani, H. (2023d). Perfect matching instances. Dataset on Zenodo, `https://doi.org/10.5281/zenodo.10400559`.

[Hadj-Djilani, 2023e] Hadj-Djilani, H. (2023e). Pimsgen instances. Dataset on Zenodo, `https://doi.org/10.5281/zenodo.10204011`.

[Hochbaum & Shmoys, 1987] Hochbaum, D. S. & Shmoys, D. B. (1987). Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1), 144–162.

[Karmarkar & Karp, 1982] Karmarkar, N. & Karp, R. M. (1982). The differencing method of set partitioning.

[Kashan & Karimi, 2009] Kashan, A. H. & Karimi, B. (2009). A discrete particle swarm optimization algorithm for scheduling parallel machines. *Computers & Industrial Engineering*, 56(1), 216–223.

[Lee & Massey, 1988] Lee, C.-Y. & Massey, J. D. (1988). Multiprocessor scheduling: combining lpt and multifit. *Discrete applied mathematics*, 20(3), 233–242.

[Martello & Toth, 1990] Martello, S. & Toth, P. (1990). Lower bounds and reduction procedures for the bin packing problem. *Discrete applied mathematics*, 28(1), 59–70.

[Michiels et al., 2003] Michiels, W., Korst, J., Aarts, E., et al. (2003). Performance ratios for the karmarkar-karp differencing method. *Electronic Notes in Discrete Mathematics*, 13, 71–75.

[Min & Cheng, 1999] Min, L. & Cheng, W. (1999). A genetic algorithm for minimizing the makespan in the case of scheduling identical parallel machines. *Artificial Intelligence in Engineering*, 13(4), 399–403.

[Yibao et al., 2002] Yibao, C., Jianchu, Y., & Yifang, Z. (2002). Ant system based optimization algorithm and its applications in identical parallel machine scheduling. *Journal of Systems Engineering and Electronics*, 13(3), 78–85.

A Proof of theorem 3.6

Proof. Denote $J_{1,k}, J_{2,k}, J_{3,k}$ the subsets for the instance I'_k of node k for a given $\bar{p}$, and $J_{1,k-1}, J_{2,k-1}, J_{3,k-1}$ the subsets for the parent node $(k-1)$ and the same $\bar{p}$. Because $L = C^b_{\max} - 1$ and $p_{i_k} = C_{i_k} + p_a \leq L/2$ we have $C_{i_k} \leq L/2$ and $p_a \leq L/2$. Denote $J_4(\bar{p}) = \{j | p_j < \bar{p}\}$ the subset of jobs that are ignored in B_α, B_β and notice that there are five possible cases for the processing times p_{i_k}, C_{i_k}, p_a:

$$p_{i_k}, C_{i_k}, p_a \in J_{3,k} \times J_{3,k-1} \times J_{3,k-1} \tag{8}$$

$$p_{i_k}, C_{i_k}, p_a \in J_{3,k} \times J_{3,k-1} \times J_{4,k-1} \tag{9}$$

$$p_{i_k}, C_{i_k}, p_a \in J_{3,k} \times J_{4,k-1} \times J_{3,k-1} \tag{10}$$

$$p_{i_k}, C_{i_k}, p_a \in J_{3,k} \times J_{4,k-1} \times J_{4,k-1} \tag{11}$$

$$p_{i_k}, C_{i_k}, p_a \in J_{4,k} \times J_{4,k-1} \times J_{4,k-1} \tag{12}$$

Denote $R(B_\alpha, k-1) = \frac{\sum_{j \in J_{3,k-1}} p_j - (L|J_{2,k-1}| - \sum_{i \in J_{2,k-1}} p_j)}{L}$ and $R(B_\beta, k-1) = \frac{|J_{3,k-1}| - \sum_{j \in J_{2,k-1}} \left\lfloor \frac{L-p_j}{\bar{p}} \right\rfloor}{\left\lfloor \frac{L}{\bar{p}} \right\rfloor}$ that are terms of the parent node I'_{k-1} B_α and B_β formulas and consider each one of the aforementioned cases.

- Case (8):

  Since

  $$\begin{aligned} J_{1,k} &= J_{1,k-1}, \\ J_{2,k} &= J_{2,k-1}, \\ J_{3,k} &= (J_{3,k-1} \cup \{p_{i_k}\}) \setminus \{p_a, C_{i_k}\} \end{aligned}$$

  and because

  $$B_{\beta,k} = |J_{1,k-1}| + |J_{2,k-1}| + max(0, \lceil R(B_{\beta,k-1}) - \frac{1}{\left\lfloor \frac{L}{\bar{p}} \right\rfloor} \rceil)$$

  We have:

  $$\begin{aligned} B_{\alpha,k} &= B_{\alpha,k-1} \\ B_{\beta,k} &\leq B_{\beta,k-1} \end{aligned}$$

- Case (9):

  Since

  $$\begin{aligned} J_{1,k} &= J_{1,k-1}, \\ J_{2,k} &= J_{2,k-1}, \\ J_{3,k} &= (J_{3,k-1} \cup \{p_{i_k}\}) \setminus \{C_{i_k}\} \end{aligned}$$

  and because

  $$B_{\alpha,k} = |J_{1,k-1}| + |J_{2,k-1}| + max(0, \lceil R(B_{\beta,k-1}) + \frac{p_a}{L} \rceil)$$

  We have:

$$B_{\alpha,k} = \begin{cases} B_{\alpha,k-1} \\ \text{ if } \lceil R(B_{\alpha,k-1}) \rceil - R(B_{\alpha,k-1}) \geq \frac{p_a}{L} > 0 \\ B_{\alpha,k} = B_{\alpha,k-1} + 1 \text{ otherwise} \end{cases}$$
$$B_{\beta,k} = B_{\beta,k-1}$$

- Case (10):

  Since

  $$J_{1,k} = J_{1,k-1},$$
  $$J_{2,k} = J_{2,k-1},$$
  $$J_{3,k} = (J_{3,k-1} \cup \{p_{i_k}\}) \setminus \{p_a\}$$

  and because

  $$B_{\alpha,k} = |J_{1,k-1}| + |J_{2,k-1}| + max(0, \lceil R(B_{\beta,k-1}) + \frac{C_{i_k}}{L} \rceil)$$

  We have:

  $$B_{\alpha,k} = \begin{cases} B_{\alpha,k-1} \\ \text{ if } \lceil R(B_{\alpha,k-1}) \rceil - R(B_{\alpha,k-1}) \geq \frac{C_{i_k}}{L} > 0 \\ B_{\alpha,k} = B_{\alpha,k-1} + 1 \text{ otherwise} \end{cases}$$
  $$B_{\beta,k} = B_{\beta,k-1}$$

- Case (11):

  Since

  $$J_{1,k} = J_{1,k-1},$$
  $$J_{2,k} = J_{2,k-1},$$
  $$J_{3,k} = J_{3,k-1} \cup \{p_{i_k}\}$$

  and because

  $$B_{\alpha,k} = |J_{1,k-1}| + |J_{2,k-1}| + max(0, \lceil R(B_{\beta,k-1}) + \frac{p_{i_k}}{L} \rceil)$$
  $$B_{\beta,k} = |J_{1,k-1}| + |J_{2,k-1}| + max(0, \lceil R(B_{\beta,k-1}) + \frac{1}{\lfloor \frac{L}{\bar{p}} \rfloor} \rceil)$$

  We have:

  $$B_{\alpha,k} = \begin{cases} B_{\alpha,k-1} \\ \text{ if } \lceil R(B_{\alpha,k-1}) \rceil - R(B_{\alpha,k-1}) \geq \frac{p_{i_k}}{L} > 0 \\ B_{\alpha,k} = B_{\alpha,k-1} + 1 \text{ otherwise} \end{cases}$$
  $$B_{\beta,k} = \begin{cases} B_{\beta,k-1} \\ \text{ if } \lceil R(B_{\beta,k-1}) \rceil - R(B_{\beta,k-1}) \geq \frac{1}{\lfloor \frac{L}{\bar{p}} \rfloor} > 0 \\ B_{\beta,k} = B_{\beta,k-1} + 1 \text{ otherwise} \end{cases}$$

- Case (12):

  Since

  $$J_{1,k} = J_{1,k-1},$$
  $$J_{2,k} = J_{2,k-1},$$
  $$J_{3,k} = J_{3,k-1}$$

  We have:

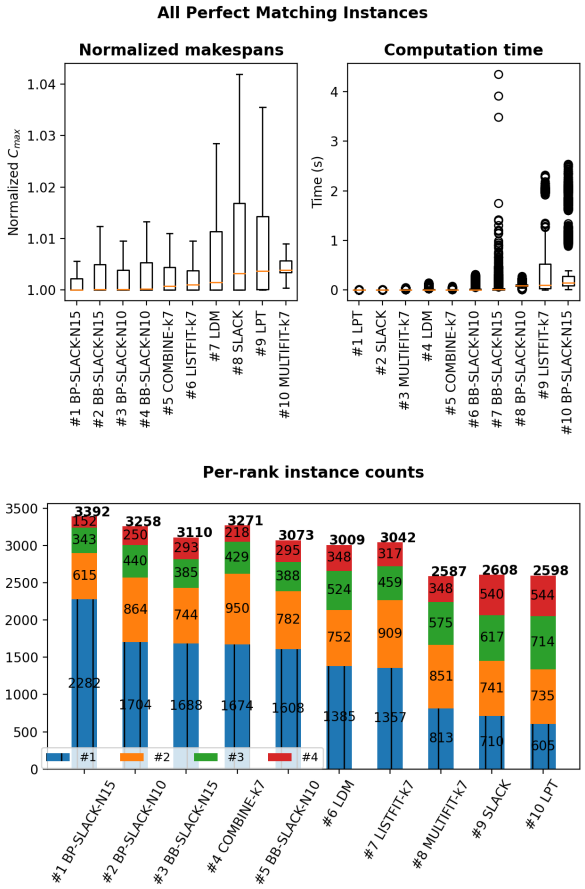
$$B_{\alpha,k} = B_{\alpha,k-1}$$
$$B_{\beta,k} = B_{\beta,k-1}$$

In all cases from (8) to (12) the bounds $B_{\alpha,k}, B_{\beta,k}$ are always such that $B_{\alpha,k} \leq B_{\alpha,k-1}+1, B_{\beta,k} \leq B_{\beta,k-1}+1$, so recalling that in the theorem we must verify $B_{\alpha,k-1} < m$ and $B_{\beta,k-1} < m$ for any $\bar{p}$ we can assert that $B_{\alpha,k} \leq m$ and $B_{\beta,k} \leq m$, hence according to (7) $C^b_{\max}$ is not a L_3 lower bound for I'_k and the node k is not to be pruned out.

□

B Results on perfect matching instances

Figure 19: Overall performance on PMI's of our branching heuristics relatively to well-known algorithms



This appendix presents the results obtained with perfect matching instances respecting the exact same (m,n) combinations as instances presented in table 2 but with a different law for picking the p_j's. Indeed, a perfect matching instance (PMI) is an instance for which it exists a solution schedule that is such that $C_{max} = \frac{\sum_{1\leq j\leq n} p_j}{m} = L_0$ which is also the completion time of all machines. The main interest of this kind of instances is that the optimal makespan is easily computable. Our PMI's were built as follows:

1. Generate an instance $I = (\{p_j\}_{1\leq j\leq n}, m)$ as in table 2.

2. Solve it with LPT and save $C_{\max}^{LPT}$. The goal is to obtain a reference makespan for our PMI.

3. For builing the PMI start by creating the m blocks representing the machine completion times all equal to $C_{\max}^{LPT}$. Then cut these blocks in random positions until n sub-blocks are obtained. The n sub-blocks identify the job processing times of the PMI.

Note that a simpler method is to compute a LPT schedule and then fills the gaps to C_{max}^{LPT} for each machine by introducing new jobs or by enlarging pre-existing ones. However this method is kind of biased because it tends to advantage LPT by design for much of the instances.

The Figure 19 presents the results obtained with PMI's running the same experiment as in 5.3. The exact instances used to produce this figures are available online [Hadj-Djilani, 2023d]. The main change we notice is that COMBINE is a way more efficient with these PMI's. It remains that BB-SLACK and BP-SLACK algorithms stay in the top five, as in 7, according to both median normalized makespans and counts of first ranks. Because we use PMI's we are able to see that the gap between solutions and optima is about 1% of $L_0 = C_{max}^*$ for BP-SLACK runs and a little more for BB-SLACK's. For comparison this gap can go up to 4% in SLACK solutions.

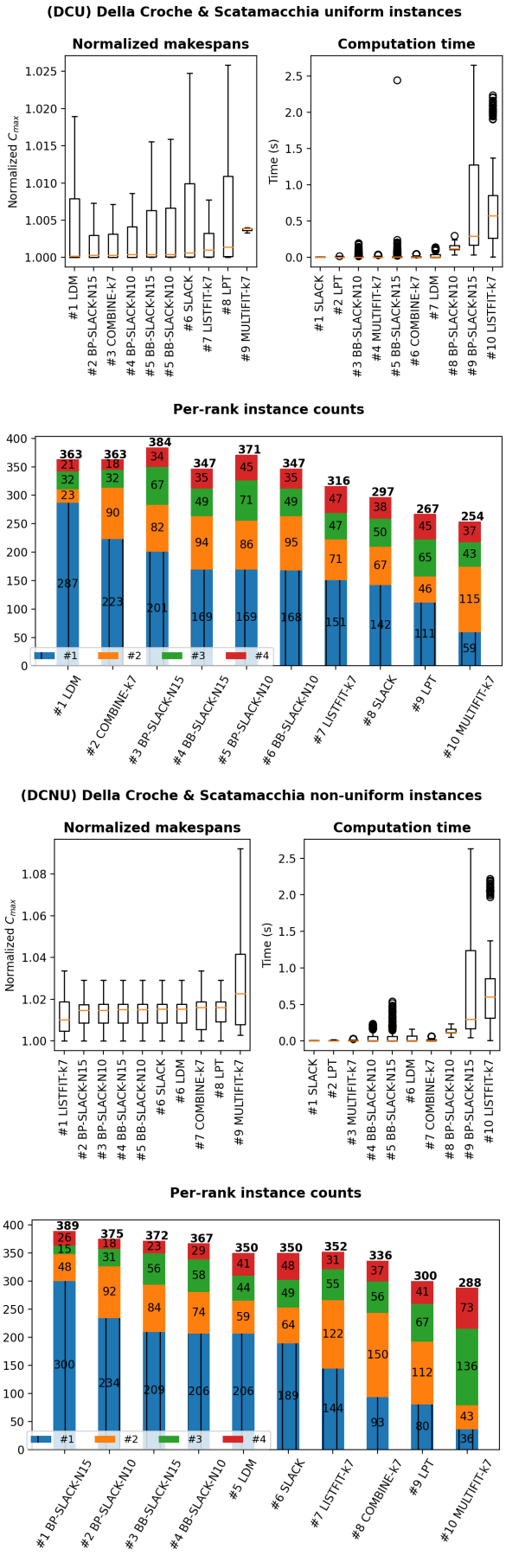
C Results on instances from University of Bologna

The figures 20 and 21 are the results obtained with instances found on the website of the University of Bologna in this page:
`https://site.unibo.it/operations-research/en/research/library-of-codes-and-instances-1`[13]

We used them to run the same experiment as in figures 8 and 18 discussed respectively in 5.3.2 and 6. Note that in our experiments we used 20 instances per valid combination of instance parameters while for the instances here only 10 instances per combination were generated. That's why we have twice more instances in our paper experiments. The results are quite similar to what obtained in the body of the paper. We only note that LISTFIT manages to make the best median normalized makespan for non-uniform instances in Figure 20 while it was only ranked 6 with our set of instances as shown in Figure 8.

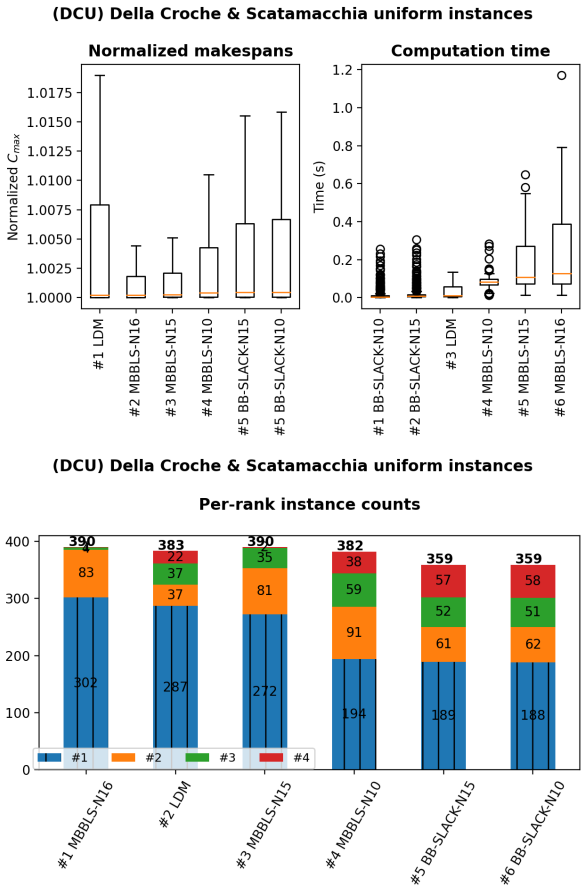
[13] In the section named "Library of Instances on the P||Cmax Problem". The instances are located in the archived linked here `https://site.unibo.it/operations-research/en/research/library-of-codes-and-instances-1/cmax.zip/@@download/file/cmax.zip`.

Figure 20: Benchmark on DCU and DCNU instances from University of Bologna (see table 2)



Nevertheless it doesn't change its rank in "Per-rank instance counts" bar chart which is 7 both in Figure 20 and Figure 8.

Figure 21: MBBLS versus LDM on DCU instances from University of Bologna (see table 2)



D Results on Gera and PIMSGen instances

Gera and PIMSGen are generators of $P||C_{max}$ instances provided by CommaLAB, University of Pisa [14].

Gera generates 10 random instances for each valid combination of $(m,n) \in \{5,10,25\} \times \{10,50,100,500,1000\}$. The p_j's are drawn in $[a = 1, b]$ according to different distribution:

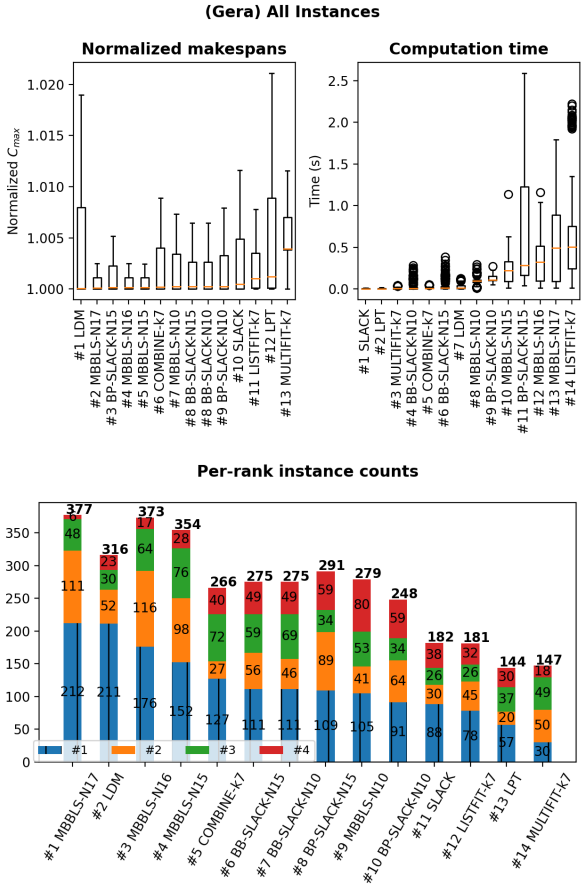
- A uniform law for which $b = 10000$ (geraun program).
- A normal law for which $b = 10000$ (geran program).
- An exponential law for which $b = 1000$ (gerae program).

It provides a total of 390 instances per run (130 per distribution). We performed the same experiment as defined in 5.3.1 using these instances. We grouped the results on all instances in the Figure 22.

Although the benchmark gives results that differ from what obtained in 5.3.2 and 6 we can see that MBBLS $N = 17$ manages to be at the top of all heuristics. Indeed, the number of first ranks

[14] At time of writing, the generators are downloaded here: `https://commalab.di.unipi.it/datasets/MS/`

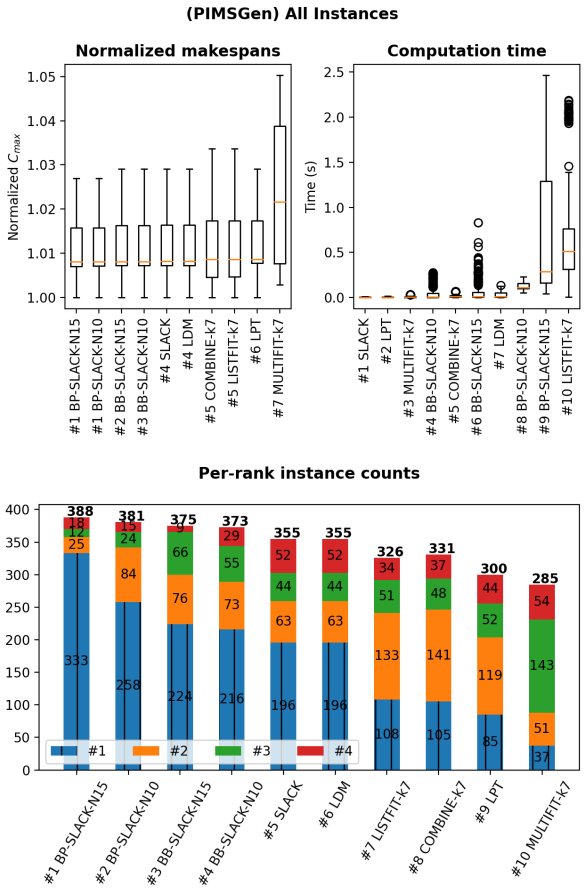
Figure 22: All Gera instances (normal, exponential and uniform laws) are grouped in this figure to show the overall performance of our branching heuristics relatively to well-known algorithms. The exact instances used to produce this figure are available online [Hadj-Djilani, 2023c].



obtained on instances is the greatest of all tested algorithms. Besides, even if LDM has the smallest median normalized makespan, the spanning of its makespans is much larger than that of MBBLS $N = 17$ and other configurations of our heuristics. Finally, that performance is made possible in a reasonable execution time. Notably we see in the computation time boxplot that MBBLS is faster than LISTFIT according to the median time.

PIMSGen generates 10 random instances for each of the 39 valid combinations $(m,n,b) \in \{5,10,25\} \times \{10,50,100,500,1000\} \times \{100,1000,10000\}$. It gives a total of 390 instances. The p_j's of an instance are integers drawn uniformly in $[0.9(b-a),\ b]$ for 99% of them while the remaining ones (1% of the instances) are picked in $[a,\ 0.2(b-a)]$ according to a uniform distribution too. Note that this instance generation law is almost the same as the one defined for DCNU instances from the table 2 (the only difference is that only 98% of the p_j's are picked in the largest subinterval for DCNU instances and not 99% as PIMSGen does). Hence Figure 23 is a quite good confirmation of the results shown in Figure 8.

Figure 23: Benchmark including all PIMSGen instances. Exact instances used to produce this figure are available online [Hadj-Djilani, 2023e]



FEMTO-ST INSTITUTE, headquarters
15B Avenue des Montboucons - F-25030 Besançon Cedex France
Tel: (33 3) 63 08 24 00 – e-mail: contact@femto-st.fr

FEMTO-ST — AS2M: TEMIS, 24 rue Alain Savary, F-25000 Besançon France
FEMTO-ST — DISC: UFR Sciences - Route de Gray - F-25030 Besançon cedex France
FEMTO-ST — ENERGIE: Parc Technologique, 2 Av. Jean Moulin, Rue des entrepreneurs, F-90000 Belfort France
FEMTO-ST — MEC'APPLI: 24, chemin de l'épitaphe - F-25000 Besançon France
FEMTO-ST — MN2S: 15B Avenue des Montboucons - F-25030 Besançon cedex France
FEMTO-ST — OPTIQUE: 15B Avenue des Montboucons - F-25030 Besançon cedex France
FEMTO-ST — TEMPS-FREQUENCE: 26, Chemin de l'Epitaphe - F-25030 Besançon cedex France

http://www.femto-st.fr