

Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers

Jan Gmys

▶ **To cite this version:**

Jan Gmys. Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. INFORMS Journal on Computing, inPress, 10.1287/ijoc.2022.1193 . hal-03689608

HAL Id: hal-03689608
https://hal.science/hal-03689608

Submitted on 7 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exactly solving hard permutation flowshop scheduling problems on peta-scale GPU-accelerated supercomputers

Jan Gmys
Inria Lille-Nord Europe
Université de Lille, CNRS/CRIStAL, jan.gmys@inria.fr,

Makespan minimization in permutation flow-shop scheduling is a well-known hard combinatorial optimization problem. Among the 120 standard benchmark instances proposed by E. Taillard in 1993, 23 have remained unsolved for almost three decades. In this paper, we present our attempts to solve these instances to optimality using parallel Branch-and-Bound (BB) on the GPU-accelerated *Jean Zay* supercomputer. We report the exact solution of 11 previously unsolved problem instances and improved upper bounds for 8 instances. The solution of these problems requires both algorithmic improvements and leveraging the computing power of peta-scale high-performance computing platforms. The challenge consists in efficiently performing parallel depth-first traversal of a highly irregular and fine-grained search tree on distributed systems composed of hundreds of massively parallel accelerator devices and multi-core processors. We present and discuss the design and implementation of our permutation-based BB and experimentally evaluate its parallel performance on up to 384 V100 GPUs (2 million CUDA cores) and 3840 CPU cores. The optimality proof for the largest solved instance requires about 64 CPU-years of computation—using 256 GPUs and over 4 million parallel search agents, the traversal of the search tree is completed in 13 hours, exploring 339×10^{12} nodes.

Key words: Permutation flow-shop scheduling; Branch-and-Bound; Supercomputing; GPU computing

1. Introduction

Many combinatorial optimization problems (e.g. scheduling, assignment or routing problems) can be modeled by using permutations to represent candidate solutions. In this work, we focus on the Permutation Flowshop Scheduling Problem (PFSP) with makespan criterion. The goal is to find a scheduling order (a permutation) for processing n jobs on m machines, such that the maximum completion time, $C_{\max}$, is minimized, given the fixed processing times $p_{jk} \geq 0$ for job J_j on machine M_k.

The problem is $\mathcal{NP}$-hard for $m \geq 3$ (Garey et al. 1976) and exact algorithms like Branch-and-Bound (BB) can only solve small-sized instances within a reasonable amount of time. BB performs an implicit enumeration of all possible solutions by dynamically constructing and exploring a tree, using four operators: branching, bounding, selection and pruning. For larger problem instances, the exhaustive exploration of the search space becomes practically infeasible on a sequential computer. In this article, we present the design and implementation of PBB@Cluster, a permutation-based BB (PBB) algorithm for heterogeneous clusters composed of multi-core processors and GPU accelerator devices. Scaling PBB@Cluster on hundreds of GPUs of the *Jean Zay* supercomputer (#64 in the Top500 ranking, Nov. 2020), we report improved solutions and proofs of optimality for dozens of benchmark instances, including the exact solution of 11 hard PFSP benchmark instances that remained open for almost three decades.

Our motivation is twofold. On the one hand, the knowledge of exact optimal solutions for benchmark instances is highly valuable, as they provide a baseline for assessing the quality of metaheuristics and other approximate methods. For the PFSP, the set of 120 benchmark instances proposed by Taillard (1993) is the most frequently used. While instances defined by less than 20 machines are relatively easy to solve (Gmys et al. 2020), most of Taillard's instances with $m = 20$ machines and $n \geq 50$ jobs are very hard and optimal solutions for 23 of them remain unknown. On the other hand, the efficient parallel design and implementation of backtracking/BB algorithms is challenging, mainly because the pattern of computation and communication captured by this method (named a "computational dwarf" in (Asanovic et al. 2009)) is highly irregular. Moreover, the efficient design of parallel BB is strongly influenced by problem-specific characteristics (search space, goal and granularity) and the targeted compute platform (Bader et al. 2005).

The potential for exploiting parallelism in BB has been recognized as early as 1975 and research activity started to intensify ten years later, as parallel processing capabilities became practically available (Pruul et al. 1988). Surveys on parallel BB in general and parallel search for discrete optimization can be found in (Gendron and Crainic 1994, Grama and Kumar 1995). Among the wide range of works in this field, two research directions are particularly relevant for this work, as our approach can be seen as a combination of both: the first deals with fine-grained tree search algorithms on massively parallel processor arrays and GPUs and the second concerns more coarse-grained applications of the BB paradigm on top of distributed memory architectures.

In the early 1990s, the design and implementation of backtracking/BB algorithms on massively parallel single instruction, multiple data (SIMD) supercomputers (MasPar, CM-2, Intel Hypercube, etc.) has attracted much attention (Rao and Kumar 1987, Karypis and Kumar 1994), with research interests focusing on data-parallel load balancing strategies (Fonlupt et al. 1994, Reinefeld and Schnecke 1994). Frequently used test-cases include puzzles (e.g. 15-puzzle) and games (e.g. Othello) which are characterized by regular fine-grained evaluation functions and highly irregular search trees. Notably, one can find many similarities between these approaches and backtracking/DFS algorithms for modern GPUs, including the fact that the latter are mainly targeting fine-grained applications (Pessoa et al. 2016, Rocki and Suda 2010, Jenkins et al. 2011).

Following hardware evolution, with the emergence of cluster and grid computing, research focus shifted towards the design of parallel BB on top of distributed, heterogeneous and volatile platforms, targeting more coarse-grained applications of BB (Crainic et al. 2006). The use of large computational grids has led to breakthroughs such as the exact solution, in 2002, of quadratic assignment problem (QAP) instances which had remained unsolved for over three decades, including the notorious *nug30* instance in 7 days, using 650 CPUs on average (Anstreicher et al. 2002). The design of a BB algorithm using a new, stronger lower bound (LB) and its parallel implementation on a large computational grid were vital in bringing about this achievement. About 15 years later, Date and Nagi (2019) used an even stronger LB to re-solve *nug30* on a GPU-powered cluster (Blue Waters@Urbana-Champaign) in 4 days and successfully improve lower bounds of challenging QAP instances with up to 42 facilities. Their approach uses GPUs to accelerate the computation of tight LBs which require solving $O(n^4)$ independent linear assignment problems.

For the PFSP, the largest attempt to exactly solve hard problem instances has been carried out by Mezmaz et al. (2007), who designed and deployed a grid-enabled PBB algorithm on more than 1000 processors. This effort led to the exact resolution of instance

Ta056 in 25 days (22 CPU-years), exploiting 328 processors on average. Attempts of similar scale to solve other open instances from Taillard's benchmark remained unfruitful, indicating that their solution requires algorithmic advances and/or much more processing power.

Following the work of Mezmaz et al. (2007), the PFSP has been used as a test-case for several PBB algorithms targeting heterogeneous distributed systems combining multi-core CPUs and GPUs (Chakroun and Melab 2015, Vu and Derbel 2016, Gmys et al. 2017). However, despite reaching speed-ups between two and three orders of magnitude over sequential execution, no new solutions for the remaining Taillard benchmark instances were reported.

This article shows that previously unsolved PFSP instances can be solved exactly through a combination of both, the efficient exploitation of a peta-scale GPU-accelerated supercomputer and the algorithmic advances presented in (Gmys et al. 2020). A crucial aspect in the design of BB algorithms is the tradeoff between the computational complexity of the lower bound (LB) and the potential of the latter to reduce the size of the explored tree. For the PFSP, the strongest LB is the one proposed by Lageweg et al. (1978) and it is used in almost all previous parallel BB approaches. However, we have shown in (Gmys et al. 2020) that using a weaker, easy-to-compute LB from one of the first BB algorithms (Ignall and Schrage 1965) allows to solve large PFSP instances more efficiently. Although weakening the LB increases the size of the explored search tree, empirical results indicate that a better overall tradeoff is achieved. Therefore, instead of strengthening the LB, our approach takes a step in the opposite direction, i.e. it uses a weaker, more fine-grained LB than previous PBB algorithms for the PFSP. To give an idea of scale, with the LB used in this work, a single node evaluation can be performed in less than 10^{-6} seconds, and trees are composed of up to $\sim 10^{15}$ nodes.

1.1. Summary of contributions

The main result of this work can be summarized as follows: 11 of 23 open PFSP instances from Taillard's benchmark are solved to optimality using a scalable GPU-accelerated PBB algorithm on up to 96 quad-GPU nodes (3840 CPU cores and nearly 2 million CUDA cores) of the *Jean Zay* supercomputer. Moreover, the best-known solutions for 8 open instances are improved. For the Vallada-Ruiz-Framinan (VRF) benchmark (Vallada et al. 2015), 55 of 292 open instances are solved for the first time and 122 best-known upper bounds are improved. Scalability experiments show that PBB@Cluster achieves a parallel efficiency of $\sim 90\%$ on 16, 64 and 128 GPUs for problem instances requiring respectively 1, 4 and 27 hours of processing on a single GPU. The largest solved instance requires over 13 hours of processing on 256 V100 GPUs, i.e. a total of 3400 GPU-hours—which amounts to an estimated equivalent CPU time of 64 years.

Although, to the best of our knowledge, our work is the first to deploy fine-grained BB on a peta-scale system, we should point out that the presented results are not "simply" a matter of brute force. Instead, PBB@Cluster builds upon research efforts that stretch over several years and deal with the following **challenges** which are mainly due to the highly irregular nature of the algorithm.

- Especially in fine-grained parallel tree search it is essential to define an efficient **data structure** for the storage and management of the "tsunami" of subproblems, dynamically generated at runtime. Indeed, it is crucial to keep the overhead of search tree management operations low as the computational cost of the latter cannot be neglected

when using a very fine-grained LB function. Therefore, PBB@Cluster is based on an innovative data structure, dedicated to permutation problems and called Integer-Vector-Matrix (IVM) (Mezmaz et al. 2014). The advantage of IVM, compared to a conventional linked-list, lies in its compact and constant memory-footprint, which is well-suited for GPUs (Gmys et al. 2016).

- Due to its highly irregular nature, scaling parallel BB up to millions of concurrent exploration agents requires efficient **load balancing** mechanisms—which in turn rely on a suitable definition of work units. We present the design and implementation of a hierarchical load balancing scheme (on the GPU and inter-node levels) with an interval-based encoding of work-units that keeps millions of GPU-based BB explorers busy. The encoding of work units as integer intervals based on the mixed-radix factorial number system (also called factoradic) has been successfully used in multi-core (Mezmaz et al. 2014), GPU (Gmys et al. 2017) and grid-based Mezmaz et al. (2007) parallel PBB algorithms.
- The algorithm is also irregular on the level of individual exploration-agents. In particular, the node evaluation function is characterized by irregular memory access patterns and diverging control flow. This makes it difficult to take advantage of **low-level parallelism** and impedes single instruction, multiple thread (SIMT) execution efficiency. For the PFSP makespan and LB evaluation functions, vectorization approaches have been proposed in (Bożejko 2009, Melab et al. 2018). Mapping strategies for reducing thread divergence in PBB@GPU were investigated in (Gmys et al. 2016). In this work, we revisit the vectorization of the fine-grained LB used in PBB@Cluster to speed up node evaluation and exploit warp-level parallelism.
- PBB prunes subproblems whose LB is greater than the best found solution so far. Therefore, it is important to **discover optimal solutions** quickly and thereby maximize the pruning rate. Any improvement of the incumbent solution may dramatically accelerate the exploration process and lead to superlinear speedups (de Bruin et al. 1995, Gmys et al. 2020). As depth-first search (DFS) alone fails in general to find high-quality solutions, we investigate the hybridization of PBB@Cluster with approximate search methods. The proposed hybridization uses independent CPU-based heuristic searches for the discovery of high-quality solutions in parallel to and in cooperation with the GPU-based exhaustive search.
- **Communication** patterns are irregular as well: several unpredictable events (new best solutions, local work exhaustion, checkpointing, termination detection) trigger communications involving messages of different kinds and sizes. Moreover, a scalable approach requires inter-node communications to be asynchronous on the worker-side, i.e. a primary design goal is to interrupt the GPU-based tree-traversal as little as possible. While the inter-node level of PBB@Cluster is conceptually similar to the coordinator-worker approach in BB@Grid (Mezmaz et al. 2007), the latter is designed for mono-core processors and uses C-based socket programming. The use of GPU-based workers and a switch to MPI (Message Passing Interface) motivates us to revisit the design of the coordinator process, redefining work units and enabling asynchronous message passing.
- A reliable global **checkpointing** mechanism is an indispensable component of PBB@Cluster. On the one hand, as the time required for solving a particular instance is unpredictable, node reservations may expire before the exploration is completed. On the other hand, the mean time between failure (MTBF) on large supercomputers keeps decreasing as we're entering the exascale era (Cappello 2009). Therefore, a

minimum requirement is to be able to re-start the exploration process from the last global checkpoint without impeding correctness.

1.2. Outline

We aim at making this article as self-contained as possible and to present not only the outcome of our research, but to expose how design choices result from an interaction between hardware and problem-dependent algorithmic constraints. First, Section 2 defines the PFSP and presents the sequential algorithm design, the IVM data structure and work units. Section 3 presents PBB@GPU, the GPU-based PBB algorithm at the core of worker processes in PBB@Cluster. In Section 4 we describe the design and implementation of PBB@Cluster's coordinator and worker processes. Experimental results are reported in Section 5 and finally, some conclusions are drawn in Section 6. Improved upper bounds and new optimal schedules for benchmark instances are provided in the Online Supplement.

2. Branch-and-Bound for the PFSP

2.1. Problem formulation

The flowshop scheduling problem (FSP) can be formulated as follows. Each of n jobs $J = \{J_1, J_2, \ldots, J_n\}$ has to be processed on m machines $M_1, M_2, \ldots, M_m$ in that order. The processing of job J_j on machine M_k, takes an uninterrupted time p_{jk}, given by a processing time matrix. Each machine can process at most one job at a time and jobs cannot be processed simultaneously on different machines. A common simplification is to consider only permutation schedules, i.e. to enforce an identical processing order on all machines, which reduces the size of the search space from $(n!)^m$ to $n!$. Considering minimization of the completion time of the last job on the last machine, called *makespan*, the resulting problem is the permutation flow-shop problem (PFSP) with makespan criterion, denoted $F_m|prmu|C_{\max}$.

Formally, denoting $\pi = (\pi(1), \ldots, \pi(n)) \in S_n$ a permutation of length n, and $C_{j,k}$ the completion time of job J_j on machine M_k, the goal is to find an optimal permutation $\pi^\star$ such that

$$C_{\max}(\pi^\star) = \min_{\pi \in S_n} C_{\max}(\pi)$$

where $C_{\max}(\pi) = C_{\pi(n),m}$. For $m = 2$, the problem can be solved in $O(n \log n)$ steps by sorting the jobs according to Johnson's rule (Johnson 1954); for $m \geq 3$ it is shown to be NP-hard (Garey et al. 1976). The completion times $C_{\pi(j),k}$ can be obtained recursively by

$$C_{\pi(j),k} = \max\left(C_{\pi(j),k-1}, C_{\pi(j-1),k}\right) + p_{\pi(j),k} \tag{1}$$

where $p_{\pi(0),k} = p_{j,0} = 0$ by convention. Thus, for a given schedule π, the makespan $C_{\max}(\pi) = C_{\pi(n),m}$ can be computed in $O(mn)$ time.

2.2. Branch-and-Bound for permutation problems

BB performs an implicit enumeration of the search space by dynamically constructing and exploring a tree, whose root node represents the initial problem, internal nodes represent subproblems (partial solutions) and leaves are feasible solutions (permutations). The algorithm starts by initializing the best solution found so far (also called the incumbent) and the data-structure used for storing the tree such that it contains only the root node.

Figure 1 shows an illustration of the four BB-operators, selection, branching, bounding, and pruning, for a permutation problem of size four. At each iteration, the **selection** operator returns the next subproblem to explore, starting with the root node. The **branching**

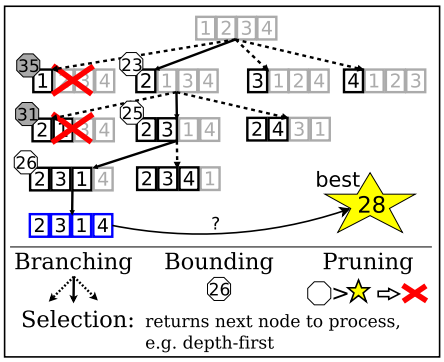


Figure 1 **Illustration of a simple Branch-and-Bound algorithm for a permutation problem of size four.**

operator decomposes the subproblem into smaller disjoint subproblems. The **bounding** operator computes lower bounds (LB) on the optimal cost of these subproblems, in the sense that no arrangement of unscheduled jobs can yield a smaller makespan than LB. Using the LB values, the **pruning** operator discards subproblems from the search that cannot lead to an improvement of the incumbent. All non-pruned subproblems are inserted into the data structure for further exploration. In the following subsections we specify the branching and bounding operators used in this work, as well as the data structure used for storing subproblems. The choice of the sequential algorithm design is based on (Gmys et al. 2020).

2.3. Branching rule

Our BB algorithm uses a position-based branching scheme, i.e. a subproblem (partial solution) is branched by selecting one free position and assigning remaining unscheduled jobs to that position. Alternatively one could investigate precedence-based approaches, such as the block-based branching of (Grabowski et al. 1983) which successively adds precedence constraints and which has been used in efficient local search algorithms for the PFSP (Nowicki and Smutnicki 1996). However, as the latter requires maintaining critical paths and block decompositions for each tree node, this approach is more challenging to implement, especially on top of GPU accelerators.

In the example of Figure 1, permutations are built from left to right, meaning that a node of depth d can be represented by a prefix partial schedule σ_1 of d jobs. The *forward* branching operation consists in generating $n-d$ child subproblems as follows:

$$\text{Forward-Branch} : \sigma_1 \mapsto \{\sigma_1 j,\, j \in J \setminus \sigma_1\}.$$

Backward branching prepends unscheduled jobs to a postfix partial schedule σ_2, i.e.

$$\text{Backward-Branch} : \sigma_2 \mapsto \{j\sigma_2,\, j \in J \setminus \sigma_2\}.$$

Our PBB algorithm uses a *dynamic* branching rule (Potts 1980), which decides dynamically, for each decomposed node, which of the two branching types is applied. With dynamic branching, subproblems are represented in the form (σ_1, σ_2) and are decomposed as follows

$$\text{Dyna-Branch} : (\sigma_1, \sigma_2) \mapsto \begin{cases} \{(\sigma_1 j, \sigma_2),\, j \in J \setminus (\sigma_1, \sigma_2)\} & \text{if Fwd-Branch} \\ \{(\sigma_1, j\sigma_2),\, j \in J \setminus (\sigma_1, \sigma_2)\} & \text{if Bwd-Branch} \end{cases}$$

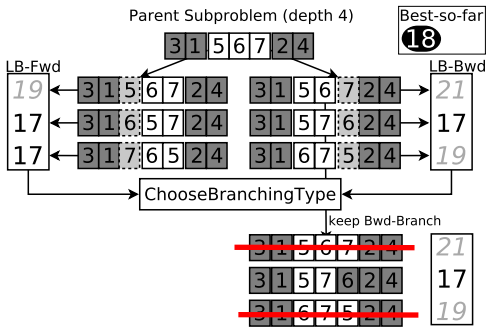


Figure 2 **Illustration of a subproblem decomposition using dynamic branching.**

Note that, to guarantee completeness of the search, all children of a subproblem are obtained by the same branching type, either forward or backward (see the Online Supplement for further details, including a proof of exhaustiveness).

To make the branching decision, both sets of children nodes are evaluated and a simple heuristic chooses which of the two sets is retained. In this work, the branching heuristic called *MinMin* (Gmys et al. 2020) is used. It chooses the set in which the minimal LB (among both sets) is realized less often, or, in case of equality the set where the sum of LBs is higher, and *forward* if the sums are equal as well. Compared to mono-directional branching, the dynamic branching requires the computation of twice as many LBs per node decomposition, but computational experiments show that the tree size can often be reduced by several orders of magnitude (Gmys et al. 2020).

Figure 2 illustrates a node decomposition, involving dynamic branching, bounding and pruning. In the example, the decomposed subproblem is $(\sigma_1, \sigma_2) = ((J_3, J_1), (J_2, J_4))$, where jobs $\{J_5, J_6, J_7\}$ remain to be scheduled and the best makespan found so far is 18. Both children sets are evaluated, yielding $\text{LB}_{\text{Fwd}} = \{19, 17, 17\}$ for *forward* and $\text{LB}_{\text{Bwd}} = \{21, 17, 19\}$ for *backward*. The smallest LB (17) occurs less frequently in LB_{Bwd}, so the *MinMin* heuristic chooses *backward* branching and the set of *forward* nodes is discarded. The computed LBs are reused by the pruning operator, which, in the example, eliminates two subproblems (instead of one in the alternative branch).

2.4. Lower Bound

The LB used in this work comes from the pioneering algorithms proposed independently by Lomnicki (1965) and Ignall and Schrage (1965). The so-called one-machine bound (LB1), was initially developed for BB algorithms using only forward branching, but it can be extended to the bi-directional subproblem representation (Potts 1980).

The computation of LB1 for a subproblem (σ_1, σ_2) can be divided into four steps, illustrated in Figure 3. We denote $|\sigma_1| = d_1$ and $|\sigma_2| = d_2$ the number of jobs scheduled in the prefix and suffix partial schedules respectively.

1. For the front, compute $C_{\sigma_1(d_1),k}$, the completion time of the last job in σ_1 on each machine, i.e. the earliest possible starting time for unscheduled jobs, given σ_1.
2. For the unscheduled jobs, compute $p(k) = \sum_{j \in J \setminus (\sigma_1, \sigma_2)} p_{j,k}$, the total remaining processing time on each machine.
3. For the back, compute $\bar{C}_{\sigma_2(d_2),k}$, the minimum time required between starting the first job in σ_2 on machine M_k and the end of operations on the last machine. These values are obtained by scheduling σ_2 *in reverse order* in front, using Equation (1) and the reversibility property of the PFSP (Potts 1980).

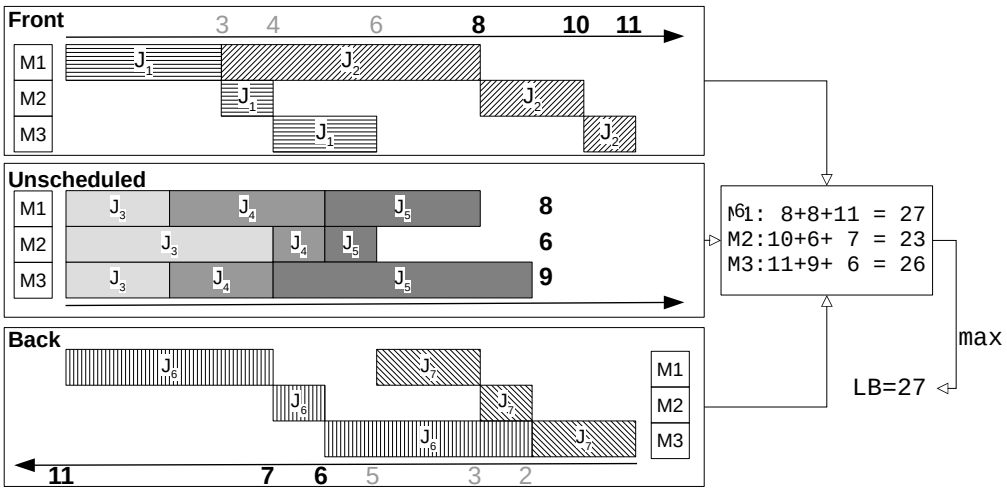


Figure 3 **Lower bound computation for a subproblem** $(\sigma_1 = (J_1, J_2), \sigma_2 = (J_6, J_7))$ **with 3 unscheduled jobs** (J_3, J_4, J_5)**. No arrangement of the unscheduled jobs can give a better makespan than** $LB = \max\{8+8+11, 10+6+7, 11+9+6\} = 27$**.**

4. Finally, LB1 is obtained by

$$LB1(\sigma_1, \sigma_2) = \max_{k=1,\dots,m} C_{\sigma_1(d_1),k} + p(k) + \bar{C}_{\sigma_2(d_2),k}.$$

Clearly, LB1 has the same time complexity as a makespan evaluation, i.e. $O(mn)$. However, reusing the quantities computed for (σ_1, σ_2), it is possible to deduce LB1 for a child subproblem in $O(m)$ steps. Therefore, the computation of LB1 *for all* $2 \times (n - d_1 - d_2)$ children of (σ_1, σ_2) also requires $O(mn)$ steps. Moreover, this incremental evaluation of the children requires only $O(1)$ additional memory per child.

LB1 is dominated by the two-machine bound LB2, proposed by Lageweg et al. (1978), which relies on the exact resolution, for different machine-pairs, of two-machine problems using Johnson's rule. In addition to the front/back computations, the different variants of LB2 require between m and $m(m-1)/2$ evaluations of (pre-sorted) two-machine Johnson schedules *for each* child node. Therefore, LB2 requires between $O(mn^2)$ and $O(m^2n^2)$ time for the evaluation of all child nodes in a decomposition step. In (Gmys et al. 2020) we found that, in combination with dynamic branching and especially for large problem instances, LB1 provides a better tradeoff between sharpness and computational effort. To give an approximate measure of comparison, using dynamic branching and LB1, Taillard's *Ta56* instance can be solved in 33 hours on a dual-socket Intel Xeon node (~ 600 CPU-hours)—with LB2 and the same branching scheme, the required CPU-time is 22 years ($300\times$ more) (Mezmaz et al. 2007).

2.5. Search strategy and data structure

The next node to be decomposed is chosen according to a predefined selection strategy. In this work, we consider only depth-first search (DFS), because memory requirements of best-first and breadth-first search grow exponentially with the problem size[1]. The data structure

[1] For example, solving *Ta058* ($n = 50$), the *critical tree* (composed of nodes with LBs smaller than the optimal cost) contains 339×10^{12} nodes, so there exists at least one level with more than 6.7×10^{12} open subproblems. Assuming that each subproblem is stored as a sequence of $n = 50$ 32-bit integers, breadth-first exploration would require at least $6.7 \times 10^{12} \times 50 \times 4$ B = 1.4 PB of memory.

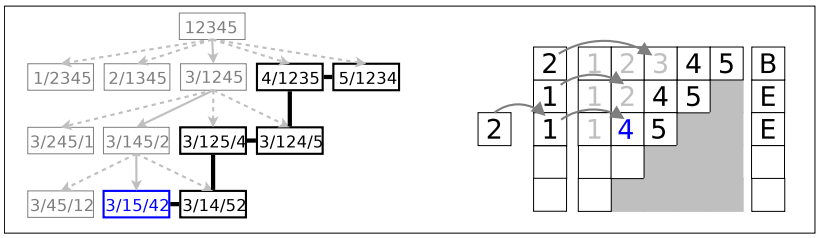


Figure 4 **Tree and IVM-based representation of the search state, solving a permutation problem of size** 5.

used for storing generated subproblems is closely related to the choice of the search strategy. In this work, we use the Integer-Vector-Matrix (IVM) data structure proposed in (Mezmaz et al. 2014). It is dedicated to permutation problems and provides a memory-efficient alternative to stacks, which are conventionally used for DFS. The working of the IVM data structure is best introduced with an example.

Figure 4 illustrates a pool of subproblems that could be obtained when solving a permutation problem of size $n=5$ with a DFS-PBB algorithm using bi-directional branching. On the left-hand side, Figure 4 shows a tree-based representation of this pool. The parent-child relationship between subproblems is represented by dashed gray arrows. The jobs before the first "/" symbol represent σ_1, the ones behind the second "/" symbol σ_2 and jobs between the two "/" symbols represent the set of unscheduled jobs in arbitrary order.

On the right-hand side, IVM indicates the next subproblem to be solved. The integer I of IVM gives the level of this subproblem, using 0-based counting (at level 0 one job is scheduled). In this example, the level of the next subproblem is 2. The vector V contains, for each level up to I, the position of the selected subproblem among its sibling nodes in the tree. In the example, jobs 3, 2 and 4 have been scheduled at levels 0, 1 and 2 respectively. The matrix M contains the jobs to be scheduled at each level: all the n jobs for the first row, the $n-1$ remaining jobs for the second row, and so on. The data structure is completed with a binary array of length n that indicates the branching type for each level. In the example, job 3 is scheduled at the beginning, jobs 2 and 4 are scheduled at the end. Thus, the IVM structure indicates that 3/15/42 is the next subproblem to be decomposed.

The IVM-based BB operators work as follows:

- To **branch** a selected subproblem, the remaining unscheduled jobs are copied to the next row of M and the current level I is incremented. The branching vector is set according to the branching decisions.
- To **prune** a subproblem, the corresponding cell in M should be ignored by the selection operator. To flag a cell of M as "pruned" its content is multiplied by -1. These flags are removed as jobs are copied to the next row.
- To **select** the next subproblem, the values of I and V are modified such that they point to the deepest leftmost non-pruned cell in M: the vector V is incremented at position I until a non-pruned cell is found or the end of the row is reached. If the end of the row is reached (i. e. $V[I]=n-I$), then the algorithm backtracks to the previous level by decrementing I and again incrementing V.

2.6. Work units

Throughout the depth-first exploration, the vector V behaves like a counter. In the example of Figure 4, V successively takes the values 00000, 00010, 00100, …, 43200, 43210 (skipping some values due to pruning). These 120 values correspond to the lexicographic numbering

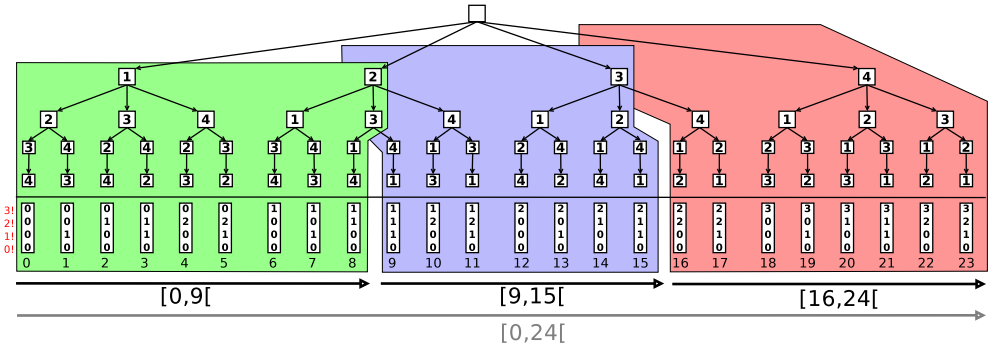


Figure 5 **Illustration of the search space encoded as the integer-interval** $[0,4![$**, for a permutation problem of size** $n=4$**. In this example, the search space is partitioned into three work units :** $[0,9[$**,** $[9,15[$**,** $[16,24[$ **(equivalently, in factoradic notation :** $[0000,1100[$**,** $[1110,2110[$**,** $[2200,3210[$**).**

of the 5! solutions in the *factorial* number system (Knuth 1997). In this mixed-radix number system, the weight of the position $k=0,1,\ldots,$ is equal to $k!$ and the digits allowed for the k^{th} position are $0,1,\ldots,k$.

For a problem of size n, each valid value of V corresponds uniquely to an integer in the interval $[0,n![$ (half-open interval including 0, excluding $n!$). Converting the position-vector V to its decimal form allows to interpret the search as an exploration, from left to right, of the integer interval $[0,n![$. Moreover, an initialization procedure allows to start the search at any position $a\in[0,n![$, and by comparing the position-vector V_a to an end-vector V_b, the search can be restricted to arbitrary intervals $[a,b[\subseteq[0,n![$.

In PBB, **work units are intervals**, that can be either represented in factoradic form $[V_a,V_b[\subseteq[(0,0,...,0),(n-1,n-2,...,2,1,0)[$ or, equivalently, in decimal form $[a,b[\subseteq[0,n![$. Figure 5 illustrates, for a problem of size $n=4$, the partition of the search space $[0,4![$ into three work units. A parallel PBB algorithm is obtained as follows. The search space $[0,n![$ is partitioned into K distinct subintervals $\{[a_i,b_i[\subset[0,n![,\ i=1,\ldots,K\}$ to be explored by K workers. As the distribution of work in $[0,n![$ is highly irregular, a work stealing approach is used (Mezmaz et al. 2014). When a worker i finishes the exploration of its interval $[a_i,b_i[$ (i.e. when $a_i=b_i$), it chooses a victim worker j and "steals" the right half $[\frac{a_j+b_j}{2},b_j[$. The work stealing victim j continues to explore the interval $[a_j,\frac{a_j+b_j}{2}[$.

2.7. Parallel models for Branch-and-Bound

The most frequently used models for the parallelization of PBB are: (1) parallel tree exploration, (2) parallel evaluation of bounds and (3) parallelization of the bounding function (Gendron and Crainic 1994).

Model (1) consists in exploring disjoint parts of the search space in parallel using multiple independent BB processes. For large trees this model yields a practically unlimited degree of parallelism (DOP). It requires efficient dynamic load balancing mechanisms to deal with the irregularity of the search tree, sharing of the best-found solution and a mechanism for (distributed) termination detection. Model (1) can be implemented either synchronously or asynchronously. In model (2), the children nodes generated at a given iteration are evaluated in parallel. The DOP is variable throughout the search as it depends on the depth of the decomposed subproblem. Model (3) strongly depends on the bounding function and may be nested within models (1) and (2).

In this work, model (1) is used hierarchically: on the first level the search space is distributed among asynchronous worker processes hosted on different compute nodes; on

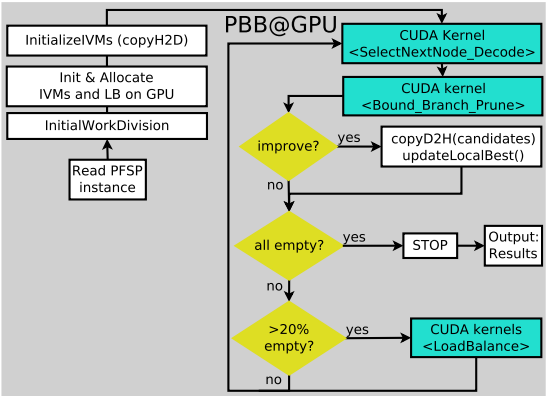


Figure 6 Outline of GPU-based PBB algorithm (PBB@GPU).

the second level, each worker process consists of several GPU-based, synchronous PBB sub-workers. On both levels, the tree is dynamically balanced among workers, best-found solutions are shared and termination conditions are handled. On the GPU-level, each independent PBB explorer is mapped onto a CUDA warp (currently 32 threads) and warp-level parallelism is exploited through a combination of models (2) and (3).

3. GPU-based Branch-and-Bound algorithm(PBB@GPU)

In PBB@GPU all four BB operators, including work stealing, are performed on the GPU. This differs from other approaches that can be found in the literature, notably the offloading of (costly) node evaluations to GPUs (Vu and Derbel 2016, Chakroun et al. 2013), and the generation of an initial *Active Set* of internal nodes on the host, which are then used as roots for concurrent GPU-based searches (Rocki and Suda 2010, Carneiro et al. 2011). In the following, we outline PBB@GPU. We put the focus on the LB computation and dynamic branching, as more detailed descriptions of selection and work stealing kernels can be found in (Gmys et al. 2016) and (Gmys et al. 2017).

3.1. Outline of PBB@GPU

The IVM data structure allows to bypass a major roadbloack for GPU-based tree search : the lack or poor performance of dynamic data structures (linked-lists, stacks, priority queues, etc.) on GPUs. IVM has a small and constant memory footprint: thousands of IVMs can be allocated in device memory. Moreover, the encoding of work units as factoradic intervals allows to implement low-overhead data-parallel work-stealing mechanisms on the GPU.

Figure 6 shows a flowchart outlining the PBB@GPU algorithm. After reading problem-specific input data, K IVM structures are allocated in GPU memory and constant data (matrix of processing times, n, m, ...) is copied to the device. Then, a collection of (at most K) intervals is initialized on the host—for instance with a single interval $\{[0, n![\}$ or an initial partitioning of the search space $\{[\frac{j\times n!}{K}, \frac{(j+1)\times n!}{K}[, j = 0, 1, \ldots, K-1\}$. If created in decimal form, the intervals are converted to factoradics in order be used as initial position- and end-vectors on the device. After this operation, PBB@GPU enters the main exploration-loop, which consists of three CUDA kernels and a few auxiliary operations.

The first kernel performs the node selection as described in Section 2.5 concurrently on all IVMs. This kernel also decodes the IVMs, producing one subproblem of the form $[\pi, d_1, d_2]$ per IVM, where π is a schedule with fixed jobs at positions $1, \ldots, d_1$ and $d_2, \ldots, n$.

The second kernel performs the decomposition step, including dynamic branching and bounding, as shown in Figure 2. IVM structures are modified in parallel to apply pruning decisions.

If an improving solution is found, then a device flag `newBest` is set. Moreover, a global device counter $0 \leq$`nbActive`$\leq K$ keeps track of the number of IVMs with non-empty intervals. Both are copied to the host at each iteration. If the `newBest` flag is set, then the candidate solution(s) is (are) copied to the host and the best solution is updated. If the current activity level is equal to zero, then PBB@GPU returns the optimal solution and exploration statistics, before shutting down. If the current activity level is below $0.8 \times K$, i.e. if more than 20% of IVMs are inactive, then a work stealing phase is triggered. The goal of this trigger-mechanism is to keep the load balancing overhead low. In Figure 6, some details, such as kernel configurations, have been spared out. Some details regarding the efficient implementation of these kernels are provided in the following subsections.

3.2. Selection kernel

The main performance issue for this kernel is branch divergence, as the amount of operations to perform varies strongly between different IVMs. Mapping each IVM to exactly one thread causes control flow divergence for threads within the same warp, leading to serialized execution of divergent branches. Experiments have shown that it is preferable to map IVMs to full warps (Gmys et al. 2016)—even if that means that all threads except the warp-leader (lane 0) are mostly inactive. The selection kernel is thus launched with $K \times$ warpSize threads, grouped in blocks of 4 warps. Besides reducing thread divergence, this makes more per-block shared memory available per IVM, allowing to bring parts of the data structure closer to the ALUs. The 32 (warpSize) available threads per IVM are used for loading data to shared memory. Moreover, despite the sequential nature of the selection operator, some sub-operations (e.g. generating a new line in the IVM matrix) benefit from warp-level parallelism.

3.3. Bounding kernel

Previous GPU-accelerated BB algorithms for the PFSP use the heavier LB2 bound (cf. Section 2), which consumes about 99% of the computation time in sequential implementations. In that situation, it is natural to focus performance optimization efforts on the bounding kernel. For instance, to deal with the variable amount of children nodes per IVM, the LB2-based algorithm proposed in (Gmys et al. 2016) introduces an auxiliary *mapping* kernel, which uses a parallel prefix sum computation to build a compact thread-data mapping. However, this approach is not suitable for the more fine-grained LB1 evaluation. Indeed, the the computational cost of the incremental LB1 evaluation is too low to justify regularizing the workload with complex overhead operations. As for the selection kernel, using a compact one-thread-per-IVM mapping results in low warp execution efficiency and requires a very large number of IVMs (K) to reach acceptable device occupancy levels.

Therefore, the bounding kernel uses a warp-centric approach, like the selection kernel, with exactly one warp per subproblem. The implementation uses the CUDA Cooperative Groups API, warp-level primitives and explicit warp-synchronization (`__syncwarp()`) (since CUDA 9). For each parent subproblem (σ_1, σ_2) of depth $d_1 + d_2 = d$ there are $n - d$ unscheduled jobs and thus $2 \times (n - d)$ child subproblems to evaluate. As explained in Section 2, the evaluation of children nodes involves: (1) the computation of partial costs for

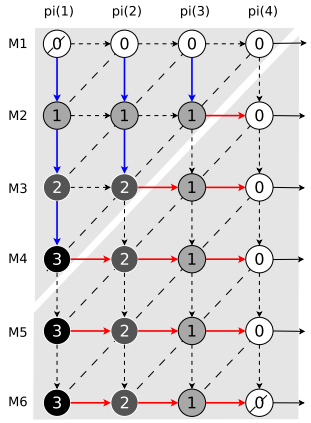


Figure 7 **Illustration of a parallel makespan evaluation for a four-job (partial) schedule and $m = 6$ machines.**

the parent subproblem and (2) the incremental evaluation of LB1 for each child subproblem. Despite the data dependencies in Equation 1, step (1) can be parallelized as proposed by Bożejko (2009) for MMX vector instructions. Step (2) is embarrassingly parallel.

Figure 7 illustrates the warp-parallel evaluation of a prefix schedule of length $d_1 = 4$ on $m = 6$ machines. Each circle represents one max-add operation (as in Equation 1), and the shades of (and labels inside) the circles indicate the lane (thread index within the warp) performing the operation. Operations connected by dashed diagonal lines are done in parallel. The horizontal and vertical arrows represent data dependencies of two types: a thin dashed arrow indicates that the lane already holds the required value from the previous iteration; a solid arrow indicates that the required value is transferred from a neighboring lane using a warp-level `shfl_up` (blue, vertical) or `shfl_down` (red, horizontal) operation. These built-in warp-synchronous functions allow to bypass shared memory and perform the (partial) makespan evaluation using only per-thread registers. The solid arrows on the right represent the storage of per-machine completion times in a (shared or global memory) array of length m.

In the example shown in Figure 7, the $6 \times 4 = 24$ operations are performed in $4+6-1 = 9$ iterations, so the theoretical speedup in this case is $\frac{24}{9} = 2.7\times$. For a detailed theoretical speedup analysis we refer the reader to (Bożejko 2009). The decomposition of a node (σ_1, σ_2), as illustrated in Figure 2, is performed at the warp-level as follows:

1. Evaluate $C_{\sigma_1(d_1),:}$ as illustrated in Figure 7.
2. Evaluate $\bar{C}_{\sigma_2(d_2),:}$ as illustrated in Figure 7.
3. Compute $n-d$ *Forward*-LBs in $m\left\lceil\frac{n-d}{32}\right\rceil$ steps.
4. Compute $n-d$ *Backward*-LBs in $m\left\lceil\frac{n-d}{32}\right\rceil$ steps.
5. Choose branching direction (warp-parallel min-reduce and warp-vote functions).
6. Apply pruning decisions for $n-d$ subproblems in parallel.

The remaining processing time per machine is obtained by subtracting from the per-machine total and integrated into steps 1 and 2. The parent subproblem $[\pi, d_1, d_2]$ and m-element arrays representing the front, back and remaining times (see Figure 3) are placed in shared memory. The processing time matrix is placed in constant memory.

3.4. Work Stealing kernels

As mentioned, a work stealing (WS) operation consists in taking the right half from a non-empty interval and assigning it to an idle worker. To perform this operation on the

GPU, device functions for elementary operations ($+$, $-$, $\div$ by scalar) on factoradic numbers are implemented. The more challenging part is to build a mapping of empty IVMs onto exploring IVMs, such that (1) no WS victim is selected twice, (2) larger intervals are preferred and (3) the mapping is build in parallel on the device.

In the victim selection step, the K IVM are seen as vertices of a hypercube, in which all empty IVMs successively poll their neighbors to acquire new work units. For illustrative purposes, let's suppose that $K = 2^{14} = 16384$. The indices of the K IVMs can be written as $(\alpha_7 \dots \alpha_1)$ in base 4. Connecting all IVMs whose base-4 indices differ in exactly one digit, a 4-ary 7-cube is obtained, where each IVM has $7 \times (4-1) = 21$ neighbors. The victim selection is carried out in 21 iterations during which each empty IVM tries to select $(\alpha_7 \dots (\alpha_i - j) \pmod 4 \dots \alpha_1)$, $i = 1, \dots, 7$, $j = 1, 2, 3$ as a work stealing victim. A non-empty IVM can be selected if and only if (1) it is not yet selected and (2) its interval is larger than (a) the average interval-length and (b) a minimum length, fixed arbitrarily to (8!). Prior to the victim selection phase, a helper kernel computes the average interval-length.

4. Distributed GPU-based algorithm (PBB@cluster)

4.1. PBB@Cluster : Coordinator process

For the inter-node level of PBB@Cluster, we revisit the PBB@Grid framework of Mezmaz et al. (2007) to enable the use of GPU-based (or multi-core) worker processes, instead of single-threaded workers. Our algorithm is implemented with MPI and uses a static number of worker processes (n_p)—contrary to PBB@Grid, which uses socket programming for inter-node communication and shell-scripts to discover available resources and launch worker processes via `ssh`.

PBB@Cluster is based on an asynchronous coordinator-worker model with worker-initiated communications. At each point in time, the coordinator keeps a list of *unassigned* work units and an *active* list, containing copies of the work units explored by different workers. As each worker is composed of multiple sub-workers, a cluster-level **work unit** is defined as a collection of intervals contained in $[0, n![$. More precisely, we define a work unit W_i as a finite union of K_i non-overlapping intervals

$$W_i = \bigcup_{j=0}^{K_i} [a_j, b_j[\quad , \text{where } \forall j : [a_j, b_j[\subseteq [0, n![\text{ and } [a_j, b_j[\cap [a_k, b_k[= \emptyset \, , j \neq k \tag{2}$$

In this definition, index i is the identifier of the work unit, and $K_i < K_i^{\max}$ the number of intervals, limited by a maximum capacity $K_i^{\max}$. The coordinator maintains (1) a list of unassigned work units $\mathcal{W}_{\text{unassigned}}$ and (2) a list of active work units $\mathcal{W} = \{W_1^c, W_2^c, \dots\}$, where W_i^c designates the coordinator's copy of a work unit W_i.

A pseudo-code of the PBB@Cluster coordinator is shown in Algorithm 1. After broadcasting an initial solution (computed or read from a file), the coordinator fills $\mathcal{W}_{\text{unassigned}}$ with the initial work, e.g. the complete interval $[0, n![$, an initial decomposition, or a list of intervals read from a file (Line 3). Then, the coordinator starts listening for incoming messages (Line 6). Under certain conditions, that will be detailed later, workers send checkpoint messages to the coordinator, containing

- the number of nodes decomposed since the last checkpoint,
- $K_i^{\max}$ the maximal number of intervals the worker can handle and
- a work unit W_i containing K_i intervals and tagged with a unique identifier i.

Algorithm 1 PBB@Cluster : Coordinator

```
 1: procedure PBB-Coordinator
 2:     /* not shown: allocations; initialize, build and broadcast initial solution; ...*/
 3:     W_unassigned ←GetInitialWorks()                          ▷ e.g. [0,n![, readFromFile(),...
 4:     n_terminated ← 0
 5:     while n_terminated < n_proc do
 6:         src, tag←MPI_Probe(ANY)                              ▷ wait for any message from any source
 7:         switch tag do                                        ▷ discriminate message types
 8:             case WORK: /* worker checkpoint */
 9:                 W_i ←ReceiveWork(src)                        ▷ wrapper for MPI_Recv and Unpack
10:                 W_tmp ←WorkerCheckpoint(W_i)                 ▷ pseudo-code provided below
11:                 if W = ∅ ∧ W_unassigned = ∅ then
12:                     SendEnd(src)                             ▷ no more work : send termination signal
13:                 else if W_tmp ≠ W_i then
14:                     SendWork(W_tmp, src)                     ▷ send new or modified W_i to worker
15:                 else
16:                     SendBest(src)                            ▷ acknowledge reception / send global best
17:                 end if
18:             case BEST: /* candidate for improved global best solution */
19:                 S←ReceiveSolution(src)
20:                 TryImproveGlobalBest(S)
21:                 SendBest(src)
22:             case END: /* worker has left computation */
23:                 S←ReceiveSolution(src)
24:                 TryImproveGlobalBest(S)
25:                 n_terminated++
26:         end switch
27:         if DoGlobalCheckpoint() then                         ▷ global checkpoint interval elapsed?
28:             SaveToDisk(W, W_unassigned)
29:         end if
30:     end while
31: end procedure
32: procedure WorkerCheckpoint(W_i)
33:     W_i^c ←FindCopy(W_i,W)
34:     W_tmp ←Intersect(W_i, W_i^c)
35:     if W_tmp = ∅ then
36:         W_tmp ←Steal(i, W)
37:     end if
38:     W_i^c ← W_tmp
39:     return W_tmp
40: end procedure
```

After receiving a checkpoint message, the coordinator intersects the received work unit W_i with the copy W_i^c. If the result of the intersection is empty, a new work unit of at most $K_i^{\max}$ intervals is generated by taking intervals from $\mathcal{W}_{\text{unassigned}}$ or by splitting a work unit from $\mathcal{W}$. The work unit resulting from the intersection and/or splitting operations (W_{tmp} in Algorithm 1) replaces the previous copy W_i^c in $\mathcal{W}$. These operations are shown as the `workerCheckpoint` operation in Algorithm 1 (Lines 10 and 32). Work unit intersection and splitting procedures are described below.

If W_{tmp} differs from the received work unit W_i, then W_{tmp} is sent back to the worker to replace W_i. If W_{tmp} is identical to the received W_i, then there is no need to send any work back and the coordinator replies only by sending the best-found global upper bound (Line 16). The remaining tasks of the coordinator, shown in Lines 18 to 29, deal with termination detection, management of the global best solution and global checkpointing. The sending/receiving of work units and the intersection procedure are the most time-consuming operations of the coordinator.

4.1.1. Work unit communication. We should note that the `MPI_Recv` and `MPI_Send` calls are not time-consuming *per se*, but the associated data marshalling is costly. A worker

checkpoint message is of type `MPI_PACKED` and consists of a metadata header and a list of K_i (for ex. ≤ 16384) intervals. As these intervals are represented by a couple of integers of the order $\sim n!$, the GNU Multiple Precision Arithmetic Library (GMP) is used. While the coordinator might as well work with factoradic numbers (i.e. integer arrays of length n), it is more convenient and faster to perform arithmetic operations (subtraction, division, addition, comparison) in decimal form. However, due to the lack of native MPI support for GMP integers, packing intervals to the communication buffer requires converting them to raw binary format and back to `mpz_t` at the receiving end. We observed that these conversions cause significant overhead.

4.1.2. Work unit intersection. The intersection of two intervals $[a_1, b_1[$ and $[a_2, b_2[$ is done by considering the maximum between both start points and the minimum between both end points, as shown in Equation 3.

$$[a_1, b_1[\cap [a_2, b_2[= [\max(a_1, a_2), \min(b_1, b_2)[\tag{3}$$

The intersection of two work units W_1 and W_2 requires pairwise intersection of the intervals contained in both sets, as shown in Equation 4.

$$\left(\bigcup_{i=1}^{K_1} [a_i^1, b_i^1[\right) \cap \left(\bigcup_{j=1}^{K_2} [a_j^2, b_j^2[\right) = \bigcup_{i=1}^{K_1}\bigcup_{j=1}^{K_2} [\max(a_i^1, a_j^2), \min(b_i^1, b_j^2)[\tag{4}$$

For arbitrary sets of intervals, $K_1 \times K_2$ elementary intersections are required to compute the intersection of two interval-lists. However, using the fact that each interval in W_1 intersects with at most one interval in W_2, and sorting intervals in increasing order, the operation in Equation 4 can be carried out in $O(K_1 + K_2)$ time. The computational cost of work unit intersections can be further reduced by taking advantage of the following observation. If a copy W_i^c hasn't been stolen from since the last worker-checkpoint, then the intersection operation becomes trivially $W_i^c \cap W_i = W_i$. Thus, the coordinator maintains a flag for each work unit in $\mathcal{W}$ to indicate whether it has been modified since the last worker checkpoint.

4.1.3. Work unit division. When no more unassigned works are available, the coordinator generates a new work unit by splitting the largest work unit from $\mathcal{W}$. For that purpose, the coordinator keeps track of the sizes of work units, defined as

$$\|W_i\| = \sum_{j=1}^{K_i} (b_j^i - a_j^i).$$

Let W_v^c be the work unit selected for splitting and $K_i^{\max}$ the maximum number of intervals the requesting worker i can handle. The new work unit is generated by taking the right halves of the first $K_{new} = \min(K_i^{\max}, K_v)$ intervals from W_v^c. The new work unit of the requesting worker i is

$$W_{new} = \bigcup_{j=1}^{K_{new}} [\frac{a_v^j + b_v^j}{2}, b_v^j[$$

and the victim's copy of the work unit becomes

$$W_v^c = \left(\bigcup_{j=1}^{K_{new}} [a_v^j, \frac{a_v^j + b_v^j}{2}[\right) \cup \left(\bigcup_{j=K_{new}+1}^{K_v} [a_v^j, b_v^j[\right)$$

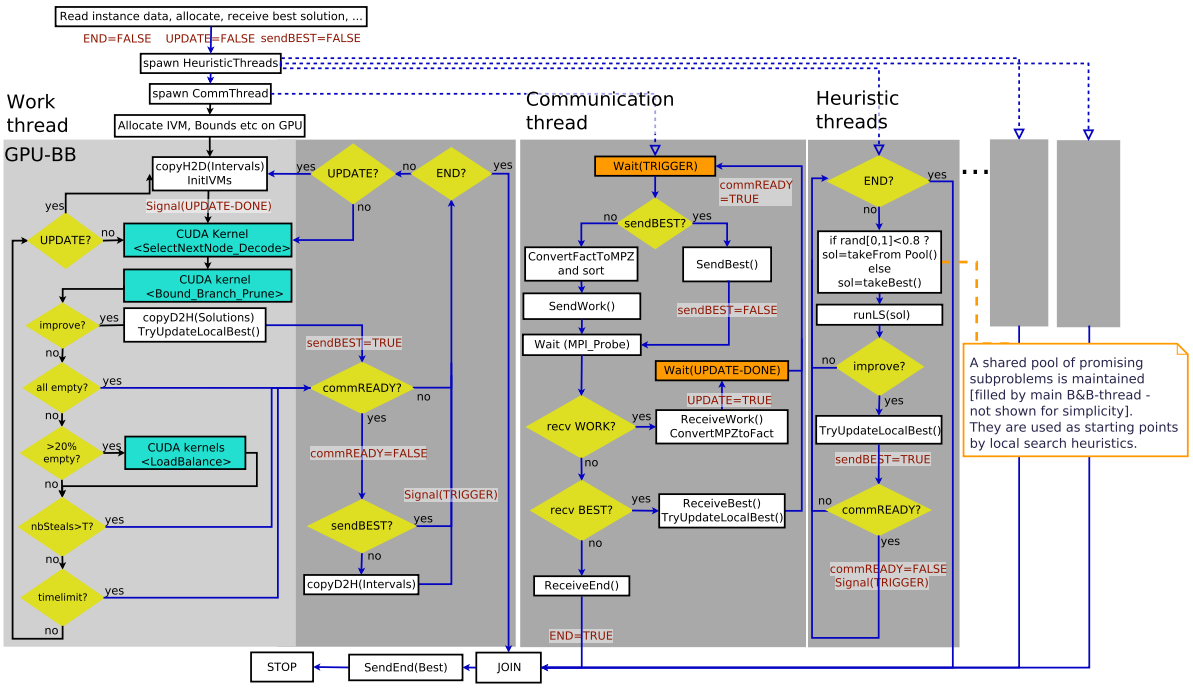


Figure 8 **PBB GPU-Worker Process.**

As mentioned above, after this operation, the victim's work unit is flagged as "modified" to make sure that the impacted worker performs a full intersection at the next checkpoint and updates its work unit.

4.1.4. Global checkpointing. Periodically, the coordinator saves the complete lists of unassigned and active work units, $\mathcal{W}_{\text{unassigned}}$ and $\mathcal{W}$, to a file. For a problem instance with $n = 100$ jobs, the size of a work unit of $K_i = 16384$ intervals is approximately

$$16384 \times 2 \times \frac{log_2(100!)}{8} B = 2.1 \text{ MB}$$

so with $n_p = 256$ workers the size of the checkpoint file grows to ~ 500 MB. In addition, the global checkpoint must contain the best found solution. When restarting PBB@Cluster from a global checkpoint, the coordinator reads the file and places the work units in $\mathcal{W}_{\text{unassigned}}$.

4.2. PBB@Cluster : Worker process

Figure 8 shows a flowchart of a worker process, composed of a PBB@GPU thread (controlling the GPU), a dedicated communication thread and multiple metaheuristic threads. The worker process is implemented using POSIX threads (`pthreads`) and the different worker components communicate through shared memory using mutexes and condition variables. For the sake of readability, details regarding synchronization and mutual exclusion primitives are spared out in Figure 8. Like PBB@GPU, the worker process starts by allocating and initializing data structures on the CPU and GPU. The initial best-found solution is received from the coordinator.

4.2.1. PBB@GPU thread The left part of Figure 8 corresponds to the PBB@GPU algorithm presented in Section 3, including a few modifications. Instead of stopping when

all work units are empty, a checkpoint communication is initiated to acquire new work. Moreover, communications with the coordinator are triggered when:

- an improved local best solution is discovered. The latter is sent to the coordinator as it might improve the global best.
- a fixed amount of GPU-based load balancing operations was performed. This threshold is set to $1/5$ of the sub-workers. The coordinator's copy of the work unit should be updated to avoid redundant exploration.
- a fixed amount of time has elapsed since the last worker-checkpoint. The purpose of this time-limit is to ensure that the global state of the search, kept by the coordinator, is updated regularly. We set this value to 30 seconds.

As shown in Figure 8, the PBB@GPU thread offloads all communications to a dedicated thread (described below). The PBB@GPU and communication threads basically interact in a producer-consumer pattern with single-item buffers. If the communicator thread is not ready (buffers are full), then the worker checks for global termination and pending updates before resuming to exploration work. Indeed, if no more local work is available, the worker thread quickly returns to the point where it checks the readiness of the communicator thread, busy-waiting for the buffer to become free. Otherwise, appropriate flags are set for the communicator thread, intervals are copied from the GPU to a send-buffer, and the worker returns to the PBB@GPU main-loop. The objective of this approach is that checkpoint operations or sending a new solution do not prevent the worker from making progress in the interval exploration.

4.2.2. Communication thread There are several reasons motivating the use of a dedicated communication thread:

- As mentioned in the previous section, the communicator handles intervals in decimal form, i.e. GMP integers, and interval-lists should be sorted to simplify the intersection operation. Using a separate communication thread, the burden of pre- and post-processing work unit communications is taken off the critical work path.
- It is one way to actually progress the message passing asynchronously (Vaidyanathan et al. 2015, Hoefler and Lumsdaine 2008).
- Although quite few best solutions are discovered throughout the search, new best solutions can be found by the PBB@GPU thread or by heuristic search threads. Offloading all communication to a single dedicated thread allows to use the `MPI_THREAD_SERIALIZED` thread-level instead of `MPI_THREAD_MULTIPLE`.
- Each message to the coordinator should be matched by an answer. In particular, if no more work is available, sending multiple subsequent work requests would cause multiple answers by the coordinator, overwriting each other. In our opinion, from a programming point of view, assuring this constraint with conventional non-blocking routines (e.g. `MPI_Isend` and `MPI_IProbe`) is at least as difficult as correctly synchronizing pthreads.

The main disadvantage is that one less CPU thread is available for computations—however, on most current systems this should be negligible. The flowchart of the communication thread is shown in the middle of Figure 8. In its default state, the communication thread waits on a condition variable to be triggered. If triggered, it then either sends the local best solution or the interval-list (after converting it from factoradic to decimal and sorting it). Then, the communicator thread waits for an answer from the coordinator.

If a WORK message is received, then the interval-list is converted to the factoradic form and the availability of an update is signaled to the work-thread. As shown in the left part of

Figure 8, the work-thread checks at each iteration whether an update is available. To ensure that the buffer can be safely reused, the communication thread blocks until the work-thread has copied the intervals to the device. Upon reception of a BEST message, which is the default answer from the coordinator, the thread attempts to update the local best solution. From coordinator to worker, BEST messages contain only the best makespan—not the corresponding schedule which is not needed by workers. The third possible message type is a termination message: it causes the communication thread to set the shared termination flag and join with the other worker-threads. Before shutting down, each worker sends a last message to the worker, containing the local best solution.

4.2.3. Heuristic threads The PBB@Cluster design presented up to this point leaves the computing power of additional CPU cores unused. For instance, each GPU-accelerated compute node on *Jean Zay* is composed of two 20-core CPUs and 4 GPUs, meaning that 32 additional CPU cores per node can be exploited. Preliminary experiments show that CPU-based BB-threads running on those cores only reach a fraction of the processing speed provided by the GPUs. PBB@Cluster therefore uses remaining CPU cores to run heuristic search algorithms. The exact BB search and heuristic searches cooperate in the following way. Periodically, the PBB@GPU thread promotes the current subproblems of all IVMs to solutions (by arbitrarily fixing the unscheduled jobs), evaluates them and adds the best resulting schedules to a fixed-size solution pool (containing up to twice as many solutions as the number of heuristic threads). These solutions are used as starting points for local searches that are allowed a fixed amount of time.

In principle, any kind of heuristic search can be used. However, two aspects are particularly important. Firstly, the heuristic searches should either be stochastic or strongly dependent on the starting solution. Otherwise, the multi-start parallel searches will find identical solutions. Secondly, for solving very hard instances, the local searches should have the ability to find very high-quality solutions if a long enough running time is allowed—rather than the ability to find good solutions very quickly. Investigating the performance of different heuristic methods in this hybridization approach goes beyond the scope of this paper.

In our attempts to solve hard problem instances, we mainly use an iterated local search (ILS) algorithm (Ruiz and Stützle 2007) using the k-insert recursive neighborhood proposed in (Deroussi et al. 2006). We also use a best-upper-bound-first BB search (truncated with stack-size- and time-limits), that prunes on LB1 *and* generates upper bounds from partial solutions by fixing unscheduled jobs in the order of appearance in the IVM-matrix. The behavior of this approach is biased by arranging jobs the first-row of the matrix according to the starting solution. Moreover, for each visited subproblem of a predefined depth, the beam-search algorithm recently proposed by (Libralesso et al. 2020) is applied on partial solutions, leaving prefix and postfix partial schedules unchanged.

Both approaches have shown good results, but no clear pattern emerged regarding the better heuristic search to use. Moreover, different methods for extracting solutions from the BB search should be investigated and the running time allowed for a heuristic search is fixed at 5 minutes. The hybridization of PBB@Cluster with approximate methods is still an experimental feature that requires more attention—we should note, however, that the hybridization allowed to solve instances whose resolution couldn't be achieved by the PBB algorithm alone.

Table 1 Instances used for single-GPU performance evaluation.

Instance	initial-UB	$n \times m$	tree-size	Instance	initial-UB	$n \times m$	tree-size
Ta021	2297	20×20	495 G	*Ta081*	6115	100×20	282 G
Ta056	3666	50×20	1444 G	*Ta101*	11156	200×20	371 G

The search is initialized with "initial-UB" ($\leq C^{\star}_{\max}$) and explores a tree of "tree-size" nodes (the set of nodes $\{\text{node}|\text{LB(node)} < \text{initial-UB}\}$.

5. Experimental evaluation

Section 5.1 provides details regarding the experimental environment. In Subsection 5.2 we experimentally evaluate the performance of the single-GPU implementation on different GPUs. In Subsection 5.3 we study the scalability of PBB@cluster on up to 384 GPUs. In Subsection 5.4 we report on the attempted resolution of the remaining open Taillard instances and discuss the results. Additional experimental results, as well as new best known solutions and proofs of optimality are given in the Online Supplement.

5.1. Experimental platform

Large-scale experiments are carried out on the GPU-accelerated partition of the *Jean Zay* supercomputer hosted at IDRIS[2]. The system has two partitions (accelerated and non-accelerated), ranked #64 and #108 respectively in the Top500 (November 2020). Accelerated nodes are equipped with **two** Intel Xeon Gold 6248 (Cascade Lake) processors and **four** Nvidia V100 SMX2 (32 GB) GPUs. Each V100 GPU has 80 streaming multiprocessors (SMs) for a total of 5120 FP32 Cuda cores clocked at 1.53 GHz (Boost Clock rate). *Jean Zay* is a HPE SGI 8600 System with Intel Omni-Path 100 GB/s interconnect. The OS is a Red Hat 8.1 Linux distribution and the job scheduler Slurm 18.08.8. For our experiments, we are limited to 384 GPUs (or 96 nodes) with a maximum duration of 20 hours for a single job. For development, testing and medium-scale experiments we also use the GPU-equipped clusters of Grid'5000, a large-scale and flexible testbed for experiment-driven research[3].

5.2. Evaluation of single-GPU performance

In this first experiment, the performance of PBB@GPU is evaluated and compared to an equivalent CPU-based PBB@multi-core implementation. PBB is initialized with an initial upper bound (UB) *smaller* than the known optimum : this ensures that the size of search trees is fixed, and small enough to be explored in 10-60 minutes on a single CPU-core. The selected instances, defined by $m = 20$ machines and $n = 20$, 50, 100 and 200 jobs, initial UBs and the corresponding tree sizes are shown in Table 1. The evaluation is performed with four different GPUs available in the Grid'5000 testbed: two gaming devices, GTX1080Ti and RTX2080Ti, based on the Pascal and Turing microarchitectures respectively; and two data-center GPUs (previously Tesla), the Pascal P100 and Volta V100 (PCIe versions). For all four, version 10.1 of the CUDA toolkit is used.

Figure 9 shows the relative speed-up of multi-core and GPU-based PBB compared to a sequential execution using a single CPU-core. The parallel multi-core version runs on a dual-socket NUMA system composed of two Intel Gold 6126 CPUs (2x12 cores) and

[2] Institute du développement et des ressources en informatique scientifique (national computing centre for the French National Centre for Scientific Research (CNRS), `https://www.idris.fr/`

[3] `https://www.grid5000.fr/`

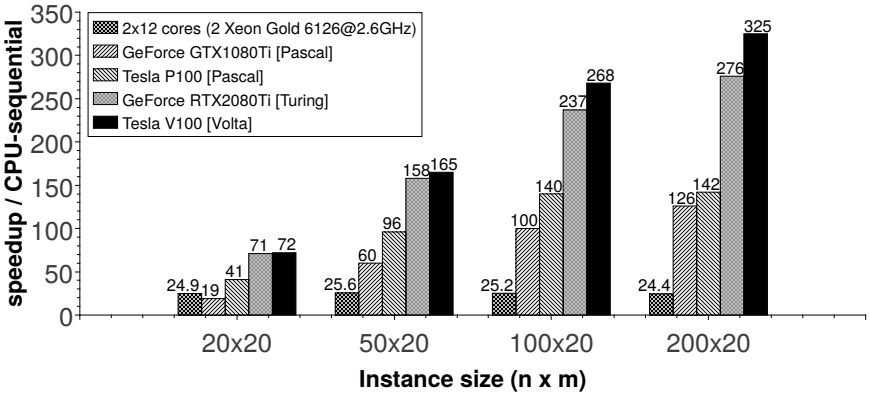


Figure 9 **Performance of PBB@GPU compared to sequential and multi-core CPU implementations, using the benchmark instances shown in Table 1.**

uses all 48 logical cores (hyperthreading enabled). The implementation uses pthread-based work stealing for load balancing between asynchronous exploration threads. In preliminary experiments, we determined that $K = 16384$ is a suitable value for the number of IVMs per GPU. One can see in Figure 9 that the multi-core B&B reaches speed-ups approximately equal to the number of physical CPU cores. The dual-socket 24-core system performs better than PBB@GPU only for the small 20×20 instance and the weakest of the four GPUs. In all other cases, PBB@GPU clearly outperforms PBB@multi-core, reaching speed-ups between $41\times$ (20×20 instance using a P100 device) and $325\times$ (200×20 instance on a V100 device) over sequential CPU execution. In other words, for instances of size 100×20 or 200×20, over 1000 CPU cores are needed to equal the processing power provided by a single quad-GPU node of *Jean Zay*.

One can notice a significant performance gap between the Pascal P100 and Volta V100 GPUs, and also between the Pascal- and Turing-based GeForce devices. To better understand the reasons for this improvement, PBB@GPU executions on the four different devices are profiled using the `nvprof` profiling tool: results are available in the Online Supplement.

5.3. Scalability experiments on Jean Zay

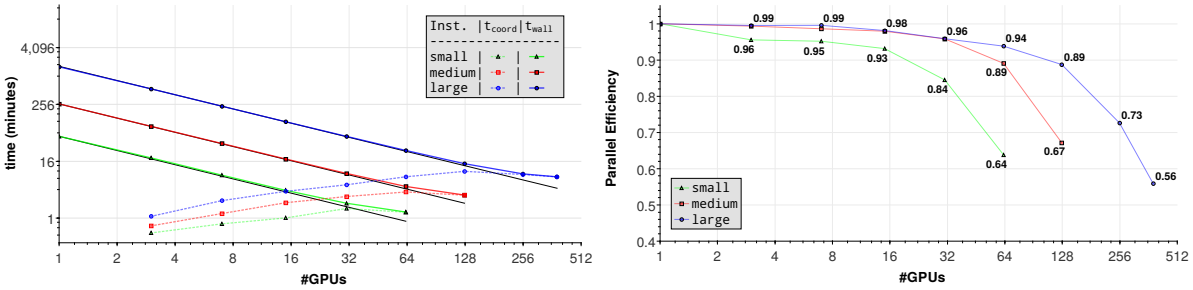
To perform a meaningful scalability analysis, we need to choose problem instances which are small enough to be solved within a reasonable amount of time on a single device and large enough to justify the use of multiple GPUs (single-GPU execution time between 1 hour and 1 day). Moreover, the selected benchmark instances should be associated with different total workloads (tree sizes), but the granularity of the workload should be the same (i.e. the number of jobs and machines defining the instances should be identical). Taillard's benchmark instances (except *Ta056*) are either too small or too large, so we selected three instances from the VRF benchmark, defined by $n = 30$ jobs and $m = 15$ machines (*30_15_2*, *30_15_5* and *30_15_9*).

To avoid speedup anomalies, we initialize PBB@Cluster with optimal solutions determined in preliminary runs. The sizes of explored critical trees (decomposed nodes) and corresponding single-GPU walltimes are shown in Table 2. To simplify the presentation of results, we refer to these instances as *small*, *medium* and *large*. The critical tree of the *small* instance is composed of 122 billion nodes and its exploration requires 54 minutes of processing on a single V100, which corresponds to an average processing speed of 37.6×10^6 nodes per second (NN/s). The *large* instance is 30 times larger, requiring over 27 hours

Table 2 **Summary of 30×15 VRF instances used for scalability experiments on Jean Zay.**

shorthand name	name	$C^{\star}_{\max}$	NN	$T_{\text{1-GPU}}$ (hh:mm)	NN/s
small	*30_15_2*	2317	122 G	0:54	37.6 M
medium	*30_15_5*	2421	564 G	4:22	35.8 M
large	*30_15_9*	2259	3660 G	27:23	37.1 M

For each instance the table gives the optimal makespan ($C^{\star}_{\max}$), size of the critical tree (NN), exploration time on one V100 device ($T_{\text{1-GPU}}$) and the corresponding processing speed (in decomposed nodes/second).



(a) Elapsed walltime and coordinator time

(b) Parallel efficiency

Figure 10 **Evaluation of scalability on Jean Zay.**

of processing at approximately the same speed. For each of the three instances, runs are performed with $1, 2, 4, \ldots, 2^k$ quad-GPU nodes until the observed parallel efficiency drops below 70%. Four MPI processes are mapped to each node and worker processes map to GPU devices via `MPI_Rank` (mod 4) and the `cudaSetDevice` API function. As the coordinator process occupies one slot on node 0, the corresponding number of GPUs is respectively $3, 7, \ldots, 2^{k+2} - 1$.

For the three instances and an increasing number of GPUs, Figure 10a shows the elapsed walltime (t_{wall}) with solid lines and the total active time of the coordinator (t_{coord}) with dotted lines. The coordinator is considered "active" when it is not waiting on `MPI_Probe`, i.e. t_{coord} includes the time process 0 spends receiving messages, converting intervals and processing worker requests. The linear scaling curve with respect to a single-GPU execution (no coordinator) is represented by black solid lines. However, as Figure 10a is drawn in log-log-scale, deviations from the ideal linear case are hard to see. Therefore, Figure 10b shows the corresponding parallel efficiency.

Parallel efficiency of at least 90% is achieved with up to 16, 32, 64 GPUs for the *small*, *medium* and *large* instances respectively; with 32, 64 and 128 GPUs, PBB@Cluster runs with efficiencies of 84, 89 and 89% respectively. For a larger number of GPUs the parallel efficiency drops off sharply, due to saturation of the coordinator process. Indeed, one can notice in Figure 10a that for ≥ 32, ≥ 64 and ≥ 128 GPUs, t_{coord} is close to t_{wall}, meaning that the coordinator is active nearly 100% of the time and becomes a sequential bottleneck. However, with 384 GPUs PBB@Cluster still reaches a speedup of 215× for the *large* instance, reducing the execution time from over 27 hours (single GPU) to about $7^1/_2$ minutes. The results of this experiment indicate that for larger instances (> 30 hours on a single device), PBB@Cluster can efficiently exploit several hundreds of GPUs, involving over millions of independent tree exploration agents ($K = 16384$ per GPU).

Table 3 Summary of solution attempts for benchmark instances Ta051-Ta060 (50×20).

				—solved—			
Instance	#GPUs	t_{elapsed}	GPUh	NN	∼CPU-time	known UB	$C^{\star}_{\max}$
Ta058	256	13h17	3399	339T	64 y	3691	3691
Ta053	128	7h59	1022	95T	19 y	3640	3640
Ta052	384	1h54	729.6	68T	14 y	3704	**3699**
Ta057	384	1h11	454.4	42T	8.5 y	3704	3704

	—remain open—	
	Best found	Comment
Ta051	3846	equal to UB from (Ravetti et al. 2012)
Ta054	3719	equal to UB from (Pan et al. 2008), (Kizilay et al. 2019)
Ta055	3610	equal to UB from (Deroussi et al. 2006)
Ta059	3741	equal to UB from (Pan et al. 2008)
Ta060	**3755**	**improved** upper bound

Exact solution of *Ta056* was first reported in Mezmaz et al. (2007). Details on re-solving *Ta056* are provided in the Online Supplement.

5.4. Resolution of open PFSP instances

The largest previously solved PFSP instance is the 50×20 instance *Ta056*, whose exact solution was first obtained by a 25-day run of PBB@Grid (Mezmaz et al. 2007), exploiting on average 328 CPUs. Over the last 5 years, we have re-solved *Ta056* several times on different platforms and with different PBB algorithms, including PBB@Cluster. Experimental results obtained with *Ta056* can be found in the Online Supplement.

In this subsection, we give feedback on our attempts to solve instances from the Taillard benchmark for which optimal solutions are unknown. There are 9 such instances in the 50-job/20-machine class, 9 in the 100×20 group and 5 in the 200×20 group. For the sake of clarity in the following presentation of results, let us briefly recall the possible outcomes of a PBB execution:

(a) A solution π with a better cost than the initial UB is found, but the algorithm does not terminate $\Rightarrow$ no proof of optimality, improved UB
(b) A solution π with a better cost than the initial UB is found and the algorithm terminates $\Rightarrow$ proof of optimality and improved UB
(c) The algorithm terminates but the initial UB is not improved $\Rightarrow$ the optimal solution is larger or equal to the initial UB
(d) The algorithm does not terminate and the initial UB is not improved $\Rightarrow$ no information

5.4.1. 50-job, 20-machine instances (Ta051-Ta060) Table 3 summarizes the execution statistics for the 9 unsolved instances of the 50×20 class—4 of them are solved to optimality for the first time. In all cases, the algorithm is initialized with the best-known solution from the literature. Clearly, even when optimal makespans for instances in this class are available, their optimality is very hard to prove. Taking for example *Ta058*, proving that no better solution than 3691 exists requires over 13 hours of processing on 256 GPUs, and 339×10^{12} node decompositions. Based on the CPU-GPU comparison shown in Figure 9, this corresponds to 64 CPU-years of sequential processing! For instances *Ta057* and *Ta053*, the best-known UB is also proven optimal.

In order to confirm the existence of a schedule with these optimal makespans, we solve *Ta057* a second time, finding the same optimal solution when initialized at $C^{\star}_{\max} + 1$. For

Table 4 Summary of solution attempts for benchmark instances Ta081-Ta090 (100×20).

			—solved—				
Instance	#GPUs	t_{elapsed}	GPUh	NN	∼CPU-time	old LB-UB	$C^{\star}_{\max}$
Ta083	64	0h24	25.8	2.2T	290 d	6252-6271	**6252**
Ta084	32	0h16	8.5	427G	95 d	6254-6269	**6254**
Ta090	128	78s	3	168G	31 d	6404-6434	**6404**
			—remain open—				
	old LB-UB	new LB-UB		ΔLB		Δ UB	
Ta081	6106-6202	6135-6173		+0.47%		-0.47%	
Ta085	6262-6314	6270-6286		+0.13%		-0.44%	
Ta086	6302-6364	6310-6331		+0.13%		-0.52%	
Ta087	6184-6268	6210-6224		+0.42%		-0.70%	
Ta088	6315-6401	6327-6372		+0.19%		-0.45%	
Ta089	6204-6247	6224-6275		+0.32%		-0.44%	
Avg				**+0.28%**		**-0.50%**	

The "old LB-UB" columns refer to the best-known lower (resp. upper) bounds in the literature, summarized on E. Taillard's website (Taillard 2015).

Ta058 and *Ta053*, additional explorations with the support of heuristic searches are performed until an optimal schedule is discovered. For *Ta052*, PBB@Cluster finds an improved schedule *and* proves its optimality in less than 2 hours, using 384 GPUs. Optimal permutations for these instances are provided in the Online Supplement.

Instances *Ta051*, *Ta054*, *Ta055*, *Ta59* and *Ta060* remain open, despite using 3-5000 GPUh per instance in solution attempts. The initial UB provided for instance *Ta060* is improved by one unit to 3755. For the remaining instances, PBB@Cluster is restarted with a larger initial UB and stopped when it finds the best-known UB. Enabling heuristic searches in PBB@Cluster, these solutions are found relatively quickly (usually within less than 1 hour). We can thus confirm the existence of the best-known solutions reported in the literature. For the sake of completeness, corresponding permutation schedules are provided at (Gmys 2021). Considering these observations, it seems likely that the best-known UBs for the open 50×20 instances are optimal, but proofs of optimality are very hard to obtain.

5.4.2. 100-job, 20-machine instances (Ta081-Ta090) For the 100-job instances *Ta081-Ta090*, prior to this work the exact solution was only known for *Ta082*. We add three instances to this list : *Ta083*, *Ta084* and *Ta090*. Solution statistics and improved upper bounds are summarized in Table 4. After initial, inconclusive solution attempts—without using heuristic search threads—we try to tackle instances in this group in two ways:

1. Using the best-known *lower bound* (as reported on E. Taillard's website (Taillard 2015)) as initial UB, PBB@Cluster proves that no better solution exists, i.e. returns without discovering a better schedule. Then, the initial UB is incremented by +1 and the search is restarted. Iterating over runs with an increasing initial UB, the best known lower bound is improved. This process is stopped when the algorithm finds and proves the optimality of a solution or when a fixed amount of time is elapsed.
2. The exploration is initialized with the best-known UB and heuristic searches are used to discover better solutions.

The first approach leads to the resolution of *Ta083*, for which the previously best-known LB (6252) is optimal. Starting from one unit above the best-known LB (6253), an optimal

schedule for *Ta083* is found and proven optimal in 24 minutes using 64 GPUs. The explored search tree is composed of 2.2×10^{12} nodes, which is much smaller than the trees explored for the 50×20 instances. Interestingly, once the optimal solution of *Ta083* is found, the search terminates almost instantly—indeed, initialized with the optimum 6252, the search completes within a few seconds by a sequential PBB algorithm.

The second approach provides improved solutions for all remaining instances of this group. However, the exact PBB search alone is not able to find these solutions and best-found solutions strongly depend on the quality of the supporting heuristic. Optimality proofs are produced for two instances, *Ta084* and *Ta090*, and in both cases the optimal makespan is equal to the best-known *lower* bound! The solution statistics shown in Table 4 correspond only to the run that resulted in the solution of the instance. For *Ta084* and *Ta090*, several PBB@Cluster executions were performed prior to that final run (decreasing the initial upper bound and restarted from previous checkpoints)—unfortunately, exploration statistics were lost when restarting the algorithm from a global checkpoint.

The fact that the search completes relatively quickly (for the solved instances) once an optimum is found, suggests that some of the remaining instances of the 100×20 class may be solved exactly—if heuristic searches are capable of finding an optimal solution. Indeed, contrary to the 50×20 class, the hardness of the three solved 100×20 instances stems from the difficulty of finding an optimal solution, while the optimality of the latter is relatively easy to prove.

For the six remaining instances, the exploration could not be completed despite using 2-10k GPUh of computation per instance. Although the required remaining time is by nature unpredictable, we use the total remaining work and LB-UB gaps as indicators for the hardness of an instance, and concentrate efforts on seemingly easier instances. For all unsolved instances, improved upper and lower bounds on the optimal makespan are reported in Table 4. One can see that for these instances, the previously best-known LBs are not optimal (all best-known LBs are improved, on average by 0.32%). Taking for example *Ta081*, we have $6135 \leq C^{\star}_{\max} \leq 6173$, which narrows down the previous LB-UB interval 6106—6202. On average, best-known UBs for the remaining 100×20 instances are improved by -0.50%. Permutation schedules for all improved UBs are provided in the Online Supplement.

It should be noted that for the 100×20 class, ARPD values (with respect to previous best-known UBs) of best-performing metaheuristics reported in the literature are in the order of $+0.5\%$ (Dubois-Lacoste et al. 2017, Kizilay et al. 2019). Our results show that, taking into account the improved UBs, actual optimality gaps of these methods are closer to $+1.0\%$.

5.4.3. 200-job, 20-machine instances (*Ta101-Ta110*) In the 200×20 class of Taillard's benchmark, 5 instances remain open, prior to this work. Four of them are solved exactly and the UB of the remaining instance is improved by 0.38%. All four exact solutions are obtained by successively running PBB@Cluster with increasing initial UBs. The exploration statistics shown in Table 5 correspond only to the last run which results in the instance's solution. The largest of these instances, *Ta101*, is solved in 18 minutes using 32 GPUs. Compared to the 50-job instances, most 200×20 instances are thus relatively "easy" to solve (although 18 minutes on 32 V100 GPU still correspond to an estimated computing time of 130 CPU-days!). For two of the four solved instances the previously best-known LB is optimal (*Ta107*, *Ta108*). Instance *Ta102* is much harder to solve. The range of possibly optimal makespan values is narrowed down to 11154–11160 (from 11143–11203), but

Table 5 Summary of solution attempts for benchmark instances Ta101-Ta110 (200×20).

			—solved—				
Instance	#GPUs	t_{elapsed}	GPUh	NN	∼CPU-time	old LB-UB	$C^{\star}_{\max}$
Ta101	32	0h18	9.6	225G	130 d	11152-11195	**11158**
Ta107	32	0h06	3	41G	41 d	11337-11360	**11337**
Ta109	32	100s	1	10G	4 d	11145-11192	**11146**
Ta108	32	28s	<1	2G	80 h	11301-11334	**11301**
			—remain open—				
	old LB-UB	new LB-UB	ΔLB		Δ UB		
Ta102	11143-11203	11154-11160	+0.10%		-0.38%		

The "old LB-UB" columns refer to the best-known lower (resp. upper) bounds in the literature, summarized on E. Taillard's website (Taillard 2015).

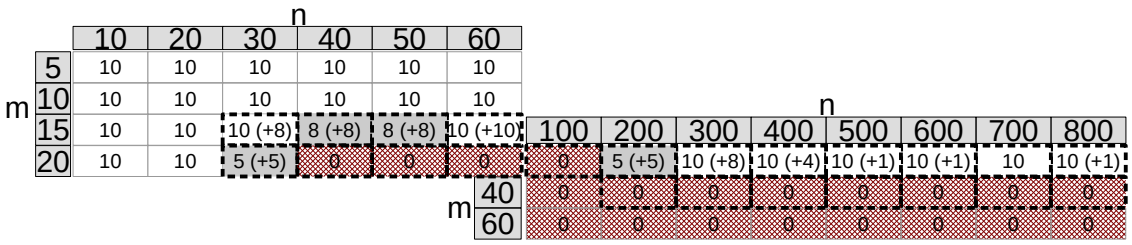


Figure 11 Summary of optimally solved VRF instances.

Note. In parentheses: number of optimal solutions reported in this work. Thick dashed cell borders indicate that the best-known solution for at least one instance in the class is improved. White: all 10 instances in the class are solved; Gray: some instances remain open; Hatched red: optimal solutions for all 10 instances remain unknown.

multiple attempts consuming several thousands of GPU-hours were unsuccessful in further increasing (resp. decreasing) the lower (resp. upper) bound.

5.4.4. VRF instances We also run PBB@Cluster on unsolved VRF instances (Vallada et al. 2015). Due to the large number of open instances in the VRF benchmark (271/480), no attempts are made to solve instances with $m = 60$ machines and the allocated computational budget per instance is smaller than for the Taillard benchmark. Overall, 55 instances are solved exactly for the first time and 122 best-known solutions (Vallada et al. 2015, Libralesso et al. 2020, Kizilay et al. 2019, Gmys et al. 2020) are improved. The largest exactly solved instance is *VRF30_20_1* (53.4×10^{12} decomposed nodes, 200 GPUh)—the previously best-known solution is optimal. Updated lists of best-known solutions and corresponding schedules are provided at (Gmys 2021). Figure 11 summarizes, for each of the 48 instance classes, the number of known exact solutions and (in parentheses) the number of optimal solutions provided in this paper. One can see that the unsolved instances with $m = 20$ machines are centered around $n = 40$-100, which is consistent with the results obtained for Taillard's benchmark. All instances with $m \geq 40$ machines remain open, although some of the best-known solutions are improved.

6. Conclusions and future work

We have presented a Branch-and-Bound (BB) algorithm for exactly solving permutation-based combinatorial optimization problems on GPU-accelerated supercomputers (PBB@Cluster). The permutation flow-shop scheduling problem (PFSP) with makespan minimization is used as a test-case. Using PBB@Cluster we solved 11 of the 23 open Taillard

benchmark instances (1993) to optimality for the first time. Moreover, best-known upper bounds are improved for 8 remaining instances. Instance *Ta056* is solved to optimality in less than 3 minutes on the *Jean Zay* supercomputer, which is a four-orders-of-magnitude improvement over the first exact solution in 2006, which required 25 days of computation, using a grid-enabled BB algorithm (Mezmaz et al. 2007).

Computational experiments demonstrate the efficiency of our approach. Using a single V100 GPU, we observe speed-ups of 325× compared to a single-threaded CPU-based implementation. The scalability of PBB@Cluster, using millions of GPU-based concurrent tree searches on up to 384 V100 GPUs (2 million CUDA cores) has been evaluated. An instance requiring 27 hours on a single GPU, is solved in 14 minutes on 128 GPUs, i.e. with 90% parallel efficiency. The largest instance solved to optimality (*Ta058*) requires an equivalent computing power of 64 CPU-years—on 256 GPUs it is solved in 13 hours, exploring a tree composed of 340×10^{12} nodes.

Although, to the best of our knowledge, this is the first deployment of fine-grained BB on a peta-scale supercomputing system, these results could not be obtained by sheer brute-force. Rather, they were made possible though a combination of recent algorithmic progress in the design of sequential BB for the PFSP (Gmys et al. 2020) and a thoroughly revisited parallel BB, specifically designed for GPUs and GPU-accelerated clusters. In this paper we addressed in a comprehensive way several challenging and performance-critical issues that arise at different levels of the algorithm-to-platform mapping, most of them related to the algorithm's highly irregular nature.

As shown in (Gmys 2017), the proposed approach can by applied to other permutation-based optimization problems, provided that the lower bounding operation can be carried out efficiently on the GPU. However, an extension to other solution encodings is not straightforward as it would require revisiting the data structures used for managing the pool of subproblems and the definition of work units.

In the short term future, we plan to investigate the hybridization of exact PBB and approximate search methods, which has shown promising results. We will also investigate the use of high-productivity PGAS-based parallel computing environments, such as Chapel, that could greatly simplify the implementation of PBB@Cluster and parallel tree-search algorithms in general.

The source code for PBB@Cluster is available at

`https://github.com/jangmys/pbb`

Acknowledgments

This work was granted access to the HPC resources of IDRIS under the allocation 2020-A0070611107 made by GENCI. Some of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Also, helpful advice by N. Melab, M. Mezmaz and D. Tuyttens on the presentation of this paper is gratefully acknowledged.

References

Anstreicher K, Brixius N, Goux JP, Linderoth J (2002) Solving large quadratic assignment problems on computational grids. *Mathematical Programming* 91(3):563–588, URL `http://dx.doi.org/10.1007/s101070100255`.

Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, Yelick K (2009) A View of the Parallel Computing Landscape. *Commun. ACM* 52(10):5667, ISSN 0001-0782, URL `http://dx.doi.org/10.1145/1562764.1562783`.

Bader DA, Hart WE, Phillips CA (2005) *Parallel Algorithm Design for Branch and Bound*, 5–1–5–44 (New York, NY: Springer New York), ISBN 978-0-387-22827-3, URL http://dx.doi.org/10.1007/0-387-22827-6_5.

Bożejko W (2009) Solving the flow shop problem by parallel programming. *Journal of Parallel and Distributed Computing* 69(5):470 – 481, ISSN 0743-7315, URL http://dx.doi.org/https://doi.org/10.1016/j.jpdc.2009.01.009.

Cappello F (2009) Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications* 23(3):212–226.

Carneiro T, Muritiba A, Negreiros M, de Campos GL (2011) A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU. *Computer Architecture and High Performance Computing, Symposium on*, 41–47 (Los Alamitos, CA, USA: IEEE Computer Society), ISSN 1550-6533, URL http://dx.doi.org/10.1109/SBAC-PAD.2011.20.

Chakroun I, Melab N (2015) Towards a heterogeneous and adaptive parallel Branch-and-Bound algorithm. *Journal of Computer and System Sciences* 81(1):72 – 84, ISSN 0022-0000, URL http://dx.doi.org/https://doi.org/10.1016/j.jcss.2014.06.012.

Chakroun I, Melab N, Mezmaz M, Tuyttens D (2013) Combining multi-core and GPU computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing* 73(12):1563 – 1577, ISSN 0743-7315, URL http://dx.doi.org/https://doi.org/10.1016/j.jpdc.2013.07.023, heterogeneity in Parallel and Distributed Computing.

Crainic TG, Le Cun B, Roucairol C (2006) *Parallel Branch-and-Bound Algorithms*, chapter 1, 1–28 (John Wiley & Sons, Ltd), ISBN 9780470053928, URL http://dx.doi.org/https://doi.org/10.1002/9780470053928.ch1.

Date K, Nagi R (2019) Level 2 Reformulation Linearization TechniqueBased Parallel Algorithms for Solving Large Quadratic Assignment Problems on Graphics Processing Unit Clusters. *INFORMS Journal on Computing* 31(4):771–789, URL http://dx.doi.org/10.1287/ijoc.2018.0866.

de Bruin A, Kindervater GAP, Trienekens HWJM (1995) Asynchronous parallel branch and bound and anomalies. Ferreira A, Rolim J, eds., *Parallel Algorithms for Irregularly Structured Problems*, 363–377 (Berlin, Heidelberg: Springer Berlin Heidelberg), ISBN 978-3-540-44915-7.

Deroussi L, Gourgand M, Norre S (2006) New effective neighborhoods for the permutation flow shop problem. Technical Report LIMOS/RR-06-09, LIMOS, Université Clermont Auvergne, URL https://hal.archives-ouvertes.fr/hal-00678053/.

Dubois-Lacoste J, Pagnozzi F, Stützle T (2017) An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Computers & Operations Research* 81:160 – 166, ISSN 0305-0548, URL http://dx.doi.org/https://doi.org/10.1016/j.cor.2016.12.021.

Fonlupt C, Marquet P, Dekeyser JL (1994) A data-parallel view of the load balancing experimental results on MasPar MP-1. Gentzsch W, Harms U, eds., *High-Performance Computing and Networking*, 338–343 (Berlin, Heidelberg: Springer Berlin Heidelberg), ISBN 978-3-540-48408-0.

Garey MR, Johnson DS, Sethi R (1976) The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research* 1(2):pp. 117–129, ISSN 0364765X.

Gendron B, Crainic TG (1994) Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research* 42(6):1042–1066, ISSN 0030364X, 15265463, URL http://www.jstor.org/stable/171985.

Gmys J (2017) *Heterogeneous cluster computing for many-task exact optimization - Application to permutation problems*. Ph.D. thesis, Université de Mons / Université de Lille, URL https://hal.inria.fr/tel-01652000.

Gmys J (2021) Permutation Flow-shop : best-known makespans and schedules for Taillard and VRF benchmarks [Data Set]. http://doi.org/10.5281/zenodo.4542886, accessed: 2021-02-16.

Gmys J, Mezmaz M, Melab N, Tuyttens D (2016) A GPU-based Branch-and-Bound algorithm using Integer-Vector-Matrix data structure. *Parallel Comput.* 59:119–139, URL http://dx.doi.org/10.1016/j.parco.2016.01.008.

Gmys J, Mezmaz M, Melab N, Tuyttens D (2017) IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems. *Concurrency and Computation: Practice and Experience* 29(9):e4019, URL http://dx.doi.org/https://doi.org/10.1002/cpe.4019, e4019 cpe.4019.

Gmys J, Mezmaz M, Melab N, Tuyttens D (2020) A computationally efficient Branch-and-Bound algorithm for the permutation flow-shop scheduling problem. *European Journal of Operational Research* 284(3):814 – 833, ISSN 0377-2217, URL http://dx.doi.org/https://doi.org/10.1016/j.ejor.2020.01.039.

Grabowski J, Skubalska E, Smutnicki C (1983) On Flow Shop Scheduling with Release and Due Dates to Minimize Maximum Lateness. *Journal of the Operational Research Society* 34(7):615–620, URL http://dx.doi.org/10.1057/jors.1983.142.

Grama A, Kumar V (1995) Parallel Search Algorithms for Discrete Optimization Problems. *ORSA Journal on Computing* 7(4):365–385, URL http://dx.doi.org/10.1287/ijoc.7.4.365.

Hoefler T, Lumsdaine A (2008) Message progression in parallel computing - to thread or not to thread? *2008 IEEE International Conference on Cluster Computing*, 213–222, URL http://dx.doi.org/10.1109/CLUSTR.2008.4663774.

Ignall E, Schrage L (1965) Application of the Branch and Bound Technique to Some Flow-Shop Scheduling Problems. *Oper. Res.* 13(3):400–412, ISSN 0030-364X, URL http://dx.doi.org/10.1287/opre.13.3.400.

Jenkins J, Arkatkar I, Owens JD, Choudhary A, Samatova NF (2011) Lessons Learned from Exploring the Backtracking Paradigm on the GPU. Jeannot E, Namyst R, Roman J, eds., *Euro-Par 2011 Parallel Processing*, 425–437 (Berlin, Heidelberg: Springer Berlin Heidelberg), ISBN 978-3-642-23397-5.

Johnson SM (1954) Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* 1(1):61–68, ISSN 1931-9193, URL http://dx.doi.org/10.1002/nav.3800010110.

Karypis G, Kumar V (1994) Unstructured Tree Search on SIMD Parallel Computers. *IEEE Trans. Parallel Distributed Syst.* 5:1057–1072.

Kizilay D, Tasgetiren MF, Pan QK, Gao L (2019) A variable block insertion heuristic for solving permutation flow shop scheduling problem with makespan criterion. *Algorithms* 12(5):100.

Knuth D (1997) The Art of Computer Programming, Volume 2: Seminumerical Algorithms. *Reading, Ma* 192, iSBN=9780201896848.

Lageweg BJ, Lenstra JK, Kan AHGR (1978) A General Bounding Scheme for the Permutation Flow-Shop Problem. *Operations Research* 26(1):53–67, URL http://dx.doi.org/10.1287/opre.26.1.53.

Libralesso L, Focke PA, Secardin A, Jost V (2020) Iterative beam search algorithms for the permutation flowshop, URL https://hal.archives-ouvertes.fr/hal-02937115, working paper or preprint.

Lomnicki ZA (1965) A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem. *Journal of the Operational Research Society* 16(1):89–100, ISSN 1476-9360, URL http://dx.doi.org/10.1057/jors.1965.7.

Melab N, Gmys J, Mezmaz M, Tuyttens D (2018) Multi-core versus many-core computing for many-task Branch-and-Bound applied to big optimization problems. *Future Generation Computer Systems* 82:472 – 481, ISSN 0167-739X, URL http://dx.doi.org/https://doi.org/10.1016/j.future.2016.12.039.

Mezmaz M, Leroy R, Melab N, Tuyttens D (2014) A multi-core parallel branch-and-bound algorithm using factorial number system. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 1203–1212, ISSN 1530-2075, URL http://dx.doi.org/10.1109/IPDPS.2014.124.

Mezmaz M, Melab N, Talbi E (2007) A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. *2007 IEEE International Parallel and Distributed Processing Symposium*, 1–9, URL http://dx.doi.org/10.1109/IPDPS.2007.370217.

Nowicki E, Smutnicki C (1996) A fast tabu search algorithm for the permutation flow-shop problem. *European Journal of Operational Research* 91(1):160–175, ISSN 0377-2217, URL http://dx.doi.org/https://doi.org/10.1016/0377-2217(95)00037-2.

Pan QK, Tasgetiren MF, Liang YC (2008) A discrete differential evolution algorithm for the permutation flowshop scheduling problem. *Computers & Industrial Engineering* 55(4):795–816.

Pessoa TC, Gmys J, Melab N, de Carvalho Junior FH, Tuyttens D (2016) A GPU-Based Backtracking Algorithm for Permutation Combinatorial Problems. Carretero J, Garcia-Blas J, Ko RK, Mueller P, Nakano K, eds., *Algorithms and Architectures for Parallel Processing*, 310–324 (Cham: Springer International Publishing), ISBN 978-3-319-49583-5.

Potts C (1980) An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research* 5(1):19 – 25, ISSN 0377-2217, URL http://dx.doi.org/https://doi.org/10.1016/0377-2217(80)90069-7.

Pruul E, Nemhauser G, Rushmeier R (1988) Branch-and-bound and parallel computation: A historical note. *Operations Research Letters* 7(2):65 – 69, ISSN 0167-6377, URL http://dx.doi.org/https://doi.org/10.1016/0167-6377(88)90067-3.

Rao VN, Kumar V (1987) Parallel Depth First Search. Part I. Implementation. *Int. J. Parallel Program.* 16(6):479499, ISSN 0885-7458, URL http://dx.doi.org/10.1007/BF01389000.

Ravetti MG, Riveros C, Mendes A, Resende MG, Pardalos PM (2012) Parallel hybrid heuristics for the permutation flow shop problem. *Annals of Operations Research* 199(1):269–284.

Reinefeld A, Schnecke V (1994) Work-load balancing in highly parallel depth-first search. *Proceedings of IEEE Scalable High Performance Computing Conference*, 773–780 (IEEE).

Rocki K, Suda R (2010) Parallel Minimax Tree Searching on GPU. Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J, eds., *Parallel Processing and Applied Mathematics*, 449–456 (Berlin, Heidelberg: Springer Berlin Heidelberg), ISBN 978-3-642-14390-8.

Ruiz R, Stützle T (2007) A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research* 177(3):2033 – 2049, ISSN 0377-2217, URL http://dx.doi.org/https://doi.org/10.1016/j.ejor.2005.12.009.

Taillard E (1993) Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64(2):278 – 285, ISSN 0377-2217, URL http://dx.doi.org/https://doi.org/10.1016/0377-2217(93)90182-M, project Management anf Scheduling.

Taillard E (2015) Summary of best known lower and upper bounds of Taillard's instances. http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/flowshop.dir/best_lb_up.txt, accessed: 2022-03-11.

Vaidyanathan K, Kalamkar DD, Pamnany K, Hammond JR, Balaji P, Das D, Park J, Jo B (2015) Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12, URL http://dx.doi.org/10.1145/2807591.2807602.

Vallada E, Ruiz R, Framinan JM (2015) New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research* 240(3):666 – 677, ISSN 0377-2217, URL http://dx.doi.org/https://doi.org/10.1016/j.ejor.2014.07.033.

Vu TT, Derbel B (2016) Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments. *Future Generation Computer Systems* 56:95 – 109, ISSN 0167-739X, URL http://dx.doi.org/https://doi.org/10.1016/j.future.2015.10.009.