Manuscript Click here to view linked References

Scheduling multi-staged jobs on parallel identical machines and a central server with sequence-dependent setup times: an application to an automated kitchen

Simon Belieres [1a], Yossiri Adulyasak[b], Jean-François Cordeau[b]

[a] TBS Business School, 20 Bd Lascrosses, 31000 Toulouse
[b] HEC Montréal, 3000 Chem. de la Cte-Sainte-Catherine, Montréal, QC H3T 2A7, Canada

Abstract

We address the problem of scheduling multi-staged jobs in a production environment with parallel identical machines and a central server with sequence-dependent setup times motivated by a real-life application in a robotic automated kitchen. The proposed model addresses a makespan minimization offline optimization problem that arises in the context of mise-en-place preparation. We present two mathematical formulations, one with a machine index and one without, as well as lower bounds. We also propose an effective general variable neighborhood search algorithm based on heuristics which aims to improve the incumbent by eliminating idle times. Through a series of experiments on random instances, we demonstrate the effectiveness of the proposed method under various combinations of instance parameters. On instances with known optimal solutions, the metaheuristic produces solutions with an average gap of 0.31% in very short computation times. On large-size instances, the metaheuristic produces solutions with an average gap of 6.04% when measured against a known lower bound. We also present a real-life case study provided by our industrial partner and we investigate how to maximize the output of the studied automated production system.

Keywords: Identical parallel machines, Central server, Sequence-dependent setup times, Scheduling, Variable Neighborhood Search, Cooking process automation

1. Introduction

The automation of manufacturing processes is an important contributor to the success of companies as it improves production activities in several ways. Economically speaking, automation is cost-effective as it often reduces labor costs while increasing productivity. In addition, the automation of manufacturing processes generally results in more sustainable production by reducing inventory, waste, unnecessary processing, and lead times. In flexible manufacturing systems, many automated production system configurations consist of a set of parallel processing machines and a central server. Applications exist, for example, in the printing industry [23], where a team of workers perform setup operations between jobs performed on the presses, such as cleaning, or resetting

[1] Corresponding author: s.belieres@tbs-education.fr

in the case of subsequent jobs with different colors or paper sizes. Similar production environments are found in other industrial applications, such as the plastic parts industry [7] or automative parts manufacturing [28].

The problem we study is motivated by the restaurant industry, which is arguably one of the largest industries in the world. In the United States alone, restaurants have annual sales of US $ 799 billion [5] and employ over 14 million employees. While cooking processes are repetitive and not free of charge, in practice, most of these processes are still operated by humans. As such, increasing the level of automation could both improve production profitability and sustainability. In addition, automation would allow decreasing the high turnover rate of employees faced by this sector, and thus reduce the associated negative effects. While it is unlikely that restaurant cooks will be replaced by humanoid chefs shortly, the development of versatile and multifunctional food processors allows for a significantly higher level of automation in the restaurant industry. These multifunctional food processors - which can perform various cooking operations such as knead, chop, blend, stir, heat, or keep warm - are able to prepare a wide range of dishes as long as they are fed the right ingredients in the right quantities at the right time. Therefore, a potential production system configuration for the automation of cooking processes consists of a set of multifunctional food processors - also referred to as pots - placed in parallel, a shelf storing the ingredients necessary for preparing the dishes, and a central server serving as a bridge between the shelf and the multifunctional food processors (see Figure 1).

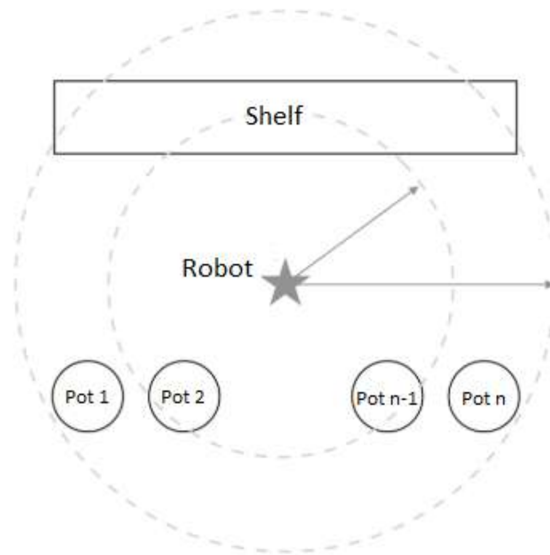


Figure 1: Automated kitchen production system configuration

In that context, one seeks to schedule a set of jobs in a production environment that includes M parallel machines and a central server. Each job, which is characterized by its corresponding recipe, is composed of multiple operations that decompose into two sets. *Loading* operations, also referred to as *type-2* operations, model the dispensing of an ingredient by the central server into a machine, and thus require these two resources

simultaneously. *Processing* operations, also referred to as *type-1* operations, model the multifunctional food processor cooking operations, and thus only require a single machine. The operations of each job are subject to precedence relationships that form a "forward flow" multi-stage graph, where the operations of a given stage are direct predecessors of the operations of the subsequent stage. In addition, each precedence graph is composed of an even number of stages, where odd stages consist of one or multiple type-2 operations and even stages consist of a single type-1 operation. Practically speaking, this represents the fact that a recipe alternates between (i) dispensing one or more ingredients into the multifunctional food processor and (ii) processing the ingredients contained in the multifunctional food processor. Neither operations nor jobs are preemptable. On the one hand, this implies that any operation started must be executed until completion. On the other hand, this implies that if one starts a job on a given machine, all the job operations must be completed on that particular machine before the machine is free again. The consecutive execution of two type-2 operations incurs a sequence-dependent setup time that reflects the time required for the central server to (i) store the current ingredient at its position on the shelf, (ii) grab a new ingredient from the shelf, (iii) move to the next machine. Note that the duration of type-2 operations depends on the quantity and type of ingredient considered. In particular, ingredient properties such as their shape, weight, or granularity, play a role in the dispensing time. For instance, liquids or rice are easier to dispense than banana or onion slices. Similarly, ingredients that need to be dispensed with high-precision, such as condiments or spices, take longer.

The above-mentioned problem shares similarities with identical parallel machine scheduling problems with a central server [32] as well as problems regarding scheduling multi-staged jobs in an environment comprising a central server, such as flow-shop problems [24] or job-shop problems [26] with a central server. As in parallel machine scheduling problems with a central server, we consider a production environment with M identical machines that can perform any job and a central server that allows to set up the jobs. However, the literature related to parallel machine scheduling focuses on jobs that consist of a single operation with fixed processing time. In this study, we consider multi-staged jobs where some operations require the use of a shared resource, such that job processing times may vary from one schedule to another. Note that, as the parallel scheduling problem with two identical machines and a common server is proved to be NP-hard (Koulamas [32], Kravchenko and Werner [33]), our extension with M machines and multi-staged jobs is NP-hard as well. Flow-shop and job-shop problems with a central server focus on scheduling similar multi-staged jobs. Nevertheless, resource management is completely different in these problems as different operations of a given job need to be performed on different machines, such that the need to assign jobs to machines is not relevant.

Meal preparation in the restaurant industry is a two-stage process: mise-en-place and peak-time. Mise-en-place happens before customer arrivals and aims to alleviate the workload by getting a head start on operations that can be carried out in advance. Part of the mise-en-place consists in preparing subsets of the recipes - e.g. prepare whipped cream and pastry for a cream puff recipe - to simplify the cooking during peak time. Mise-en-

place preparation relies on a forecast of the peak-time demand and can be formulated as an offline optimization problem. Since mise-en-place aims to prepare subsets of the recipes without particular priority and as efficiently as possible, an appropriate objective function is makespan minimization. Meanwhile, the preparation of meals during peak time takes place in a highly dynamic environment and depends on real-time customer demands. Consequently, peak-time operations should be formulated as an online optimization problem, with an objective function that minimizes customer waiting times. In this paper, we focus on the case of the offline optimization problem with makespan minimization.

Our contribution is threefold. First, we introduce two mathematical formulations for scheduling multi-stage jobs in a production environment with parallel identical machines and a central server with sequence-dependent setup times, as well as lower bounds. Second, we propose an effective metaheuristic algorithm for this difficult real-life scheduling problem. Third, we provide an extensive computational study performed on both random instances and instances provided by our industrial partner. We use random instances to assess the performance of our solution algorithm and analyze how instance parameters impact the problem difficulty. We then analyze the solutions obtained by our method on a real-life instance and discuss the appropriate number of machines to maximize the output of our partner production system. The remainder of the paper is organized as follows. In Section 2, we review the literature related to scheduling with a central server. Section 3 aims to define the studied problem and present the corresponding mixed-integer programming formulations. Section 4 focuses on the proposed solution algorithm. We provide an extensive computational study in Section 5, and conclusions as well as a discussion of future work in Section 6.

2. Literature review

We first review the literature on parallel machine scheduling with a central server. We then review the literature on problems regarding scheduling multi-staged jobs in an environment comprising a central server, i.e. flow-shop problems or job-shop problems with a central server.

2.1. Parallel machine scheduling with a central server

Parallel machine scheduling is a family of scheduling problems that consist of scheduling jobs with fixed processing times on M machines. Machines can be classified into three categories. Machines are *identical* if they have similar processing times, *uniform* if they have different speeds but work at a consistent rate, and *unrelated* otherwise. In the most basic version, each job has to be processed without preemption, which yields two types of decisions: job-machine assignment and sequencing. Parallel machine scheduling with a central server is an extension where machines share a common resource for setting up jobs. In the case of makespan minimization and sequence-dependent setup times, the parallel machine scheduling problem with a central server is denoted by $P, S1|p_j, s_{ij}|C_{max}$ using the standard three-field notation described by Graham et al. [16].

Early works on that problem focus on complexity analysis. Koulamas [32] introduces the parallel machine scheduling problem with a central server and focuses on a variant with two parallel identical machines as well as sequence-independent setup times. The objective function, denoted as IT, aims to minimize machine idle times. Koulamas [32] shows that the problem of $P2, S1|s_i|IT$ is strongly NP-hard, and proposes a beam search heuristic that leverages a problem reduction procedure. Kravchenko and Werner [33] study a makespan minimization variant where all setup times are equal to one. They propose a pseudopolynomial algorithm for the problem $P2, S1|s_i = 1|C_{max}$, and demonstrate that the same problem with an arbitrary number of machines, i.e. $P, S1|s_i = 1|C_{max}$, is unary NP-hard. In a further article, Kravchenko and Werner [34] extend their complexity results to many variants with the objective function IT. They also study the case with multiple servers and demonstrate that $Pm, Sk|s_i = 1|C_{max}$ is binary NP-hard.

Glass et al. [15] analyze the complexity of parallel machine scheduling problems with a central server in the case of dedicated machines. They provide heuristics with a worst-case performance ratio bound of $\frac{3}{2}$ for the case with two machines, and a worst-case performance ratio bound of 2 for the general case. Hall et al. [17] and Brucker et al. [9] derive new complexity results for special cases with various objective functions, namely $L_{max}, \sum C_j, \sum w_j C_j, \sum U_j, \sum w_j U_j, \sum T_j$, and $\sum w_j T_j$. Abdekhodaee and Wirth [1] study variants with short processing times, i.e. all job processing times are shorter than all setup times, and equal-length jobs. They provide polynomial-time algorithms for these problems, as well as heuristics of log-linear complexity for the general case. Abdekhodaee et al. [2] focus on cases with equal processing times and equal setup times, which were proved to be NP-complete.

One stream of the literature related to parallel machine scheduling with a central server focuses on solution algorithms to solve the case with two machines. Similarily to the complexity-related studies, most of these works focus on the version with sequence-independent setup times. Abdekhodaee et al. [3] introduce heuristics, a genetic algorithm, and Gilmore-Gomory procedures for $P2, S1|s_i|C_{max}$. They compare algorithm performance on instances with 50 and 100 jobs, and varying server loads, i.e. ratio of the mean setup time to the mean processing time. They find that the genetic algorithm performs better in the case of small server loads, while the Gilmore-Gomory procedure is more effective otherwise. Gan et al. [14] present two mixed-integer linear programming (MILP) formulations for the same problem, as well as two branch-and-price schemes. They observe that solving the MILP formulations is effective for small-size instances only and that branch-and-price schemes achieve superior performance as the number of jobs increases. Hasani et al. [19] propose a block-based MILP formulation and demonstrate its superior performance compared to that of Gan et al. [14], on instances with up to 250 jobs. Hasani et al. [21] develop a simulated annealing algorithm and a genetic algorithm that outperforms previous methods on instances with up to 1000 jobs, as well as greedy heuristics [22] for very large-size instances with up to 100,000 jobs. The same authors [20] also develop a MILP formulation and a tabu search for the case where the objective function aims to minimize machine idle times.

Huang et al. [23] investigate parallel machine scheduling with a central server and an arbitrary number of machines. They consider sequence-dependent setup times, makespan minimization, and propose two lower bounds, a MILP formulation, and a hybrid genetic algorithm. While dedicated machines are considered, i.e. job assignment is not a decision, the computational study indicates that obtaining small gaps is difficult for instances with 100 jobs, particularly in the case of heterogeneous setup times. Kim and Lee [29] study the case with sequence-independent setup times and identical parallel machines. They propose two MILP formulations based on the server setup sequence and the server waiting times, respectively, as well as dominance conditions and a metaheuristic hybridizing simulated annealing and tabu search. Their approach is compared to the heuristics in [1] as well as the genetic algorithm in [3], and appears to be superior on instances with up to 40 jobs and 8 machines. Instances with high server loads are the most challenging, yielding gaps greater than 13% in the worst case. Hamzadayi and Yildiz [18] consider the case with sequence-independent setup times. They propose a MILP formulation as well as solution approaches based on simulated annealing and a genetic algorithm. They computationally demonstrate that the MILP is tractable for small-size instances only. The metaheuristics outperform list scheduling heuristics. However, no lower bounds are proposed to assess the quality of the obtained solutions. Elidrissi et al. [11] study the case with sequence-dependent setup times and provide four MILP formulations based on network variables, linear ordering variables, completion time variables, and time-indexed variables, respectively. The formulation based on time-indexed variables outperforms the other formulations and solves 5 out of 10 instances with 100 jobs and 5 machines within a 1-hour time limit. For the 5 remaining instances, the same formulation yields an average optimality gap of approximately 35%. The authors also introduce a MILP formulation for the case of regular job sets, for which instances with up to 500 jobs and 5 machines are solved with very short computational times.

The problem considered in this paper and parallel machine scheduling problems with a central server are similar as both aim to schedule a set of jobs that can be processed on a set of identical parallel machines while needing a central server for setup operations. However, parallel machine scheduling problems with a central server focus on jobs with one operation while we have multi-staged jobs with precedence-constrained operations, as in flow-shop scheduling and job-shop scheduling.

2.2. Multi-stage job scheduling with a central server

The flow-shop scheduling problem aims to schedule multi-staged jobs, i.e. jobs composed of multiple operations, on M machines. The jobs are similar in the sense that they all have the same number of operations, and these operations need to be processed in the same order through the machines. In the variant with a central server, job transportation from one machine to another is considered and executed by a central resource - also referred to as *central robot*. The first studies have focused on the complexity of special cases. Kise [30] and Kise et al. [31] focus on a variant with two machines, a central server, makespan minimization, and fixed server transportation times. They demonstrate that this variant, i.e. $F2; R1|t_j = T|C_{max}$, is NP-hard in the ordinary sense. Hurink and Knust [24] extend this result by showing that $F2; R1|t_j = T|C_{max}$, is NP-hard in the strong sense. Due to the

difficulty of solving special cases, most of the articles dedicated to general variants focus on heuristic approaches, see for example Tang and Liu [46], Naderi et al. [41], Behnamian et al. [6], Elmi and Topaloglu [12], Zabihzadeh and Rezaeian [48].

The job-shop scheduling problem is a generalization of the flow-shop scheduling problem where all jobs do not necessarily follow the same pattern. Specifically, different jobs may consist of a different number of operations, and different jobs may have different processing orders through the machines. Hurink and Knust [25] first introduce the single-machine scheduling problem that consists of optimizing the central server schedule in the case where orders for the job-shop machines are already fixed. They demonstrate that the decision version of this subproblem is already NP-hard as it is a generalization of the asymmetric traveling salesman problem with time windows. The authors model the problem employing disjunctive graphs, develop a tabu search algorithm using neighborhood structures based on the block approach, and propose efficient ways to evaluate the proposed neighborhoods. The heuristic solutions are compared to a known lower bound, yielding an average gap of 0.6% on instances with 30 operations. For instances with 90 operations, the average gap increases to 6.4%, which highlights the difficulty of solving that subproblem alone. Hurink and Knust [26] address the job-shop scheduling problem with a central server and propose two tabu search algorithms. The first approach is integrated and performs local moves on the machine and the central server simultaneously. The second approach is sequential, with an outer stage focusing on local moves for the machines and an inner stage focusing on local moves for the central server. Both solution methods are compared to a known lower bound. The integrated approach appears to perform better, and yield average gaps of more than 20% on the largest instances, which include 10 machines and 10 jobs. The job-shop scheduling problem with central servers finds applications in the context of flexible manufacturing systems where automated guided vehicles transport the jobs between machines [36]. As for flow-shop scheduling problems with central servers, most approaches proposed in the literature are heuristics, and they often rely on disjunctive graph representation, see for example Lacomme et al. [35], Zhang et al. [50], Rossi [43], Afsar et al. [4], Nouri et al. [42].

While flow-shop scheduling and job-shop scheduling problems with a central server share similarities with our problem - as they aim to schedule multi-staged jobs in a production environment with parallel identical machines and a central server with sequence-dependent setup times - they also are fundamentally different. Specifically, these problems do not integrate job-machine assignment decisions as they focus on jobs with operations that need to be processed by different machines, thus yielding different constraints.

3. Problem definition and mathematical formulations

In this section, we first describe the problem under study. We then provide an example instance and illustrate the impact of the scheduling environment on the production schedule. We finally present two mathematical models of the problem.

3.1. Problem description

This problem aims to schedule a set of jobs K in an environment that includes a set of M machines and a central server. Each job $k \in K$ is composed of multiple tasks that are linked by precedence relationships. Thus, the completion of a given job requires that its tasks are executed in an order that satisfies the precedence relations. In the industrial application that motivates this problem, a job corresponds to a recipe to be prepared, and the associated tasks are the different steps to achieve to complete that recipe. The set of tasks is referred to as $T = \{1, ..., n\} \cup \{0, n+1\}$, where each task $i \in T$ has a processing time of p_i units of time, and 0 and $n+1$ are dummy tasks with null processing times. Note that the set of tasks of a job $k \in K$ is referred to as T_k. Therefore, each job $k \in K$ has a minimum duration d_k obtained by summing the processing time of all tasks in T_k. Tasks are nonpreemtive, and they can be partitioned into three subsets: type-1 tasks, $T^1 = \{t_1^1, ..., t_{n_1}^1\}$, type-2 tasks, $T^2 = \{t_1^2, ..., t_{n_2}^2\}$, and dummy tasks, $T^* = \{0, n+1\}$. Type-1 tasks model processing operations that must be peformed on a machine $m \in \{1, ..., M\}$ during p_i units of time. On the other hand, type-2 tasks model loading operations that require to be processed simultaneously by the server and a machine $m \in \{1, ..., M\}$ during p_i units of time.

The precedence graph of each job $k \in K$ can be seen as a "forward flow" network alternating between (i) a set of type-2 tasks and (ii) a single type-1 task. In the context of automated kitchens, this translates into the fact that each recipe alternates between two phases that consist of (i) dispensing one or multiple ingredients in the multifunctional food processor and (ii) processing the ingredients contained in the multifunctional food processor. Therefore, the precedence graph of each job $k \in K$ consists of stages $L_k = \{\lambda_k^1, ..., \lambda_k^{|L_k|}\}$. Each job $k \in K$ has an even strictly positive number of stages, i.e. $\forall k \in K, |L_k| = 2x, x \in Z^*$. Precedence graph multi-staged structures imply that, for each job $k \in K$ and each task i belonging to λ_k^n, the nth stage of k's precedence graph, the direct predecessors of i, namely $P(i)$, are the tasks in λ_k^{n-1}, the $n-1^{th}$ stage of k's precedence graph. Recalling that even stages are composed of a group of type-2 tasks and odd stages are composed of a single type-1 task, for each job $k \in K$ the first stage λ_k^1 is composed of a group of type-2 tasks, the penultimate stage $\lambda_k^{|L_k-1|}$ is composed of a group of type-2 tasks, and the last stage $\lambda_k^{|L_k|}$ is composed of a single type-1 task, also referred to as $last_k$. In the next section, we provide an example instance with two jobs and we detail the different stages of each job.

The scheduling environment is composed of M machines and a central server. The machines are identical, in the sense that they have the same processing speed and that each machine can process each task $i \in T$. Each job $k \in K$ is assigned to a machine $m \in \{1, ..., M\}$, in the sense that if one of its first tasks $i \in \lambda_k^1$ is executed by machine m, then all tasks $i \in T_k$ must be executed by machine m. Consequently, and to ensure that jobs are successfully carried out, a machine m that has started a job k cannot process the tasks of another job k' until all tasks of k are done. In the context of automated kitchens, this translates into the fact that a multifunctional food processor must complete all steps of a recipe before starting a new one. A machine is referred to as *occupied* if a job is being processed, and *free* otherwise.

The central server can consecutively process a pair of type-2 tasks belonging to different jobs, as long as the precedence relationships are respected. The consecutive execution of two loading operations $i \in T^2$ and $j \in T^2$ incurs a sequence-dependent setup time s_{ij}, which correspond to the transfer time required for the central server between tasks i and j. In the context of automated kitchens, s_{ij} captures the time taken by the central server to put back task i's ingredient to its place, get task j's ingredient, and transfer it to a machine. Setup times s_{0i} correspond to the transfer time required for the central server between the scheduling environment initial state and the start of task i. Setup times $s_{i,n+1}$ are null, and correspond to the transfer time required for the central server between the end of task i and the scheduling environment final state. We assume setup times are symmetric and respect the triangle inequality. The main notations are listed in Table 1.

Table 1: Notations

Parameter	Description
$T = \{1, ..., n\} \cup \{0, n+1\}$	Set of tasks indexed by $i, j \in T$
$T^1 = \{t_1^1, ..., t_{n_1}^1\}$	Set of type-1 tasks
n_1	Number of type-1 tasks
$T^2 = \{t_1^2, ..., t_{n_2}^2\}$	Set of type-2 tasks
n_2	Number of type-2 tasks
$T^* = \{0, n+1\}$	Dummy tasks
p_i	Processing time of task i
$P(i)$	Direct predecessors of task i
s_{ij}	Setup time between type-2 tasks i and j
K	Set of all jobs indexed by $k, l \in K$
d_k	Minimum duration of job k
T_k	Set of tasks of job k
$L_k = \{\lambda_k^1, ..., \lambda_k^{\lvert L_k \rvert}\}$	Set of stages of job k
λ_k^1	First stage of job k (type-2 tasks)
$\lambda_k^{\lvert L_k - 1 \rvert}$	Penultimate stage of job k, i.e. last type-2 tasks of job k
$last_k$	Last task of job k (type-1 task)
M	Number of machines indexed by $m \in \{1, ..., M\}$
R	Sufficiently large number

3.2. Example instance

Two jobs must be executed on two parallel machines. The first job includes tasks 1, 2, 3, 4 and 5. The second job includes tasks 6, 7 and 8. Tasks 0 and 9 refer to the dummy tasks with null processing times. Table 2 indicates the job, processing time, type, and direct predecessors associated with each task. Table 3 indicates the setup times s_{ij} that occur for the central server as it processes consecutively task j after task i. Figure 4 illustrates the precedence graph associated with each job, including the dummy tasks. Job 1 has four stages $L_1 = \{\lambda_1^1, \lambda_1^2, \lambda_1^3, \lambda_1^4\} = \{\{1\}; \{2\}; \{3, 4\}; \{5\}\}$, and $last_1 = 5$. Job 2 has two stages $L_2 = \{\lambda_2^1, \lambda_2^2\} = \{\{6, 7\}; \{8\}\}$, and $last_2 = 8$.

Figure 2 displays the production schedules associated with four different scheduling environments. Specifically,

Task	Job	Processing time	Type	Direct pred.
0	-	0	-	-
1	1	3	2	0
2	1	7	1	1
3	1	4	2	2
4	1	2	2	2
5	1	8	1	3,4
6	2	5	2	0
7	2	2	2	0
8	2	10	1	6,7
9	-	0	-	5,8

Table 2: Description of the tasks

Task	0	1	3	4	6	7	9
0	-	1	5	5	3	1	0
1	1	-	2	5	3	2	0
3	5	2	-	5	3	3	0
4	5	5	5	-	3	1	0
6	3	3	3	3	-	5	0
7	1	2	3	1	5	-	0
9	0	0	0	0	0	0	-

Table 3: Central server setup times

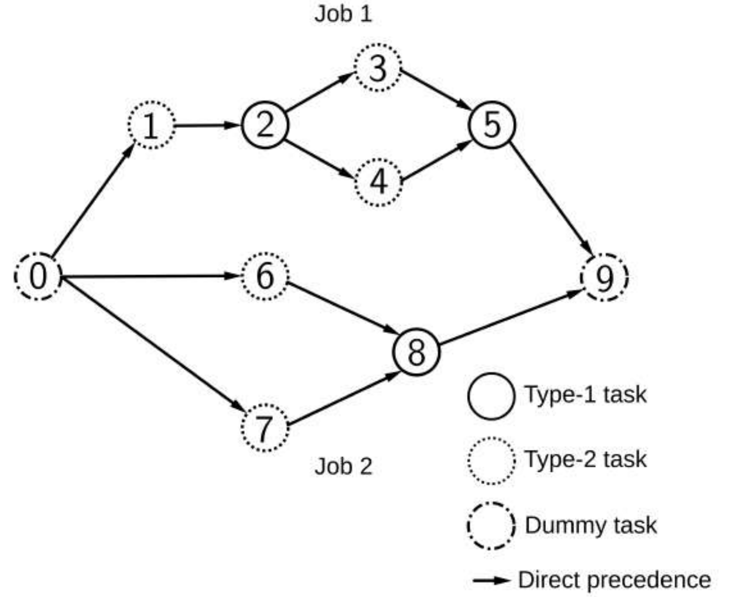
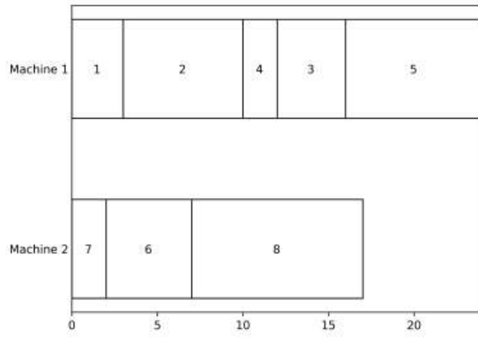


Table 4: Precedence graph

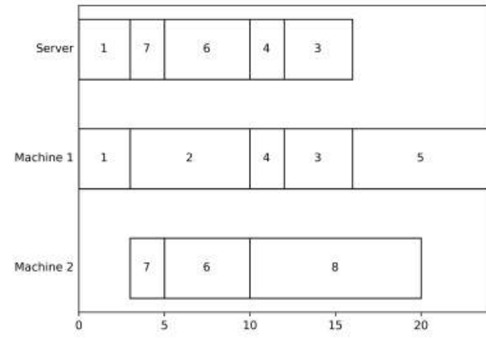
it illustrates how the various components of our problem affect the solutions. Figure 2a displays a feasible schedule in the context of a classical parallel production environment, where the requirement of using a central server to process type-2 tasks is eliminated, and the setup times are therefore not considered. If one considers a parallel production environment with no common resource nor setup times, then for each job $k \in K$ started on a machine, all tasks $i \in T_k$ can be performed consecutively without idle times, as long as the precedence relations are respected. Consequently, for each job $k \in K$, one can aggregate its tasks $i \in T_k$ into a single one, which is equivalent to the basic parallel machine scheduling problem. Figure 2b displays a modification of the previous schedule to accommodate for a scheduling environment that includes a central server as well as the need to use it for processing type-2 tasks, but no setup times. Since two type-2 tasks cannot occur simultaneously, several tasks are delayed. However, the makespan remains unchanged with a value of 24. Taking into account the sequence-dependent setup times increases the makespan to 40, as illustrated in Figure 2c. The optimal schedule is displayed in Figure 2d, it has a makespan of 35. From this example, we can see that the problem components, namely, jobs with multiple precedence-constrained tasks, the use of a shared server, and sequence-dependent setup times, make production scheduling highly complicated.

3.3. Mathematical models

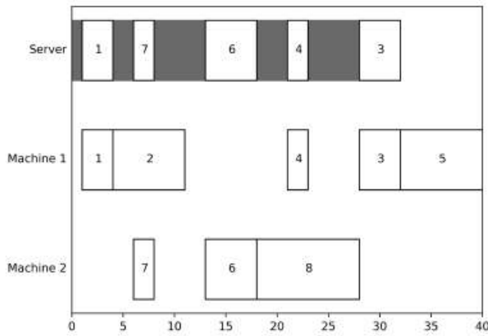
We first adjust the well-known disjunctive graph model [44] to represent the sequence of type-2 tasks to be executed by the central server. We use the proposed disjunctive graph to formulate two MILPs for the considered problem. The first formulation requires a machine index, whereas the second formulation does not. Lower bounds for both formulations are described in Appendix A.



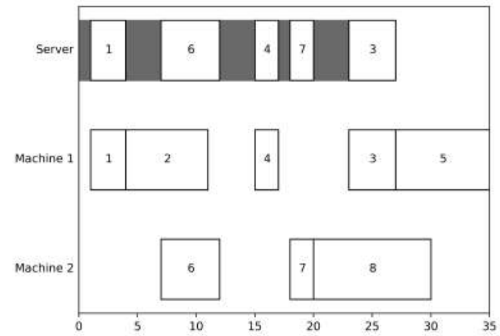
(a) Basic schedule



(b) Schedule with a central server



(c) Schedule with a central server and setup times



(d) Optimal schedule

Figure 2: Production schedules

3.3.1. Disjunctive graph representation

Disjunctive graphs [44] are widely used in the literature for describing instances of the job shop scheduling problem [27], as well as various other resource-constrained scheduling problems [10, 13, 38]. A disjunctive graph G is composed of vertices, N, which represent the tasks to be scheduled, as well as arcs, $A = C \cup D$, which are partitioned into conjunctions and disjunctions. A conjunction $i \rightarrow j \in C$ indicates that task j must be executed before task i. It is represented in the disjunctive graph by a directed arc (i, j). On the other hand, a disjunction $i \leftrightarrow j \in D$ indicates that tasks i and j can be performed in either order, but not simultaneously. It is represented by a bidirectional arc, or two directed arcs: (i, j) and (j, i). We present a disjunctive graph $G = (N, A = C \cup D)$ that models schedules of the type-2 tasks.

The set N contains a vertex for each type-2 task $i \in T^2$, as well as the dummy vertices. These dummy nodes, 0 and $n + 1$, model the beginning and the end of the central server operations, respectively. Conjunctions can be inferred between type-2 tasks based on the precedence graph of each job k. Recalling that job precedence graphs are "forward flow" networks alternating between (i) a group of type-2 tasks and (ii) one type-1 task, by

definition, each type-2 task has exactly one direct predecessor, and exactly one direct successor. Let i and j be two tasks of job k. If the direct successor of i is also the direct predecessor of task j (e.g. tasks 1 and 3 in Figure 4), then the central server must execute i before j, such that we add the conjunction $i \rightarrow j$ to C. Also, there are conjunctions between dummy node 0 and all first type-2 tasks (e.g. tasks 1, 6, 7 and 8 in Figure 4), as well as from all last type-2 tasks (e.g. tasks 3, 4, 6, 7 and 8 in Figure 4) to dummy node $n+1$. Specifically, for each job k, $0 \rightarrow i$ and $j \rightarrow n+1$ are added to C for all $i \in \lambda_k^1$ and all $\lambda_k^{|L_k - 1|}$, respectively. Regarding disjunctions, two tasks i and j belonging to two different jobs (e.g. tasks 1 and 6 in Figure 4) can be performed in either order, but not simultaneously by the central server. In that case, we add the disjunction $i \leftrightarrow j$ to D. Similarly, two tasks i and j that are part of the same job k may be performed in either order by the central server. In particular, if both i and j are in the same stage of k's precedence graph (e.g. tasks 3 and 4 in Figure 4), there are no precedence relations between i and j, such that we add the disjunction $i \leftrightarrow j$ to D. We illustrate the disjunctive graph $G = (N, A = C \cup D)$ associated to the example problem of Section 3.2 in Figure 3.

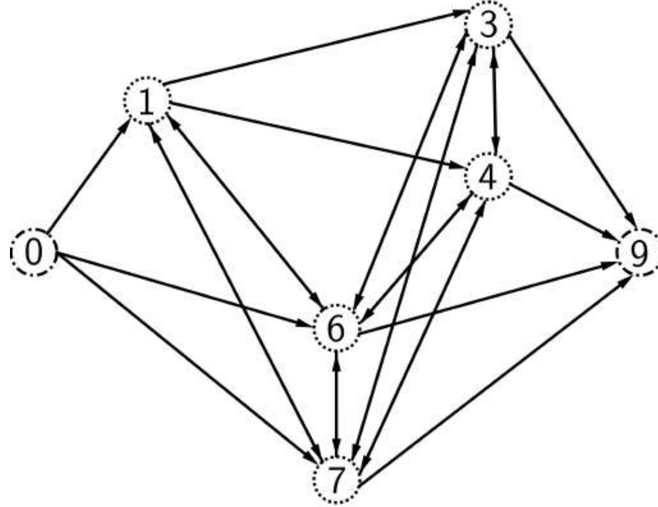


Figure 3: Central server's disjunctive graph

Unidirectional arrows and bidirectional arrows represent conjunctions and disjunctions, respectively. In this disjunctive graph, each path/permutation $\pi = (0, \pi_1, ..., \pi_{n_2}, n+1)$ from 0 to n+1 that visits all the nodes in N represents a sequence of type-2 tasks to be executed by the central server. As the proposed disjunctive graph does not consider job scheduling decisions and the precedence relations they induce, some permutations may not be feasible. Back to the example problem of Section 3, if one schedules job 2 after job 1 on machine 1, then tasks 3 and 4 become predecessors of tasks 6 and 7, such that the permutation $permutation = (0, 1, 6, 7, 3, 4, 9)$ violates precedence constraints induced by the job scheduling decisions. Nevertheless, regardless of the job scheduling decisions made, for each feasible sequence of type-2 tasks, there exists an associated permutation in G. We thus introduce two valid MILPs formulated over G.

3.3.2. Formulation with a machine index

We let the continuous decision variable c indicate the schedule makespan. Continuous decision variables c_k are defined for each job $k \in K$ and indicate job completion times. Continuous decision variables s_i are defined for each task $i \in T$ and indicate task starting times. Binary variables x_{ij} are defined for each arc $(i,j) \in A$. They indicate whether the type-2 tasks i and j are executed consecutively by the central server. Binary variables y^m_{0k} and $y^m_{k|K|+1}$ are defined for each machine $m \in \{1,...,M\}$ and each job $k \in K$. They indicate whether job k is the first job or the last job on machine m, respectively. Binary variables y^m_{kl} are defined for each machine $m \in \{1,...,M\}$ and each pair of jobs $k,l \in K$. They indicate whether job l is scheduled directly after job k on machine m. Finally, binary variables z^m_k appear for each machine $m \in \{1,...,M\}$ and each job $k \in K$, and indicate if job k is assigned to machine m. The first formulation reads as follows:

$$\min \qquad c \tag{1}$$

$$c \geq c_k, \quad \forall k \in K \tag{2}$$

$$s_j \geq s_i + p_i, \quad \forall j \in T, \forall i \in P(j) \tag{3}$$

$$c_k \geq s_{last_k} + p_{last_k}, \quad \forall k \in K \tag{4}$$

$$\sum_{m=1}^{M} z^m_k = 1, \quad \forall k \in K \tag{5}$$

$$y^m_{0k} + \sum_{\substack{l=1 \\ l \neq k}}^{|K|} y^m_{lk} = z^m_k, \quad \forall m \in \{1,...,M\}, \forall k \in K \tag{6}$$

$$\sum_{\substack{l=1 \\ l \neq k}}^{|K|} y^m_{kl} + y^m_{k|K|+1} = z^m_k, \quad \forall m \in \{1,...,M\}, \forall k \in K \tag{7}$$

$$\sum_{k=1}^{|K|} y^m_{0k} \leq 1, \quad \forall m \in \{1,...,M\} \tag{8}$$

$$s_j \geq c_k - R\left(1 - \sum_{m \in \{1,...,M\}} y^m_{kl}\right), \quad \forall (k,l) \in K^2, j \neq k, \forall j \in \lambda^1_k \tag{9}$$

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i')\in A} x_{ji'} = \begin{cases} 1 & \text{if } j = 0 \\ -1 & \text{if } j = n+1 \\ 0 & \text{otherwise} \end{cases}, \quad \forall j \in N \tag{10}$$

$$\sum_{(i,j)\in A} x_{ij} = 1, \quad \forall j \in N \tag{11}$$

$$s_j \geq s_i + p_i + s_{ij} - R\left(1 - x_{ij}\right), \quad (i,j) \in D \tag{12}$$

$$c \geq 0,\ c_k \geq 0\ \forall k \in K,\ s_i \geq 0\ \forall i \in T \tag{13}$$

$$x_{ij} \in \{0,1\}\ \forall (i,j) \in A,\ y^m_{0k} \in \{0,1\}\ \forall k \in K\ \forall m \in \{1,...,M\},\ y^m_{kl} \in \{0,1\}\ \forall k \in K\ \forall l \in K\ \forall m \in \{1,...,M\} \tag{14}$$

$$y^m_{k|K|+1} \in \{0,1\}\ \forall k \in K\ \forall m \in \{1,...,M\}s,\ z^m_k \in \{0,1\}\ \forall k \in K\ \forall m \in \{1,...,M\} \tag{15}$$

The objective function (1) aims to minimize the makespan. Constraints (2) guarantee that the makespan is greater than the completion time of each job, and thus reflects the duration of the schedule. Constraints (3) ensure that the direct precedences of all tasks are respected. Constraints (4) guarantee that the completion time of each job must be equal to or greater than the latest completion time of its last tasks. Constraint (5) ensures that each job is assigned to exactly one machine. Constraints (6) ensure that, if a job is assigned to a machine, it is either the first job on that machine or preceded by another job. Constraints (7) ensure that, if a job is assigned to a machine, it is either the last job on that machine or succeded by another job. Constraint (8) limits the number of first jobs on the machines to the number of machines. Constraints (9) impose that task starting times take account of the job scheduling decisions. Specifically, it states that if a job l is preceded by another job k on the same machine, then the first tasks of jobs l cannot start before the completion time of job k, i.e. the latest completion time of the last tasks of k.

Constraints (10), (11) and (12) ensure that the x-variables provide a permutation of the type-2 tasks, and that sequence-dependent setup time are taken into account. Constraints (10) are standard flow conservation constraints. Constraints (11) guarantee that the flow arriving to each node, apart from dummy node 0, is equal to 1. Constraints (12) ensure that setup times are respected between each pair of type-2 tasks executed consecutively by the central server. Precisely, it ensures that the starting time of task j must be greater than or equal to the completion time of i plus the setup time s_{ij} if j is executed directly after i. The combination of these constraints with constraints (3), (4) and (9) ensure that the x-variables provide a precedence-feasible permutation of the type-2 tasks, i.e. a permutation that satisfies job inner precedence constraints as well as precedence constraints induced by the job scheduling decisions.

Constraints (13) define c, c_k, and s_i as positive continuous variables. Constraints (14) and (15) define $x_{ij}, y^m_{0k}, y^m_{kl}, y^m_{k|K|+1}$, and z^m_k as binary variables.

3.3.3. Formulation without a machine index

The previous MILP has the disadvantage that its number of variables increases with the number of machines. Alternatively, one can formulate job scheduling constraints with y-variables that do not comprise a machine index and omit the z-variables, as proposed by Biskup et al. [8] for the basic parallel machine scheduling problem. We propose an alternative MILP without a machine index by extending the formulation of Biskup et al. [8]. This

formulation includes the same continuous decision variables and x-variables as the formulation with an index. Binary variables y_{0k} and $y_{k|K|+1}$ are defined for each job $k \in K$, and they indicate whether job k is the first job or the last job on any machine, respectively. Binary variables y_{kl} are defined for each pair of jobs $k, l \in K$, they indicate whether job l is scheduled directly after job k on any machine. The second formulation reads as follows:

(1)-(4)

$$\sum_{k=1}^{|K|} y_{0k} \leq M \tag{16}$$

$$y_{0k} + \sum_{\substack{l=1 \\ l \neq k}}^{|K|} y_{lk} = 1, \quad \forall k \in K \tag{17}$$

$$\sum_{\substack{l=1 \\ l \neq k}}^{|K|} y_{kl} + y_{k|K|+1} = 1, \quad \forall k \in K \tag{18}$$

$$y_{kl} + y_{lk} \leq 1, \quad \forall k \in K, \forall l \in K, k \neq l \tag{19}$$

$$\sum_{k=1}^{|K|} y_{k|K|+1} \leq M \tag{20}$$

$$y_{0k} + \sum_{\substack{l=1 \\ l \neq k}}^{|K|} y_{kl} + y_{k|K|+1} \leq 2, \quad \forall k \in K \tag{21}$$

(9)-(12)

$$c \geq 0, \;\; c_k \geq 0 \; \forall k \in K, \;\; s_i \geq 0 \; \forall i \in T \tag{22}$$

$$x_{ij} \in \{0,1\} \; \forall (i,j) \in A, \;\; y_{0k} \in \{0,1\} \; \forall k \in K, \;\; y_{kl} \in \{0,1\} \; \forall k \in K \; \forall l \in K, \;\; y_{k|K|+1} \in \{0,1\} \; \forall k \in K \tag{23}$$

The objective function (1), as well as constraints (2), (3), (4), (9), (10), (11) and (12) remain unchanged. Constraints (16) ensure that the number of first jobs on the machines is lower than or equal to the number of machines. Constraints (17) ensure that each job either starts on a machine or is preceded by another job. Constraints (18) ensure that each job is either last on a machine or is followed by another job. Constraints (19), (20), (21) are valid inequalities proposed by Biskup et al. [8] to strenghten the formulation. Constraints 19 ensure that two jobs cannot precede each other at the same time. Constraints 20 ensure that each machine only has one last job. Constraints 21 ensure that the first job on a machine may only have one successor. Similarly to the previous formulation, constraints (13) define c, c_k, and s_i as positive continuous variables. Constraints (23) define x_{ij}, y_{0k}, y_{kl}, and $y_{k|K|+1}$ as binary variables.

4. Methodology

We propose a general variable neighborhood search (GVNS) approach based on a hierarchical decomposition of the problem. This decomposition consists of (i) scheduling jobs on the machines and (ii) scheduling the central server while satisfying the precedence relations. In particular, we show that the central server scheduling subproblem resulting from job scheduling decisions is equivalent to the single-machine scheduling problem studied by Hurink and Knust [25]. We first describe the problem decomposition and explain how the solutions are represented and evaluated. We briefly describe the GVNS algorithm, a variant of variable neighborhood search (VNS) that uses variable neighborhood descent (VND) as a local search operator. Finally, we describe the improvement heuristics included in the proposed VND.

4.1. Hierarchical decomposition: solution representation and evaluation

We represent each solution with two components based on the proposed problem decomposition. The first solution component, $\phi = (\phi_1, ..., \phi_M)$, defines the job scheduling decisions. Each vector ϕ_m, with $m \in \{1, ..., M\}$, indicates which jobs are assigned to machine m, and in which order. These job scheduling decisions induce additional precedence relations between the type-2 tasks that must be taken into account when defining the second solution component, π, which provides a feasible permutation of type-2 tasks. We formally explain how the job scheduling decisions impact the precedence relations of the disjunctive graph, and we illustrate it with an example. We then show that the central server scheduling subproblem resulting from job scheduling decisions is equivalent to the strongly NP-hard single-machine scheduling problem studied by Hurink and Knust [25]. For this reason, we use the same decoding strategy to construct a minimal makespan schedule from a solution (ϕ, π). Finally, we summarize the decoding strategy and show that our solution representation corresponds to the solution space of the original problem.

As previously stated, job scheduling decisions have an impact on the arcs of the disjunctive graph as they induce precedence between type-2 tasks of different jobs. Specifically, if a job l is executed directly after a job k on machine m, then the last task of job k must be completed before starting the first task of job l. In addition, if two jobs k and l are assigned to the same machine m, then either k or l precedes the other, so that there can be no disjunctions between the type-2 tasks of k and those of l. Consequently, given ϕ, the graph $G^\phi = (N, A = C^\phi \cup D^\phi)$ is constructed as follows. We infer conjunctions between type-2 tasks that are part of the same job k as done in Section 3.3.1. In addition, for two jobs k and l executed consecutively on the same machine, for each task $i \in \lambda_k^{|L_k-1|}$, and for each task $j \in \lambda_l^1$, we add a conjunction $i \to j$ to C^ϕ. As before, for two type-2 tasks i and j in the same stage of job k's precedence graph, we add the disjunction $i \leftrightarrow j$ to D^ϕ. Then, we add a disjunction $i \leftrightarrow j$ to D^ϕ for each pair of type-2 tasks i and j belonging to distinct jobs k and l that are not assigned to the same machine. We consider the example problem of Section 3.2, and a single machine. Let $\phi = (\phi_1) = (1, 2)$ be the job scheduling decisions. We illustrate the resulting disjunctive graph $G^\phi = (N, A^\phi = C^\phi \cup D^\phi)$ in Figure 4.

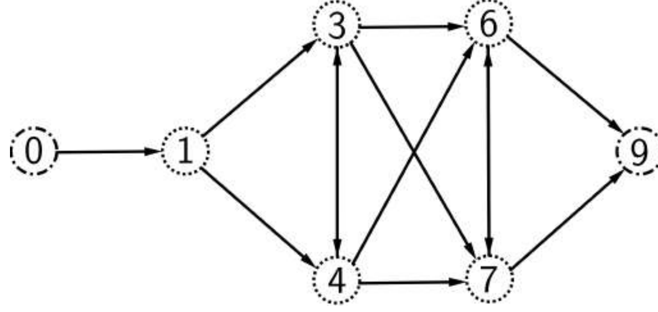


Figure 4: Central server's disjunctive graph

Hurink and Knust [26] study the job-shop problem with a single transport robot. They consider $|K|$ jobs that must be performed in a scheduling environment composed of M machines and a central server. Each job $k \in K$ is composed of n_k tasks, t_i^k, that must be performed in the order $t_1^k \rightarrow t_2^k \rightarrow ... \rightarrow t_{n_k}^k$. Each task t_i^k needs to be processed by a dedicated machine so that each job needs to change from one machine to another between two consecutive tasks. This transportation operation is performed by a central server and incurs a transport time. Therefore, each job $k \in K$ comprises $n_k - 1$ transport tasks, tt_i^k, that must be executed between regular tasks, yielding the following precedence relations: $t_1^k \rightarrow tt_1^k \rightarrow t_2^k \rightarrow ... \rightarrow tt_{n_k-1}^k \rightarrow t_{n_k}^k$. Finally, sequence-dependent setup times are considered for the central server and used to model empty moving times from one machine to another. While the problem considered in this paper and that studied by Hurink and Knust [26] share many similarities, they are also significantly different. In particular, the job shop with a single robot does not involve decisions regarding the allocation of tasks to machines. Therefore, one knows in advance which tasks must be performed by which machine. The problem is to determine in which order the different tasks should be executed - for the machines and the central server - which can be modeled through a single disjunctive graph. On the other hand, in our problem job scheduling decisions impact the precedence relations, such that the disjunctive graph needs to be modified accordingly. Hurink and Knust [26] propose an integrated solution approach, as well as a two-level approach, where the first level sets orders for the machine tasks and the second level sequences the central server tasks. The latter problem is formalized in [25], and is equivalent to our central server scheduling subproblem.

The problem studied in [25] aims to order a set of precedence-related tasks on a central server while addressing time-lags and sequence-dependent setup times. The problem is denoted as $1|prec(l_{ij}), r_j, s_{ij}|max\{c_j + q_j\}$ in the three-field notation, with $prec(l_{ij})$ standing for positive time-lags between tasks i and j. Such time-lags exist for all conjunctions and model the need to wait for l_{ij} units of time between the end of i and the beginning of j. In our problem, conjunctions between type-2 tasks i and j exist if the direct successor of i, e.g. type-1 task i', is also the direct predecessor of j. Thus, we set l_{ij} to $p_{i'}$, which models the fact that task i' must be executed between the end of i and the beginning of j. Sequence-dependent setup times between tasks i and j are reflected by s_{ij},

and are set to the values of the central server setup times. The release time of task j is denoted by r_j , which can be set to s_{0j} in our case, i.e. the transfer time required for the central server between the scheduling environment initial state and the start of task i. Finally, q_j stands for the duration to be observed after the completion of task j, which can be set to $p_{j'}$ in our case, where type-1 task j' is the direct successor of j. For each task j, $c_j + q_j$ reflects the completion time of the subsequent type-1 task j', such that the objective function, i.e. minimizing the maximum value of $c_j + q_j$, reflects the schedule makespan. As stated by Hurink and Knust [25], this scheduling problem is strongly NP-hard as it generalizes several NP-hard problems.

Given job scheduling decisions ϕ and an acyclic orientation of the disjunctions of the graph G^{ϕ}, one can determine the makespan minimizing schedule associated by identifying a critical path in the oriented graph. As demonstrated in Hurink and Knust [25], defining an acyclic orientation of the disjunctions is equivalent to determining a permutation of the type-2 tasks $\pi = (0, \pi_1, ..., \pi_{n_2}, n{+}1)$ that is precedence-feasible. The associated makespan minimizing schedule $S^{(\phi,\pi)}$ can be obtained by applying the decoding strategy of Hurink and Knust [25]. Let $S_0^{(\phi,\pi)}$ be the starting time of the first dummy task, i.e. the beginning of the schedule. Let $S_{n+1}^{(\phi,\pi)}$ be the starting time of the second dummy task, i.e. the end of the schedule. Let i be the ith type-2 task, let u be the position of task i in permutation π be the starting time of task i. The starting time of the first dummy task, $S_0^{(\phi,\pi)}$ is set to 0. We then calculate for $u = 1, ..., n_2$ the starting time of task $j := \pi_u$ as:

$$S_j^{(\phi,\pi)} = max\left[\max_{\{i|i\to j\in C^{\phi}\}}\{S_i^{(\phi,\pi)} + p_i + p_{i'}\}, S_{\pi_{u-1}}^{(\phi,\pi)} + p_{\pi_{u-1}} + s_{\pi_{u-1}j}\right].$$

In the first term, i' denotes the type-1 task that is both the direct predecessor of j and the direct successor of i. The first term ensures that all the time-lags are respected, while the second term ensures that setup times between consecutive type-2 tasks are respected. The starting time of dummy task $n + 1$ is set to :

$$S_{n+1}^{(\phi,\pi)} = max\left[\max_{\{i|i\to n+1\in C^{\phi}\}}\{S_i^{(\phi,\pi)} + p_i + p_{i'}\}, S_{\pi_{n_2}}^{(\phi,\pi)} + p_{\pi_{n_2}} + s_{\pi_{n_2}n+1}\right]$$

and yields the schedule makespan, i.e. $c(S^{(\phi,\pi)}) = S_{n+1}^{(\phi,\pi)}$, which is obtained through a calculation with complexity of $O(n + |C|) = O(n^2)$. Hurink and Knust [25] show that the solution space of the central server scheduling subproblem may be represented by the set of all precedence-feasible permutations of the type-2 tasks. On the other hand, it is obvious that the solution space of the job scheduling decisions can be represented with ϕ. Therefore, the solution space of the considered problem may be represented by the set of solutions (ϕ, π).

4.2. A general variable neighborhood search

Variable Neighborhood Search (VNS) is a metaheuristic framework proposed by Mladenović and Hansen [40] whose basic idea is to systematically change the neighborhood structure during the search. It has been successfully applied to a large number of difficult combinatorial optimization problems, including scheduling problems such as the job-shop problem [47], the flow-shop problem [37], or the open-shop problem [39]. The basic VNS is an iterative algorithm, where each iteration consists of (i) generating a random neighbor x' of the incumbent x,

which is also referred to as the *Shake* step, (ii) determining a new solution x'' by applying a local search heuristic to x', and (iii) comparing solutions x and x'' to identify the next neighborhood structure, which is also referred to as the *NeighborhoodChange* step. Specifically, if the new solution x'' outperforms the incumbent, it replaces it and the algorithm continues with the first neighborhood structure. Otherwise, the incumbent remains unchanged and the algorithm moves to the next neighborhood structure. Many extensions of VNS have been proposed in the literature. One of these extensions is the general VNS (GVNS), where the local search step is replaced by a variable neighborhood descent (VND) procedure. VND sequentially applies local search heuristics to different neighborhood structures until a local optimum to all the neighborhood structures is found. The pseudo-code of the GVNS approach as well as the *NeighborhoodChange* function are given in Algorithms 1 and 2.

Algorithm 1 General Variable Neighborhood Search

```
Data: Initial solution, x, number of shaking neighborhood structures, k_max, number of improvement heuristics, l_max, time
      limit, t_max, maximum time without improvement, t_max^impr
t_last^impr ← CpuTime();
repeat
  | k ← 1;
  | repeat
  |   | x' ← Shake(x, k);
  |   | x'' ← VND(x', l_max, t_last^impr);
  |   | k, x ← NeighborhoodChange(x, x'', k, t_last^impr);
  | until k > k_max
  | t ← CpuTime();
until t > t_max OR t − t_last^impr > t_max^impr
Result: Incumbent solution x
```

Algorithm 2 Neighborhood Change

```
Data: Incumbent solution, x, new solution, x', current neighborhood structure index, k, last improvement time, t_last^impr
if f(x') < f(x) then
  | x ← x';
  | k ← 1;
  | t_last^impr ← CpuTime();
else
  | k ← k + 1;
```

As stated previously, the classical VND consists in sequentially applying local search heuristics to different neighborhood structures. Specifically, at each iteration of the classical VND, one enumerates the solutions in the neighborhood of the current solution to determine a direction of descent using a *first improvement* strategy, i.e. stop the local search and update the incumbent whenever an improving neighbor is found, or a *best improvement* strategy, i.e. evaluate all the neighbors and select the best one. In this study, we propose a VND where the

descent is performed through local search heuristics, as well as improvement heuristics. Rather than defining a neighborhood and enumerating all the solutions in that neighborhood, the latter heuristic procedures perform a succession of steps to the current solution to reduce idle times and potentially improve the incumbent. The steps of the VND are presented in Algorithm 3.

Algorithm 3 Variable Neighborhood Descent

Data: Incumbent solution, x, number of improvement heuristics, l_{max}, last improvement time, t_{last}^{impr}

$l \leftarrow 1$;

repeat

  $x' \leftarrow Heuristic_l(x)$;

  $l, x \leftarrow NeighborhoodChange(x, x', l, t_{last}^{impr})$;

until $l > l_{max}$

Result: Incumbent solution x

To initialize the GVNS, we construct an initial solution (ϕ,π) using a heuristic described in Appendix B. Then, the VND uses five heuristics which are described in Section 4.3. Among these heuristics, the first four are improvement heuristics introduced for this problem. The last one is the local search heuristic proposed in [25]. In Section 4.4, we present six operators to shake the solutions in the course of the GVNS.

4.3. VND heuristics

The VND heuristics modify the current solution (ϕ,π) to improve the makespan of schedule $S^{(\phi,\pi)}$, which can be obtained by applying the decoding strategy described in Section 4.1. The first two heuristics modify the job schedule decisions ϕ, and reconstruct the permutation π so that it respects the precedence relations induced by the new job scheduling decisions. The three remaining heuristics modify the permutation π. As stated previously, the proposed heuristic procedures aim to leverage idle times identified in the current schedule. For a given resource, an idle time is a time during which no action is being performed. We formally define three categories of idle times. *Central server idle times* correspond to time intervals during which the server is not processing a type-2 task nor in setup. *Machine idle times* correspond to time intervals during which no type-1 task is being processed and are decomposed into machine *occupied idle times* and machine *free idle times*. In the first case, no type-1 task is being processed by the machine but a job is in progress. In the second case, no type-1 task is being processed by the machine and the machine is free. Recall that, given job scheduling decisions ϕ and a precedence-feasible permutation π, the associated makespan minimizing schedule $S^{(\phi,\pi)}$ prescribes the starting times of the type-2 tasks only. While these starting times are sufficient to compute *central server idle times*, one must also define the starting times of the type-1 tasks to determine *machine idle times*. For each type-1 task $j \in T^1$, we compute its earliest starting time as follows:

$$S_j^{(\phi,\pi)} = \max_{\{i \in P(j)\}} \{S_i^{(\phi,\pi)} + p_i\}.$$

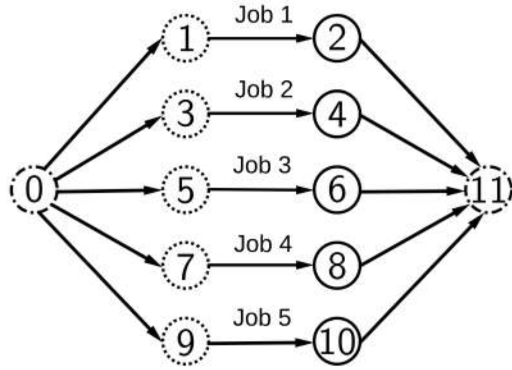


Figure 5: Precedence graph

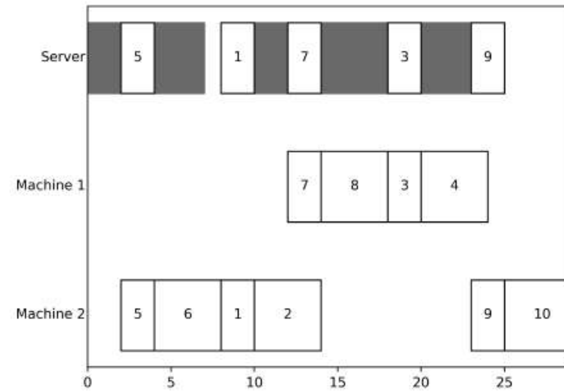


Figure 6: Obtained schedule

We illustrate idle times with an example. We consider five identical jobs to be executed on two parallel machines. Each job is composed of one type-2 task and one type-1 task that last 2 and 4 units of time, respectively. Table 5 indicates the job, processing time, type, and direct predecessors associated with each task. Table 6 indicates the setup times s_{ij} that occur for the central server as it processes consecutively task j after task i. Figure 5 illustrates the precedence graph associated with each job, including the dummy tasks. Let $\phi = (\phi_1, \phi_2) = ((4, 2), (3, 1, 5))$ be the job scheduling decisions. Let $\pi = (0, 5, 1, 7, 3, 9)$ be a precedence-feasible permutation. Figure 6 displays the obtained schedule $S^{(\phi,\pi)}$, which has a makespan of 29. *Central server idle times* are $[7, 8]$ and $[25, 29]$. *Idle times* for the first machine are $[0, 12]$ and $[24, 29]$. *Idle times* for the second machine are $[0, 2]$ and $[14, 23]$. All idle times associated with the machines are *free idle times*.

Task	Job	Processing time	Type	Direct pred.
0	-	0	-	-
1	1	2	2	0
2	1	4	1	1
3	2	2	2	0
4	2	4	1	3
5	3	2	2	0
6	3	4	1	5
7	4	2	2	0
8	4	4	1	7
9	5	2	2	0
10	5	4	1	9
11	-	0	-	2,4,6,8,10

Table 5: Description of the tasks

Task	0	1	3	5	7	9	11
0	-	2	3	2	4	1	0
1	2	-	2	3	2	3	0
3	3	2	-	4	4	3	0
5	2	3	4	-	2	1	0
7	4	2	4	2	-	1	0
9	1	3	3	1	1	-	0
11	0	0	0	0	0	0	-

Table 6: Central server setup times

4.3.1. Job assignment heuristic

This heuristic aims at improving the current solution makespan by adjusting the job scheduling decisions. To do so, the heuristic identifies the machine that finishes last, i.e. the makespan machine, and assigns its last job to one of the other machines. To reduce the negative impact that this reassignment can have on the makespan, the heuristic reschedules the job to the machine with the longest *free idle time*. Specifically, it adjusts job scheduling

decisions to reflect the decision of rescheduling the job during that longest *free idle time*, and it reconstructs the permutation accordingly, as well as the associated schedule. If the obtained solution does not improve the incumbent, the procedure stops. Otherwise, the obtained solution replaces the incumbent, and the procedure re-iterates. The heuristic pseudo-code is given in Appendix C.1.

Given a solution (ϕ,π), the first step is to identify the makespan machine m^*. Then, the heuristic computes idle times for the non-makespan machines and identifies the longest interval among the *free idle times*. Let $\dot{m}$ be the non-makespan machine associated with the longest *free idle time*, $[a, b]$, which occurs between the end of task i^* and the start of task j^*. Since these idle times occur between two jobs, one can link the longest *free idle time* with the job scheduling decisions of machine $\dot{m}$, and leverage this information to rearrange the jobs. In the case of the schedule illustrated in Figure 6, the heuristic identifies that the makespan machine is machine 2, the longest *free idle time* on the non-makespan machines is $[a, b] = [0, 12]$, which occurs on machine 1, between the end of task 0 and the start of task 7. Given that $\phi_1 = (4, 2)$, and that the first task of job 4 starts at time 12, the heuristic identifies that this longest *free idle time* occurs before job 4 starts.

Next, the heuristic constructs new job scheduling decisions obtained by rescheduling the last job of the makespan machine, k^*, to the identified longest *free idle time*. In the case of the scheduling illustrated in Table 5, this would result in new job scheduling decisions $\phi' = (\phi'_1, \phi'_2) = ((5, 4, 2), (3, 1))$, where job 5 is rescheduled before job 4 on machine 1. Because new job scheduling decisions ϕ' induce a disjunctive graph $G^{\phi'}$ with different conjunctions than those of G^{ϕ}, the permutation π is unlikely to satisfy the new precedence relations. Consequently, the heuristic rearranges the permutation π to reflect the decision of rescheduling job k during the identified *free idle time*. To do so, the heuristic constructs a partial permutation π' obtained by removing from π all the tasks associated with the rescheduled job k^*. These tasks are saved in a second vector π", and kept in the same order as in π. Specifically, if $\pi = (0, \pi_1, ..., \pi_{u_i}, ..., \pi_{u_j} ..., \pi_{n_2}, n + 1)$ such that π_{u_i} and π_{u_j} belong to job k, then there are integers u_k, u_l, such that $\pi"_{u_k} = \pi_{u_i}, \pi"_{u_l} = \pi_{u_j}$, and $u_k < u_l$. Then, as long as vector π" is not empty, the heuristic performs the following steps to complete the rearranged permutation π'. First, it constructs the partial schedule associated with π' according to the conjunctions of $G^{\phi'}$. Second, it updates i^*, the last task executed on machine m^* before task j^*. Third, it updates a as the completion time of i^*, and b as the starting time of j^*, so that $[a, b]$ reflect the identified *free idle time* in the partial schedule. Fourth, it removes the first task in π" and places it in the partial permutation π' before the first type-2 task whose starting time is greater than or equal to a. This procedure ensures that the job is rescheduled as early as possible while respecting the inner precedence constraints, as well as the precedence constraints relative to the job before/after the identified *free idle time*. Figure 7 displays the schedule obtained after applying the heuristic to the solution associated with the schedule illustrated in Figure 6.

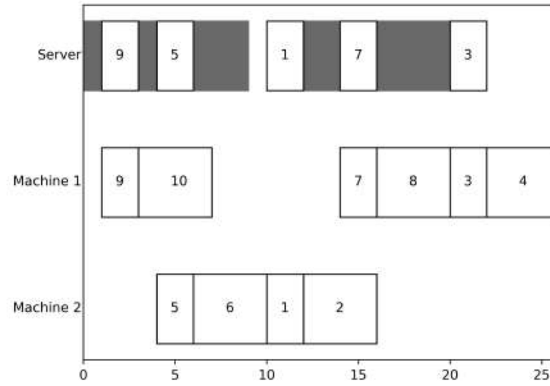


Figure 7: Improved schedule

4.3.2. Job ordering heuristic

This heuristic aims to improve the makespan by changing the order of two jobs consecutively scheduled on the same machine. Because doing so for each pair of consecutive jobs can be very time-consuming, we operate this move for promising pairs only. Assuming that a machine m observes an idle time $[a, b]$ between two tasks i and j, with i being a type-1 task and j being a type-2 task. As the arm cannot perform task j before time b, and the machine is available from time a, replacing task i with another type-1 task with a longer duration would lead to better utilization of machine m, and potentially a schedule of higher quality. Nevertheless, because job precedence graphs are "forward flow" networks alternating between (i) a group of type-2 tasks and (ii) a single type-1 task, one cannot swap task i with another type-1 task i' scheduled on machine m, as this would necessarily cause the violation of a precedence constraint. Therefore, the heuristic identifies the job associated with task i, and it either swaps with (i) the next job scheduled on machine m, or (ii) with the previous job on machine m. The heuristic pseudo-code is given in Appendix C.2.

Given a solution (ϕ,π), at each iteration the heuristic first identifies all machine idle times between a type-1 task and a type-2 task. Then, for each machine m, it iterates on the I^{max} largest idle times $[a, b]$ sorted in decreasing order. For each of these idle times, it identifies the type-1 task i^* that terminates at time a on machine m, as well as the associated job k^*. If there is another job k^{**} scheduled on machine m before job k^*, new job scheduling decisions are obtained by swapping k^* and k^{**}. These new job scheduling decisions yield a new disjunctive graph $G^{\phi'}$ with conjunctions that are not satisfied by the current permutation π. In particular, as k^* and k^{**} are jobs executed consecutively on the same machine, swapping them yields a violation of precedence constraints involving those jobs, i.e. tasks associated to k^* should be executed before tasks of k^{**} according to the new job scheduling decisions, while they are executed after those of k^{**} in the current solution. Therefore, the heuristic rearranges the permutation to satisfy the new precedence relations. The procedure

consists in swapping the tasks of k^* and k^{**} in the permutation, i.e. altogether tasks of k^* and k^{**} keep the same positions of the permutation, but in a different order. Assuming that job k^{**} is composed of $|T_{k^{**}}|$ tasks having indices $u = (u_1, ..., u_{|T_{k^{**}}|})$ in the current permutation π, and that job k^* is composed of $|T_{k^*}|$ tasks having indices $v = (v_1, ..., v_{|T_{k^*}|})$ in the current permutation π. As job k^{**} precedes job k^* in the current solution, which is precedence-feasible, we have that $u_i < v_j, \forall i \in \{1, ..., |T_{k^{**}}|\}, \forall j \in \{1, ..., |T_{k^*}|\}$. Let us consider vector $w = (w_1, ..., w_{|T_{k^{**}}|+|T_{k^*}|}) = (u_1, ..., u_{|T_{k^{**}}|}, v_1, ..., v_{|T_{k^*}|})$. In the rearranged permutation π', tasks of job k^* have indices $(w_1, ..., w_{|T_{k^{**}}|})$, tasks of job k^{**} have indices $(w_{|T_{k^{**}}|+1}, ..., w_{|T_{k^{**}}|+|T_{k^*}|})$, while all other tasks keep the same index. This ensures that the rearranged permutation π' is feasible w.r.t the new job scheduling decisions ϕ'. Then, the obtained solution (ϕ', π') updates the current solution (ϕ, π) if it yields a better makespan. Otherwise, if there is a job k^{**} scheduled on machine m after job k^*, a similar procedure is applied by swapping k^* and k^{**}. Whenever an improved solution is found, the procedure starts over.

4.3.3. Machine idle times heuristic

This heuristic aims at improving the current solution makespan by focusing on reducing the *machine idle times* found on the makespan machine, m^*. We first explain how to reduce an individual *machine idle time*. We then detail the steps of the proposed procedure. The heuristic pseudo-code is given in Appendix C.3.

Recall that a *machine idle time* corresponds to a time interval during which no type-1 task is being processed. Let us consider idle time $[a, b]$ on machine m^* which occurs between the end of task i^* and the start of task j^*. In the case of the schedule illustrated in Figure 6, the second machine finishes last, and it observes a *machine idle time* from time 14 to time 23, between tasks $i^* = 2$ and $j^* = 9$. By construction, each type-1 task starts directly after a type-2 task, such that if there is an idle time between i^* and j^*, j^* is necessarily a type-2 task. Let i' be the last type-2 task executed before the idle time starts. Back to the case of the schedule illustrated in Figure 6, $i' = 7$. In the permutation π, one can identify the indices of tasks i' and j^*, i.e. u_i and u_j such that $\pi_{u_i} = i'$ and $\pi_{u_j} = j^*$. By construction, we have $u_i < u_j$ as i' is executed before j^*. If $u_j > u_{i+1}$, there is at least one type-2 task performed between i' and j^*. Let $n = j - i - 1$ be the number of type-2 tasks between i' and j^* in the permutation. By shifting j^* by m units to the left of the permutation, with $1 \leq m \leq n$, one allows executing j^* earlier, and thus reducing the idle time between tasks i' and j^*. The larger m is, the earlier j^* is performed, and the more the idle time is reduced. Note that shifting j^* by a maximum of $n = j - i - 1$ units to the left of the permutation necessarily yields a precedence-feasible permutation. Indeed, because of the idle time $[a, b]$ on machine m^*, one infers that none of the type-2 tasks performed between i' and j^* is performed on machine m^*, meaning there are only disjunctions between these tasks and j^*.

At each iteration, the heuristic identifies the makespan machine m^* and all associated idle times. It then compresses these idle times ranked by ascending starting times. For each idle time $[a, b]$, such that there is at least one type-2 task between i' and j^* in the permutation (i.e. $n = j - i - 1 > 0$), a new permutation is created by shifting j^* to the left, and accepted if it does not worsen the makespan. The heuristic creates a new permutation

π' by left-shifting j^* as much as possible, i.e. by $m = n$ units, which allows compressing the idle time the most. If π' provides a worse makespan than π, m is reduced by 1, and the heuristic creates a new permutation π' by left-shifting j^* by m units. The procedure is applied until a permutation π' with a makespan at least as good as the current permutation π is found, or all positions between i' and j^* have been tried. If no such permutation is found, the heuristic moves to the next idle time. Otherwise, the permutation is accepted, and the procedure starts over. Figure 8 displays the schedule obtained after applying the heuristic to the solution associated with the schedule illustrated in Figure 6.

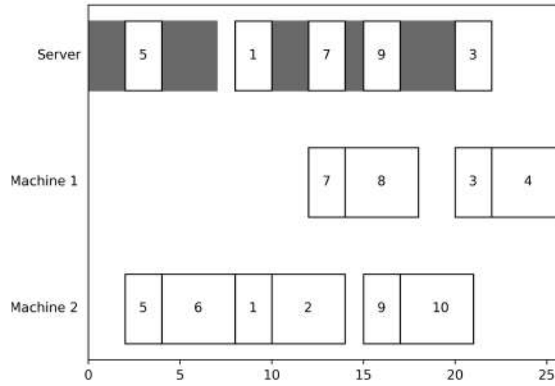


Figure 8: Improved schedule

4.3.4. Central server idle times heuristic

This procedure aims to reduce idle times in the same way as the previous heuristic, but for the central server. We first explain how to reduce an individual *central server idle time*. We then detail the steps of the proposed procedure. The heuristic pseudo-code is given in Appendix C.4.

Let $[a, b]$ be a *central server idle time*, which occurs between the end of type-2 task i^* and the start of type-2 task j^*, with m^* being the machine associated with j^*. The presence of idle time $[a, b]$ indicates that j^* cannot start right after the end of i^* plus the setup time $s_{i^*j^*}$. Therefore, by construction of schedule starting times, one can infer that the idle time is due to a time-lag related to the execution of a type-1 task on machine m^*. In the case of the schedule illustrated in Figure 6, the second machine finishes last, a *central server idle time* appears from time 7 to time 8, with $i^* = 5$, $j^* = 1$, and $m^* = 1$. The *central server idle time* is due to a time-lag related to the execution of type-1 task 6.

To reduce such *central server idle time*, one can insert between i^* and j^* a type-2 task executed after j^* on another machine than m^*. Suppose that j^{**} is the first type-2 task performed on a machine $m^{**} \neq m^*$ after j^*. If there is an idle time $[a', b']$ on machine m^{**} just before the start of j^{**}, and if this time interval intersects with $[a, b]$, then one may reduce the *central server idle time* by building a permutation obtained by placing j^{**} between

i^* and j^* in π. Specifically, one can identify the new starting time of j^{**} and use this information to identify if the move is worth it. In the new schedule, as i^* and j^{**} are executed consecutively by the central server, a setup time of $s_{i^*j^{**}}$ must be observed after the end of the i^* task. Task i^* ends at time $a - s_{i^*j^*}$, such that j^{**} cannot start before $a - s_{i^*j^*} + s_{i^*j^{**}}$. On the other hand, j^{**} cannot start before a'. Consequently, the starting time of j^{**} in the new permutation π' corresponds to the minimum value between a' and $a - s_{i^*j^*} + s_{i^*j^{**}}$. If this value is lower than b, then the move allows to reduce the idle time on the central server after the end of task i^*.

Back to the case of the schedule illustrated in Figure 6, the first type-2 task performed on the first machine after $j^* = 1$ is $j^{**} = 7$. There is an idle time on the first machine from time 0 to time 12 before the start of j^{**}. Here, $i^* = 5$ ends at time 4, and the setup time between $i^* = 5$ and $j^{**} = 7$ is $s_{i^*j^{**}} = 2$. Therefore, in the schedule built from permutation π', where $j^{**} = 7$ is interposed between $i^* = 5$ and $j^* = 1$, task $j^{**} = 7$ now starts at time 6, which allows removing the idle time on the central server after the end of task i^*. Again, note that the new permutation π' does not violate precedence constraints. Since j^{**} is the first type-2 task executed after j^* on machine m^{**}, all tasks between j^* and j^{**} in the permutation π are executed on machines different from m^* and m^{**}. Therefore, there are only disjunctions between these tasks and j^{**}. Similarly, as $m^* \neq m^{**}$, there is a disjunction between j^{**} and j^*, such that placing j^{**} between i^* and j^* does not violate any conjunction.

At each iteration, the heuristic identifies all *central server idle times*. It then compresses these idle times ranked by ascending starting times. For each idle time $[a, b]$ between type-2 tasks i^* and j^*, the heuristic detects the first type-2 tasks executed on other machines than m^* that can be performed by the central server before j^*. If no such type-2 task is found, the heuristic proceeds to the next idle time. Otherwise, amongst the identified type-2 tasks, the heuristic chooses the one with the earliest starting time. It constructs the new permutation π', as well as the associated schedule. If the new solution makespan is not worse than that of the current solution, the new permutation is accepted, and the procedure starts over. Figure 9 displays the schedule obtained after applying the heuristic to the solution associated with the schedule illustrated in Figure 6.

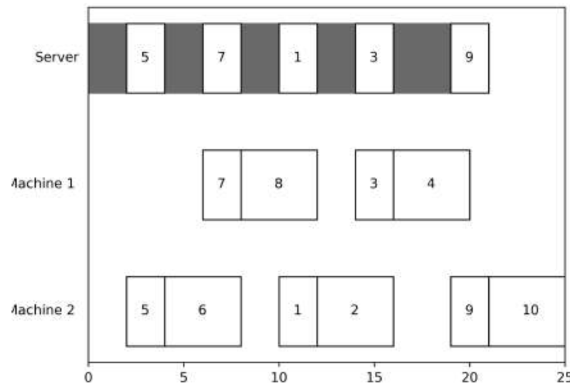


Figure 9: Improved schedule

4.3.5. Block-based local search

The last step of the VND applies the local search proposed by Hurink and Knust [25], which relies on the so-called block approach. Given job scheduling decisions ϕ, a precedence-feasible permutation of the type-2 tasks π, and the resulting critical path $p^{G^{\phi},\pi} = (0, p_1, ..., p_k, n+1)$, a block is defined as a subsequence of tasks $(p_b, ..., p_f)$ of at least two successive tasks for which (i) consecutive tasks of the subsequence are executed by the central server without idle times (ii) there is no conjunction between two consecutive tasks, and (iii) enlarging the subsequence by one task breaks one of the two previous properties. For their single-machine scheduling problem, Hurink and Knust [25] demonstrate that, if a permutation π' yields a better makespan than another permutation π, then in π' at least two tasks of a block of the critical path associated with π are processed in the opposite order as in π. The authors leverage this property and propose a neighborhood defined by three operators, consisting of (i) interchanging two adjacent tasks of a block, (ii) shifting the first block task to the right, and (iii) shifting an internal block task to the end of the block. To accelerate the local search, the authors propose a two-step evaluation of the neighbor solutions. Specifically, the authors first compute lower bounds on the neighbor solution makespans and accurately compute the makespan value of a neighbor solution only if its lower bound is not worse than the incumbent. For more details on the local search procedure, we refer the interested reader to [25].

4.4. Shaking operators

We use six different operators for shaking the incumbent in the course of the GVNS. The first operator modifies the central server permutation π, while the five remaining operators randomize the job scheduling decisions ϕ. Since different job scheduling decisions involve different conjunctions between type-2 jobs, each shaking operator on the job scheduling decisions is followed by a procedure to reconstruct the permutation. Specifically, for a solution (ϕ, π), and new job scheduling decisions ϕ', the procedure identifies, i^* and j^*, the first and last tasks of π that do not satisfy the precedence relations induced by ϕ'. Then, the procedure repairs π by randomly reconstructing the subsequence between i^* and j^* while respecting all the precedence relations. The first operator randomly determines two type-2 tasks i^* and j^*, and it determines a new π' obtained by randomly reconstructing the subsequence between i^* and j^* in π, while respecting all the precedence relations. The second operator switches two jobs on a random machine. The third operator selects a job on a given machine and schedules it on another machine at a random position. The fourth operator shuffles the order of the jobs on a given machine. The fifth operator shuffles the job scheduling decision between two machines, i.e. it identifies all the jobs scheduled on two given machines and reschedules these jobs randomly on these two machines. The last operator shuffles all the job scheduling decisions.

5. Computational study

We present a computational study performed on randomly generated small-size instances, randomly generated large-size instances, and a case study. We first explain how the instance generator operates and we describe the

two sets of random instances. We then present experiments conducted on the small-size instances to compare the proposed mathematical models and the GVNS. We also present experiments conducted on large-size instances to assess the performance of the GVNS according to the different instance parameters, as well as to analyze how the instance parameters impact the solution structures. Finally, we provide experimental results on instances of an automated kitchen provided by our industrial partner. All algorithms are coded in C++ and executed on an Intel E5-2683 2.1GHz processor with 8GB of memory under Linux 18.04. Integer programs were solved using CPLEX 12.8 and executed with stopping criteria of a proven optimality gap of 1% or less and a maximum run-time of 2 hours. The GVNS as well as the mathematical formulations are initiated with the heuristic solution obtained using the procedure described in Appendix B. For the GVNS, the time limit t_{max} is set to 15 minutes. The maximum time without improvement, t_{max}^{impr}, is set to 15 seconds.

5.1. Instances

The randomly generated instances are produced by a generator that operates as follows. The first two parameters used are $|K|$, the number of jobs to schedule, and M, the number of machines in the scheduling environment. Then, the generator constructs each job $k \in K$ by defining the different tasks that compose it, as well as the precedence relations between these tasks. Recalling that job precedence graphs have a multi-staged structure, with even stages being composed of a group of type-2 tasks and odd stages being composed of a single type-1 task, jobs are generated based on two parameters: max^L and $max_L^{T^2}$. The first parameter is a strictly positive even integer that indicates the maximum number of stages per job. The second parameter is a strictly positive integer which indicates the maximum number of type-2 tasks per even stage. For each job $k \in K$, the number of stages is an even integer randomly chosen between 2 and max^L. For each even stage, the number of type-2 tasks is a strictly positive integer randomly chosen between 1 and $max_L^{T^2}$.

The processing time p_i of each type-1 task $i \in T^1$ is randomly chosen in the interval $[\gamma_1^1, \gamma_2^1]$. The processing time p_i of each type-2 task $i \in T^2$ is randomly chosen in the interval $[\gamma_1^2, \gamma_2^2]$. Sequence-dependent setup times, s_{ij}, are set to $\alpha \times \frac{p_i+p_j}{2}$ with α being chosen in the interval $[\delta_1, \delta_2]$ according to a uniform distribution. The interval $[\delta_1, \delta_2]$ plays on the relative duration of setup times compared to type-2 tasks, as larger values of α imply longer setup times.

For our experiments, we generate small-size instances based on the following parameter values: $M = \{2, 3, 4\}$, $|K| = \{10, 20\}$, $max^L = \{2, 4\}$, $max_L^{T^2} = \{1, 3\}$, $[\gamma_1^1, \gamma_2^1] = \{[1, 50], [50, 100], [1, 100]\}$, $[\gamma_1^2, \gamma_2^2] = \{[1, 50], [50, 100], [1, 100]\}$ and $[\delta_1, \delta_2] = \{[0.01, 0.05], [0.05, 0.15], [0.15, 0.30]\}$. For each problem, five instances are generated, which yields a total of 3240 small-size instances. We also generate large-size instances based on the following parameter values: $M = \{3, 5, 10\}$, $|K| = \{20, 40, 60, 80\}$, $max^L = \{2, 4, 6\}$, $max_L^{T^2} = \{1, 3, 5\}$, $[\gamma_1^1, \gamma_2^1] = \{[1, 50], [50, 100], [1, 100]\}$, $[\gamma_1^2, \gamma_2^2] = \{[1, 50], [50, 100], [1, 100]\}$ and $[\delta_1, \delta_2] = \{[0.01, 0.05], [0.05, 0.15], [0.15, 0.30]\}$. For each problem, five instances are generated, which yields a total of 14580 large-size instances.

5.2. Results on the small-size instances

We first assess the performance of both mathematical formulations embedding the lower bounds described in Appendix A. The first and second mathematical formulations are referred to as "Index" and "No-Index", respectively. For each formulation, we present the number of instances solved by CPLEX within the optimality tolerance. These numbers are averaged over instances with the same number of jobs in Table 7, and they are averaged over instances with the same number of machines in Table 8. The best values are indicated in bold.

Table 7: Instances solved according to the number of jobs

$\|K\|$	Index	No-Index
10	169	**201**
20	21	**77**

Table 8: Instances solved according to the number of machines

$\|M\|$	Index	No-Index
2	**99**	76
3	55	**79**
4	36	**123**

CPLEX has a weak performance as only 190 and 278 of the 3240 small-size instances are solved within the optimality tolerance when using formulations "Index" and "No-Index", respectively. Overall, the formulation without an index yields better performance than its indexed counterpart, except for the instances with two machines. In total, 318 instances are solved within optimality tolerance.

To further investigate the performance of the mathematical formulations, we compare both formulations on the unsolved instances. For both formulations, we indicate the optimality gap at termination, the relative improvement of the initial lower bound, and the relative improvement of the initial upper bound. Recalling that the lower bound LB is described in Appendix A and obtained by computing the maximum value between a lower bound on the central server, LB_1, and a lower bound on the machines, LB_2, (see Appendix A). In addition, the initial upper bound is obtained from the construction heuristic in Appendix B. Relative improvements are computed as $gap_{lb} = (lb^{init} - lb^x)/lb^{init}$ and $gap_{ub} = (ub^x - ub^{init})/ub^x$, where lb^{init} is the initial lower bound, ub^{init} is the initial upper bound, and lb^x and ub^x are the lower bound and upper bound at termination for formulation x. Values are averaged over instances with the same number of jobs and machines and indicated in Table 9.

Reported average gaps at termination for the unsolved instances range between 6.34% and 9.35%. CPLEX significantly improves the initial heuristic solution for both formulations, especially for instances with 10 jobs. However, CPLEX finds it difficult to improve the initial lower bound. These results suggest that the gaps at termination are essentially due to an inability to improve the lower bound. Overall, we find that the formulation without an index outperforms the formulation with an index as it yields better gaps at termination. Specifically, CPLEX manages to improve the initial heuristic solution much more effectively in the case of the formulation without an index. For that formulation, the improvement of the initial upper bound is greater as the number of machines increases. We also observe that both formulations yield a smaller gap at termination when the number

Table 9: MILP formulation performances on the unsolved instances

		opt. gap. (%)		gap_{lb} (%)		gap_{ub} (%)	
$\|K\|$	$\|M\|$	Index	No-Index	Index	No-Index	Index	No-Index
10	2	7.59	**7.36**	**0.16**	0.00	3.36	**3.77**
10	3	7.51	**6.62**	**0.07**	0.01	2.72	**3.74**
10	4	7.66	**6.34**	**0.06**	0.02	2.26	**3.72**
20	2	9.35	**8.98**	0.00	0.00	1.38	**1.79**
20	3	8.78	**7.64**	0.00	0.00	0.76	**2.00**
20	4	8.70	**6.91**	0.00	0.00	0.35	**2.27**

of machines increases. This can be explained by LB_1 being tighter, and thus LB potentially being of better quality. Indeed, increasing the number of machines leads to solutions with a smaller makespan while not affecting the lower bound on the central server. In the case where LB_1 is greater than LB_2, the increase in the number of machines leads to a tighter lower bound LB and therefore better gaps. Overall, both formulations have difficulties scaling with the increasing number of jobs.

We assess the performance of the GVNS on the instances solved within the optimality tolerance. For these instances, we compute the gap between the objective values of the optimal solution and the solution computed by the GVNS. This primal gap is computed as $100 \times (ub^{GVNS} - ub^*)/ub^{GVNS}$, where ub^{GVNS} is the objective value of the final solution returned by the GVNS, and ub^* is the optimal objective function value. Figure 10 presents the cumulative percentage of instances with a primal gap that is lower than or equal to the corresponding value.

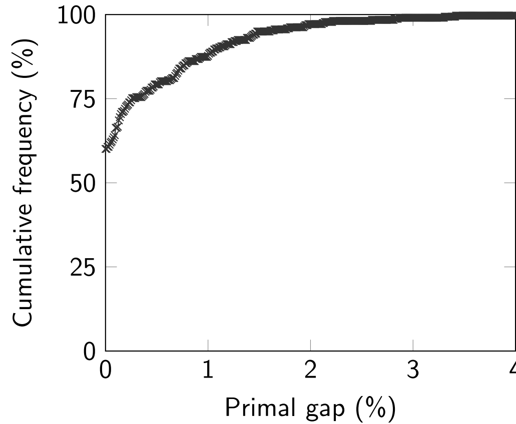


Figure 10: Cumulative percentage of instances within a given primal gap

The GVNS computes optimal solutions for 191 out of the 318 closed instances and yields an average primal gap of 0.31%, while only spending a fraction of the CPU time required by the most effective formulation, less than 1%. In addition, the primal gap standard deviation is 0.42%, and remains below 1% for almost 90% of the

instances. We also assess the performance of the GVNS on the unsolved instances and indicate the corresponding results in Table 10. For formulation x, we compute (i) the relative improvement of the objective value between the GVNS solution and the solution returned by CPLEX after two hours, i.e. $impr_{ub} = (ub^x - ub^{GVNS})/ub^x$, as well as (ii) the ratio between computation times, i.e. time-ratio $= time^{GVNS}/time^x$, where $time^{GVNS}$ is the GVNS time at termination, and $time^x$ is CPLEX time at termination (i.e. 2 hours). For each method, we also report *Best*, i.e. the percentage of instances for which the considered method found the best upper bound over all methods.

Table 10: Comparison of the MILP formulations and the GVNS on the unsolved instances

		Index		No-Index				
$\lvert K\rvert$	$\lvert M\rvert$	$impr_{ub}$ (%)	time-ratio (%)	$impr_{ub}$ (%)	time-ratio (%)	Index *Best*(%)	No-Index *Best*(%)	GVNS *Best*(%)
10	2	1.52	0.50	1.12	0.49	12.61	13.72	**80.53**
10	3	1.58	0.48	0.56	0.48	12.15	23.03	**73.99**
10	4	1.64	0.47	0.18	0.47	9.44	42.49	**60.73**
20	2	2.60	0.54	2.21	0.50	5.32	3.61	**92.02**
20	3	2.46	0.49	1.24	0.48	1.91	8.97	**89.89**
20	4	2.48	0.47	0.55	0.46	1.65	30.52	**71.13**
Total		2.05	0.49	0.98	0.48	7.18	20.39	**78.05**

For both formulations, the solutions returned by CPLEX after two hours are clearly worse than those computed by the GVNS. On average, the GVNS determines solutions that are 2.05% better than those obtained when solving the formulation with an index, while only spending 0.49% of the corresponding CPU time. The improvement is less significant when considering the formulation without an index, for which CPLEX significantly improves the initial upper bound. Nevertheless, GVNS finds solutions 0.98% better than those obtained when solving the formulation without an index, in a fraction of the CPU time. Note that the performance gap increases with the number of jobs considered, which motivates developing a metaheuristic approach for solving large-size instances. If GVNS finds the best solution among all methods for 78.05% of the instances, this number is close to 70% for instances with 10 jobs and approximately of 85% for instances with 20 jobs.

5.3. Results on the large-size instances

We perform experiments on the large-size instances to assess the effectiveness of the GVNS method and to gain insights into how instance parameters impact the problem difficulty. Several measures are reported in Table 11 and averaged over instances with the same parameter value, for all the considered parameters. We compute the gap between the heuristic solution objective value and LB, and we indicate the computation time of the GVNS algorithm. To compare the quality of the lower bounds, we report the percentage of instances for which the lower bound LB is equal to LB_1 and LB_2, respectively. We also report a ratio LB_1/LB_2 that indicates the gap between these two lower bounds.

Table 11: Performance of the GVNS on the large-size instances

Instances		Gap (%)	Time (sec.)	LB_1 (%)	LB_2 (%)	$\frac{LB_1}{LB_2}$
$\lvert K \rvert$	20	5.75	39.99	98.00	2.00	3.8
	40	6.05	181.68	99.00	1.00	3.79
	60	6.16	355.82	99.00	1.00	3.78
	80	6.23	474.74	99.00	1.00	3.77
$\lvert M \rvert$	3	6.79	248.25	95.72	4.28	1.89
	5	5.82	258.35	100.00	0.00	3.15
	10	5.53	282.57	100.00	0.00	6.31
max^L	2	5.77	161.75	98.56	1.44	3.79
	4	6.14	277.39	98.46	1.54	3.78
	6	6.24	350.02	98.70	1.30	3.78
$max_L^{T^2}$	1	6.42	117.68	95.72	4.28	3.13
	3	5.94	286.84	100.00	0.00	3.92
	5	5.79	384.64	100.00	0.00	4.30
$[\gamma_1^1.\gamma_2^1]$	[1.50]	5.51	269.43	100.00	0.00	4.47
	[50.100]	5.98	261.35	99.42	0.58	3.71
	[1.100]	6.66	258.39	96.30	3.70	3.18
$[\gamma_1^2.\gamma_2^2]$	[1.50]	7.35	250.66	95.72	4.28	3.04
	[50.100]	7.16	266.47	100.00	0.00	3.87
	[1.100]	3.63	272.04	100.00	0.00	4.44
$[\delta_1.\delta_2]$	[0.01.0.05]	1.46	256.79	98.56	1.44	3.81
	[0.05.0.15]	5.39	262.64	98.56	1.44	3.78
	[0.15.0.30]	11.30	269.74	98.60	1.40	3.76

Overall, the average gap is 6.04% for the large-size instances, which confirms the good performance of the proposed GVNS method. In addition, the algorithm performance is only slightly impacted by the variation of the values considered for the first five parameters. Specifically, the gap between the GVNS solution and the lower bound worsens slightly as one increases the number of jobs, the number of stages, or the type-1 task durations, and improves marginally as the number of machines or the number of type-2 tasks per stage increases. On the other hand, the performance is significantly affected by variations of the value of the last parameter, i.e. the setup range, $[\delta_1, \delta_2]$. The gap appears to be positively correlated with the setup operation durations, as it takes an average value of 1.46% for small setup times, 5.39% for medium setup times, and 11.30% for large setup times. From this result, one can infer that the proposed metaheuristic is particularly suitable for applications where setup times are short relative to the type-2 task durations.

This algorithm property is suitable for the industrial application that motivates this paper. Indeed, in the

context of operating a robotic automated kitchen, ingredient dispensing times (i.e. the duration of the type-2 tasks) are much longer than setup times, especially because ingredient grammages must be accurately respected to complete the recipes. In addition, one can note that gaps at termination are significantly smaller for instances with a greater variance in type-2 task durations. The computation times required for the large-size instances are reasonable as they yield an overall value of 263 seconds. Three parameters significantly impact the computation times, namely, the number of jobs, the number of stages per job, and the number of type-2 tasks per stage. Regarding lower bounds, the bound related to the central server clearly outperforms that related to the machines, as it is stronger for 98.61% of the instances. Overall, LB_1 is greater than LB_2 by a factor of 3.78.

We now investigate how instance parameters impact solution structures. For each instance and each job k, we compute a ratio between the actual duration of the job and d_k, i.e. the minimum duration of this job. With $start_k$ and end_k being the beginning and the end of job k, respectively, we denote this ratio as t-dev$^k = (end_k - start_k)/d_k$. It is equal to 1 when the job duration is minimal, and it increases as the job duration deviates from d_k. We denote the average value and the standard deviation over all jobs as t-dev^{av} and t-dev^{sd}, respectively. For each instance and each machine, we refer to m-use as the percentage of time during which the machine is processing a task. We denote the average value and the standard deviation over all jobs as m-use^{av} and m-use^{sd}, respectively. For each instance, we refer to cs-use as the percentage of time during which the central server is processing a type-2 task or performing a setup operation. Measures t-dev^{av}, t-dev^{sd}, m-use^{av}, m-use^{sd}, and cs-use are reported in Table 12 and averaged over instances with the same parameter value.

Table 12: Solution structures for the large-size instances

Instances		t-dev^{av}	t-dev^{sd}	m-use^{av}(%)	m-use^{sd}(%)	cs-use(%)
$\lvert K \rvert$	20	2.92	1.6	32.61	5.33	99.00
	40	2.92	1.65	32.50	2.98	99.24
	60	2.89	1.59	32.59	2.31	99.30
	80	2.89	1.6	32.62	1.99	99.32
$\lvert M \rvert$	3	1.72	0.52	50.69	3.05	98.34
	5	2.53	1.26	31.33	3.41	99.62
	10	4.47	3.05	15.73	3.00	99.68
max^L	2	2.47	1.54	32.69	3.95	99.16
	4	3.03	1.63	32.57	2.95	99.22
	6	3.22	1.66	32.49	2.56	99.26
$max_L^{T^2}$	1	2.01	1.18	40.48	3.84	98.30
	3	3.04	1.84	30.46	3.19	99.56
	5	3.66	1.82	26.82	2.44	99.78
$[\gamma_1^1.\gamma_2^1]$	[1.50]	3.35	2.06	26.39	3.07	99.90
	[1.100]	2.85	1.61	32.81	3.37	99.53
	[50.100]	2.52	1.17	38.55	3.02	98.21
$[\gamma_1^2.\gamma_2^2]$	[1.50]	2.44	1.29	39.98	3.61	98.26
	[1.100]	2.99	1.79	30.58	3.33	99.58
	[50.100]	3.29	1.75	27.19	2.52	99.79
$[\delta_1.\delta_2]$	[0.01.0.05]	2.74	1.44	34.69	3.14	99.07
	[0.05.0.15]	2.87	1.55	33.01	3.16	99.19
	[0.15.0.30]	3.11	1.85	30.05	3.16	99.38

Overall, we observe that the usage of the central server is significantly higher than that of the machines. This is expected as the central server must process many more tasks than the other machines, and is thus a bottleneck resource. Nevertheless, the use of the arm is very close to 100% regardless of the case considered, which indicates that the solutions obtained can hardly be improved w.r.t. the bottleneck resource. On average, job durations tend to be three times longer than their minimum achievable. This value is stable as the number of jobs considered increases. On the other hand, t-dev^{av} increases with the number of machines, the number of stages per job, the number of type-2 tasks per stage, the duration of the type-2 tasks, and the duration of the setup times. Recall that the number of machines is an upper bound on the number of jobs that can be performed at the same time, the more machines the more likely it is that the central server will be required simultaneously by different jobs, which leads to an increase in t-dev^{av}. We also observe that t-dev^{sd} increases significantly with $|M|$, which indicates that job duration deviations are greatly affected by the number of machines considered. Unsurprisingly, t-dev^{av} also increases with max^L and $max_L^{T^2}$, as increasing the number of stages per job and the number of type-2 tasks

per stage leads to longer and more complex jobs. Similarly, increasing the duration of the type-2 tasks and the setup times increases the central server workload, which in turn increases the probability that the central server is occupied while it is needed, and thus increases t-dev^{av}. On the other hand, we observe that t-dev^{av} decreases when the durations of the type-1 tasks are larger. This can be explained by the fact that increasing type-1 task durations leads to higher minimum job durations d_k while not affecting the workload of the central server. One can also note that t-dev^{sd} decreases as type-1 task durations increase, suggesting that the obtained schedules are more homogeneous w.r.t. the deviation between job durations and their minimum duration.

On average, the machine usage is slightly higher than 30%. The value of m-use^{av} is quite stable w.r.t. the number of jobs. On the other hand, the standard deviation relative to machine usage, m-use^{sd}, tends to decrease w.r.t. the number of jobs, which means that the workload is better balanced between the different machines when the number of jobs increases. The same trends are observed w.r.t. the number of steps per job. The average machine usage naturally decreases with the number of machines considered, as in that case, one needs to distribute the same workload over a larger number of resources. Conversely, the machine workload balance is stable w.r.t. the number of machines. Similar results are observed for the duration of setup times. Unsurprisingly, increasing the type-1 task durations leads to a higher average machine usage, as longer type-1 tasks correspond to an increase in the total machine workload. On the other hand, this parameter has little impact on the machine workload balance.

5.4. Application to a real-world instance

We test the metaheuristic on a real-world instance provided by our industrial partner with a number of parallel machines (i.e. multifunctional food processors) varying from 1 to 20. This case study corresponds to a mise-en-place preparation for 400 portions. Specifically, the mise-en-place preparation of 100 dishes is performed, with each dish having 4 portions. Detailed information cannot be provided due to confidentiality issues. Table 13 provides the makespan change according to the number of multifunctional food processors, as well as measures t-dev^{av}, t-dev^{sd}, m-use^{av}, m-use^{sd}, and cs-use.

Unsurprisingly, increasing the number of cooking devices improves the automated production system output. In particular, the obtained makespan is greater than 19 hours in the worst case and lower than 4 hours in the best case. The reduction in makespan is relatively large when the first machines are added, and gradually decreases as the number of machines increases. The makespan reduction becomes minor from the 8th machine added and is null from the 19th machine added. We empirically observe that the makespan improvement goes through an increase in the use of the central server, to the detriment of the use of the cooking devices. For the best solutions, the average use of cooking devices is relatively low, which indicates that the machines are waiting for the central server and yields greater job duration deviations.

Table 13: Makespans and solution structures for the real-life application

$\lvert M \rvert$	Makespan (hours)	t-dev^{av}	t-dev^{sd}	m-use^{av}(%)	m-use^{sd}(%)	cs-use(%)
1	19.25	1.05	0.04	0.98	-	0.23
2	10.04	1.13	0.22	0.94	0.00	0.44
3	7.42	1.42	0.45	0.85	0.02	0.57
4	6.24	1.73	0.69	0.75	0.02	0.67
5	5.67	1.73	0.98	0.67	0.04	0.76
6	5.06	2.18	2.70	0.62	0.04	0.85
7	4.74	2.33	1.57	0.57	0.03	0.88
8	4.53	2.88	1.76	0.52	0.04	0.90
9	4.39	3.05	1.66	0.48	0.03	0.92
10	4.33	3.25	2.19	0.43	0.04	0.94
11	4.33	3.68	2.34	0.40	0.02	0.91
12	4.20	3.74	2.63	0.37	0.06	0.94
13	4.16	4.26	3.03	0.35	0.07	0.95
14	4.14	4.36	2.98	0.33	0.05	0.96
15	4.11	4.39	2.91	0.31	0.07	0.97
16	4.06	4.57	2.81	0.29	0.06	0.98
17	4.00	5.06	3.26	0.28	0.06	0.99
18	3.94	4.97	3.96	0.27	0.06	1.00
19	3.93	4.99	3.04	0.25	0.04	1.00
20	3.93	5.13	3.10	0.24	0.05	1.00

6. Conclusions and perspectives

This paper introduced a problem that consists of scheduling multi-staged jobs in a production environment with parallel identical machines and a central server with sequence-dependent setup times. The studied problem finds applications in the restaurant industry, for which the automation of cooking production processes represents a clear opportunity to improve both production profitability and sustainability. Specifically, it aims to schedule operations in a production environment that consists of (i) a set of multifunctional food processors, (ii) a shelf storing the ingredients, and (iii) a central server serving as a bridge between the multifunctional food processors and the shelf. We have presented two mathematical formulations, one with a machine index and one without, as well as lower bounds for both formulations. We also have developed an effective GVNS algorithm to solve this problem. In the proposed approach, the variable neighborhood descent employs several original improvement heuristics designed to eliminate idle times. Through an extensive series of experiments carried out on random instances, we tested the performance of the proposed metaheuristic under various combinations of instance parameters and we computationally demonstrated the effectiveness of our method. The results show that the proposed approach computes solutions near to the optimum in the case of short sequence-dependent setup times

and that its performance remains stable for most instance parameters. We also presented a case study provided by our industrial partner and analyzed the appropriate number of machines to consider to maximize the output of its production system.

To the best of our knowledge, this is the first paper that addresses the scheduling of multi-staged jobs in a production environment with parallel identical machines and a central server with sequence-dependent setup times. This problem arises in multiple industrial applications and warrants further research. We model an offline optimization problem with makespan minimization that applies in the context of mise-en-place preparation. A clear path for future research is to focus on the scheduling problem occurring during peak-time, which corresponds to an online optimization variant with waiting times minimization [45, 49]. Furthermore, the mathematical formulations and the solution method proposed should be extended to accommodate more complex production environments that include machine eligibility constraints, or multiple central servers.

Acknowledgements

Computations were made on the supercomputer "Cedar" managed by Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), the Ministère de l'économie, de la science et de l'innovation du Québec (MESI) and the Fonds de recherche du Québec – Nature et technologies (FRQ-NT).

References

[1] Abdekhodaee, A. H. and Wirth, A. (2002). Scheduling parallel machines with a single server: some solvable cases and heuristics. *Computers & Operations Research*, 29(3):295–315.

[2] Abdekhodaee, A. H., Wirth, A., and Gan, H. S. (2004). Equal processing and equal setup time cases of scheduling parallel machines with a single server. *Computers & Operations Research*, 31(11):1867–1889.

[3] Abdekhodaee, A. H., Wirth, A., and Gan, H.-S. (2006). Scheduling two parallel machines with a single server: the general case. *Computers & Operations Research*, 33(4):994–1009.

[4] Afsar, H. M., Lacomme, P., Ren, L., Prodhon, C., and Vigo, D. (2016). Resolution of a job-shop problem with transportation constraints: a master/slave approach. *IFAC-PapersOnLine*, 49(12):898–903.

[5] Association, N. R. (2022). National Restaurant Association kernel description.

[6] Behnamian, J., Ghomi, S. F., Jolai, F., and Amirtaheri, O. (2012). Minimizing makespan on a three-machine flowshop batch scheduling problem with transportation using genetic algorithm. *Applied Soft Computing*, 12(2):768–777.

[7] Bektur, G. and Saraç, T. (2019). A mathematical model and heuristic algorithms for an unrelated parallel machine scheduling problem with sequence-dependent setup times, machine eligibility restrictions and a common server. *Computers & Operations Research*, 103:46–63.

[8] Biskup, D., Herrmann, J., and Gupta, J. N. (2008). Scheduling identical parallel machines to minimize total tardiness. *International Journal of Production Economics*, 115(1):134–142.

[9] Brucker, P., Dhaenens-Flipo, C., Knust, S., Kravchenko, S. A., and Werner, F. (2002). Complexity results for parallel machine problems with a single server. *Journal of Scheduling*, 5(6):429–457.

[10] Burdett, R. L. and Kozan, E. (2010). A disjunctive graph model and framework for constructing new train schedules. *European Journal of Operational Research*, 200(1):85–98.

[11] Elidrissi, A., Benmansour, R., Benbrahim, M., and Duvivier, D. (2021). Mathematical formulations for the parallel machine scheduling problem with a single server. *International Journal of Production Research*, 59(20):6166–6184.

[12] Elmi, A. and Topaloglu, S. (2014). Scheduling multiple parts in hybrid flow shop robotic cells served by a single robot. *International Journal of Computer Integrated Manufacturing*, 27(12):1144–1159.

[13] Fortemps, P. and Hapke, M. (1997). On the disjunctive graph for project scheduling. *Foundations of Computing and Decision Sciences*, 22(3):195–209.

[14] Gan, H.-S., Wirth, A., and Abdekhodaee, A. (2012). A branch-and-price algorithm for the general case of scheduling parallel machines with a single server. *Computers & Operations Research*, 39(9):2242–2247.

[15] Glass, C. A., Shafransky, Y. M., and Strusevich, V. A. (2000). Scheduling for parallel dedicated machines with a single server. *Naval Research Logistics (NRL)*, 47(4):304–328.

[16] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. Elsevier.

[17] Hall, N. G., Potts, C. N., and Sriskandarajah, C. (2000). Parallel machine scheduling with a common server. *Discrete Applied Mathematics*, 102(3):223–243.

[18] Hamzadayi, A. and Yildiz, G. (2017). Modeling and solving static m identical parallel machines scheduling problem with a common server and sequence dependent setup times. *Computers & Industrial Engineering*, 106:287–298.

[19] Hasani, K., Kravchenko, S. A., and Werner, F. (2014a). Block models for scheduling jobs on two parallel machines with a single server. *Computers & Operations Research*, 41:94–97.

[20] Hasani, K., Kravchenko, S. A., and Werner, F. (2014b). Minimising interference for scheduling two parallel machines with a single server. *International Journal of Production Research*, 52(24):7148–7158.

[21] Hasani, K., Kravchenko, S. A., and Werner, F. (2014c). Simulated annealing and genetic algorithms for the two-machine scheduling problem with a single server. *International Journal of Production Research*, 52(13):3778–3792.

[22] Hasani, K., Kravchenko, S. A., and Werner, F. (2016). Minimizing the makespan for the two-machine scheduling problem with a single server: Two algorithms for very large instances. *Engineering Optimization*, 48(1):173–183.

[23] Huang, S., Cai, L., and Zhang, X. (2010). Parallel dedicated machine scheduling problem with sequence-dependent setups and a single server. *Computers & Industrial Engineering*, 58(1):165–174.

[24] Hurink, J. and Knust, S. (2001). Makespan minimization for flow-shop problems with transportation times and a single robot. *Discrete Applied Mathematics*, 112(1-3):199–216.

[25] Hurink, J. and Knust, S. (2002). A tabu search algorithm for scheduling a single robot in a job-shop environment. *Discrete Applied Mathematics*, 119(1-2):181–203.

[26] Hurink, J. and Knust, S. (2005). Tabu search algorithms for job-shop problems with a single transport robot. *European Journal of Operational Research*, 162(1):99–111.

[27] Jain, A. S. and Meeran, S. (1999). Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113(2):390–434.

[28] Kim, H.-J. and Lee, J.-H. (2021). Scheduling uniform parallel dedicated machines with job splitting, sequence-dependent setup times, and multiple servers. *Computers & Operations Research*, 126:105115.

[29] Kim, M.-Y. and Lee, Y. H. (2012). Mip models and hybrid algorithm for minimizing the makespan of parallel machines scheduling problem with a single server. *Computers & Operations Research*, 39(11):2457–2468.

[30] Kise, H. (1991). On an automated two-machine flowshop scheduling problem with infinite buffer. *Journal of the Operations Research Society of Japan*, 34(3):354–361.

[31] Kise, H., Shioyama, T., and Ibaraki, T. (1991). Automated two-machine flowshop scheduling: A solvable case. *IIE Transactions*, 23(1):10–16.

[32] Koulamas, C. P. (1996). Scheduling two parallel semiautomatic machines to minimize machine interference. *Computers & Operations Research*, 23(10):945–956.

[33] Kravchenko, S. A. and Werner, F. (1997). Parallel machine scheduling problems with a single server. *Mathematical and Computer Modelling*, 26(12):1–11.

[34] Kravchenko, S. A. and Werner, F. (1998). Scheduling on parallel machines with a single and multiple servers. *Otto-von-Guericke-Universitat Magdeburg*, 30(98):1–18.

[35] Lacomme, P., Larabi, M., and Tchernev, N. (2007). A disjunctive graph for the job-shop with several robots. In *MISTA conference*, volume 20, pages 285–292. Citeseer.

[36] Lacomme, P., Larabi, M., and Tchernev, N. (2013). Job-shop based framework for simultaneous scheduling of machines and automated guided vehicles. *International Journal of Production Economics*, 143(1):24–34.

[37] Li, J.-q., Pan, Q.-k., and Wang, F.-t. (2014). A hybrid variable neighborhood search for solving the hybrid flow shop scheduling problem. *Applied Soft Computing*, 24:63–77.

[38] Liu, S. Q. and Kozan, E. (2009). Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Computers & Operations Research*, 36(10):2840–2852.

[39] Mejía, G. and Yuraszeck, F. (2020). A self-tuning variable neighborhood search algorithm and an effective decoding scheme for open shop scheduling problems with travel/setup times. *European Journal of Operational Research*, 285(2):484–496.

[40] Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100.

[41] Naderi, B., Tavakkoli-Moghaddam, R., and Khalili, M. (2010). Electromagnetism-like mechanism and simulated annealing algorithms for flowshop scheduling problems minimizing the total weighted tardiness and makespan. *Knowledge-Based Systems*, 23(2):77–85.

[42] Nouri, H. E., Driss, O. B., and Ghédira, K. (2016). Simultaneous scheduling of machines and transport robots in flexible job shop environment using hybrid metaheuristics based on clustered holonic multiagent model. *Computers & Industrial Engineering*, 102:488–501.

[43] Rossi, A. (2014). Flexible job shop scheduling with sequence-dependent setup and transportation times by ant colony with reinforced pheromone relationships. *International Journal of Production Economics*, 153:253–267.

[44] Roy, B. and Sussmann, B. (1964). Les problèmes d'ordonnancement avec contraintes disjonctives. *Note DS no. 9 bis, SEMA, Paris, France*.

[45] Su, C. (2013). Online lpt algorithms for parallel machines scheduling with a single server. *Journal of Combinatorial Optimization*, 26(3):480–488.

[46] Tang, L. and Liu, P. (2009). Flowshop scheduling problems with transportation or deterioration between the batching and single machines. *Computers & Industrial Engineering*, 56(4):1289–1295.

[47] Yazdani, M., Amiri, M., and Zandieh, M. (2010). Flexible job-shop scheduling with parallel variable neighborhood search algorithm. *Expert Systems with Applications*, 37(1):678–687.

[48] Zabihzadeh, S. S. and Rezaeian, J. (2016). Two meta-heuristic algorithms for flexible flow shop scheduling problem with robotic transportation and release time. *Applied Soft Computing*, 40:319–330.

[49] Zhang, L. and Wirth, A. (2009). On-line scheduling of two parallel machines with a single server. *Computers & Operations Research*, 36(5):1529–1553.

[50] Zhang, Q., Manier, H., and Manier, M.-A. (2012). A genetic algorithm with tabu search procedure for flexible job shop scheduling with transportation constraints and bounded processing times. *Computers & Operations Research*, 39(7):1713–1723.

Appendix A. Lower bounds

We strengthen both formulations with two lower bounds on the optimal objective function value. Let $\dot{S}$ be a set that contains the setup times s_{0i} for all $i \in T^2$, and let $\dot{s}$ be the the smallest value of $\dot{S}$. Let $\ddot{S}$ be a set that contains the setup times s_{ij} for all $i \in T^2, j \in T^2$, such that $j > i$. Let $\ddot{s}$ be the sum of the $n_2 - 1$ smallest values of $\ddot{S}$.

Lemma 1. *A lower bound related to the central server, LB_1, is $\dot{s} + \ddot{s} + \sum_{i \in T^2} p_i$*

Lemma 2. *A lower bound related to the machines, LB_2, is $\frac{\sum_{k \in K} d_k}{M}$*

As a result, a lower bound on the makespan of the problem, LB, is $\max\{LB_1, LB_2\}$.

Example 1. *Assume the same example problem of Section 3.2. $\dot{S} = \{1, 5, 5, 3, 1\}$ and $\dot{s} = 1$. In addition, $\ddot{S} = \{2, 5, 3, 2, 5, 3, 3, 3, 1, 5\}$ and $n_2 = 5$, such that $\ddot{s} = 8$. Thus, $LB_1 = \dot{s} + \ddot{s} + \sum_{i \in T^2} p_i = 1 + 8 + 16 = 25$. On the other hand, jobs 1 and 2 have minimum durations of $d_1 = 24$, and $d_2 = 17$, respectively. 2 machines are considered, such that $LB_2 = \frac{\sum_{k \in K} d_k}{M} = \frac{24+17}{2} = 20.5$. The lower bound on the makespan of the problem is $LB = max\{25, 20.5\}$*

Appendix B. Construction heuristic

The algorithm starts by creating an empty solution (ϕ,π) and two sets, ϕ^{init} and π^{init}, which contain all the jobs and type-2 tasks of the considered instance, respectively. All machines are defined as *free*, i.e. they are not processing any job. The available time of all the machines and the central server are set to 0. Then, the algorithm iteratively adds a type-2 task to π, and potentially a job to ϕ, until the solution is complete. At each iteration, it identifies the machines with the earliest available times. If one of these machines is free, then (i) it assigns to that machine the job in ϕ^{init} with the longest process time, and (ii) it defines the machine as *occupied*. Then, the algorithm gives priority to the job in progress with the longest remaining processing time. For this job k^* executed on machine $\bar{m}$, the algorithm identifies the precedence-feasible and non-processed type-2 task i^* that can be executed the earliest, i.e. the precedence-feasible non-processed type-2 task of k^* that minimizes setup time. The task is added to the permutation, and the schedule is updated accordingly. In addition, if all the tasks belonging to the same stage as i^* in the precedence graph of job k^*, are scheduled, then the type-1 task of the subsequent stage can be executed and the schedule is updated accordingly. The available times of the central server and machine $\bar{m}$ are updated. Finally, if all tasks of the k^* job are scheduled, machine $\bar{m}$ is defined as free.

Algorithm 4 Construction heuristic

Data: Instance

$\pi \leftarrow \emptyset$;

$\phi \leftarrow (\phi_1, ..., \phi_M)$ with $\phi_m = \emptyset, \ \ \forall m \in \{1, ..., M\}$;

$\phi^{init} \leftarrow K$, set of non-processed jobs;

$\pi^{init} \leftarrow T^2$, set of non-processed type-2 tasks;

Set all machines as free;

Set all machine available times to 0;

Set central server available time to 0;

while π^{init} *is not empty* **do**

  $t^* \leftarrow$ earliest available time amongst machines;

  if *there is a free machine* $m^* \in \{1, ..., M\}$ *with an available time equal to* t^* *AND* $\phi^{init} \neq \emptyset$ **then**

    Identify j^*, the longest job in ϕ^{init};

    Assign job j^* to machine m^*, i.e. add job j^* to ϕ_{m^*} and remove j^* from ϕ^{init};

    Set machine m^* as occupied

  Identify k^*, the job with longest remaining processing time amongst jobs in the making

  Identify machine $\bar{m}$, the machine to which job k^* is assigned;

  Identify i^* the precedence-feasible non-processed type-2 task of k^* that can start the earliest;

  Schedule i^* as early as possible, add task i^* to π and remove i^* from π^{init};

  Schedule the type-1 task that succedes i^* if possible;

  Update the central server available time;

  Update the available time of machine $\bar{m}$;

  if *all tasks of* k^* *are scheduled* **then**

    Set machine $\bar{m}$ as available

Result: Heuristic solution, (ϕ,π)

Appendix C. Improvement heuristic pseudo-codes

Appendix C.1. Job assignment heuristic pseudo-code

Algorithm 5 Job assignment heuristic

Data: Incumbent solution, (ϕ,π)

```
improved ← true;
repeat
    improved ← false;
    m* ← Makespan machine of (φ,π);
    [a, b] ← Longest free idle time amongst non-makespan machines;
    ṁ ← Machine associated with [a, b];
    i* ← task ending at time a on machine ṁ;
    j* ← task starting at time b on machine ṁ;
    Identify k*, the last job scheduled on m*
    Construct φ′ by reassigning k* to machine ṁ at the position matching free idle time [a, b];
    Construct incomplete permutation π′ by removing all tasks of k* from π;
    Construct incomplete permutation π′′ with all tasks of k* from π, sorted in the same relative order;
    repeat
        Construct the partial schedule associated with (φ′,π′);
        Update i*, the last task executed on machine ṁ before task j* in the partial schedule;
        Update free idle time [a, b] occuring between task i* and j* in the partial schedule;
        Remove the first task of π′′ and place it in permutation π′ before the first type-2 task whose starting time is ≥ a ;
    until π′′ is empty;
    if (φ′,π′) yields a better makespan than (φ,π) then
        (φ, π) ← (φ′, π′);
        improved ← true;
until improved = false;
```

Result: Incumbent solution, (ϕ,π)

Appendix C.2. Job ordering heuristic pseudo-code

Algorithm 6 Job ordering heuristic

Data: Incumbent solution, (ϕ,π), I^{max}, parameter

improved $\leftarrow$ true;

repeat

  Identify all machine idle times

  for *each machine* $m \in \{1, ..., M\}$ **do**

    for *the* I^{max} *largest idle times* $[a, b]$ *on machine m in descending order* **do**

      Identify task i^* that ends at time a on machine m;

      Identify job k^* associated with task i^*;

      if i^* *is a type-1 task AND* j^* *is a type-2 task* **then**

        if *there is a job* k^{**} *scheduled before* k^* *on m* **then**

          Construct ϕ' by interverting k^* and k^{**} in ϕ_m;

          Construct π' by interverting tasks of k^* and k^{**};

          if *(*ϕ'*,*π'*) yields a better makespan than (*ϕ*,*π*)* **then**

            $(\phi, \pi') \leftarrow (\phi', \pi')$;

            improved $\leftarrow$ true;

            break;

        if *there is a job* k^{**} *scheduled after* k^* *on m* **then**

          Construct ϕ' by interverting k^* and k^{**} in ϕ_m;

          Construct π' by interverting tasks of k^* and k^{**};

          if *(*ϕ'*,*π'*) yields a better makespan than (*ϕ*,*π*)* **then**

            $(\phi, \pi') \leftarrow (\phi', \pi')$;

            improved $\leftarrow$ true;

            break;

      if *improved* **then**

        break;

until *improved = false;*

Result: Incumbent solution, (ϕ,π)

Appendix C.3. Machine idle times heuristic pseudo-code

Algorithm 7 Machine idle times heuristic

```
Data: Incumbent solution, (φ,π)
improved ← true;
repeat
    improved ← false;
    m* ← Makespan machine of (φ,π);
    Identify all machine idle times [a, b] on m* and rank them by ascending starting times;
    for all machine idle times [a, b] on m* do
        j* ← type-2 task starting at time b;
        i' ← last type-2 task that ends before or at time a;
        u_i, u_j ← indices of i', j* in permutation π;
        n ← u_j − u_i − 1;
        if (n > 0) then
            for m ← n to 1 do
                Construct π' by shifting j* by m units to the left;
                Construct the schedule associated with (φ,π');
                if (φ,π') yields a makespan better or equal to that of (φ,π) then
                    (φ, π) ← (φ, π');
                    improved ← true;
                    break;
        if improved then
            break;
until improved = false;
Result: Incumbent solution, (φ,π)
```

Appendix C.4. Central server idle times heuristic pseudo-code

Algorithm 8 Central server idle times heuristic

Data: Incumbent solution, (ϕ,π)

improved $\leftarrow$ true;

repeat

 improved $\leftarrow$ false;

 Identify all central server idle times $[a, b]$ and rank them by ascending starting times;

 for *all central server idle times* $[a, b]$ **do**

  $i^* \leftarrow$ type-2 task executed right before a;

  $j^* \leftarrow$ type-2 task starting at time b;

  $m^* \leftarrow$, machine on which is executed j^*;

  $j^{**}, \leftarrow$ -1;

  $earliest_{j^{**}} \leftarrow b$;

  for *each machine* m **do**

   if $m \neq m^*$ *and there is a task after* j^* *in permutation that is performed on* m **then**

    $j \leftarrow$ first task after j^* in permutation that is performed on m;

    if *there is an idle time* $[a', b']$ *on* m *before* j *and* $[a, b] \cap [a', b'] \neq \emptyset$ **then**

     if $min(a', a - s_{i^*j^*} + s_{i^*j}) < earliest_{j^{**}}$ **then**

      $j^{**} \leftarrow j$;

      $earliest_{j^{**}} \leftarrow min(a', a - s_{i^*j^*} + s_{i^*j})$;

  if $j^{**} \neq -1$ **then**

   Construct π' by placing j^{**} between i^* and j^* in π;

   if *(ϕ,π') yields a makespan better or equal to that of (ϕ,π)* **then**

    $(\phi, \pi) \leftarrow (\phi, \pi')$;

    improved $\leftarrow$ true;

    break;

until *improved = false;*

Result: Incumbent solution, (ϕ,π)

Highlights

- Extension of the parallel machine scheduling problem
- Industrial applications found in kitchen automation
- Effective general variable neighborhood search algorithm
- Computational results reported on both random and real-world instances
- Analysis of the effects of different instance parameters

Simon Belieres: Conceptualization, Methodology, Software, Writing - original draft.
Yossiri Adulyasak: Conceptualization, Methodology, Writing - original draft.
Jean-François Cordeau: Conceptualization, Methodology, Writing - original draft.

Simon Belieres
Toulouse Business School
20 Bd Lascrosses, 31068 Toulouse, France
+33 6 10 64 03 57
s.belieres@tbs-education.fr

September 22, 2022

To the Editorial Board of the *Computers & Operations Research*,

We would like to submit the manuscript entitled *"Scheduling multi-staged jobs on parallel identical machines and a central server with sequence-dependent setup times: an application to an automated kitchen"* authored by Yossiri Adulyasak, Jean-François Cordeau, and I, Simon Belieres, to be considered for publication as a research article in the *Computers & Operations Research.*

This article introduces a difficult parallel machine scheduling problem that is motivated by a real-life application in a robotic automated kitchen. Specifically, it considers the scheduling of multi-task jobs that are performed on parallel machines, with certain tasks competing for a central resource. In the industrial application, each job corresponds to a recipe to be prepared, and the associated tasks are the steps that must be achieved to complete that recipe. The considered production system configuration consists of (i) a set of multifunctional food processors, i.e. the parallel machines, (ii) a shelf storing the ingredients necessary for preparing the dishes, and (iii) a robotic arm serving as a bridge between the shelf and the multifunctional food processors, i.e. the central resource.

Our contribution is threefold. We first introduce two mathematical formulations for the considered problems, as well as lower bounds. Second, we propose an effective metaheuristic algorithm. On instances with known optimal solutions, the metaheuristic produces solutions with an average gap of 0.31% in very short computation times. On large-size instances, the metaheuristic produces solutions with an average gap of 6.04% when measured against the best-known lower bound. Third, we provide an extensive computational study performed on both random instances and instances provided by our industrial partner. Random instances allow us to assess the effectiveness of our solution approach as well as to analyze the influence that each instance parameter has on the problem difficulty. We also analyze the solutions obtained by our method on a real-life instance and provide insights regarding the considered production system configuration.

We believe this research would be of interest to the readers of *Computers & Operations Research.* From an academic perspective, it introduces a challenging extension of the parallel machine scheduling problem, as well as an efficient solution approach and a detailed analysis of the effects of different instance parameters. The proposed study is also of interest to practitioners as it focuses on an industrial application with great market potential, the restaurant industry being arguably one of the largest industries in the world. While robotic kitchen automation is still a nascent space, the investments of many venture capital firms in kitchen automation startups suggest that this field will develop rapidly.

This work has not been published earlier and it is not under consideration for publication elsewhere. Each named author has substantially contributed to conducting the underlying

research and drafting this manuscript. The proposed research meets all applicable standards with regard to the ethics of experimentation and research integrity, and the following is being certified/declared true. None of the authors has financial or personal relationships with other people or organizations that could inappropriately influence or bias the content of the paper.

We thank you for your consideration of this manuscript.

S. Belieres, Y. Adulyasak, J.-F. Cordeau