Parallel Computing for the Non-permutation Flow Shop Scheduling Problem with Time Couplings Using Floyd-Warshall Algorithm

Wojciech Bożejko, Jarosław Rudy, and Radosław Idzikowski

Abstract In this chapter a variant of the classic Non-permutation Flow Shop Scheduling Problem is considered. Time couplings for operations are introduced, determining the minimal and maximal allowed machine idle time between processing of subsequent jobs. The mathematical model of the problem and a graph representation of its solution are presented. Next, several properties of the problem, including a method for computation of the goal function on a CREW PRAM model of parallel computation, are formulated and proven. Finally, the proposed method is discussed in terms of its theoretical effects on the time needed to calculate the goal function and search one of the well-known neighborhoods for use in local search solving methods.

Keywords Flow Shop · Time couplings · Parallel computing · Discrete optimization · Scheduling

1 Introduction

The Flow Shop Scheduling Problem (FSSP), and its more specific permutation variant [21], is one of the most well-known scheduling problems in the field of discrete optimization and operations research. It has many practical applications, being able

W. Bożejko · R. Idzikowski (✉)
Department of Control Systems and Mechatronics, Faculty of Electronics, Wrocław University of Science and Technology, 27 Wybrzeże Wyspiańskiego St., 50-372 Wrocław, Poland
e-mail: radoslaw.idzikowski@pwr.edu.pl

W. Bożejko
e-mail: wojciech.bozejko@pwr.edu.pl

J. Rudy
Department of Computer Engineering, Faculty of Electronics, Wrocław University of Science and Technology, 27 Wybrzeże Wyspiańskiego St., 50-372 Wrocław, Poland
e-mail: jaroslaw.rudy@pwr.edu.pl

G. Bocewicz et al. (eds.), *Performance Evaluation Models for Distributed Service Networks*, Studies in Systems, Decision and Control 343,
https://doi.org/10.1007/978-3-030-67063-4_1

to model various real-life production processes, starting from classic assembly line manufacturing [27] to construction projects management [6].

Due to its popularity, practical applications and difficulty (FSSP is considered NP-hard optimization problem), the FSSP and many of its variants (e.g. cyclic [8], multi-objective [18, 25], stochastic [11]), had been a topic of very active research by various groups of scientists. Moreover, various modeling and solving methods have been considered for this problem, ranging from integer programming [28] and metaheuristic method [26] to parallel computation [5] and fuzzy sets theory [22].

However, sometimes existing variants of the FSSP are sometimes not sufficient to model real-life situations and unusual constraints, such as machines with specific working conditions or machine/vehicle operating/renting cost. One example of this is the concreting process. In this case one of the operations consists of pouring concrete on a designated location. The concrete mixer truck needs to load and mix the next portion of the mixture before the task can proceed to the next location. Moreover, the concrete needs to be mixed for a certain amount of time. Too short or too long mixing can result in incorrect concrete parameters or even damage the concrete mixer. Mixing process is thus and additional task with minimal and maximal duration taking place between other tasks. Such a restriction can be generally described as time coupling: an additional relation between the completion time of an operation on a machine and the starting time of the next operation on the same machine. In result, the final schedule will contain gaps required for those additional tasks (like concrete mixing) with their allowed duration given by a closed from-to interval.

The minimal idle time for a machine can be modeled as setup times [23], however this does not model the maximal idle time. Other types of time couplings often encountered in the literature are the no-idle constraint [19] (machine has to start processing the next job immediately after completion of the previous one) and no-wait [12] (next operation of a job has to be processed immediately after completion of the previous one). Other constraints considered for similar scheduling problems that resemble time couplings or its specific cases include limited-wait [10, 24] (maximal inter-operation time in a job) and inserted-idle [14] (deliberate various-sized inter-machine idle periods). Naturally, it is also possible to use more than one such restriction together, for example combining no-wait and limited-idle constraints together [13]. Approaches exist that consider machine-fixed time couplings. And example of that is paper by Bożejko et al., where Branch-and-bound and Tabu search solving methods were proposed for the Permutation Flow Shop Problem [7]. Further results from the area of complex scheduling problems can be found in the works of Bach et al. [1], Bocewicz [2], and Bocewicz et al. [3, 4] as well as Pempera and Smutnicki [20].

In recent years the there have been a considerable development in parallel computing . Example include such technologies as NVidia CUDA (GPU devices with up to several thousand parallel cores), Xeon Phi vector processor (up to several dozens parallel processors) or massively parallel distributed systems, grids and cluster. Thus, parallel computation has become a common way to improve running time and effectiveness of many discrete optimization algorithms, including solving methods for Flow Shop and similar scheduling problems.

For example, Luo and Baz [15] have employed a two level parallel Genetic Algorithm using a hybrid GPU-CPU system to solve large instances of Flexible Flow Shop Scheduling Problem. Results indicated that the approach remains competitive at reduced computation time. Similarly, Luo et al. [16] proposed genetic algorithm for solving an dynamic version of the Flexible Flow Shop Problem with emphasis on energy efficiency. The method, designed for consistency with NVidia CUDA greatly reduces computation time while providing competitive results. Steinhöfel et al. [26] proposed a parallel Simulated Annealing algorithm for the Job Shop Scheduling Problem with makespan criterion. The authors employed a method for computing the goal function by finding the longest path using n^3 parallel processors. Moreover, the authors showed that bound on the value of the goal function can be used to further reduce computation time, by bounding the number of edges on the longest path in the graph. Finally, Flow Shop Scheduling was also used as a benchmark for testing general parallel computation methods. For example, Melab et al. have used this problem to test the effectiveness of their Branch-n-Bound method [17]. Two energy-consuming equivalent parallel systems were used: a MIC architecture with Intel Xeon Phi 5110P coprocessor and GPU system with NVidia Tesla K40. Results of experiments indicated that GPU approach outperforms the MIC coprocessor.

In this paper we aim to model Non-permutation Flow Shop Scheduling Problem with minimal-maximal time coupling and makespan criterion. We will formulate several problem properties aiding us in efficient calculation of the goal function. Finally, we will propose a method of parallel computation of the goal function using a modification of the Floyd-Warshall algorithm.

The remainder of the paper is structured as follows. In Sects. 2 and 3 we formalize the problem, presenting its mathematical model and solution graph. In Sect. 4 we formulate several problem properties with regards to solution feasibility, solution graph and two methods of the goal function computation: sequential and parallel. In Sect. 5 we discuss the possible application and speedup of the parallel method. Finally, Sect. 6 contains the conclusions.

2 Problem Formulation

In this section we will formulate the mathematical model of the FSSP-TC problem, including notation, problem constraints and the goal function. All values are positive integers unless otherwise specified.

The problem can be described as follows. Let $\mathscr{J} = \{1, 2, \ldots, n\}$ and $\mathscr{M} = \{1, 2, \ldots, m\}$ be sets of n *jobs* and m *machines* respectively. For each job j by $\mathscr{O}_j$ we denote the set of m *operations* of that job:

$$\mathscr{O}_j = \{l_j + 1, l_j + 2, \ldots, l_j + m\}, \tag{1}$$

where $l_j = m(j-1)$ is the total number of operations in all jobs prior to j. Thus, there are nm operations in total with $\mathscr{O} = \{1, 2, \ldots, nm\}$ being the set of all operations. Sets $\mathscr{O}_j$ are thus a partition of $\mathscr{O}$.

Table 1 Example instance for the FSSP-TC with $n = 4$ and $m = 3$

i	$p_{i,1}$	$p_{i,2}$	$p_{i,3}$	$p_{i,4}$	$\hat{r}_i$	$\hat{d}_i$
1	2	2	5	2	3	4
2	1	2	4	2	1	2
3	1	8	3	2	2	3

Each operation of job j has to be processed on a different, specific machine. The order of visiting machines for each job is the same and given as:

$$1 \to 2 \to 3 \to \ldots \to m-1 \to m. \tag{2}$$

Thus, operations from the set $\mathscr{O}_j$ are processed in the order given by sequence:

$$(l_j + 1, l_j + 2, \ldots, l_j + m). \tag{3}$$

Job j on machine i (i.e. operation $l_j + i$) has to be processed for time $p_{i,j}$ without interruption. Several additional constraint exist. First, machine can process at most one operation and at most one operation of a job can be processed at any given time (operations do not overlap). Second, time at which an operation is processed is tied to the time at which previous operation on that machine is processed. This constraint is called a time coupling. In this specific case, let $\hat{r}_i \geq 0$ and $\hat{d}_i \geq \hat{r}_i$ denote the minimal and maximal time machine i has to wait before processing the next operation. This means the wait time before processing next operation is in interval $[\hat{r}_i, \hat{d}_i]$. In particular if $\hat{d}_i = 0$ this constraint is reduced to classic no-idle constraint known from the literature. Values of $p_{i,j}$, $\hat{r}_i$ and $\hat{d}_i$ for an exemplary FSSP-TC instance with $n = 4$, $m = 3$ are shown in Table 1.

The task is to determine the order of processing jobs for each machine. Let π_i be an n-element sequence (permutation) describing the order of processing of jobs on machine i, such that $\pi_i(j)$ is the job that will be processed as j-th on i. Then the order of processing of jobs is given by an m-element tuple $\pi = (\pi_1, \pi_1, \ldots, \pi_m)$. Also, let $I_i(j)$ denote on what position j appears in π_i:

$$\pi_i(j) = k \iff I_i(k) = j. \tag{4}$$

In other words, job j is processed on machine i as $I_i(j)$-th job.

The processing order π is used to determine the processing schedule for all operation, which is given by a matrix of operation starting times S of size $m \times n$:

$$S = [S_{i,j}]^{m \times n}, \tag{5}$$

where element $S_{i,j}$ is the starting time of j-th operation to be processed on machine i according to the processing order π. Similarly, we can define matrix C of operation completion times:

$$C = [C_{i,j}]^{m \times n}, \tag{6}$$

where $C_{i,j}$ is the completion time of j-th operation to be processed on i in π.

The schedule given by S is feasible if it satisfied the following conditions:

$$S_{i,j} \geq C_{i,j-1} + \hat{r}_i, \tag{7}$$

$$S_{i,j} \leq C_{i,j-1} + \hat{d}_i, \tag{8}$$

$$S_{i,j} \geq C_{i-1,I_{i-1}(\pi_i(j))}, \tag{9}$$

$$C_{i,j} = S_{i,j} + \rho_{i,j}, \tag{10}$$

with starting conditions $S_{i,0} = S_{0,j} = 0$ and $\rho_{i,j}$ being a shorthand notation for $p_{i,\pi_i(j)}$. Inequalities (7)–(8) ensure that operations on a given machine do not overlap, are processed in the order given by π_i and obey the time coupling constraints. Inequality (9) guarantees that operations in a job do not overlap and are processed in the order specified by formulas (2)–(3). Finally, Eq. (10) ensures that each operation is processed by the required time without interruption. The precise method of determining schedule S from processing order π and the feasibility of schedules will be discussed further in the paper.

For the instance shown in Table 1 and example processing order:

$$\pi = \Big((1,2,3,4), (2,1,3,4), (2,3,4,1)\Big), \tag{11}$$

the possible schedule S is as follows:

$$S = \begin{bmatrix} 0 & 5 & 10 & 18 \\ 8 & 12 & 15 & 20 \\ 10 & 20 & 25 & 29 \end{bmatrix}. \tag{12}$$

The resulting schedule is shown as a Gantt chart in Fig. 1.

Let $C_{\max}(\pi)$ denote, for a given processing order π, the maximum of completion times of all operations (the makespan):

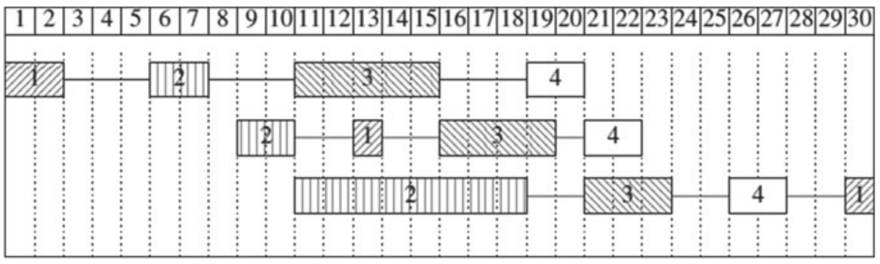


Fig. 1 Gantt chart for the exemplary schedule and problem instance

$$C_{\max}(\pi) = \max_{j \in \mathscr{J}, i \in \mathscr{M}} C_{i,j}. \tag{13}$$

Due to constraints (7)–(10), it can be shown that this formula simplifies to:

$$C_{\max}(\pi) = C_{m,n}. \tag{14}$$

To solve the FSSP-TC problem, one needs to find the processing order π^* which minimizes the makespan $C_{\max}(\pi)$:

$$C_{\max}(\pi^*) = \min_{\pi \in \Pi} C_{\max}(\pi), \tag{15}$$

where Π is the set of all feasible processing orders and π^* is called the optimal processing order. The makespan for the exemplary instance from Table 1, processing order (11) and schedule (12) is equal to 30.

The FSSP-TC problem with the makespan criterion defined in this section will be denoted as $F|\hat{r}_i, \hat{d}_i|C_{\max}$ in the Graham notation for theoretical scheduling problems.

3 Graph Representation

In this section we will introduce and discuss the graph model used to represent solutions and constraints for the FSSP-TC problem. The graph structure is based on a similar graph model for the classic Flow Shop Scheduling Problem and will be used to prove several problem properties later in the paper.

Let us consider an arbitrary processing order π, for which we will now construct a weighted directed graph $G(\pi) = (V, E)$, with V being the set of nm vertices and E being the set of $3nm - 2m - n$ directed edges (arcs). The general exemplary structure of this solution graph for some instance is shown in Fig. 2.

Let us start with the vertices, which can be thought of as if placed on a 2-dimensional grid. In result the vertices can be denoted using a pair of coordinates (i, j). Thus, vertex (i, j) is placed in the j-th "column" and i-th "row" of $G(\pi)$ and it represents the j-th job to be processed on machine i. It means that jobs in i-th row are ordered according to π_i. All vertices are weighted with the weight of vertex (i, j) being equal to $\rho_{i,j}$.

The arcs of the graph $G(\pi)$ represent the problem constraints and can be divided into three disjoint subsets:

- horizontal "forward" arcs from vertices (i, j) to $(i, j + 1)$ with weight $\hat{r}_i$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n - 1$. There are $(n - 1)m$ such arcs.
- horizontal "reverse" arcs from vertices $(i, j + 1)$ to (i, j) with weight $\hat{D}_{i,j}$:

$$\hat{D}_{i,j} = -\rho_{i,j} - \rho_{i,j+1} - \hat{d}_i. \tag{16}$$

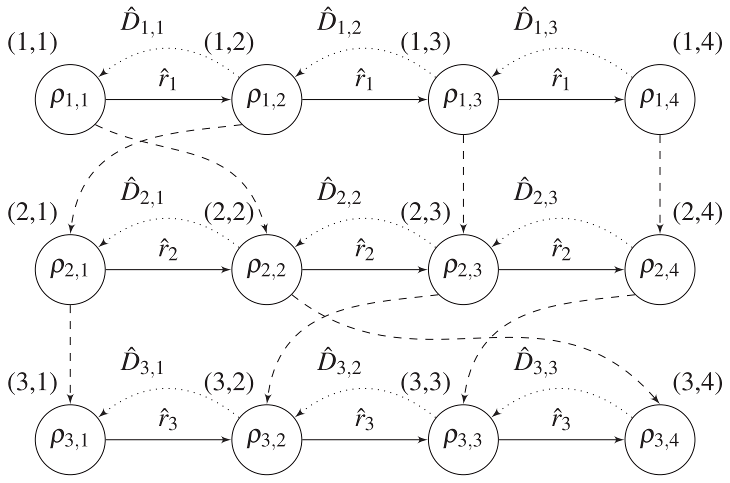


Fig. 2 Exemplary structure of the graph $G(\pi)$

for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n-1$. There are $(n-1)m$ such arcs.
- weightless arcs from vertices $(i, I_i(j))$ to $(i+1, I_{i+1}(j))$ for $i = 1, 2, \dots, m-1$ and $j = 1, 2, \dots, n$. There are $n(m-1)$ such arcs.

As with graph for similar scheduling problems, the length of the longest path (defined as the sum of arc and vertex weights along the path, including the starting and ending vertex) ending at vertex (i, j) is equal to $C_{i,j}$. If we do not include the weight of the final vertex (i, j), then we get $S_{i,j}$ instead. Moreover, the length of the longest (critical) path in $G(\pi)$ is equal to value $C_{\max}(\pi)$.

Weights $\rho_{i,j}$ represent operation times. From constraints (9)–(10) we have:

$$C_{i+1,j} - C_{i,j} \geq \rho_{i,j+1}. \tag{17}$$

Due to that the "vertical" arcs have weight 0 (since value $\rho_{i,j+1}$ is added once we have entered vertex $(i, j+1)$) and are thus effectively weightless. Next, from constraints (7) and (10) it follows that:

$$C_{i,j+1} - C_{i,j} \geq \hat{r}_i + \rho_{i,j+1}, \tag{18}$$

and thus forward arcs have weight $\hat{r}_i$. The most interesting is constraint (8). Due to this constraint it follows that:

$$C_{i,j} - C_{i,j+1} \leq -\rho_{i,j+1} - \hat{d}_i. \tag{19}$$

Thus, it seems the weight of the "reverse" arc should be equal to $-\rho_{i,j+1} - \hat{d}_i$. However, the arc will end at vertex (i, j) and its weight $\rho_{i,j}$ will be automatically included in the length of the path and yield incorrect value. In order to mitigate it, we include additional term $-\rho_{i,j}$ and the final arc weight is:

$$-\rho_{i,j} - \rho_{i,j+1} - \hat{d}_i, \tag{20}$$

which matches the value $\hat{D}_{i,j}$ described before.

Let us notice that change of the processing order π can affect the order (permutation) of vertices in every row. This will affect the weights of vertices and which vertices weightless arcs connect to. It will also change weights $\hat{D}_{i,j}$ of reverse arcs. Forward arcs and their values are independent of π. Finally, let us notice that graph $G(\pi)$ contains cycles. Cycles with positive length (understood as the sum of weights of vertices and arcs belonging to the cycle) are not allowed as the resulting graph would not have a longest path at all. We will look into this issue in the next section.

4 Problem Properties

In this section we will formulate and proof several properties and theorems for the FSSP-TC problem, including cycles in graph $G(\pi)$, feasibility of schedules and methods of sequential and parallel computation of the goal function.

We will start with the issue of existence of the longest path in graph $G(\pi)$. Unlike in the classic Flow Shop Scheduling Problem, where there are no cycles in the solution graph, there are clearly cycles in graph $G(\pi)$ for the FSSP-TC problem. Such cycles are allowed as long as there are no cycles with positive length. It is indeed true as stated by the following property.

Property 1.1 *Let π be a processing order for problem $F|\hat{r}_i, \hat{d}_i|C_{\max}$. Then the solution graph $G(\pi)$ for that problem does not contain a cycle with positive length.*

Proof $G(\pi)$ does not contain arcs going upward, thus a cycle is always restricted to a single row. Cycles must contain at least two vertices. Let us consider two cases: (1) cycles with two vertices, and (2) cycles with three or more vertices.

A cycle with two vertices contains vertices (i, j) and $(i, j+1)$ (for $j < n$) as well as two arcs between those vertices. The weights in that cycle are $\rho_{i,j}$, $\rho_{i,j+1}$, $\hat{r}_i$ and $\hat{D}_{i,j}$. It is easy to see that the length of such cycle is equal $\hat{r}_j - \hat{d}_i \leq 0$.

Let us now consider cycles with $k > 2$ subsequent vertices from (i, j) to $(i, j + k - 1)$. This cycle also contains $k - 1$ forward arcs of weight $\hat{r}_j$ and $k - 1$ reverse arcs. Let us now calculate the length of sych cycle.

Without the loss of generality we will start the cycle from vertex $(i, j + k - 1)$. Let us consider "inner" vertices of the cycle i.e. vertices $(i, j + 1)$ through $(i, j + k - 2)$. Let (i, c) denote such inner vertex. When walking the cycle we visit this vertex twice, adding the weight $2\rho_{i,c}$ to the cycle. However, we also add weight $-\rho_{i,c}$ twice: once from the arc leading to (i, c) and once from the arc leaving (i, c). Thus, those values negate each other. In result, it is equivalent to a situation where inner vertices have weight 0 and the reverse arcs have only the $\hat{d}_i$ term, except for the arc going from

$(i, j + k - 1)$ and the arc going into (i, j). Those two arcs retain the terms $-\rho_{i,j+k-1}$ and $-\rho_{i,j}$, respectively. However, those two weights will be negated by the weights of the "border" vertices (i, j) and $(i, j + k - 1)$ (we visit those vertices only once).

To summarize this, we can compute the length of the cycle as if all vertices belonging to the cycle had weight 0 and the reverse arcs had only term $\hat{d}_i$. The length of such a cycle is thus:

$$(k - 1)(\hat{r}_i - \hat{d}_i) \leq 0. \tag{21}$$

■

Therefore, we know all cycles have non-positive length and thus the longest path indeed exists in $G(\pi)$. However, remaining cycles are still a potential issue when computing the goal function. Fortunately, we can disregard all remaining cycles for the purpose of the calculation of the goal function due to the following property.

Property 1.2 *Let π be a processing order for problem $F|\hat{r}_i, \hat{d}_i|C_{\max}$ and* $\text{len}(c)$ *be the length of path c (sum of vertex and arc weights on that path) in graph $G(\pi)$. Then if there exists path p in $G(\pi)$ with length* $\text{len}(p)$ *such that p contains a cycle, then there exists path P in $G(\pi)$ that does not contain a cycle with length* $\text{len}(P) \geq \text{len}(p)$.

Proof Let us first consider that p has a single cycle. Such a path can be decomposed into three paths: p_1 (before the cycle), p_c (the cycle) and p_2 (after the cycle), where p_1 or p_2 can be empty. Then we can build P from joining paths p_1 and p_2, ignoring the cycle. From the Property 1.1 we have $\text{len}(p_c) \leq 0$ and thus $\text{len}(P) \geq \text{len}(p)$.

The single-cycle case can be easily generalized for paths with multiple cycles by applying that logic multiple times, removing a single cycle with each step. ■

Thus, if a path ending at vertex (m, n) contains a cycle, then either it is not a critical path (if at least one of its cycles has negative length) or we can find another path with the same length, but with all cycles removed.

Next, we will present an algorithm for computing the value of the makespan sequentially (using a single processor) and prove its computational complexity.

Theorem 1.1 *Let π be a processing order for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem. The schedule S and makespan $C_(max)(\pi)$ for π can be determined in time $O(nm)$.*

Proof The algorithm is divided into m phases, one for each machine. After phase i the starting and completion times for all machines up to i are determined, meaning determining values $S_{l,j}$ and $C_{l,j}$ for $j = 1, 2, \ldots, n$ and $l = 1, 2, \ldots, i$. Also, every time a given value $S_{i,j}$ is updated, the corresponding value $C_{i,j}$ is updated as according to constraint (10) (that constraint is thus always satisfied).

The first phase is started by setting $S_{1,1} = 0$. Next, we iterate over remaining jobs in the order specified by π_1 and setting:

$$S_{1,j} = C_{1,j-1} + \hat{r}_1, \quad j = 2, 3, \ldots, n. \tag{22}$$

After this constraint (7) for machine 1 is satisfied. Since it is the first machine and $\hat{d}_1 \geq \hat{r}_1$, then all other constraints are also met and the phase is complete. It is easy to see this phase is done in time $O(n)$.

Every subsequent phase $i > 1$ is divided into two subphases. We start the first subphase by setting $S_{i,1} = C_{i-1,1}$. Next, we iterate over the remaining jobs (according to the order specified in π_i) and setting:

$$S_{i,j} = \max\{C_{i,j-1} + \hat{r}_i, C_{i-1,j}\}, \quad j = 2, 3, \ldots, n. \tag{23}$$

After this, the first subphase is complete in time $O(n)$.

It is easy to see that after this is done, all constraint for machine i are satisfied, except for constraint (8). If two jobs on machine i are separated by more than $\hat{d}_i$ then we have to move them closer to each other. However, we cannot move the later job to be scheduled earlier, as doing so would violate one or both of the constraints (7) and (9). Instead, we will move the earlier job to be processed later, so the separation between them will be exactly $\hat{d}_i$. This is always possible as moving a job to be processed later does not violate any constraints ($\hat{d}_i \geq \hat{r}_i$).

The second subphase is thus aimed at correcting any possible violation of constraint (8). However, moving a job to be processed later widens the gap separating it from the previous job and make it violate its own constraint (8) (if it was not already violated). This would make it possible to shift the same job up to $n - 1$ times in the worst-case. Fortunately, this can be fixed by iterating over the jobs in the reverse order compared to the first subphase. We thus set:

$$C_{i,j-1} = \max\{S_{i,j} - \hat{d}_i, C_{i,j-1}\}, \quad j = n, n-1, \ldots, 2. \tag{24}$$

After this the second subphase is finished in time $O(n)$ with all constraints met.

In total, after m phases are complete, all values $S_{i,j}$ and $S_{i,j}$ (including value $C_{\max}$ are determined. The algorithm completes in time $O(nm)$.

The above theorem also leads to the following immediate conclusions.

Corollary 1.1 *For any processing order π for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem there exists at least one feasible schedule S (and thus all processing orders are feasible).*

Proof The algorithm from Theorem 1.1 is applicable for any processing order π as the only thing that changes with the change of π are the actual values $\rho_{i,j}$ (as we remember, $\rho_{i,j}$ is a shorthand for $p_{i,\pi_i(j)}$) that are used to update $C_{i,j}$ based on the current value of $S_{i,j}$. The algorithm thus produces a feasible schedule S for any π.

Corollary 1.2 *Le π be a processing order for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem. The schedule S obtained for π through the algorithm from Theorem 1.1 is left-shifted.*

Proof This follows because operation starting times are always set to the earliest time that does not violate one or more problem properties. Thus, it is impossible to schedule any operation to be started earlier without changing the processing order π and S is thus left-shifted. ■

For our last result in this section, we will propose a parallel algorithm for computing the goal function for the FSSP-TC problem which can be applied for the Concurrent Read, Exclusive Write Parallel Random Access Machine (CREW PRAM) model of parallel computation. We start by reminding the commonly known fact about the time complexity of computing the minimum of a sequence.

Fact 1.1 *The minimum value of an n-element sequence can be determined on a CREW PRAM machine in time $O(\log n)$ using $O(n/\log n)$ processors.*

Proof In phase one, we divide the sequence into $O(n/\log n)$ blocks (subsequences) of length $O(\log n)$. Each processor computes the minimum of each block in a sequential manner in time $O(\log n)$. Thus, we obtain $O(n/\log n)$ values. In the second phase, we have to compute the minimum of them. Since we have $O(n/\log n)$ processors, this can be done in time $O(\log n)$. Thus, both phases in total take time $O(\log n)$ using $O(n/\log n)$ processors.

With this and the graph $G(\pi)$ we can formulate the following theorem for time complexity of a parallel algorithm for determining of $C_{\max}$ for the FSSP-TC problem.

Theorem 1.2 *For a fixed processing order $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$ the value of $C_{\max}(\pi)$ for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem can be determined in time $O(\log^2(nm))$ using a CREW PRAM computation model with $O\left(\frac{(nm)^3}{\log(nm)}\right)$ processors.*

Proof In order to compute $C_{\max}(\pi)$ we will compute the length of the longest path in the graph $G(\pi)$. The proposed parallel method of computing the longest path is based on the sequential Floyd-Warshall algorithm for finding the shortest paths in graphs [9]. The algorithm allows for negative edge weights and cycles as long as there is no negative cycle (i.e. a cycle with negative sum of its edge weights).

Finding the longest path in $G(\pi) = (V, E)$ is equivalent to finding the shortest path in graph $G'(\pi) = (V, E')$. Graph $G'(\pi)$ is exactly like $G(\pi)$, except its edge weights are negated:

$$\forall (i, j) \in E : \quad (i, j) \in E' \land \psi'(i, j) = -\psi(i, j), \tag{25}$$

where $\psi(u, v)$ and $\psi'(u, v)$ are the weights of edge (u, v) in graphs $G(\pi)$ and $G'(\pi)$ respectively. From Property 1.1 we know that the graph $G(\pi)$ for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem has no positive cycles, meaning that $G'(\pi)$ has no negative cycles. Thus, the Floyd-Warshall algorithm is viable in this case.

To make the notation more clear, we will transform the graph $G'(\pi)$ into $G^*(\pi)$ by changing the vertex numbering. In result, vertices of $G^*(\pi)$ will be indexed using single number u instead of a pair (i, j) as for $G'(\pi)$. The numbering translation from (i, j) to u is as follows:

$$u = (i - 1)n + j. \tag{26}$$

Thus, vertices are numbered starting from the top left, numbering all vertices in a row before proceeding to the next row. The reverse translation is as follows:

$$i = \left\lfloor \frac{u-1}{n} \right\rfloor + 1, \quad j = u - \left\lfloor \frac{u-1}{n} \right\rfloor n. \tag{27}$$

In result we obtain graph $G^*(\pi) = (W, E^*)$, with $W = \{1, 2, \dots, nm\}$ being the set of nm vertices. The weight of vertex $u \in W$ is denoted ρ_u and is equal to weight of the corresponding vertex from $G'(\pi)$, i.e. $\rho_u = -\rho_{i,j}$. The set of edges E^* can be partitioned into sets E^0, E^r and E^d representing vertical (technological), forward horizontal and reverse horizontal edges from graph $G'(\pi)$ respectively:

$$E^0 = \bigcup_{u=1}^{nm-n} \{(u, u+n)\}, \tag{28}$$

$$E^r = \bigcup_{k=1}^{m} \bigcup_{u=(k-1)n}^{kn-1} \{(u, u+1)\}, \tag{29}$$

$$E^d = \bigcup_{k=1}^{m} \bigcup_{u=(k-1)n}^{kn-1} \{(u+1, u)\}. \tag{30}$$

Weight $\psi(u, v)$ of edge (u, v) in graph $G^*(\pi)$ is given as follows:

$$\psi(u, v) = \begin{cases} 0 & \text{if } (u, v) \in E^0, \\ -\hat{r}_i & \text{if } (u, v) \in E^r, \\ -\hat{D}_{i,j} & \text{if } (u, v) \in E^d. \end{cases} \tag{31}$$

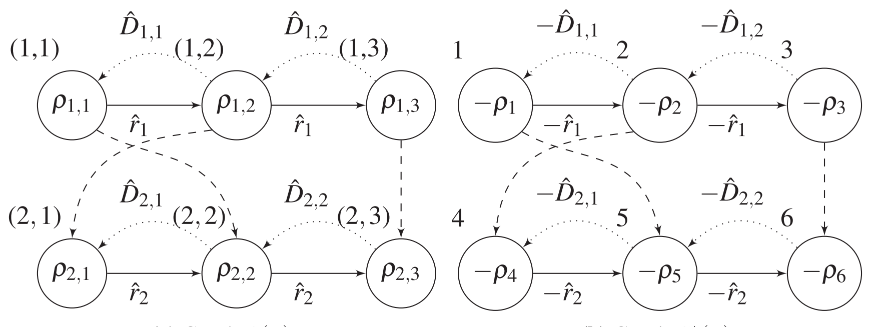
where i and j can be translated from u using formula (27). The example of a simple graph $G(\pi)$ and corresponding graph $G^*(\pi)$ are shown on Fig. 3a, b.

With graph $G^*(\pi)$ defined, we will now introduce matrix $A = [a_{u,v}]$ of size $nm \times nm$, where $a_{u,v}$ will represent the length of longest path between vertices u and v in graph $G^*(\pi)$. Values of A should be initialized as follows:

$$a_{u,v} = \begin{cases} 0 & \text{if } u = v, \\ \psi(u, v) - \rho_u & \text{if } u \neq v \wedge (u, v) \in E^*, \\ \infty & \text{if } u \neq v \wedge (u, v) \notin E^*. \end{cases} \tag{32}$$

The reason for including value ρ_u is that the Floyd-Warshall algorithm recognized weighted edges, but not weighted vertices. Thus, the weight ρ_u is added to every edge that starts at vertex u. The initial contents of matrix A for the graph from Fig. 3b is shown in Fig. 3c.

Matrix A will be used to compute the shortest path in $G^*(\pi)$ which, after negating it, will be the longest path in the original graph $G(\pi)$. Each of the $(nm)^2$ initial values of A is calculated independently from the others. Thus, initialization of A can be done in time $O(1)$ on a CREW PRAM computation model using $O((nm)^2)$ processors, each performing a single assignment.



(a) Graph $G(\pi)$ **(b)** Graph $G^*(\pi)$

$$\begin{bmatrix} 0 & -\hat{r}_1 - \rho_2 & \infty & \infty & -\rho_5 & \infty \\ -\hat{D}_{1,1} - \rho_1 & 0 & -\hat{r}_1 - \rho_3 & -\rho_4 & \infty & \infty \\ \infty & -\hat{D}_{1,2} - \rho_2 & 0 & \infty & \infty & -\rho_6 \\ \infty & \infty & \infty & 0 & -\hat{r}_2 - \rho_5 & \infty \\ \infty & \infty & \infty & -\hat{D}_{2,1} - \rho_4 & 0 & -\hat{r}_2 - \rho_6 \\ \infty & \infty & \infty & \infty & -\hat{D}_{2,2} - \rho_5 & 0 \end{bmatrix}$$

(c) Matrix A

Fig. 3 Transformation of the original solution graph $G(\pi)$ into derived graph $G^*(\pi)$ and initial values of matrix A for some instance with $n = 3$ and $m = 2$

Furthermore, we define a 3-dimension array $T = [t_{u,w,v}]$ of size $nm \times nm \times nm$, which will be used to compute the transitive closure of the path lengths in $G^*(\pi)$. In other words, value $t_{u,w,v}$ will be used to store and update the length of the shortest path from vertex u to vertex v that goes through vertex w.

The core part of the algorithm is based on repeating the following steps in a loop:

1. Update $t_{u,w,v}$ for all triples (u, w, v) as per the following formula:

$$t_{u,w,v} = a_{u,w} + a_{w,v}, \tag{33}$$

2. Update $a_{u,v}$ for all pairs (u, v) based on the following formula:

$$a_{u,v} = \min\{a_{u,v}, \min_{1 \le w \le nm} t_{u,w,v}\}. \tag{34}$$

Our task is to determine the value $a_{1,nm}$ (length of path from vertex 1 to vertex nm). In order to do this, it is sufficient to run the above two steps $\lceil \log(nm - 1) \rceil$ times. This is because in each step the algorithm finds out the shortest path between vertices placed further from each other. After the first iteration the algorithm will compute the shortest paths that consist of a single edge. After the second iteration, the algorithm will compute the shortest paths composed of up to two edges. After the third iteration it will compute shortest paths that are composed of up to 4 edges and so on. Therefore, after the k-th iteration of the loop the algorithm will compute the shortest paths that are composed of up to 2^k edges.

The longest possible path (in the sense of the number of edges it is composed of) in graph $G^*(\pi)$ will go from left to right through the entire first machine, then go back from right to left through the entire second machine, then from left to right on the third machine and so on. This path will thus move in a zigzag pattern, going through all vertices. Such a path will have no more than $nm - 1$ edges. Thus, the algorithm will compute the longest path after $\lceil \log(nm - 1) \rceil$ iterations.

Next, we will discuss how the two steps indicated by formulas (33) and (34) can be done in parallel. The first step could be executed in time $O(1)$ on CREW PRAM if we assigned $O((nm)^3)$ processors to it, each doing a single assignment for a single triple (u, w, v). However, since we have only $\lceil (nm)^3 / \log(nm) \rceil$ processors, each processor will have to process not a single (u, w, v) triple, but $\lceil \log(nm) \rceil$ such triples. The time complexity of this step will thus be $O(log(nm))$.

The goal of step 2 is to calculate the minimum of $nm + 1$ values, which can be done (according to Fact 1.1) on a CREW PRAM in time $O(log(nm))$ using $O(nm/\log(nm))$ processors. Such minimum has to be computed for $(nm)^2$ different (u, v) pairs and computation for each pair is independent from the others. Thus this step can be done in time $O(log(nm))$ using $(nm)^2 O(nm/\log(nm)) = O((nm)^3/\log(nm))$ processors in total.

As already mentioned, the steps 1 and 2 of updating matrices A and T have to be repeated $\lceil \log(nm - 1) \rceil$ times in a loop, thus the total time complexity of this loop is:

$$\lceil \log(nm - 1) \rceil O(log(nm)) = O(log^2(nm)), \tag{35}$$

using $(nm)^2 O(nm/\log(nm)) = O((nm)^3/\log(nm))$ processors.

Finally, let us notice that weight ρ_u of vertex u is only used in formula (32), if u has outgoing edges. This is true for all vertices except for nm, which has no outgoing edges. Thus, value ρ_{nm} is not taken into account by the algorithm and the value $a_{1,nm}$ is not true value of $C_{\max}$. However, we know that vertex nm is always included in the critical path and since it has no outgoing edges then it is visited only once and as the last edge. Thus, we can compute the final value of $C_{\max}(\pi)$ in time $O(1)$ as follows:

$$C_{\max}(\pi) = -a_{1,nm} + \rho_{nm}. \tag{36}$$

Thus, the final time complexity of the entire algorithm (including initialization of A, the loop and the final corrections) is $O(log^2(nm))$ using $O(\frac{(nm)^3}{\log(nm)})$ processors. ■

To summarize the results presented in this section, we have shown that the presence of cycles in solution graph $G(\pi)$ is not a problem, showed a sequential and parallel algorithms for computing $C_{\max}(\pi)$ in time $O(nm)$ and $O(\log^2(nm))$ respectively and that schedules obtained are feasible and left-shifted.

5 Discussion

In this section we will discuss the possible results of applying the proposed parallel computation method in solving algorithms for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem.

The first point of interest is the speedup S we can achieve by computing the single value of $C_{\max}$ in parallel. This speedup is defines as:

$$S = \frac{T_s}{T_p}, \tag{37}$$

where T_s is the time required to compute $C_{\max}$ with a traditional (i.e. sequential) method using a single processor and T_p is the time required to compute $C_{\max}$ with the proposed parallel method using $\frac{(nm)^3}{\log(nm)}$ processors. The theoretical speedup values for several problem sizes commonly considered in the literature are shown in Table 2. We see that the proposed parallel computation method can provide considerable speedup (up to 10 for the considered problem sizes).

The computation of a single $C_{\max}$ value at a time is a part of nearly every solving algorithm for the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem. However, there is a group of methods, called local search methods, that is based on searching the entire neighborhood of a given solution to find the best solution. For the $F|\hat{r}_i, \hat{d}_i|C_{\max}$ problem one of such neighborhood is the so-called Adjacent Pair Interchange (API) neighborhood. On each machine there are $n - 1$ possible adjacent job pairs and there are m machines, thus the API neighborhood contains $(n - 1)m$ solutions. Thus, it is possible to further shorten the computation by computing every solution from the neighborhood in parallel. We can define speedup S' for this similar to the previous one as follows:

$$S' = \frac{T'_s}{T'_p}, \tag{38}$$

Table 2 Theoretical speedups for the proposed parallel method compared to sequential approach

$n \times m$	T_s	T_p	S	T'_s	T'_p	$S(p)'$
10×5	50	36	1.39	2250	40	56.25
20×10	200	64	3.13	38000	69	550.72
30×15	450	81	5.56	195750	86	2276.16
40×20	800	100	8.00	624000	106	5886.79
50×20	1000	100	10.00	980000	106	9245.28

Table key:
$n \times m$ – problem size,
T_s – time of computing $C_{\max}$ of a single solution using sequential algorithm,
T_p – time of computing $C_{\max}$ of a single solution using parallel algorithm,
S – single solution speedup (T_s/T_p),
T'_s – time of API neighborhood search using sequential algorithm,
T'_p – time of API neighborhood search using parallel algorithm,
S' – neighborhood search speedup (T'_s/T'_p)

where T'_s is time of searching all $(n-1)m$ solutions one after another using a single processor and T'_p is the time of searching the neighborhood fully in parallel using $\frac{(nm)^3}{\log(nm)}$ processors for each neighborhood solution. The values of S' are also shown in Table 2 as well. We see that the theoretical obtainable speedup is very high (up to several thousands for common problem sizes). This is mostly due to the neighborhood size, but is further enhanced by the proposed parallel method.

While offering considerable to very high speedup, the proposed method requires a very high number of processors. Even for the 10×5 the method requires 25 000 parallel processors to compute the value of $C_{\max}$ and this number only increases with the growth of the problem size. Thus, the proposed method is not yet viable to be employed for real-life scheduling problems at the current state of parallel computing technologies, which is why we focus on the theoretical approach in this paper.

However, while the proposed method remains mostly theoretical at the moment it is still possible to apply it using massively parallel distributed systems. Similarly, experimental research of the proposed method can be carried out using such systems. Example of such parallel computation environments:

- GPU devices using Nvidia CUDA technology. For example Tesla K40c and RTX 2080Ti offer 2880 and 4352 CUDA parallel cores respectively. It should be noted that it is possible to install several GPU devices on a single machine. This, should allow to up to 20 000 parallel cores. However, GPU cores are usually much slower than CPU cores.
- Multicore and Manycore CPUs. While many CPU are limited to around 20 cores, there exist devices like Xeon Phi x200 coprocessor (64 cores) and ADM Ryzen Threadripper 3990X processor (128 logical cores).
- Computer and supercomputer clusters. Such environments consist of hundreds of multicore computer nodes. For example, Wrocław Center for Networking and Supercomputing (referred to as WCSS) provides access to over 22 000 cores through over 900 24/28-core computing nodes with total computation power of 860 TFLOPS with 76 TB of RAM (from 64 to 512 GB) per node. Fast inter-node connection through InfiniBand.

It is also important to consider the software required. Most of the mentioned parallel environments can be access with C/C++ programming language, with the help of the CUDA, OpenMp/MPI libraries, depending on the parallel method used. Some of the environments are also supported by newer programming languages like python, but this is not guaranteed for all environments. Aside from that, cluster and supercomputer centers often need to accessed using specific interfaces, in which case the software required is heavily also dependent on the particular cluster used.

To summarize, it is possible to apply the proposed method, though care must be taken concerning inter-node communication, which could become a bottleneck. Moreover, due to Brant's law it is also possible to apply the method in system with lower number of cores, but the method will run slower, as described by that law.

It should be also noted that the proposed method has several limitations. The ones already mentioned that stems from real-life applications are the hardware necessary to run the method and inter-node bottlenecks. Moreover, the method has low efficiency (measured as speedup divided by the number of processors employed). For example, for $n = 20$ and $m = 10$ the efficiency is only 0.0006.

6 Conclusions

In this paper we have considered a variant of the well-known Non-permutation Flow Shop Scheduling Problem with makespan criterion. The additional constraint, called time couplings, limits the minimal and maximal idle time for each machine. We have presented a mathematical model of the problem and a graph model of the problem solution. Next, we proved several properties of the problem concerning feasibility of solutions and time required to compute the value of the makespan. We have also proved that solution graph, despite having cycles, does not contain cycles with positive length, making it possible to compute the longest path in that graph.

The main results is a proposed method of computation of the makespan on a CREW PRAM computation model using a modification of the Floyd-Warshall algorithm for finding shortest paths in graphs. This method, coupled with large size of solution neighborhood for the considered problem, allows for very high speedup (in thousands) of the process of searching the solution neighborhood. However, the method also faces several limitations due to massive number of processors it requires, low efficiency and inter-node communication bottleneck.

It should also be noted, that the proposed method could be generalized to be applicable for other types of scheduling problems. Other variants of the Flow Shop Scheduling Problem (Permutation and Non-permutation, setup and transport times or more general time couplings) would be the easiest. While not as easy, the method could also be generalized to problems like Job Shop Scheduling Problem and Open Shop Scheduling Problem or even other related discrete permutation-oriented optimization problem like Traveling Salesman Problem or Vehicle Routing Problem.

The next step of research on this topic could include: (1) physical implementation of the proposed method and experimental research, (2) implementation of the full algorithm encompassing more than just goal function computation, and (3) consideration of different neighborhoods than Adjacent Pair Interchange.

References

1. Bach, I., Bocewicz, G., Banaszak, Z.A., Muszyński, W.: Knowledge based and cp-driven approach applied to multi product small-size production flow. Control Cybern. **39**(1), 69–95 (2010)
2. Bocewicz, G.: Robustness of multimodal transportation networks. Maint. Reliab. **16**, 259–269 (2014)
3. Bocewicz, G., Muszyński, W., Banaszak, Z.: Models of multimodal networks and transport processes. Bull. Pol. Acad. Sci. Tech. Sci. **63**(3), 635–650 (2015)

4. Bocewicz, G., Nielsen, P., Banaszak, Z., Thibbotuwawa, A.: Routing and scheduling of unmanned aerial vehicles subject to cyclic production flow constraints. Adv. Intell. Syst. Comput. **801**, 75–86 (2019)
5. Bożejko, W., Gnatowski, A., Pempera, J., Wodecki, M.: Parallel tabu search for the cyclic job shop scheduling problem. Comput. Ind. Eng. **113**, 512–524 (2017)
6. Bożejko, W., Hejducki, Z., Wodecki, M.: Flowshop scheduling of construction processes with uncertain parameters. Arch. Civ. Mech. Eng. **19**(1), 194–204 (2019)
7. Bożejko, W., Idzikowski, R., Wodecki, M.: Flow Shop Problem with Machine Time Couplings, vol. 987. Springer, Cham (2020)
8. Bożejko, W., Smutnicki, C., Uchroński, M., Wodecki, M.: Cyclic Two Machine Flow Shop with Disjoint Sequence-Dependent Setups, pp. 31–47. Springer International Publishing, Cham (2020)
9. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education (2001)
10. Gicquel, C., Hege, L., Minoux, M., van Canneyt, W.: A discrete time exact solution approach for a complex hybrid flow-shop scheduling problem with limited-wait constraints. Comput. Oper. Res. **39**(3), 629–636 (2012)
11. Gonzàilez-Neira, E., Montoya-Torres, J.: A simheuristic for bi-objective stochastic permutation flow shop scheduling problem. J. Proj. Manag. **4**, 57–80 (2017)
12. Grabowski, J., Pempera, J.: Some local search algorithms for no-wait flow-shop problem with makespan criterion. Comput. Oper. Res. **32**(8), 2197–2212 (2005)
13. Harbaoui, H., Khalfallah, S., Bellenguez-Morineau, O.: A case study of a hybrid flow shop with no-wait and limited idle time to minimize material waste. In: 2017 IEEE 15th International Symposium on Intelligent Systems and Informatics (SISY), pp. 207–212 (2017)
14. Kanet, J.J., Sridharan, V.: Scheduling with inserted idle time: problem taxonomy and literature review. Oper. Res. **48**(1), 99–110 (2000)
15. Luo, J., El Baz, D.: A dual heterogeneous island genetic algorithm for solving large size flexible flow shop scheduling problems on hybrid multicore CPU and GPU platforms. Math. Probl. Eng. **1–13**(03), 2019 (2019)
16. Luo, J., Fujimura, S., Baz, D.E., Plazolles, B.: GPU based parallel genetic algorithm for solving an energy efficient dynamic flexible flow shop scheduling problem. J. Parallel Distrib. Comput. **133**, 244–257 (2019)
17. Melab, N., Gmys, J., Mezmaz, M., Tuyttens, D.: Many-Core Branch-and-Bound for GPU Accelerators and MIC Coprocessors, pp. 275–291. Springer International Publishing, Cham (2020)
18. Mishra, A.K., Shrivastava, D., Bundela, B., Sircar, S.: An efficient Jaya algorithm for multi-objective permutation flow shop scheduling problem. In: Venkata Rao, R., Taler, J. (eds.) Advanced Engineering Optimization Through Intelligent Techniques, pp. 113–125. Springer Singapore, Singapore (2020)
19. Pan, Q.K., Ruiz, R.: An effective iterated greedy algorithm for the mixed no-idle permutation flowshop scheduling problem. Omega (U. K.) **44**, 41–50 (2014)
20. Pempera, J., Smutnicki, C.: Open shop cyclic scheduling. Eur. J. Oper. Res. **269**(2), 773–781 (2018)
21. Potts, C.N., Shmoys, D.B., Williamson, D.P.: Permutation vs. non-permutation flow shop schedules. Oper. Res. Lett. **10**(5), 281–284 (1991)
22. Rudy, J.: Cyclic scheduling line with uncertain data. In: Lecture Notes in Computer Science, pp. 311–320 (2016)
23. Ruiz, R., Stützle, T.: An Iterated Greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. Eur. J. Oper. Res. **187**(3), 1143–1159 (2008)
24. Santosa, B., Siswanto, N., Fiqihesa: Discrete particle swarm optimization to solve multi-objective limited-wait hybrid flow shop scheduling problem. IOP Conf. Ser. Mater. Sci. Eng. **337**, 012006 (2018)

25. Smutnicki, C., Pempera, J., Rudy, J., Żelazny, D.: A new approach for multi-criteria scheduling. Comput. Ind. Eng. **90**, 212–220 (2015)
26. Steinhöfel, K., Albrecht, A., Wong, C.-K.: Fast parallel heuristics for the job shop scheduling problem. Comput. Oper. Res. **29**, 151–169 (2002)
27. Wang, P.-S., Yang, T., Chang, M.-C.: Effective layout designs for the Shojinka control problem for a TFT-LCD module assembly line. J. Manuf. Syst. **44**, 255–269 (2017)
28. Ünal, A.T., Ağral, S., Taşn, Z.C.: A strong integer programming formulation for hybrid flowshop scheduling. In: Journal of the Operational Research Society, pp. 1–11 (2019)