Parallel Algorithm with Blocks for a Single-Machine Total Weighted Tardiness Scheduling Problem

Mariusz Uchorński

Department of Control Systems and Mechatronics, Wroclaw University of Science and Technology, 50-370 Wroclaw, Poland; mariusz.uchronski@pwr.edu.pl

Abstract: In this paper, the weighted tardiness single-machine scheduling problem is considered. To solve it an approximate (tabu search) algorithm, which works by improving the current solution by searching the neighborhood, is used. Methods of eliminating bad solutions from the neighborhood (the so-called block elimination properties) were also presented and implemented in the algorithm. Blocks allow a significant shortening of the process of searching the neighborhood generated by insert type moves. The designed parallel tabu search algorithm was implemented using the MPI (Message Passing Interface) library. The obtained speedups are very large (over 60,000×) and superlinear. This may be a sign that the parallel algorithm is superior to the sequential one as the sequential algorithm is not able to effectively search the solution space for the problem under consideration. Only the introduction of diversification process through parallelization can provide an adequate coverage of the entire search process. The current methods of parallelization of metaheuristics give a speedup which strongly depends on the problem’s instances, rarely greater than number of used parallel processors. The method proposed here allows the obtaining of huge speedup values (over 60,000×), but only when so-called blocks are used. The above-mentioned speedup values can be obtained on high performance computing infrastructures such as clusters with the use of MPI library.

Keywords: parallel optimization; scheduling; Message Passing Interface (MPI)

1. Introduction

Problems of scheduling tasks on a single machine with cost goal functions, despite the simplicity of formulation, mostly belong to the class of the most difficult (NP-hard) discrete optimization problems. In the literature many types of such problems differing in task parameters, functional properties of machines and criteria are considered. Starting from the simplest ones with minimizing the number of late tasks (problem denoted by 1||∑U_i) to complex ones with machine setups and time windows, where machine downtime may occur. Their optimization boils down to determining tasks starting times (or their order) that minimize the sum of penalties (costs of performing tasks). Optimal algorithms solve, within a reasonable time, examples with several tasks not exceeding 50 (80 in a multiprocessor environment, see [1]), therefore in practice there are almost exclusively approximate algorithms used. In the author’s opinion, the best of them are based on local search methods (as opposed to randomized methods, which will potentially give a different result each time you run, sometimes very good and sometimes poor).

In the problem considered in this work there is a set of tasks that must be performed on one machine. Each task has its requested execution time, deadline and weight of the objective function for tardiness. One should determine the order of performing tasks, minimizing the sum of delay costs. It is undoubtedly one of the most studied problems of scheduling theory. It belongs to the class of strongly NP-hard problems. First work on this subject, Rinnoy Kan et al. [2] was published in the 1970s. Despite the passage of over 40 years this problem is still attractive to many researchers. Therefore, its various variants are considered in the papers published in recent years: Cordone and Hosteins [3].
Rostami et al. [4], Poongothai et al. [5], Gafarow and Werner [6] polynomial algorithms for special cases) Ertem et al. [7]. A comprehensive review of the literature on scheduling problems with critical lines is presented in the following papers: Chenga, et al. [8], Adamu and Adewumi [9]. Single-machine scheduling problems with random execution times or desired completion dates are also considered in the literature (Rajba and Wodecki [10], Bożejko et al. [11,12]). In turn, the parallel algorithms for the single-machine scheduling problems are considered: parallel dynasearch algorithm [13], parallel population training algorithm [14], parallel path relinking for the problem version with setups [15]. The variant of single-machine scheduling is also studied in the port logistics Iris et al. [16–18].

The paper presents new properties of the problem under consideration, which will be used in local search algorithms. There was a neighborhood proposed based on insert moves and procedures for splitting permutations (solutions) into blocks (subpermutations). Their use leads to a significant reduction in neighborhood size. Elimination of “bad” solutions performed in such a way, considerably accelerates the calculations.

A taxonomy of parallel tabu search algorithms was proposed by Voß [19] referring to the classic classification of Flynn [20] characterizing parallel architectures (SIMD, MIMD, MISD and SISD models). Voß’s classification is independent from the classic search procedures taxonomy (single-walk and multiple-walk, see Alba [21]), differentiating the algorithms into:

- **SPSS (Single (Initial) Point Single Strategy)**—allowing for parallelism only at the lowest level, such as objective functions calculation or parallel neighborhood search,
- **SPDS (Single (Initial) Point Different Strategies)**—all processes start with the same initial solution, but they use different search strategies (e.g., different lengths of tabu list, different items stored in the tabu list, etc.)
- **MPSS (Multiple (Initial) Point Single Strategy)**—processors begin operation from different initial solutions, using the same search strategy,
- **MPDS (Multiple (Initial) Point Different Strategies)**—the widest class, embracing all previous categories as its special cases.

Here we propose the use of a parallel algorithm, tabu search multiple-walk version based on the MPSS model (Multiple starting Points Single Strategy). The MPI (Message Passing Interface) library is used for communication between computing processes run on several processors cores which execute its own tabu search processes.

**Contributions**

To sum up, the paper presents a new method of generating sub-neighborhoods based on the elimination properties of the block in a solution. Searching for sub-neighborhoods in the parallel tabu search algorithm gives a significant acceleration of calculations without losing the quality of the generated solutions. Compared to paper [1], where blocks have been introduced, here we consider the division into blocks of the entire permutation, and not only its middle fragment (for unfixed tasks, as in the work [1]). Moreover, the properties of blocks are used to eliminate solutions from the neighborhood.

**2. Formulation of the Problem**

In the formulation part and throughout the paper, we use the following notations:

- \( n \) - number of tasks,
- \( \mathcal{J} \) - set of tasks,
- \( p_i \) - time of task execution,
- \( w_i \) - task’s tardiness cost factor,
- \( d_i \) - requested task completion time,
- \( \pi \) - permutation of tasks,
- \( \Phi \) - set of all permutation of elements from \( \mathcal{J} \),
\[ N(\pi) \] - neighborhood of solution \( \pi \),
\[ S_i \] - task starting time \( i \in J \),
\[ C_i \] - task completion time \( i \in J \),
\[ T_i \] - task tardiness \( i \),
\[ \pi^T \] - semi block of early tasks,
\[ \pi^D \] - semi block of tardy tasks,
\[ B \] - partition of permutation into blocks,
\[ F(\approx) \] - sum of tardiness costs (criterion).

**Total Weighted Tardiness Problem**, (in short TWT) can be formulated as follows:

**TWT problem**: Each task from the set \( J = \{1, 2, \ldots, n\} \) is to be executed on one machine. Wherein the following restrictions must be met:

(a) all jobs are available at time zero,
(b) the machine can process at most one job at a time,
(c) preemption of the jobs is not allowed,
(d) associated with each job \( j \in J \) there is

(i) processing time \( p_j \),
(ii) due date \( d_j \),
(iii) positive weight \( w_j \).

The order in which tasks will be performed must be determined, which minimizes the sum of the cost of delays. As in [22], we denote the problem as \( 1|| \sum w_i T_i \).

Any solution to the considered problem (the order in which the tasks are performed on the machine) can be represented by the permutation of tasks (elements of the set \( J \)). By \( \Phi \) we denote the set of all such permutations.

Let \( \pi \in \Phi \) be some permutation of tasks. For task \( \pi(i) \) \( (i = 1, 2, \ldots, n) \), let:

- \( C_{\pi(i)} \) be the completion time of task execution,
- \( T_i = \max \{0, C_i - d_i\} \) - tardiness,
- \( f_i(C_i) = w_i \cdot T_i \) - tardiness cost.

In the considered problem there should be determined the order in which the tasks will be performed (permutation \( \pi \in \Phi \)) by machine, minimizing sum of tardiness costs, i.e., sum

\[ F(\pi) = \sum_{i=1}^{n} f_{\pi(i)}(C_{\pi(i)}) = \sum_{i=1}^{n} w_{\pi(i)} T_{\pi(i)}. \] (1)

The problem of minimizing the cost of tardiness is \( NP \)-hard (Lawler [23] and Lenstra et al. [22]). There were many papers published devoted to researching this problem. Emmons [24] introduced a partial order relationship on a set of tasks, thus limiting the process of searching the optimal solutions to some subset of the set of solutions. These properties are used in the best metaheuristic algorithms. Optimal algorithms (based on the dynamic programming method or branch and bound) were published by: Rinnoy Kan et al. [25], Potts i Van Wassenhove [26] oraz Wodecki [1]. Some of them were presented in the review of Abdul-Razaq et al. [27]. They are, however not very useful in solving most examples found in practice, because the calculation time increases exponentially relatively to the number of tasks. Hence, in a reasonable time one can solve only examples with the number of tasks not exceeding 50 tasks (80 with the use of parallel algorithm [1]). There is extensive literature devoted to algorithms determining approximate solutions within acceptable time. Methods, on which the constructions of these algorithms are based, can be divided into the class of construction and correction style method.

Construction algorithms usually have low computational complexity. However, designated solutions may differ significantly (even by several hundred percent) from optimal ones. Most commonly used construction algorithms used in solving TWT problems are presented in the works of Fischer [28], Morton and Pentico [29] and in Potts' and Van Wassenhove review [30].
In the correction algorithms we start with a solution (or a set of solutions) and we try to improve them by local search. Obtained in this way solution is the starting point in the next iteration of the algorithm. The most known implementations of the correction method to solving TWT problem are metaheuristics: tabu searches (Crauwels et al. [31]), simulated annealing (Potts and Van Wassenhove [30], Matsuo et al. [32]), genetic algorithm (Crauwels et al. [31]), the ant algorithm (Den Basten et al. [33,34]). A very interesting and effective implementation was also presented in the work of Congram et al. [35] and then it was developed by Grosso et al. [36]. Its main advantage is the use of neighborhood browsing procedure with an exponential number of elements in polynomial time.

3. Definitions and Properties of the Problem

For permutation \( \pi \in \Phi \) \( C_{\pi(i)} = \sum_{j=1}^{n} p_{\pi(j)} \) is completion time of execution of task \( \pi(i) \) \((i = 1, 2, \ldots, n)\) in permutation \( \pi \). The task \( \pi(i) \) is early, if its completion time is not greater than the requested completion time (i.e., \( C_{\pi(i)} \leq d_{\pi(i)} \)) and late, if this time is greater than the requested completion time i.e., \( C_{\pi(i)} > d_{\pi(i)} \).

Hence, Total Weighted Tardiness (TWT) Problem consists of determining the optimal permutation \( \pi^* \in \Phi \), which minimizes the value of the criterion function \( F \) on the set \( \Phi \), such for which

\[
F(\pi^*) = \min \{ F(\pi) : \pi \in \Phi \}.
\]  

First, we introduce certain methods of aggregating tasks for generating blocks. In any permutation \( \pi \in \Phi \), there are subpermutations (subsequences of following tasks) for which:

1. execution of each task from subpermutation ends before its desired completion time (all tasks are early), or
2. execution of each task from subpermutation ends after its desired deadline (all tasks are tardy).

In the further part we present two types of blocks: early tasks blocks and tardy tasks ones. They will be used to eliminate worse solutions.

3.1. Blocks of Tasks

This section briefly introduces the definitions and properties of blocks and algorithms for their determination. They were described in detail in the work by Wodecki [1].

Blocks of early tasks
Subpermutation of tasks \( \pi^T \) in permutation \( \pi \in \Phi \) is \( T \)-block, if:

(a) each task \( j \in \pi^T \) is early and \( d_j \geq C_{last} \), where \( C_{last} \) is the completion time of the last task in \( \pi^T \),

(b) \( \pi^T \) is maximum subpermutation (in sense of number of elements) satisfying the restriction (a).

It is easy to see that if \( \pi^T \) is \( T \)-block, then the inequality \( \min \{ d_j : j \in \pi^T \} \geq C_{last} \) is satisfied. Therefore, in any permutation of elements from \( \pi^T \) every task in permutation \( \pi \) is early. Using this property, we present the algorithm determining the first \( T \)-block in permutation \( \pi \).

Input of Algorithm 1 is permutation \( \pi \), and output some \( T \)-block of this permutation. In line 1 first due date job is determined. Next in lines 4–7, it is checked whether adding another due date task to the block will keep the following property: in any permutation of these tasks, all tasks are on time. The computational complexity of the algorithm is \( O(n) \).
Theorem 1 ([1]). For any permutation $\pi \in \Phi$ there is a partition of $\pi$ into subpermutations such, in which each of them is:

(i) $T$-block or

(ii) $D$-block.

The algorithm for splitting permutation $\pi \in \Phi$ into blocks has computational complexity $O(n)$.

Property 1. From block definition and from Theorem 1:

1. Each task belongs to a certain $T$ or $D$-block,
2. Different blocks contain various elements,
3. Two T or D blocks can appear directly next to each other,
4. A block can contain only one task.
5. The partition of permutations into blocks is not explicit. According to the block definitions and the Theorem 1:
   \[ d_{π(i)} < S_{\text{first}} + p_{π(i)}. \]

Therefore, the cost function of performing this task
\[ f_{π(i)}(x) = w_{π(i)}(x - d_{π(i)}), \text{ for } x \geq S_{\text{first}} + p_{π(i)}, \]
is a linear function. It follows from Smith’s theorem [37] that tasks in \( π^D \) occur in optimal order if and only if
\[ \frac{w_{π(i-1)}}{p_{π(i-1)}} \geq \frac{w_{π(i)}}{p_{π(i)}}, \quad i = a + 1, a + 2, \ldots, b \] (3)
where subpermutation \( π^D = (π(a), π(a + 1), \ldots, π(b)), 1 \leq a < b \leq n. \)

Permutation \( π \in Φ \) is ordered (in short D-OPT) due to the partition into blocks, if in each D-block each pair of neighboring tasks meets the relation (3), hence they appear in the optimal order.

**Theorem 2** ([1]). A change in the order of tasks in any block of permutation D-OPT does not generate permutations with a smaller value of criterion function.

From the above statement follows the so-called block elimination property. It will be used while generating neighborhoods.

**Corollary 1.** For any D-OPT permutation \( π \in Φ \), if \( β \in Φ \) and
\[ F(β) < F(π), \] (4)
then in permutation \( β \) at least one task of some block from partition \( π \) was swapped before the first or the last task of this block.

Therefore, by generating from D-OPT permutation \( π \in Φ \) new solutions to the TWT problem we will only swap element of the block before the first or after the last element of this block.

**4. Moves and Neighborhoods**

The essential element found in approximate algorithms solving NP-hard optimization problems based on the local search method is neighborhood—mapping:
\[ N : Φ_n \mapsto 2^Φ, \]
attributing to every element \( π \in Φ \) a certain subset of \( N(π) \) set of acceptable solutions \( Φ \), \( N(π) \subseteq Φ. \)

The number of elements of the neighborhood, the method of their determination and browsing has a decisive impact on efficiency (calculation time and criterion values) of the algorithm based on the local search method. Classic neighborhoods are generated by transformations commonly known as moves, i.e., “minor” changes of certain permutation elements consisting of:
1. Swapping positions of two elements in permutation—swap move \( s^k_l \), changes the position of two elements \( π(k) \) and \( π(l) \) (found respectively in \( k \) and \( l \) in \( π \) positions), generating permutation \( s^k_l(π) = π^k_l \). In short, it will be called s-move. The computational complexity of s-move execution is \( O(1) \).
2. Moving the element in the permutation to a different position—insert move $i_k^l$, swaps element $\pi(k)$ (from position $k$ in $\pi$) to position $l$, generating permutation $i_k^l(\pi) = \pi_k^l$. The insert type move will be abbreviated to $i$-move. Its computational complexity is $O(1)$.

One way to determine the permutation neighborhood is to define the set of moves that generate them. If $\mathcal{M}(\pi)$ is a certain set of moves specified for permutation $\pi \in \Phi$, then

$$\mathcal{N}(\pi) = \{ m(\pi) : m \in \mathcal{M}(\pi) \}$$

is neighborhood $\pi$ generated by moves from $\mathcal{M}(\pi)$.

In each iteration of an algorithm based on a local search method using the neighborhood (moves) generator, a subset of a set of solutions - neighborhood is determined. Let

$$\mathcal{B} = [B^1, B^2, \ldots, B^n],$$

be a partition of ordered ($D$-OPT) permutation $\pi$ into blocks.

We consider the task $\pi(j)$ belonging to a certain block from the division $\mathcal{B}$. Moves that can bring improvement of the criterion values consist of swapping the task $\pi(j)$ (before) the first or (after) the last task of this block. Let $\mathcal{M}_{bf}^j$ and $\mathcal{M}_{af}^j$ be sets of these moves (i.e., respectively all such $i$-moves and $s$-moves). These sets are symbolically shown in Figure 1. The task $\pi(a^k)$ is the first whereas a $\pi(b^k)$ is the last element of $B^k$ block, which also includes the considered task $\pi(j)$.

Let

$$\mathcal{M}(\pi) = \bigcup_{j=1}^{n} \mathcal{M}_{bf}^j \cup \bigcup_{j=1}^{n} \mathcal{M}_{af}^j,$$

be a set of all moves that can bring improvement (see Corollary 1), i.e., moves before or after blocks of some $\pi$ permutation.

Some properties of $i$-moves and $s$-moves were proven, which can be used to determine subneighborhood. These are both elimination criteria and procedures for determining sets of moves and their representatives.
4.1. Properties of Insert Moves

The move $m^* \in M(\pi)$ is a representative of a set of moves from a certain set of moves $\mathcal{W} \subset M(\pi)$, if
\[
\forall r \in \mathcal{W}, \quad F(r(\pi)) \geq F(m^*(\pi)).
\]

Let us assume that permutation $\pi \in \Pi$ is $\mathcal{D}$-OPT, and the neighborhood is generated by insert type of (i-moves). If $i_l^k$ is a representative of the moves of the set $M^k \subset M(\pi)$ ($i_l^k \notin M^k$), then moves belonging to $M^k$ are removed from the set $M_{bf}^k \cap M_{af}^k$, i.e., they are modified as follows:
\[
M_{bf}^k \leftarrow M_{bf}^k \setminus M^k \quad \text{and} \quad M_{af}^k \leftarrow M_{af}^k \setminus M^k.
\]

This procedure makes it possible, in the process of generating the neighborhood, to omit the elements that do not directly improve the value of the criterion.

**Theorem 3.** If task $\pi(k)$ after swapping to the position $l$ (after the move $i_l^k$), $1 \leq k < l \leq n$, in permutation $\pi$ is early (i.e., $C(\pi(l)) \leq d(\pi(k))$), then for a pair of moves $i_{l-1}^k$, $i_l^k$ there is
\[
F(\pi_{l-1}^k) \geq F(\pi_l^k).
\]

**Proof.** Let us assume that the task $\pi(k)$ after swapping to position $l$ (i.e., executing the move $i_l^k$) in permutation $\pi$ is early. We consider two moves: $i_l^k$ and $i_{l-1}^k$. In permutations $\pi_l^k$ and $\pi_{l-1}^k$ generated by these moves there occurs:
\[
\pi_l^k(j) = \pi_{l-1}^k(j) \quad \text{for } j = 1, 2, \ldots, l-2, l+1, \ldots, n,
\]
\[
\pi_{l-1}^k(l-1) = \pi_l^k(l) = \pi(k), \quad \pi_l^k(l-1) = \pi_l^k(l) = \pi(l).
\]

We present both permutations.

**Position** → ... $k-1$, $k$, ... $l-2$, $l-1$, $l$, $l+1$...

\[
\pi_{l-1}^k = (\ldots \pi(k-1), \pi(k+1) \ldots \pi(l-1), \pi(k), \pi(l), \pi(l+1) \ldots)
\]
\[
\pi_l^k = (\ldots \pi(k-1), \pi(k+1) \ldots \pi(l-1), \pi(l), \pi(k+1), \pi(l+1) \ldots)
\]

Since tasks $\pi_{l-1}^k(l-1), \pi_l^k(l)$ are assumed to be early and task completion time $\pi_{l-1}^k(l)$, $C_{\pi_{l-1}^k(l)} = C_{\pi_l^k(l)} + p(\pi(k))$, hence $F(\pi_{l-1}^k) \geq F(\pi_l^k)$. \( \square \)

**Theorem 4.** Let $\pi(a)$ be the first and $\pi(b)$ the last element of some $\mathcal{I}$-block of permutation $\pi$. If for task $\pi(k)$ ($0 < k < a$) the requested completion time $d(\pi(k)) < C_{\pi(a)}$, then
\[
F(\pi_l^k) \leq F(\pi_{l+1}^k), \quad l = a - 1, a, a + 1, \ldots, b - 1, b.
\]

**Proof.** For task $\pi(k)$, $0 < k < a$, we consider two moves $i_l^k$ and $i_{l+1}^k$, $l = a - 1, a, \ldots, b$. Generated by these moves permutations $\pi_l^k$ and $\pi_{l+1}^k$ there is: $\pi_l^k(j) = \pi_{l+1}^k(j)$ for $j = 1, 2, \ldots, l-1, l+2, \ldots, n$ and $\pi_l^k(l+1) = \pi_{l+1}^k(l+1)$. Since $\pi_l^k(l) = \pi_{l+1}^k(l+1) = \pi(k)$, therefore the completion time of the task $\pi_{l+1}^k(l+1)$, $C_{\pi_{l+1}^k(l+1)} = C_{\pi_l^k(l)} + p(\pi(l+1))$, hence $F(\pi_l^k) \leq F(\pi_{l+1}^k)$. \( \square \)

For task $\pi(k)$ ($1 \leq k \leq n$), let
\[
\tau(k) = \begin{cases} 0, & \max\{j : 1 \leq j \leq n \land C_{\pi_l^k(j)} \leq d_{\pi(k)}\}, \\ \text{if } d_{\pi(k)} < p(\pi(k)), & \text{otherwise.} \end{cases}
\]
Thus, this is the last position in the permutation π, on which the task π(k) is swapped, after execution of i-move, is early. Therefore, the task π(k) is early in each permutation π^k_l, l = 1, 2, . . . , τ(k).

**Corollary 2.** For a sequence of moves \( i^k_1, i^k_2, \ldots, i^k_{\tau(k)} \) of task π(k) there:
\[
\mathcal{F}(\pi^k_1) \geq \mathcal{F}(\pi^k_2) \geq \ldots \geq \mathcal{F}(\pi^k_{\tau(k)}),
\]
where parameter τ(k) is designated by (6).

**Proof.** The move \( i^k_l \), \( l = 1, 2, \ldots, \tau(k) \) generates permutation \( \pi^k_l \), in which the task π(k) is early. Using the Theorem 3, it is easy to show that \( \mathcal{F}(\pi^k_1) \geq \mathcal{F}(\pi^k_2) \geq \ldots \geq \mathcal{F}(\pi^k_{\tau(k)}) \). □

Therefore, the move \( i^k_l \) is a representative of a set \( \{i^k_1, i^k_2, \ldots, i^k_{\tau(k)-1}\} \), hence these moves can be omitted by modifying accordingly the sets:
\[
\mathcal{M}^k_{bf} \leftarrow \mathcal{M}^k_{bf} \setminus \{i^k_1, i^k_2, \ldots, i^k_{\tau(k)-1}\},
\]
\[
\mathcal{M}^k_{af} \leftarrow \mathcal{M}^k_{af} \setminus \{i^k_1, i^k_2, \ldots, i^k_{\tau(k)-1}\}.
\]

**Corollary 3.** Let π(a) i π(b) be the first and the last task T-bloku in permutation π. If \( 1 \leq k < a \) and \( a \leq \tau(k) \leq b \), where parameter τ(k) is defined in (6), then for the sequence of moves \( i^k_{\tau(k)+1}, i^k_{\tau(k)+2}, \ldots, i^k_b \) there is:
\[
\mathcal{F}(\pi^k_{\tau(k)+1}) \leq \mathcal{F}(\pi^k_{\tau(k)+2}) \leq \ldots \leq \mathcal{F}(\pi^k_{b-1}) \leq \mathcal{F}(\pi^k_b).
\]

**Proof.** The move \( i^k_l \), \( l = \tau(k) + 1, \tau(k) + 2, \ldots, b \) generates permutation \( \pi^k_l \), in which the task π(k) is tardy. Using the Theorem 4, it is easy to show that
\[
\mathcal{F}(\pi^k_{\tau(k)+1}) \leq \mathcal{F}(\pi^k_{\tau(k)+2}) \leq \ldots \leq \mathcal{F}(\pi^k_{b-1}) \leq \mathcal{F}(\pi^k_b).
\]

The move \( i^k_{\tau(k)+1} \) is a representative of a set \( \{i^k_{\tau(k)+1}, i^k_{\tau(k)+2}, \ldots, i^k_{b-1}, r^k_b\} \). It is then possible to assume
\[
\mathcal{M}^k_{bf} \leftarrow \mathcal{M}^k_{bf} \setminus \{i^k_{\tau(k)+2}, i^k_{\tau(k)+3}, \ldots, r^k_b\}.
\]
Computational complexity of algorithms checking for each proven property 2–3 is \( O(n) \).

4.2. **Properties of Swap Moves**

To eliminate some s-moves from neighborhoods there will be blocks and elimination criteria used.

There is a partial relation \( \rightarrow \) introduced on the set of tasks \( \mathcal{J} \). Let \( \Gamma^{-}(i) \) and \( \Gamma^{+}(i) \) be respectively, the sets of predecessors and successors of tasks \( i \in \mathcal{J} \) in relation \( \rightarrow \). Properties enabling to determine the elements of the relationship \( \rightarrow \) are called in the literature elimination criteria.

**Theorem 5.** If one of the conditions is met:

(a) \( p_r \leq p_j, \ w_r \geq w_j, \ d_r \leq d_j, \ lub \)
(b) \( p_r \leq p_j, \ w_r \geq w_j, \ d_r \leq \sum_{i \in \Gamma^{-}_j} p_i + p_j, \ lub \)
(c) \( w_r \geq w_j, \ d_r \leq d_j, \ d_r \geq \sum_{i \in \mathbb{N}, \Gamma^{+}_j} p_i + p_j, \)

then there is an optimal solution in which the task \( r \) precedes \( j \), i.e., \( r \rightarrow j \).
Proof. Condition (a) is proved by Shwimer [38]. Conditions (b) and (c) are a generalized version of theorem 1 and 2 from Emmons’ work [24]. □

After establishing a new relationship between the task \( i \) and \( j \) to avoid a cycle, there should be a transitive closure of relations given, i.e., one should modify the sets accordingly:

\[
\forall l \in \{i\} \cup \Gamma^-_i, \quad \Gamma^+_i \leftarrow \Gamma^+_i \cup \{j\} \cup \Gamma^+_{\Gamma^-_i},
\]

\[
\forall l \in \{j\} \cup \Gamma^+_j, \quad \Gamma^-_j \leftarrow \Gamma^-_j \cup \{i\} \cup \Gamma^-_{\Gamma^+_j}.
\]

Using the relation \( \rightarrow \) (using the statement 5), from the set of s-moves generating the neighborhood, there will be some elements removed.

Corollary 4. For any s-move \( s^k_l \in \mathcal{M}_af^k \cup \mathcal{M}_bf^k \), if

\[
\pi(l) \in \Gamma^+_n(k) \quad \text{then} \quad \mathcal{M}_af^k \leftarrow \mathcal{M}_af^k \setminus \{s^k_l\},
\]

and if

\[
\pi(l) \in \Gamma^-_n(k) \quad \text{then} \quad \mathcal{M}_bf^k \leftarrow \mathcal{M}_bf^k \setminus \{s^k_l\}.
\]

Proof. It follows directly from the Theorem 5. □

5. Construction of Algorithm

The local search algorithm starts with some startup solution. Then its neighborhood is generated and a certain element is selected from it, which is assumed as the starting solution for the next iteration. Thus, the space of solutions is browsed with the “moves”, from one element to the other. This process continues until certain stop criterion is met. In this way a sequence (trajectory) of solutions is created, from which the best element is the result of an algorithm running.

One of the deterministic and most commonly used implementations of the local search method is tabu search (TS for short). Its main ideas were presented by Glover in the works of [39,40] and monographs Glover and Lagoon [41]. To avoid ‘looping’ (going back to the same solution), a short-term memory mechanism is introduced, tabu (solutions or prohibited movements). performing motion—determining the starting solution in the next iteration—its attributes are remembered on the list. Generating new neighborhood solutions, whose attributes are on the list, are omitted, except for those meeting the so-called aspiration criterion (i.e., “exceptionally favorable”). The basic elements of this method are:

- **move**—function transforming startup solution into another (one of the elements of the neighborhood),
- **neighborhood**—set of solutions determined by moves belonging to a certain set,
- **tabu list**—list containing attributes of recently considered startup solutions,
- **stop criterion**—in practice, it is a fixed number of iterations or algorithm calculation time.

Let \( x \in \mathcal{X} \) be any startup solution \( \mathcal{LT} \) tabu list and \( \Psi \) selection criterion (i.e., function enabling comparing elements of the neighborhood), usually it is a goal function \( \mathcal{F} \).

Computational complexity of a single algorithm iteration depends on the number of elements of the neighborhood, the procedure for generating its elements and the complexity of the function that calculates the criterion value. A detailed description of the implementation of this algorithm for a single-machine task scheduling problem is presented in the work of Bozejko et al. [42]

5.1. Construction Algorithms

Most construction algorithms are quick and simple to implement, unfortunately the solutions they designate are usually “far from optimal”. Hence, they are almost exclusively used to determine “good” startup solutions for other algorithms. In case of the TWT
problem, the following algorithms (or their simple modifications and different hybrids) has been used for years:

- **SWPT**—Shortest Weighted Processing Time, (Smith [37]),
- **EDD**—Earliest Due Date, (Baker [43]),
- **COVERT**—Cost Over Time, (Potts and Van Wassenhove [30]),
- **AU**—Apparent Urgency, (Potts and Van Wassenhove [30]),
- **META**—Metaheuristic, (Potts and Van Wassenhove [30]).

The first two have a static priority function and computational complexity of $O(n \ln n)$, whereas in the other two there is dynamic priority function used and their computational complexity is $O(n^2)$.

### 5.2. Tabu Search Algorithm

Considered in this work problem were solved using tabu search algorithm. Next, we will describe each element of the algorithm in more detail.

**Neighborhood.** In TS algorithm there is neighborhood used, generated by swap and insert moves. Let $B$ be a division of D-OPT permutation $\pi$ into blocks. Using the elimination block properties for $i-1$ moves we determine (Corollaries 2–3) a set of moves before and after the block $M(\pi) = \bigcup_{j \in I} (M^i_{bf} \cup M^i_{af})$. (9)

Next:
1. According to Corollary 2 and 3 $M(\pi)$ we remove some subsets of $i-1$ moves leaving only their representatives.
2. We remove $s-1$ moves, whose performance means that one of the conditions of the Theorem 5 is not met

Therefore, the neighborhood of permutation $\pi$

$$\mathcal{N}(\pi) = \{m^k_l(\pi) : m^k_l \in M(\pi)\}. \quad (10)$$

The procedure for determining the neighborhood has a complexity $O(n^2)$.

**Startup solution.** For each example, in TS algorithm, there were adopted startup solutions determined by the best construction algorithms: SWPT, EDD, COVERT, AU and META algorithms. Their description is presented in Section 5.1.

**Stop condition.** A stop condition for completing the calculations of both algorithms was the maximum number of iterations. In the process of parallel implementations of TS algorithm was a strategy of many computing processors working in parallel used. The maximum number of iterations for each processor was $\lceil 1000/p \rceil$, where $p$ is the number of processors.

**Tabu List in TS algorithm.** To prevent the formation of cycles too quickly (i.e., a return to the same permutation, after a small number of iterations of the algorithm), some attributes of each move are remembered on the so-called tabu list (abbreviated to LT). It is supported on a FIFO queue basis. Making a move $m^r_j \in M(\pi)$ (i.e., generating from $\pi \in \Phi$ permutation $\pi'_j$), we write some attributes of this move on the list, i.e., threesome: $(\pi(r), j, F(\pi'_j))$. Let us assume that we are considering the move $m^k_l \in M(\beta)$ generating permutation $\beta^k_l$. If on the list LT there is a threesome $(r, j, Y)$ such that $\beta(k) = r$, $l = j$ and $F(\beta^k_l) \geq Y$, then this move is eliminated (removed) from the set $M(\beta)$.

### 6. Parallelization of Algorithms

The proposed tabu search algorithm (in short TSA) has been parallelized using the MPI library using the scheme of independent search processes with different starting points (MSSS according to Voß [19] classification) On the cluster platform, there were parallel non-cooperative processes implemented with mechanism for diversifying startup solutions based on the Scatter Search idea. Each of the processors modified the solution generated by the
MERTA algorithm by performing a certain number of swap moves proportional to the size of the problem and the number of a processor. A parallel reduction mechanism (MPI_Bcast) was used to collect the data. MWPTSA (Multiple-Walk Parallel Tabu Search Algorithm) parallel algorithm's pseudocode is given in Algorithm 3, and the scheme in Figure 2.

**Algorithm 3: Multiple-Walk Parallel Tabu Search Algorithm (MWPTSA)**

```
Input : \( \Pi^0 = \{\pi^0_1, \pi^0_2, \ldots, \pi^0_p\} \) – set of initial solutions;
\( p \) – number of MPI processes;

Output : \( \pi^* \) – the best-found solution;

1 parfor \( i = 1, 2, \ldots, p \) do
2 \( \pi_i \leftarrow \pi^0_i \); (current solution, local for the process)
3 \( \pi^*_i \leftarrow \pi^0_i \); (local best solution found so far)
4 repeat
5 \( LT_i \leftarrow \emptyset \); (tabu list)
6 Generate neighborhood \( \mathcal{N}(\pi_i) \) taking under consideration tabu list \( LT_i \) as well as aspiration criterion;
7 \( \delta = \arg \min_{\pi \in \mathcal{N}(\pi_i)} F(\pi) \);
8 if \( F(\delta) < F(\pi^*_i) \) then
9 \( \pi^*_i \leftarrow \delta \);
10 \( \pi \leftarrow \delta \);
11 Add move attributes to the local tabu list \( LT_i \);
12 until Stop criterion (achieving solution with assumed or better value of the cost function);
13 Reduce \( \pi^*_i, i = 1, 2, \ldots, p \), to the best value with using tree-based parallel calculating scheme;
14 return \( \Theta^* \)
```

7. Computational Experiments

The parallel tabu search MWWTS algorithm has been implemented in C++ using the MPI library. The calculations were made on the BEM cluster installed in the Wrocław Network and Supercomputing Center. Parallel computing tasks were run on Intel Xeon E5-2670 2.30GHz on processors under the control of the PBS queue system.

Examples of test data of various sizes and degrees of difficulty, on which the calculations were made, were divided into two groups:
(a) the first group includes 375 examples of three different sizes \((n = 40, 50, 100)\). Together with the best solutions, they are placed on the OR-Library [44] website.

(b) the second group embracing test instances for \(n = 200, 500, 1000\) was generated in following way:

- \(p_i\) – random integer from range \([1, 100]\) with uniform distribution,
- \(w_i\) – random integer from range \([1, 10]\) with uniform distribution,
- \(d_i\) – random integer from range \(\left[ P(1 - T_F - R_{DD}/2), P(1 - T_F + R_{DD}/2) \right]\) with uniform distribution.

Where \(P = \sum_{i=1}^{o} p_i\), \(R_{DD} = 0.2, 0.4, 0.6, 0.8, 1.0\) (relative range of due dates) and \(F_T = 0.2, 0.4, 0.6, 0.8, 1.0\) (average tardiness factor). For each of the 25 pairs of values of \(R_{DD}\) and \(F_T\) five instances were generated. Overall, 325 instances were generated – 125 for each value of \(n\). Test instances was published on web page [45].

Computational experiments of the algorithms presented in the work were carried out in several stages. First, the construction algorithms SWPT, EDD, COVERT, AU and META algorithms, presented in Section 5.1, were compared. Calculations were made on examples from group a). The obtained results (percentage relative error (11)) are presented in Table 1, compared to the reference solutions taken, as well as benchmarks data, from the OR-Library [44].

| Instance Group | SWPT   | COVERT | AU    | EDD   | META   |
|----------------|--------|--------|-------|-------|--------|
| wt40           | 579.92 | 152.39 | 15.77 | 138.55| 10.46  |
| wt50           | 4315.94| 951.99 | 21.93 | 155.21| 10.21  |
| wt100          | 4692.65| 1417.07| 23.64 | 143.28| 12.90  |
| average        | 3196.17| 840.49 | 20.45 | 145.68| 11.19  |

The best solutions were determined by the META algorithm [30]. The solutions of this algorithm are the starting point of the TSA algorithm. When running these algorithms on multiple processors, each of them started from a different initial solution. Differentiation of initial solutions was made by running a sequential algorithm with a small number of iterations \(10i, i = 1, 2, \ldots, p\), where \(p\) is the number of processors.

To determine the values of the parameters of approximation algorithms, preliminary calculations were made on a small number of randomly determined examples. Based on the analysis of the results obtained, the following were adopted:

- length of tabu list: 7,
- length of the list for a long-term memory: 5,
- algorithm for determining the startup solution: META,
- algorithm’s stop condition: computation time \(t = 120\) s.

As the solution of the parallel algorithm there was the best solution obtained by individual processors. The percentage relative error was determined for each solution:

\[
PRD = \frac{F_{ref} - F_{alg}}{F_{ref}} \cdot 100\%,
\]

where:

- \(F_{ref}\)—value of the reference solution obtained with META algorithm,
- \(F_{alg}\)—the value of the solution determined by tested algorithm.

Table 2 contains PRD improvement values relative to the solution obtained by the META algorithm, which is also a starting solution for TSA (that is why they are negative) for bigger instances of the sizes \((n = 200, 500 and 1000)\).
Table 2. Computational results for sequential tabu search algorithm \((t = 120 \text{ s}, \text{META as reference})\).

| Instance Group | \(wt200\) | \(wt500\) | \(wt1000\) |
|----------------|----------|----------|-----------|
|                | \(TS\)  | \(TS_b\) | \(TS\)  | \(TS_b\) | \(TS\)  | \(TS_b\) |
| 001–025        | −18.43   | −18.49   | −15.55   | −16.85   | −5.21   | −7.54    |
| 026–050        | −11.38   | −11.29   | −13.68   | −13.88   | −4.05   | −3.54    |
| 051–075        | −9.84    | −9.78    | −9.18    | −9.14    | −3.79   | −3.65    |
| 076–100        | −5.77    | −5.73    | −3.32    | −3.46    | −4.60   | −3.82    |
| 101–125        | −2.80    | −2.79    | −3.99    | −3.93    | −1.32   | −0.78    |
| average        | −9.64    | −9.62    | −9.15    | −9.45    | −3.79   | −3.87    |

Table 3 contains PRD improvement values relative to the solution obtained by the META algorithm, which is also a starting solution for parallel TSA (that is why they are negative) for bigger instances of the sizes \((n = 200, 500 \text{ and } 1000)\). Experiments have been conducted for the number of processors \(p = 8, 16, 32 \text{ and } 64\). One can observe that increasing of the number of processors results in improving of the quality of the obtained solutions especially for algorithm with blocks.

Table 3. Computational results for parallel tabu search algorithm \((t = 120 \text{ s}, \text{META as reference})\).

| Number of Processors | \(wt200\) | \(wt500\) | \(wt1000\) |
|----------------------|----------|----------|-----------|
|                      | \(TS\)  | \(TS_b\) | \(TS\)  | \(TS_b\) | \(TS\)  | \(TS_b\) |
| 1                    | −9.64    | −9.62    | −9.15    | −9.45    | −3.79   | −3.87    |
| 8                    | −9.65    | −9.63    | −9.17    | −9.53    | −3.88   | −3.91    |
| 16                   | −9.65    | −9.64    | −9.17    | −9.55    | −3.88   | −3.92    |
| 32                   | −9.65    | −9.66    | −9.17    | −9.55    | −3.88   | −3.94    |
| 64                   | −9.66    | −9.71    | −9.19    | −9.59    | −3.88   | −3.95    |

Tables 4 and 5 contain speedup results. The values have been calculated from the relationship \(s = t_s/t_p\), where
- \(t_s\) time during which the sequential algorithm obtains the \(F\) solution,
- \(t_p\) is the time after which the parallel algorithm obtains the \(F\) solution.

From the results presented in the table, it can be seen that high speedups are obtained for smaller size problems (\(TS_b\) values over 60,000 for \(n = 40\)). The block algorithm gives better acceleration (on average more than 2600 times better). Such results are probably the result of the fact that the size of the problem solution space \((40!, 50!, 100!, 200!, 500!, 1000!)\) increases much faster than the number of processors increases \((8, 16, 32, 64)\). The increase in speedup within a single size of the problem is also not large as shown above. The large disproportion in speedup of the algorithm with and without blocks results from the fact that the block algorithm strongly limits the space of the solutions being reviewed.

As one can see increasing the size of the problem reduces the value of the speedup obtained – which is probably due to the limited computation time for a larger space for search solutions; however, increasing the number of processors does not have a big impact on acceleration values even when increasing the size of the problem.
Table 4. Speedup values for parallel tabu search algorithm.

|       | p = 8     | p = 16     |
|-------|-----------|------------|
|       | \(s_{TS}\) | \(s_{TS_b}\) | \(s_{TS}\) | \(s_{TS_b}\) |
| \(wt40\) | 182.60    | 61,491.57  | 185.57     | 61,466.13  |
| \(wt50\)  | 39.17     | 28,943.19  | 39.54      | 23,866.83  |
| \(wt100\) | 5.63      | 2150.46    | 5.78       | 2234.89    |
| \(wt200\) | 3.99      | 58,752.36  | 4.24       | 55,967.32  |
| \(wt500\) | 3.42      | 21.80      | 3.58       | 22.21      |
| \(wt1000\)| 2.23      | 1280.58    | 2.27       | 1724.82    |

Table 5. Speedup values for parallel tabu search algorithm.

|       | p = 32    | p = 64     |
|-------|-----------|------------|
|       | \(s_{TS}\) | \(s_{TS_b}\) | \(s_{TS}\) | \(s_{TS_b}\) |
| \(wt40\) | 193.85    | 60,606.01  | 186.14     | 60,595.42  |
| \(wt50\)  | 40.01     | 53,659.18  | 38.98      | 50,530.73  |
| \(wt100\) | 5.94      | 2768.76    | 5.88       | 2793.09    |
| \(wt200\) | 4.13      | 56,021.05  | 4.13       | 55,445.96  |
| \(wt500\) | 3.47      | 22.10      | 3.53       | 22.90      |
| \(wt1000\)| 2.29      | 985.34     | 2.30       | 1020.59    |

8. Conclusions

Based on the problem analysis and computational experiments performed, we can suggest the following conclusions. Tabu search method using block properties allows the solution of the problem not only faster than classic tabu search metaheuristic without using these properties, but also with huge speedups. Speedups achieved of the parallel tabu search algorithm with blocks are much greater than for classic tabu search algorithm, which confirms the legitimacy of using block elimination properties. The fact of achieving such huge speedups should be the subject of further research. Also proposed blocks elimination criteria can be used for constructing efficient parallel algorithms solving other NP-hard scheduling problems.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Test instances for a single-machine total weighted tardiness scheduling problem generated during the study are available online: https://zasobynauki.pl/zasoby/51561 (accessed on 25 February 2021).

Acknowledgments: The paper was supported by the National Science Centre of Poland, grant OPUS no. 2017/25/B/ST7/02181. Calculations have been carried out using resources provided by Wroclaw Centre for Networking and Supercomputing (http://wcss.pl (accessed on 25 February 2021)), grant No. 96.

Conflicts of Interest: The authors declare no conflict of interest.

References
1. Wodecki, M. A Branch-and-Bound Parallel Algorithm for Single-Machine Total Weighted Tardiness Problem. *Int. J. Adv. Manuf. Technol.* 2008, 37, 996–1004. [CrossRef]
2. Rinnoy, Kan, A.H.G.; Lageweg, B.J.; Lenstra, J.K. Minimizing total costs in one-machine scheduling. *Oper. Res.* 1975, 25, 908–927. [CrossRef]
3. Cordone, R.; Hosteins, P. A bio-objective model for the singla-machine scheduling with rejection cost and total tardiness minimization. *Comput. Oper. Res.* 2019, 102, 130–140. [CrossRef]
4. Rostami S., Creemers S., Leus R., Precedence theorems and dynamic programming for the single machine weighted tardiness problem. *Eur. J. Oper. Res.* 2019, 272, 43–49. [CrossRef]
36. Grosso, A.; Della Croce, F.; Tadei, R. An enhanced dynasearch neighborhood for single-machine total weighted tardiness scheduling problem. *Oper. Res. Lett.* **2004**, *32*, 68–72. [CrossRef]

37. Smith, W.E. Various Optimizers for Single-Stage Production, *Nav. Res. Logist. Q.* **1956**, *3*, 59–66. [CrossRef]

38. Shwimer, J. On the N-Job, One-Machine, Sequence-Independent Scheduling Problem with tardiness Penalties: a Branch-and-Bound Solution. *Manag. Sci.* **1972**, *18*, 301–313. [CrossRef]

39. Glover F. Tabu search. Part I. *ORSA J. Comput.* **1989**, *1*, 190–206. [CrossRef]

40. Glover F. Tabu search. Part II. *ORSA J. Comput.* **1990**, *2*, 4–32. [CrossRef]

41. Glover, F.; Laguna, M. *Tabu Search*; Kluwer: Alphen aan den Rijn, The Netherlands, 1997.

42. Bożejko, W.; Grabowski, J.; Wodecki, M. Block approach-tabu search algorithm for single machine total weighted tardiness problem, *Comput. Ind. Eng.* **2006**, *50*, 1–14.

43. Baker, K.R. *Introduction to Sequencing and Scheduling*; Wiley: New York, NY, USA, 1974.

44. OR Library. Available online: [http://people.brunel.ac.uk/~mastjjb/jeb/info.html](http://people.brunel.ac.uk/~mastjjb/jeb/info.html) (accessed on 25 February 2021).

45. Uchorński, M. Test Instances for a Single-Machine Total Weighted Tardiness Scheduling Problem. Available online: [https://zasobynauki.pl/zasoby/51561](https://zasobynauki.pl/zasoby/51561) (accessed on 25 February 2021).