Computers & Industrial Engineering 65 (2013) 466–474

Contents lists available at SciVerse ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

Parallel tabu search algorithm for the hybrid flow shop problem ☆

Wojciech Bożejko *, Jarosław Pempera, Czesław Smutnicki

Institute of Computer Engineering, Control and Robotics, Wrocław University of Technology, Janiszewskiego 11-17, 50-372 Wrocław, Poland

ARTICLE INFO

Article history:
Received 3 April 2012
Received in revised form 10 April 2013
Accepted 11 April 2013
Available online 19 April 2013

Keywords:
Parallel metaheuristics
Scheduling
Hybrid flow shop
Tabu search

ABSTRACT

The paper deals with the parallel variant of the scheduling algorithm dedicated to the hybrid flow shop problem. The problem derives from practice of automated manufacturing lines, e.g. for printed packages. The overall goal is to design a new algorithm which merges the performance of the best known sequential approach with the efficient exploitation of parallel calculation environments. In order to fulfill the above aim, there are two methods proposed in this paper: the original fast method of parallel calculation of the criterion function and the local neighborhood parallel search method embedded in the tabu search approach. The theoretical analysis, as well as the original implementation, with the use of vector processing instructions SSE2 supported by suitable data organization, are presented below. Numerical properties of the proposed algorithm are empirically verified on the multi-core processor.

1. Introduction

The *hybrid flow shop* scheduling problem, called also the flow shop problem with parallel machines, is a combination (thus generalization) of the classic *flow shop* problem (each job flows through the sequential structure of service stages having single machine at each stage) and *parallel shop* problem (at each stage there is a set of identical machines working in parallel). Both component problems are well examined in the scheduling theory and have quite rich literature; this, however, does not refer to their combination. Skipping consciously the long list of papers dealing with this subject, we only refer the reader to recent reviews and best up-to-now algorithms described in Ruiz and Vázquez-Rodríguez (2010).

There are only a few papers dealing with single-walk parallel algorithms for the job scheduling problems. Bożejko, Pempera, and Smutnicki (2009) proposed a single-walk parallelization of the simulated annealing metaheuristic for the job shop problem. Steinhöfel, Albrecht, and Wong (2002) presented the method of parallel cost function determination for the job shop problem. Bożejko (2012, 2009) considered methods of parallel cost function calculation for the permutational flow shop problem, which constitutes a special case of the problem considered here.

The hybrid problem is commonly considered as the fundamental one for modeling real automated manufacturing lines. More realistic models supplemented by a few additional constraints, e.g. buffering (Sawik, 1993), have also been considered. Since the stated problem originates from the flow shop problem, it certainly constitutes the strongly NP-hard case. There are a few criteria used to evaluate the schedule quality. The most popular is makespan, because of its practical significance, possibility of utilization of some special properties followed chiefly from the critical path notion, relative simplicity of solution algorithms and good quality of approximate methods for instances of real sizes. As to the hybrid flow shop problem, in the case modeled and considered here, there are only few papers dealing with it. There have also been reports on practical applications of the flexible scheduling and assembly systems in works of Sawik (1999) and Wittrock (1988).The state-of-the-art method follows from Nowicki and Smutnicki (1998). This quite sophisticated algorithm, supported by a few theoretical features of the problem, is capable of solving instances up to 300 jobs, 10 stages and 50 machines (3000 operations) with quality of approximately 5% (relative percentage deviation to optimal solutions, in average) in a reasonable running time. In contrast, Azizoğlu, Çakmak, and Kondakci (2001), proposes branch and bound algorithm, capable of solving instances with very small sizes (up to 15 jobs and 10 machines) in the time yet acceptable by practitioners. Also, new hybrid approaches can be met in the literature, i.e. (Khademi Zare & Fakhrzad, 2011; Shahvari, Salmasi, Logendran, & Abbasi, 2012).

Although there are no doubts that the future belongs to approximate approaches, there is still room for improving their efficiency, speed and quality. Moreover, since their running time increases disturbingly for large size instances, one reposes hope in parallel approaches, perceiving them as the new promising tendency in the design of efficient algorithms, see e.g. (Bożejko et al., 2009).

Quality of the best solutions determined by approximate algorithms depends, in most cases, on the number of solutions being analyzed, therefore on the time of computations. Time and quality

☆ This manuscript was processed by Area Editor Maged M. Dessouky.

* Corresponding author. Tel.: +48 603672142.
E-mail address: wojciech.bozejko@pwr.wroc.pl (W. Bożejko).

0360-8352/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cie.2013.04.007

demonstrate opposite tendencies in the sense that obtaining a better solution requires significant increase in computing time. The construction of parallel algorithms makes it possible to increase significantly the number of solutions considered (in a unit of time) using effectively multi-processor computing environment.

Here we propose a new Parallel Tabu Search (ParTS) scheduling algorithm which belongs to the local search methods class. This algorithm starting from presented in Nowicki and Smutnicki (1998), trends to solve problem instances of very large size by introducting new properties advantageous for parallel computing environment. The ParTS is also dedicated for the use in multi-processors systems as well as in the single-processor systems with vector processing SSE2 instructions set. The algorithm is especially customized to small-grain parallel architectures, with fast communication between processors, such as multi-core processors, processors with vector processing instructions and GPUs.

2. The problem

Let us consider a manufacturing system with a structure consisting of m machine centers given by the set $M = \{1, \ldots, m\}$, where each center $k \in M$ is equipped with $m_k \geqslant 1$ identical machines given by the set $M_k = \{1, \ldots, m_k\}$. The production task is given by the set of jobs (i.e. parts or customer orders) $J = \{1, 2, \ldots, n\}$. Each job has to be processed on a machine in 1, 2, ..., m centers in that order. Job $j, j \in J$, consists of a sequence of m operations $O_{j1}, O_{j2}, \ldots, O_{jm}$; operation O_{jk} corresponds to the processing of j job on a machine in k center during p_{jk} uninterrupted processing time. Each machine can execute at most one job at a time and each job can be processed on at most one machine at a time. We want to find an assignment of jobs to machines and a schedule on each machine such that the maximum completion time (makespan) is minimal.

Let us denote by J_{ki} the set of jobs allocated to machine i in center k. Clearly, for any $k \in M$, sets $J_{k1}, \ldots, J_{km_k}$ constitute a partition of the set J, i.e. $J_{ki} \cap J_{kj} = \emptyset$, $i \neq j$, $i,j = 1, \ldots, m_k$ and $J = \bigcup_{i=1}^{m_k} J_{ki}$. The sequence of processing jobs on i-th machine in the k-th center can be represented by a permutation $\pi_{ki} = (\pi_{ki}(1), \ldots, \pi_{ki}(n_{ki}))$ of elements from the set J_{ki}, where $n_{ki} = |J_{ki}|$. Thus, jobs processing at center k can be completely represented by the set of m_k permutations $\pi_k = (\pi_{k1}, \ldots, \pi_{km_k})$, while the overall job processing order can be described by the m-tuple $\pi = (\pi_1, \ldots, \pi_m)$. Note that π is able to carry information about processing order as well as jobs assignment, since $J_{ki} = \{\pi_{ki}(1), \ldots, \pi_{ki}(n_{ki})\}$, where $n_{ki} = |\pi_{ki}|$.

Traditionally, the schedule is represented by job starting and/or completion times. In our case, for each given π, the schedule can be described by the matrix of jobs starting times $S_{jk} \geqslant 0$, $j = 1, \ldots, n$, $k = 1, \ldots, m$, satisfying the following constraints:

$$S_{j,k} \geqslant S_{j,k-1} + p_{j,k-1} \quad j = 1, \ldots, n, \ k = 2, \ldots, m, \tag{1}$$

$$S_{\pi_{ki}(s),k} \geqslant S_{\pi_{ki}(s-1),k} + p_{\pi_{ki}(s-1),k} \quad \begin{matrix} s = 2, \ldots, n_{ki}, \ i = 1, \ldots, m_k, \\ k = 1, \ldots, m. \end{matrix} \tag{2}$$

Constraint (1) follows from the technological processing order of operations inside a job, whereas (2) – from the capacity of machines. Our overall aim is to find the processing order $\pi^* \in \Pi$ such that

$$C_{\max}(\pi^*) = \min_{\pi \in \Pi} C_{\max}(\pi), \tag{3}$$

where $C_{\max}(\pi) = \max_{1 \leqslant j \leqslant n}(S_{jm} + p_{jm})$ is the makespan (maximum of jobs completion times) and Π is the set of all processing orders. The set Π is defined as follows:

$$\Pi = \{\pi = (\pi_1, \ldots, \pi_m) : \pi_k = (\pi_{k1}, \ldots, \pi_{km_k}), (\pi_{ki} \in P(J_{ki}), i \in M_k), (J_{ki}, i \in M_k \text{ is a partition of theset } J), k \in M\}, \tag{4}$$

where $P(X)$ denotes the set of all permutations of the set X. Since makespan is the regular criteria, schedule for the given π is shifted to the left on the time axis and can be found using ASAP (as soon as possible) rule. Thus, based on inequalities (1) and (2), we are proposing the following recursive formula for calculating the schedule:

$$C_{\pi_{ki}(j),k} = \max\{C_{\pi_{ki}(j-1),k}, C_{\pi_{ki}(j),k-1}\} + p_{\pi_{ki}(j),k} \tag{5}$$

$j = 1, 2, \ldots, n_{ki}$, $i = 1, 2, \ldots, m_k$, $k = 1, 2, \ldots, m$, with initial conditions $\pi_{ki}(0) = 0$, $C_{0,k} = 0$, $C_{j,0} = 0$. The number of values of $C_{\pi_{ki}(j),k}$ for the fixed processing order π_{ki} on i-th machine in k-th center is $O(\sum_{k=1}^{m}\sum_{i=1}^{m_k} n_{ki}) = O(\sum_{k=1}^{m} n) = O(nm)$ because the sum of operations numbers in each center $k \in M$ equals $\sum_{i=1}^{m_k} n_{ki} = n$ (as we mentioned above, $n_{ki} = |J_{ki}|, J = \bigcup_{i=1}^{m_k} J_{ki}$ and $|J| = n$). Fortunately, the recursion can be easily omitted by computation these values in the proper order of indexes, as given above. Therefore, the computational complexity of (5) is $O(nm)$.

3. Properties

Both: special properties employed in the algorithm and accelerator (sequential as well as parallel) refer to the graph model defined for each processing order π. The graph $G(\pi) = (N, T \cup E(\pi))$ has the set of nodes $N = J \times M$ and set of arcs $T \cup E(\pi)$, where

$$T = \bigcup_{j=1}^{n}\bigcup_{k=2}^{m}\{(j,k-1),(j,k)\}, \tag{6}$$

$$E(\pi) = \bigcup_{k=1}^{m}\bigcup_{i=1}^{m_k}\bigcup_{j=2}^{n_{ki}}\{(\pi_{ki}(j-1),k),(\pi_{ki}(j),k)\}. \tag{7}$$

The node $(j, k) \in N$ has the weight $p_{j,k}$. Arcs from the set T represent the processing order of operations in jobs and correspond to constraint (1), whereas arcs from set $E(\pi)$ represent the processing order of operations on machines and correspond to constraint (2). All arcs from both subsets have weight zero.

It is easy to prove that $C_{j,k}$ equals the length of the longest path going to the node (j, k) in this graph (including this node weight). The makespan $C_{\max}(\pi)$ equals the length of the longest (critical) path in $G(\pi)$.

Fig. 1 illustrates a directed graph for an example of three centers $k = 1, 2, 3$ and six jobs $j = 1, 2, \ldots, 6$ with processing times given in Table 1. The critical path in $G(\pi)$ is drawn as a bold line.

Each path in the graph $G(\pi)$ can be represented by a sequence of nodes. Let us denote the critical path (arbitrarily selected) in $G(\pi)$ by $(n_o, n_{o+1}, \ldots, n_{p-1}, n_p)$, $n_i = (j_i, k_i) \in N$, $i = o, \ldots, p$. A special type of sub-path called block is determined by the maximal sequence $(n_s, \ldots, n_t)$ such that $k_s = k_{s+1} = \cdots = k_{t-1} = k_t$ and $(n_i, n_{i+1}) \in E(\pi)$, $i = s, \ldots,$

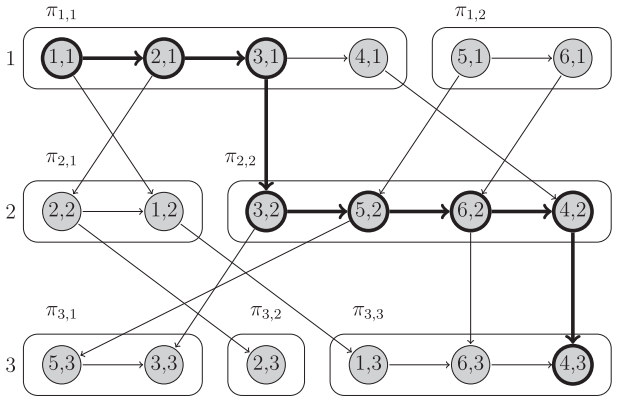

Fig. 1. Directed graph $G(\pi)$ for the $C_{\max}(\pi)$ computation.

Table 1
Data for the instance.

j	k	p_{jk}	j	k	p_{jk}
1	1	60	4	1	30
1	2	60	4	2	40
1	3	30	4	3	50
2	1	30	5	1	20
2	2	10	5	2	90
2	3	40	5	3	70
3	1	40	6	1	30
3	2	30	6	2	50
3	3	40	6	3	30

$t-1$. A block corresponds to a sequence of operations (jobs) processed on the same machine with no idle time.

We propose the new properties developed especially for the hybrid flow shop problem, which are based on so called block approach firstly introduced by Grabowski, Skubalska, and Smutnicki (1983). We will make use of them to reduce the size of the researched neighborhood as well as to construct an effective parallel accelerator.

Property 1. *For any processing order π, there exists exactly one block of operations in each center.*

Proof. Each critical path in the graph $G(\pi)$ consists of blocks linked by arcs from T. A technological arc $((j,k-1),(j,k))$ from T links two nodes which represents two operations performed in two successive centers $k-1$ and k. Therefore each two successive blocks of operations are performed on two successive centers, i.e. the first one in center 1, the next one in center 2, etc. □

Property 2. *Let $\pi \in \Pi$ be any processing order with blocks $B_k = (n_{s_k}, \ldots, n_{t_k})$, $k = 1,2,\ldots,m$. If $C_{max}(\pi) > C_{max}(\beta)$ for any other processing order $\beta \in \Pi$, then in β:*

- *at least one operation $o \in B_k$ precedes the first operation of the block (n_{s_k}), for $k \in \{2, 3, \ldots, m\}$, or*
- *at least one operation $o \in B_k$ succeeds the last operation of the block (n_{t_k}), for $k \in \{1, 2, \ldots, m-1\}$, or*
- *at least one operation $o \in B_k$ is performed on other machine(s) from the center k, for $k \in \{1, 2, \ldots, m\}$.*

Proof (by contradiction). The length of any path in $G(\pi)$ equals to the sum of weight of nodes (weight of each arc is zero). Let $u = (n_{s_1}, \ldots, n_{t_1}, n_{s_2}, \ldots, n_{t_2}, \ldots, n_{s_m}, \ldots, n_{t_m})$ be a critical path in $G(\pi)$. The sequence $n_{s_k}, n_{s_k+1}, \ldots, n_{t_k-1}, n_{t_k}$ constitutes the block on a machine in the center k. Let us assume by contradiction that $C_{\max}(\pi) < C_{\max}(\beta)$ and β does not fulfill any of 3 bullets (conditions) of the Property 2. Then, there exists the path (not necessary critical) $v = (n_{s_1}, \ldots, n_{t_1}, n_{s_2}, \ldots, n_{t_2}, \ldots, n_{s_m}, \ldots, n_{t_m})$ in the graph $G(\beta)$, which is different from the path u taking under consideration the order of nodes inside the blocks, i.e. nodes $n_{s_k+1}, \ldots, n_{t_k-1}, k = 1, \ldots, m$. By using commutative properties of addition we conclude that the length of the path v equals to the length of critical path u. Therefore, we have $C_{\max}(\beta) \geqslant C_{\max}(\pi)$ which is contradictory with the assumption, that $C_{\max}(\pi) < C_{\max}(\beta)$.

Let $Q_{j,k}$ be a length of the longest path going out from the node (j, k) (including this node weight) in $G(\pi)$. All values $Q_{j,k}$ can be found in the time $O(nm)$ by using the following recursive formula calculated for $j = n_{ki}, \ldots, 1$, $i = m_k, \ldots, 1$, $k = m, \ldots, 1$,

$$Q_{\pi_{ki}(j),k} = \max\{Q_{\pi_{ki}(j+1),k}, Q_{\pi_{ki}(j),k+1}\} + p_{\pi_{ki}(j),k} \tag{8}$$

with initial conditions $\pi_{ki}(n_{ki}+1) = 0$, $Q_{0,k} = 0$, $Q_{j,\,m+1} = 0$.

Property 3. *Let $L_{j,k} = C_{j,k} + Q_{j,k} - p_{j,k}$ be the length of the longest path in the graph $G(\pi)$ passing through the node (j, k). We have $C_{max}(\pi) \geqslant L_{j,k}$.*

Property 4. *For any center $k \in M$, we have $C_{max}(\pi) = max_{1 \leqslant j \leqslant n} L_{jk}$.*

The Property 3 is obvious. The length of the longest path passing through any node constitutes a lower bound of the $C_{\max}(\pi)$ value, whereas the Property 4 is a simple consequence of the Property 3 and the fact that any critical path has to pass through an operation of these performed in the center.

4. The algorithm

The tabu search algorithm TSNS is commonly considered as the most effective solution method for the considered problem. Its high efficiency is obtained due to so called *reduced neighborhood* based on the block properties (presented in the previous section) and the *accelerator* designed for $C_{\max}$ computation for all neighbors of the base solution. A detailed description of all TSNS components can be found in Nowicki and Smutnicki (1998). Hereinafter we adduce only these elements of the original algorithm which are essential for introducing ParTS algorithm.

4.1. The reduced neighborhood

A lot of recent papers have shown obvious benefits from the usage of the insertion-type of moves applied to multiple-machines scheduling problems. The pentad $v = (k, a, x, b, y)$ associated with a processing order $\pi \in \Pi$ defines an insert move such that a job $\pi_{ka}(x)$ is deleted from the position $x, 1 \leqslant x \leqslant n_{ka}$, in the permutation π_{ka} and then inserted in the position y, $1 \leqslant y \leqslant n_{kb}+1$, in the permutation π_{kb}. The number of all such moves equals approximately n^2m. Taking into account the above criteria for each solution, we need $O(nm)$ time to evaluate goal function value; the cost of searching such neighborhood is extremely excessive $O(n^3m^2)$. That is why any useful properties are especially welcome.

The main idea of the reduction of the neighborhood size, introduced by Nowicki and Smutnicki (1998), consists in eliminating some moves, for which it is known *a priori* that no improvement of $C_{\max}(\pi)$ is possible. For the insertion-type move $v = (k, a, x, b, y)$, let $z = \pi_{ka}(x)$ and e_k (f_k) be the position of the first (last) operation of the block B_k of operations performed on the machine $h_k \in M_k$. Thus, the block has the form $B_k = (\pi_{kl}(e_k), \ldots, \pi_{kl}(f_k))$, where $l = h_k$. With respect to the Property 2 the move $v = (k, a, x, b, y)$, $z = \pi_{ka}(x)$ can be eliminated, if at least one of the following conditions is satisfied:

1. $z \notin B_k$,
2. $z \in B_k$, and $|B_k| = 1$,
3. $z \in B_k \setminus \{\pi_{ka}(e_k), \pi_{ka}(f_k)\}$, where $a = h_k$, $b = h_k$ and $e_k < x$, $y < f_k$,
4. $z \in B_1$ and $y = 1$, where $a = h_1$, $b = h_1$,
5. $z \in B_m$ and $y = n_{k,h_k}$, where $a = h_m$, $b = h_m$,
6. $z = \pi_{ka}(e_k)$, where $a = h_k$, $b = h_k$ and $y < x$,
7. $z = \pi_{ka}(f_k)$, where $a = h_k$, $b = h_k$ and $y > x$,
8. $z = \pi_{ka}(e_1)$, where $a = h_1$, $b = h_1$ and $y < f_1$,
9. $z = \pi_{ka}(f_m)$, where $a = h_m$, $b = h_m$ and $e_m < y$.

The reduced set of moves $W(\pi)$ satisfying the conditions of the Property 2 can be defined as follows:

$$W(\pi) = \bigcup_{k \in M} \bigcup_{x=e_k}^{f_k} \bigcup_{b \in M_k} W_{k,x,b} \tag{9}$$

where $W_{k,x,b} = \bigcup_{y=1}^{n_{kb}+1} \{(k, x, h_k, y, b)\}$ for $b \neq h_k$ and $W_{k,x,b} = L_{k,x} \cup R_{k,x}$ for $b = h_k$. The set of moves to the left is defined as

$L_{k,x} = \bigcup_{y=1}^{e_k-1}\{(k,x,h_k,y,h_k)\}$, and it is empty for $x = e_k$, $k = 2, \ldots, m$ and for $x = e_1, \ldots, f_1$, $k = 1$. The set of moves to the right is defined as $R_{k,x} = \bigcup_{y=f_k}^{n_{k,h_k}}\{(k,x,h_k,y,h_k)\}$ and is empty for $x = f_k$, $k = 1, \ldots, m-1$ and for $x = e_m, \ldots, f_m$, $k = m$.

The set of moves $W(\pi)$ generates a neighborhood $N(W(\pi), \pi) = \{\pi_v: v \in W(\pi)\}$ which consists of $w = |W(\pi)|$ processing orders and it consumes $O(wnm)$ computation time. In the worst case, a critical path consists of m sequences, each consists of n operations and each requires n insertion. So, $w = n^2m$ and computation complexity is $O(n^3m^2)$. The computation time strongly depends on distribution of the blocks.

In fact, the large original neighborhood is clustered into small disjoint subsets (clusters). The representative $r(X)$ of the cluster X is the best solution from this cluster. Next, each representative is classified into one of two general categories: unforbidden and forbidden. Additionally, forbidden but profitable moves belong to the unforbidden moves, especially the forbidden moves, which generate the new best solution. The new base solution in selected only from the set of unforbidden moves.

The representative set of moves $\widehat{W}(\pi)$ can be defined as follows:

$$\widehat{W}(\pi) = \bigcup_{k\in M}\bigcup_{x=e_k}^{f_k}\bigcup_{b\in M_k} r(W_{k,x,b}(\pi)), \tag{10}$$

where $C_{\max}(\pi_{r(X)}) = \min_{v\in X} C_{\max}(\pi_v)$. Checking of the tabu status for each solution generated by the set $W(\pi)$ can be executed in the time $O(|W(\pi)|)$, i.e. $O(1)$ per each solution. The utilization of $\widehat{W}(\pi)$ reduces this time to $O(|\widehat{W}(\pi)|)$ which is essential for parallel accelerator described below.

4.2. The sequential accelerator

The neighborhood $N(\widehat{W}(\pi), \pi)$ requires a great computational effort to be searched, namely, $O(\hat{w}nm)(\hat{w} = |\widehat{W}(\pi)|)$ per single neighborhood in the considered case, assuming that all neighbors will be evaluated explicitly by the Eq. (5). An advanced single neighborhood search method proposed in Nowicki and Smutnicki (1998) reduces the computations time potentially n times. In this section we are proposing new theoretical foundations and new description of the method.

Property 5. *Let $z(x) = \pi_{ka}(x)$. The length of the longest path in the graph $G(\pi)$ passing through the node $(z(x), k)$ equals to*

$$L_{z(x)k} = \max\{C_{\pi_{ka}(x-1),k}, C_{z(x),k-1}\} + p_{z(x)k} + \max\{Q_{\pi_{ka}(x+1),k}, Q_{z(x),k+1}\}. \tag{11}$$

where $\pi_{ka}(0) = 0$, $\pi_{ka}(n_{ka}+1) = 0$ for $k \in M$, $a \in M_k$, $C_{0,k} = 0$, $Q_{0,k} = 0$ for $k \in M$, $C_{j,0} = 0$, $C_{j,m+1} = 0$, $Q_{j,0}$, $Q_{j,m+1} = 0$ for $j \in J$.

Proof. It is enough to observe that with each node there are at most two incoming paths associated. The first path passes through the immediate technological predecessor represented by node $(z(x), k-1)$, whereas the second path passes through the immediate machine predecessor represented by node $(\pi_{ka}(x-1), k)$. The arcs which link this node with $(z(x), k)$ node have weight zero. Therefore, we have $C_{z(x)k} = \max\{C_{\pi_{ka}(x-1),k}, C_{z(x),k-1}\} + p_{z(x)k}$. From the similar analysis of outgoing paths, we have $Q_{z(x)k} = \max\{Q_{\pi_{ka}(x+1),k}, Q_{z(x),k+1}\} + p_{z(x)k}$. Substitution obtained $C_{z(x)k}$ and $Q_{z(x)k}$ to $L_{z(x)k} = C_{z(x)k} + Q_{z(x)k} - p_{z(x)k}$ completes the proof. □

The execution of the move $v = (k, a, x, b, y)$, $z(x) = \pi_{ka}(x)$ in the solution π can be divided into two phases: deletion and insertion. Without loss of generality in our above considerations, we focus on the given center k and given machines a and b. To make the description more intuitive, we use the superscript $(^{\uparrow(x)})$ to denote the processing order $\pi^{\uparrow(x)}$ (and all associated variables) obtained from the π by the deletion operation $O_{z(x),\,k}$ from position x in permutation π_{ka}, whereas we use the superscript $(^{\downarrow(x,y)})$ to denote the processing order $\pi^{\downarrow(x,\,y)}$ (and all associated variables) obtained from the $\pi^{\uparrow(x)}$ by the insertion operation $O_{z(x),\,k}$ on position y in permutation $\pi^{\uparrow(x)}_{kb}$.

Property 6. *For the fixed π and $v = (k, a, x, b, y)$, we have $C^{\uparrow(x)}_{j,s} = C_{j,s}, s = 1, \ldots, k-1, j \in J$ and $Q^{\uparrow(x)}_{j,s} = Q_{j,s} s = k+1, \ldots, m, j \in J$.*

From the similar analysis as above one can prove that for the deletion phase of a move $v = (k, a, x, b, y)$ the length of the longest incoming path does not change for all operations processed on machines other than a and all operations processed on the machine a in positions $1, \ldots, x-1$, i.e. processed before the deleted operation $z(x)$. Also the length of the longest outgoing path does not change for all operations processed on machines other than a and all operations processed on the machine a in positions $x+1, \ldots, n_{ka}$, i.e. processed after the deleted operation $z(x)$. Finally, after deletion of the operation $z(x)$, $x = e_k, \ldots, f_k$, from the block B_k, we have

$$C^{\uparrow(x)}_{\pi_{kl}(r),k} = \begin{cases} \max\left(C^{\uparrow(x)}_{\pi_{kl}(r-2),k}, C_{\pi_{kl}(r),k-1}\right) + p_{\pi_{kl}(r),k} & \text{for } l = h_k,\ r = x+1, \\ \max\left(C^{\uparrow(x)}_{\pi_{kl}(r-1),k}, C_{\pi_{kl}(r),k-1}\right) + p_{\pi_{kl}(r),k} & \text{for } l = h_k,\ r = x+2, \ldots, n_{ka}, \\ C_{\pi_{kl}(r),k} & \text{otherwise}, \end{cases} \tag{12}$$

$$Q^{\uparrow(x)}_{\pi_{kl}(r),k} = \begin{cases} \max\left(Q^{\uparrow(x)}_{\pi_{kl}(r+2),k}, Q_{\pi_{kl}(r),k+1}\right) + p_{\pi_{kl}(s),k} & \text{for } l = h_k,\ r = x-1, \\ \max\left(Q^{\uparrow(x)}_{\pi_{kl}(r+1),k}, Q_{\pi_{kl}(r),k+1}\right) + p_{\pi_{kl}(s),k} & \text{for } l = h_k,\ r = 1, \ldots, x-2, \\ Q_{\pi_{kl}(r),k} & \text{otherwise}, \end{cases} \tag{13}$$

and

$$L^{\uparrow(x)}_{j,k} = C^{\uparrow(x)}_{j,k} + Q^{\uparrow(x)}_{j,k} - p_{j,k} j \neq z, \quad j \in J. \tag{14}$$

Note, that Property 6 does not define $C^{\uparrow(x)}_{\pi_{ka}(x),k}$, $Q^{\uparrow(x)}_{\pi_{ka}(x),k}$, $L^{\uparrow(x)}_{\pi_{ka}(x),k}$..

Fig. 2 shows the graph $G(\pi^{\uparrow(x)})$ for processing order $\pi^{\uparrow(x)}$ obtained from π by applying deletion phase of move $v = (2, 2, 2, 1, 2)$. The deleted node and all adjacent and modified arcs are drawn in a bold line. Determination of the value of each expression (12)–(14) requires $O(1)$ time, whereas update of all values requires no more than $O(n)$ for each deletion.

Let us consider the insertion phase of move $v = (k, a, x, b, y)$. From Property 5, we have

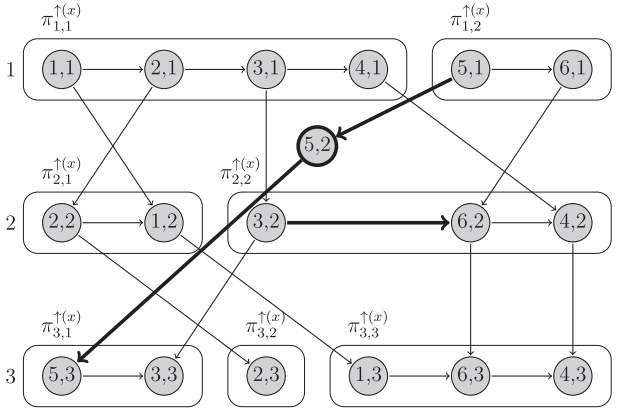

Fig. 2. Directed graph $G(\pi^{\uparrow(x)})$-deletion phase.

$$L^{\downarrow(x,y)}_{z(x),k} = \max\left\{C^{\downarrow(x,y)}_{\pi^{\downarrow(x,y)}_{kb}(y-1),k}, C^{\downarrow(x,y)}_{z(x),k-1}\right\} + p_{z(x),k} + \max\left\{Q^{\downarrow(x,y)}_{\pi^{\downarrow(x,y)}_{kb}(y+1),k}, Q^{\downarrow(x,y)}_{z(x),k+1}\right\}. \quad (15)$$

Property 7. *For the fixed π and $v=(k, a, x, b, y)$ we have*

$$C^{\downarrow(x,y)}_{\pi^{\downarrow(x,y)}_{kb}(y-1),k} = \begin{cases} C^{\uparrow(x)}_{\pi_{kb}(y),k} & \text{for } b = h_k,\ y > x, \\ C^{\uparrow(x)}_{\pi_{kb}(y-1),k} & \text{otherwise}, \end{cases} \quad (16)$$

$$Q^{\downarrow(x,y)}_{\pi^{\downarrow(x,y)}_{kb}(y-1),k} = \begin{cases} Q^{\uparrow(x)}_{\pi_{kb}(y+1),k} & \text{for } b = h_k,\ y > x, \\ Q^{\uparrow(x)}_{\pi_{kb}(y),k} & \text{otherwise}, \end{cases} \quad (17)$$

where $C^{\downarrow(x,y)}_{z(x),k-1} = C_{z(x),k-1}$ and $Q^{\downarrow(x,y)}_{z(x),k+1} = Q_{z(x),k+1}$.

Proof. The processing order $\pi^{\downarrow(x,y)}_{kb}$ obtained from the π by applying a move $v=(k, a, x, b, y)$ in π takes one of the following forms:

case $a \neq b$

$$\pi^{\downarrow(x,y)}_{kb} = (\pi_{kb}(1), \ldots, \pi_{kb}(y-1), \pi_{ka}(x), \pi_{kb}(y), \ldots, \pi_{kb}(n_{kb})) \quad (18)$$

case $a = b$, $y > x$

$$\pi^{\downarrow(x,y)}_{kb} = (\pi_{kb}(1), \ldots, \pi_{kb}(x-1), \pi_{kb}(x+1), \ldots, \pi_{kb}(y), \pi_{ka}(x), \pi_{kb}(y+1), \ldots, \pi_{kb}(n_{kb})) \quad (19)$$

case $a = b$, $y < x$

$$\pi^{\downarrow(x,y)}_{kb} = (\pi_{kb}(1), \ldots, \pi_{kb}(y-1), \pi_{kb}(x), \pi_{kb}(y), \ldots, \pi_{kb}(x-1), \pi_{kb}(x+1), \ldots, \pi_{kb}(n_{kb})). \quad (20)$$

Thus, the straight predecessor (successor) of $z(x) = \pi_{ka}(x)$ in $\pi^{\downarrow(x,y)}_{kb}$ is $\pi_{kb}(y-1)$ ($\pi_{kb}(y)$) for $l = h_k$, $y > x$ and $\pi_{kb}(y)$ ($\pi_{kb}(y+1)$) otherwise. Let us denote a predecessor of $z(x)$ by $pred(z(x))$ and a successor of $z(x)$ by $succ(z(x))$ in $\pi^{\downarrow(x,y)}_{kb}$.

In the insertion phase a single arc $((pred(z(x)), k), (succ(z(x)), k))$ in $\pi^{\uparrow(x)}_{kb}$ is replaced by two arcs $((pred(z(x)), k), (z(x), k))$ and $((z(x), k), (succ(z(x)), k))$ in $G\left(\pi^{\downarrow(x,y)}_{kb}\right)$. Taking into account the structure of $\pi^{\uparrow(x)}_{kb}$ and direction of added arcs, it is easy to observe that the insertion of node $(z(x), k)$ does not change any path incoming to nodes $(pred(z(x)), k)$ and it does not change any path outgoing from nodes $(succ(z(x)), k)$, therefore we obtain (16) and (17). From the similar observation we have $C^{\downarrow(x,y)}_{z(x),k-1} = C^{\uparrow(x)}_{z(x),k-1}$ and $Q^{\uparrow(x,y)}_{z(x),k+1} = Q^{\uparrow(x)}_{z(x),k+1}$. Finally, from Property 6, we have $C^{\downarrow(x,y)}_{z(x),k-1} = C_{z(x),k-1}$ and $Q^{\downarrow(x,y)}_{z(x),k+1} = Q_{z(x),k+1}$. □

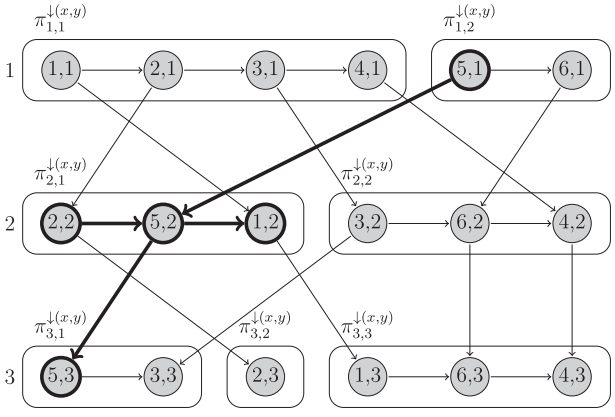

Fig. 3. Directed graph $G(\pi^{\downarrow(x,y)})$-insertion phase.

Fig. 3 shows the graph $G(\pi^{\downarrow(x,y)})$ for processing order $\pi^{\downarrow(x,y)}$ obtained from π by applying the move $v=(2, 2, 2, 1, 2)$. The inserted node and all adjacent and modified arcs are drawn in a bold line. Finally, the length of the longest path passing through the operation $z(x) = \pi_{ka}(x)$ after its insertion in the position y on the machine b in center k equals

$$L^{\downarrow(x,y)}_{zk} = \begin{cases} \max\left\{C^{\uparrow(x)}_{\pi_{kl}(y),k}, C_{z(x),k-1}\right\} + p_{z(x),k} + \max\left\{Q^{\uparrow(x)}_{\pi_{kl}(y+1),k}, Q_{z(x),k+1}\right\} & \text{for } l = h_k,\ y > x, \\ \max\left\{C^{\uparrow(x)}_{\pi_{kl}(y-1),k}, C_{z(x),k-1}\right\} + p_{z(x),k} + \max\left\{Q^{\uparrow(x)}_{\pi_{kl}(y),k}, Q_{z(x),k+1}\right\} & \text{otherwise}. \end{cases} \quad (21)$$

From the Property 4 we have

$$C_{\max}(\pi^{\downarrow(x,y)}) = \max\left\{L^{\downarrow(x,y)}_{z(x),k}, L^{\uparrow(x)}_{\max}\right\}, \quad (22)$$

where $L^{\uparrow(x)}_{\max} = \max\left\{L^{\uparrow(x)}_{jk} : j \neq z(x), j \in J\right\}$ is the length of the longest path in $G(\pi^{\uparrow(x)})$. Note that Eqs. (21) and (22) can be calculated in the time $O(1)$ for a single insertion and they require the computing time $O(n + m - (f_k - e_k))$ for all insertions of the operation.

Fig. 4 shows the essential fragment of pseudo-code of accelerator. In the former six steps the values of $C^{\uparrow(x)}_{*,k}$, $Q^{\uparrow(x)}_{*,k}$, $L^{\uparrow(x)}_{*,k}$ and $L^{\uparrow(x)}_{\max}$ are determined for all deletions. The execution time of lines 3, 5 and 6 is $O(d)$, where $d = (f_k - e_k + 1)$ is the number of jobs in block B_k, so the total execution time lines 1–6 is $O(nd)$. The time required to execute line 7 is $O(d)$ and line 8 is $O(nd)$.

The insertion phase that can be divided into two parts follows the same code. The first part is dedicated to machine h_k (lines 11–13), whereas the second to the remaining machines (lines 15–17). The parts differ in the range of insertion positions. In the former case the positions inside of the block are omitted (see Property 2). The main processing of insertion phase is performed in 12 (16) and 13(17) lines of code. Each of these lines requires $O(d)$ time and the total execution time of the second phase is $O(d(n + m - s))$.

4.3. The parallel accelerator

In this section, we are proposing a genuine method of parallelization of the advanced single neighborhood search method presented in the previous section. We focus on the vector processing

Phase I.
1. for $l = 1, \ldots, m_k$
2. for $r = 1, \ldots, n_{kl}$
3. for $x = e_k, \ldots, f_k$ determine $C^{\uparrow(x)}_{\pi_{kl}(r),k}$ from (12)
4. for $r = n_{kl}, \ldots, 1$
5. for $x = e_k, \ldots, f_k$ determine $Q^{\uparrow(x)}_{\pi_{kl}(r),k}$ from (13)
6. for $x = e_k, \ldots, f_k$ determine $L^{\uparrow(x)}_{\pi_{kl}(r),k}$ from (14)
7. for $x = e_k, \ldots, f_k$ do $L^{\uparrow(x)}_{\pi_{ka}(x),k} = -\infty$
8. for $x = e_k, \ldots, f_k$ do $L^{\uparrow(x)}_{\max} = \max\{L^{\uparrow(x)}_{j,k} : j \in J\}$

Phase II.
9. for $l = 1, \ldots, m_k$
10. if $l = h_k$
11. for $y = 1, \ldots, e_k, f_k, \ldots, n_{kl}$
12. for $x = e_k, \ldots, f_k$ determine $L^{\downarrow(x,y)}_{z(x),k}$ from (21)
13. for $x = e_k, \ldots, f_k$ determine $C_{\max}(\pi^{\downarrow(x,y)}) = \max\{L^{\downarrow(x,y)}_{z(x),k}, L^{\uparrow(x)}_{\max}\}$
14. else
15. for $y = 1, \ldots, n_{kl} + 1$
16. for $x = e_k, \ldots, f_k$ determine $L^{\downarrow(x,y)}_{z(x),k}$ from (21)
17. for $x = e_k, \ldots, f_k$ determine $C_{\max}(\pi^{\downarrow(x,y)}) = \max\{L^{\downarrow(x,y)}_{z(x),k}, L^{\uparrow(x)}_{\max}\}$

Fig. 4. Schema of the sequential accelerator.

model of parallel computations, which is the most demanding from developers' perspective and easy in realization on the hardware. The model selected is the special case of the single instruction multiple data (SIMD) model.

It is noteworthy to observe, that the proposed small-grained approach has a theoretical upper bound of the obtained speedups (so also a lower bound of parallel execution times). We are considering a parallel algorithm which employs a single process (thread) to guide the search. The thread performs, in a cyclic way, (iteratively) two leading tasks:

(i) goal function evaluation for a single solution or a set of solutions,
(ii) management, e.g. solution filtering and selection, collection of history, updating.

The part (ii) takes statistically up to 1–3% of the total iteration time in most metaheuristics, however the part (i) can be accelerated in a multithread environment. If (ii) takes β percentage of the 1-processor algorithm and if it is not parallelizable, the speedup of the parallel algorithm for *any* number of processors p cannot be greater than $\frac{1}{\beta}$ (according to Amdahl's law). In practice, if part (ii) takes 2% of the total execution time, the speedup can achieve at most the value of 50.

In parallel accelerator, similarly to sequential accelerator, computations are divided into two phases: deletion and insertion. In deletion phase, for a block B_k, $k = 1, \ldots, m$ variables $C^{\uparrow(x)}_{j,k}$, $Q^{\uparrow(x)}_{j,k}$, $L^{\uparrow(x)}_{j,k}$, $j \in J$ are determined in parallel. The block B_k constitutes a compact sub-sequence of operations, i.e. its positions are respectively $e_k, \ldots, f_k$ in the permutation π_{kh_k}. For each position $x = e_k, \ldots, f_k$ we assign one vector processor, i.e. processor number 0 is assigned to the position e_k, processor 1 is assigned to $e_k + 1$, etc. In each step of vector machine computations, calculations for single job j and single machine k are performed. For this reason, we call this phase job-oriented vectoring processing.

In the second phase (insertion), generally all deleted operations are inserted in all positions in all permutations which define processing order in a center k (see the definition of a set of moves). Therefore, for each machine in a center k, each operation is inserted into identical positions, i.e. an operation from the position e_k is inserted in position 1, $e_k + 1$ is inserted in position 1, $e_k + 2$ is inserted in position 1,..., f_k is inserted in position 1; e_k is inserted in position 2, $e_k + 1$ is inserted in position 2, and so on, what can be directly parallelized with using a number of vector processors connected with target positions. In each step of a vector machine computations, the computation of $C^{\downarrow(x,y)}_{z(x),k}$, $Q^{\downarrow(x,y)}_{z(x),k}$ or $L^{\downarrow(x,y)}_{z(x),k}$ is performed for the single position y on a machine in the center k. For this reason, we call this phase position-oriented vectoring processing.

Note that the process of mapping of scalar values to the vectors is developed especially for maximizing of the efficiency of vectoring processing and minimizing of the frequency of access to variables from outside of vectors performed.

The quick overview of pseudo code of accelerator suggests, that parallelization sequential accelerator is very simple. It seems that the replacement of the loop *for* $x = e_k, \ldots, f_k$ by concurrent processing for $x = e_k, \ldots, f_k$ is sufficient. Unfortunately, the main limitation related to parallel vector processing significantly complicates utilization of this type of parallel processing. In vectoring parallel processing all processors execute the same instruction of code in the same tick of computing, some exceptions appearing for some elements of vector require scalar correction.

Let

$$\overline{C}^{\uparrow}_{r,k} = \left(C^{\uparrow(e_k)}_{r,k}, \ldots, C^{\uparrow(f_k)}_{r,k}\right), \quad \overline{Q}^{\uparrow}_{r,k} = \left(Q^{\uparrow(e_k)}_{r,k}, \ldots, Q^{\uparrow(f_k)}_{r,k}\right),$$
$$\overline{L}^{\uparrow}_{r,k} = \left(L^{\uparrow(e_k)}_{r,k}, \ldots, L^{\uparrow(f_k)}_{r,k}\right),$$

be the vectors of variables corresponding to deletion phase and $\overline{L}^{\downarrow(y)}_{z,k} = \left(L^{\downarrow(e_k,y)}_{z(e_k),k}, \ldots, L^{\downarrow(f_k,y)}_{z(f_k),k}\right)$, $\overline{C}_{\max}(\pi^{\downarrow(y)}) = \left(\overline{C}_{\max}(\pi^{\downarrow(e_k,y)}), \ldots, \overline{C}_{\max}(\pi^{\downarrow(f_k,y)})\right)$, be the vectors of variables corresponding to insertion phase of computing.

Property 8. *For a fixed block B_k in a center $k \in M$ represented by the triple (h_k, e_k, f_k), let $C^{\uparrow(x)}_{jk}$, $Q^{\uparrow(x)}_{jk}$, $L^{\uparrow(x)}_{jk}$ and $L_{max}(\pi^{\uparrow(x)})$, $x = e_k, \ldots, f_k$ be proper values calculated after a deletion of an operation $\pi_{kh_k}(x)$. All these values can be found on vector machine with $d = f_k - e_k + 1$ processors in the time $O(n + d)$.*

Proof. Let

$$\overline{C}^{\uparrow}_{\pi_{kl}(r),k} = \max\left\{\overline{C}^{\uparrow}_{\pi_{kl}(r-1),k}, \overline{C}_{\pi_{kl}(r),k-1}\right\} + \bar{p}_{\pi_{kl}(r),k} \tag{23}$$

where

$$\overline{C}_{*,*} = \left(C^{(e_k)}_{*,*}, \ldots, C^{(f_k)}_{*,*}\right), \quad C^{(s)}_{*,*} = C_{*,*}, \bar{p}_{*,*} = \left(p^{(e_k)}_{*,*}, \ldots, p^{(f_k)}_{*,*}\right),$$
$$p^{(s)}_{*,*} = p_{*,*}, s = e_k, \ldots f_k,$$

be a vector version of (12). The maximum operator is calculated for each dimension of vectors. For $l = h_k$ and $r = x + 1, x = e_k, \ldots, f_k$, the vector processing requires scalar correction of element $C^{\uparrow(x)}_{\pi_{kl}(r-1),k}$. One should take $C^{\uparrow(x)}_{\pi_{kl}(r-2),k}$, instead $C^{\uparrow(x)}_{\pi_{kl}(r-1),k}$, i.e., the value determined two iterations back. Now, we provide similar consideration for $Q^{\uparrow(x)}_{*,*}$ and $L^{\uparrow(x)}_{*,*}$ values. The vector version of (13) and (14) are

$$\overline{Q}^{\uparrow}_{\pi_{kl}(r),k} = \max\left\{\overline{Q}^{\uparrow}_{\pi_{kl}(r+1),k}, \overline{Q}_{\pi_{kl}(r),k+1}\right\} + \bar{p}_{\pi_{kl}(r),k} \tag{24}$$

and

$$\overline{L}^{\uparrow}_{j,k} = \overline{C}^{\uparrow}_{j,k} + \overline{Q}^{\uparrow}_{j,k} - \bar{p}_{j,k} j \neq z, \quad j \in J. \tag{25}$$

For $l = h_k$ and $r = x - 1$, $x = e_k, \ldots, f_k$, the vector processing requires scalar correction of element $Q^{\uparrow(x)}_{\pi_{kl}(r),k}$. Summarizing, for determining all values of deletion phase the computation requires $O(n)$ steps of vectoring processing and $O(s)$ scalar correction. □

Property 9. *For a fixed block B_k in a center $k \in M$ represented by the triple (h_k, e_k, f_k), let $L^{\downarrow(x,y)}_{z(x),k}, x = e_k, \ldots, f_k, y = 1, \ldots, n_{ki} + 1$ for $l \neq h_k$, $l \in M_k$ and $y = 1, \ldots, e_k, f_k, \ldots, n_{kh_k}$ for $l = h_k$ be the length of the lon-*

Phase I.
1. for $l = 1, ..., m_k$
2. for $r = 1, ..., n_{kl}$
3. determine $\overline{C}^{\uparrow}_{\pi_{kl}(r),k}$ from (23)
3a. if $l = h_k$ and $e_k < r < f_k$ correct $C^{\uparrow(r)}_{\pi_{kl}(r),k}$ from (12)
4. for $r = n_{kl}, ..., 1$
5. determine $\overline{Q}^{\uparrow}_{\pi_{kl}(r),k}$ from (24)
5a. if $l = h_k$ and $e_k < r < f_k$ correct $Q^{\uparrow(r)}_{\pi_{kl}(r),k}$ from (13)
6. determine $\overline{L}^{\uparrow}_{\pi_{kl}(r),k}$ from (25)
7. for $x = e_k, ..., f_k$ do $L^{\uparrow(x)}_{\pi_{ka}(x),k} = -\infty$
8. determine $\overline{L}^{\uparrow}_{\max} = \max\{\overline{L}^{\uparrow}_{j,k} : j \in J\}$

Phase II.
9. for $l = 1, ..., m_k$
10. if $l = h_k$
11. for $y = 1, ..., e_k, f_k, ..., n_{kl}$
12. determine $\overline{L}^{\downarrow(y)}_{z,k}$ from (26)
13. determine $\overline{C}_{\max}(\pi^{\downarrow(y)}) = \max\{\overline{L}^{\downarrow(y)}_{z,k}, L^{\uparrow}_{\max}\}$
14. else
15. for $y = 1, ..., n_{kl} + 1$
16. determine $\overline{L}^{\downarrow(y)}_{z,k}$ from (26)
17. determine $\overline{C}_{\max}(\pi^{\downarrow(y)}) = \max\{\overline{L}^{\downarrow(y)}_{z,k}, L^{\uparrow}_{\max}\}$

Fig. 5. Schema of the parallel accelerator.

gest path passing through operation $O_{z(x),k}, z(x) = \pi_{kh_k}(x)$ *after its insertion in the position* y *on machine* $b \in M_k$. *All these values can be found on the vector machine with* $d = f_k - e_k + 1$ *processors in the time* $O(n + m - d)$.

Proof. Let

$$\overline{L}_{zk}^{\downarrow(y)} = \begin{cases} \max\left\{\overline{C}^{\uparrow}_{\pi_{kl}(y),k}, \overline{C}_{z,k-1}\right\} + \bar{p}_{z,k} + \max\left\{\overline{Q}^{\uparrow}_{\pi_{kl}(y+1),k}, \overline{Q}_{z,k+1}\right\} & \text{for } l = h_k,\ y > x, \\ \max\left\{\overline{C}^{\uparrow}_{\pi_{kl}(y-1),k}, \overline{C}_{z,k-1}\right\} + \bar{p}_{z,k} + \max\left\{\overline{Q}^{\uparrow}_{\pi_{kl}(y),k}, \overline{Q}_{z,k+1}\right\} & \text{otherwise.} \end{cases} \tag{26}$$

where

$$\overline{C}_{z,*} = (C_{z(e_k),*}, \ldots, C_{z(f_k),*}), \quad \overline{Q}_{z,*} = (Q_{z(e_k),*}, \ldots, Q_{z(f_k),*}), \quad \bar{p}_{z,*} = (p_{z(e_k),*}, \ldots, p_{z(f_k),*})$$

be a vector version of (21). In this case, the vectoring processing does not require any scalar correction. The total number of insertion is $n + m - 1 - d$, so the total computation time of vectoring machine is $O(n + m - d)$. □

The time required for the goal function value determination for neighbor solutions for $d = f_k - e_k + 1$ jobs of a block, for the sequential accelerator, is $O(d(n + m - d))$. For the parallel accelerator it is $O(n + m - d)$, therefore the theoretical speedup equals $O(d)$. Moreover, in the sequential accelerator, the computation time required to determine $C_{\max}()$ values for solutions from the neighborhood equals $O(1)$ per each solution. So, this time equals $O(1/d)$ per each solution in the parallel accelerator.

The Figure 5 shows pseudo code of parallel version of accelerator. For the given processing order π and given block B_k on machine h_k in stage k, the values of $C_{\max}()$ function for all insertions of the jobs from block B_k are computed. In the case of sequential version, the results of computations are stored in multidimensional matrix of integer values, whereas in the case of parallel version, in multidimensional matrix of vectors.

5. Computational experiments

We have implemented two main versions of the tabu search algorithm: the pure sequential *sTS* and the parallelized *pTS*. The *sTS* and *pTS* vary in method of neighborhood computation. The *pTS* algorithm uses the parallel accelerator to speed up the computation and similarly to *sTS* belongs to the class of single-walk algorithms. Additionally, we have implemented multiple-walk versions of these algorithms called *msTS* and *mpTS*.

Programs were executed on the Compaq 8510w Mobile Workstation PC with Intel Core 2 Duo 2.60 GHz processor and the Microsoft Windows XP operating system and compiled with Microsoft Visual C++ 2005. For vector computations we have used SSE2 (Streaming Single Instruction, Multiple Data Extensions 2) set of instructions. There can be distinguished three groups of vector instructions operated on the vector: 8×2, 4×4, 2×8, where the first number denotes the size of vector and the second the size of data in bytes. Unfortunately, one of the most frequently performed operations in the *pTS* algorithm, the function maximum, is computed only on 8-bit and 16-bit integers.

The execution of SSE2 instruction is managed inside the processor, therefore, setups and delays are negligible, moreover the execution time of these instructions is comparable with corresponding scalar instructions. To execute one *pTS* algorithm, we use only one core of processor. We considered the distribution of computation on many cores. However, preliminary tests showed that the synchronization of calculations performed by the operating system slows down the algorithm significantly.

The algorithms were tested on 110 benchmark instances created on the basis of Taillard tests (Taillard, 1993) for the flow shop problem. The benchmark set contains 11 groups of 'hard' instances of different sizes $n \times m$: 20×5, 20×10, 20×20, 50×5, 50×10, 50×20, 100×5, 100×20, 200×10, 200×20. The group 500×20 of original examples with the largest number of tasks has been omitted, because the 16-bit integers are not sufficient to encode instance data, especially completion times. The centers have a fixed uniform number of machines $m_k = c$, $k \in M$, $c = 2, 3, 4$. This assumption does not help to solve the problem, however, it allows us to keep the hardness of Tailard's instances in the case of hybrid flow shop problem. It is easy to see, that as in the original examples, no center will be a bottleneck.

The initial solution for the proposed algorithms can be found by any method. For our needs we use the algorithm based on the list scheduling. We considered especially two ways of creating lists: based on LPT-rule (longest processing time) ($L(LPT)$) and based on the sequence of job generated by NEH algorithm for permutation flow shop problem ($L(NEH)$). The assignment of operation to the machines was made on the principle: assign machine released at the earliest. The simple diversification method is used in the *pTS* algorithm. The algorithm restarts, whenever *nimp* iterations were performed without improving the best solution found so far. For each restart the initial solution is generated by list scheduling algorithm. The list for this algorithm is generated randomly.

In multiple-walk version of algorithms, each thread of *sTS* or *pTS* starts from a different initial solution. All threads independently (asynchronously) update the best solution found so far. Whenever a new best solution is found, the number *nimp* of corresponding threads is increased 3-times. We perform two computational tests of algorithms. The main purpose of the first of them is to determine speedup of parallel algorithms, while the purpose of the second is to evaluate quality of solutions generated by *TS* algorithms.

Table 2
Results of comparison of execution times of algorithms.

Group	m_k	*CPU*(*pTS*)	*CPU*(*sTS*)	Speedup	*APU*	*ABS*
20×5	2	0.22	0.48	2.2	2.9	3.0
20×10	2	0.34	0.69	2.0	1.9	1.9
20×20	2	0.66	1.25	1.9	1.5	1.5
50×5	2	0.50	1.68	3.4	4.4	6.1
50×10	2	0.83	2.32	2.8	3.4	3.7
50×20	2	1.43	3.38	2.4	2.1	2.2
100×5	2	1.12	5.69	5.1	5.9	12.5
100×10	2	1.61	6.29	3.9	4.4	6.3
100×20	2	2.90	8.99	3.1	3.2	3.8
200×10	2	3.67	19.45	5.3	5.7	11.0
200×20	2	7.02	29.45	4.2	4.6	7.3
20×5	3	0.22	0.45	2.0	2.3	2.3
20×10	3	0.37	0.70	1.9	1.6	1.6
20×20	3	0.70	1.23	1.8	1.4	1.4
50×5	3	0.50	1.53	3.1	3.8	4.6
50×10	3	0.81	2.12	2.6	2.6	2.8
50×20	3	1.58	3.65	2.3	2.2	2.3
100×5	3	0.98	4.35	4.4	4.7	7.9
100×10	3	1.59	5.54	3.5	3.8	4.6
100×5	3	3.09	8.70	2.8	2.9	3.6
200×10	3	3.90	18.72	4.8	5.0	9.3
200×20	3	7.88	31.75	4.0	5.3	11.0
20×5	4	0.20	0.41	2.1	1.8	1.8
20×10	4	0.42	0.70	1.7	1.7	1.7
20×20	4	0.76	1.25	1.6	1.3	1.3
50×5	4	0.47	1.18	2.5	2.9	3.0
50×10	4	0.84	1.89	2.3	2.2	2.2
50×20	4	1.68	3.48	2.1	2.7	2.7
100×5	4	0.92	3.40	3.7	4.5	5.8
100×10	4	1.90	6.47	3.4	4.2	6.4
100×5	4	3.42	10.05	2.9	4.2	5.7
200×10	4	4.74	22.76	4.8	5.7	13.4
200×20	4	7.93	30.59	3.9	5.1	10.9

5.1. Speedup measures

In the first test, the algorithms were terminated after performing 10,000 iterations and started for identical initial solution i.e. all threads perform identical path of search. Based on two core processor we execute two threads. For each tested instance and for each run of algorithms, we collected the following values:

- $CPU(A)$ – total computation time for the algorithm $A \in \{sTS, pTS, msTS, mpTS\}$,
- ABS – average block size,
- APU – average number of elements of vector processor.

Based on the CPU values we calculate the speedup ratio as $CPU(sTS)/CPU(pTS)$ for comparison execution times of two single-walk algorithms. Table 1 shows the results of comparison between the algorithms. Next, $2 \cdot CPU(mA)/CPU(A)$ speedup ratio measurement is determined for comparison execution times of multiple-walk mA and single-walk algorithm A, $mA \in \{msTS, mpTS\}$, where $A \in \{sTS, pTS\}$. The second of speedup ratios for all instances was close to 2.2 for pair ($mpTS$, pTS) and for pair ($msTS$, sTS).

The main observation is that the proposed method of parallelization reduces significantly the computation time on the single multi-core processor.

The superiority of pTS (parallelized) over sTS (non-parallelized) becomes more evident for instances with big average block size, especially for the instances with big jobs number, small number of centers and small number of machines in each center.

For the small instances the benefit of parallelization is small and close to 2, whereas for the big instances we observe the speedup values which are achieving 5.3. This means that the parallel algorithm calculates neighborhoods up to 5.3 times faster. The speedup increases with the increasing number of jobs and decreases with the number of centers as well as the number of machines in centers.

5.2. Performance measures

In the second test, the pTS and $mpTS$ algorithms were run with the limit of 100,000 iterations and $nimp = 3000$. The initial solution for the pTS algorithm and for the first thread of $mpTS$ algorithm were generated by $L(NEH)$ algorithm, while the initial solution for the second thread of $mpTS$ algorithm was generated by $L(LPT)$.

For each instance of the problem we determine the best solution found by tested algorithms and its makespan. Thus, based on the reference values obtained in this way, the measure of the algorithm quality was calculated.

$PRD(H) = 100\%(C^H - C^*)/C^*$–the value of the percentage relative difference between makespan C^H and reference value C^*, where C^H is the makespan of solution generated by algorithm $H \in \{L(NEH), L(LPT), pTS, mpTS\}$.

The results of averaged in groups PRD values are given in Table 2. Additionally, we can observe the behavior of PRD values in iterations (1000, 50,000, 100,000) for TSW algorithms. (See Table 3).

The results prove that TS algorithms converge to good solutions very fast. The PRD values for list scheduling algorithms are not less

Table 3
Results of PRD values.

Group	m_k	$L(LPT)$	$L(NEH)$	pTS			$mpTS$		
				1000	50,000	100,000	1000	50,000	100,000
20×5	2	29.56	10.65	2.42	1.15	1.08	2.42	0.41	0.06
20×10	2	26.16	7.75	1.10	0.04	0.04	1.10	0.04	0.04
20×20	2	13.97	3.61	0.26	0.10	0.10	0.26	0.10	0.10
50×5	2	19.49	3.86	0.62	0.03	0.03	0.62	0.03	0.03
50×10	2	23.11	5.56	0.62	0.01	0.01	0.62	0.01	0.01
50×20	2	21.46	4.61	0.48	0.00	0.00	0.48	0.00	0.00
100×5	2	13.21	2.21	0.11	0.01	0.01	0.11	0.01	0.01
100×10	2	17.48	3.07	0.34	0.00	0.00	0.34	0.00	0.00
100×5	2	16.02	3.02	0.12	0.00	0.00	0.12	0.00	0.00
200×10	2	14.70	1.68	0.04	0.00	0.00	0.04	0.00	0.00
200×20	2	16.00	1.99	0.09	0.00	0.00	0.09	0.00	0.00
Average		19.20	4.36	0.56	0.12	0.12	0.56	0.05	0.02
20×5	3	31.13	15.03	2.93	0.15	0.15	2.93	0.00	0.00
20×10	3	23.89	9.94	1.65	0.41	0.04	1.65	0.36	0.00
20×20	3	12.26	4.25	1.30	1.00	0.45	1.30	0.63	0.00
50×5	3	21.73	6.59	1.24	0.18	0.18	1.24	0.18	0.00
50×10	3	23.56	8.34	1.51	0.02	0.02	1.51	0.02	0.02
50×20	3	19.71	5.65	0.14	0.01	0.01	0.14	0.01	0.01
100×5	3	14.57	3.80	0.55	0.02	0.02	0.55	0.02	0.02
100×10	3	17.51	4.31	0.38	0.02	0.02	0.38	0.02	0.02
100×5	3	15.82	4.61	0.37	0.00	0.00	0.37	0.00	0.00
200×10	3	14.83	2.53	0.16	0.00	0.00	0.16	0.00	0.00
200×20	3	13.42	1.03	0.00	0.00	0.00	0.00	0.00	0.00
Average		18.95	6.01	0.93	0.16	0.08	0.93	0.11	0.01
20×5	4	31.16	17.81	3.86	0.86	0.78	3.86	0.27	0.12
20×10	4	21.38	9.07	2.37	0.72	0.11	2.37	0.72	0.04
20×20	4	10.46	4.00	1.22	0.77	0.25	1.22	0.33	0.00
50×5	4	22.42	8.38	1.44	0.00	0.00	1.44	0.00	0.00
50×10	4	23.24	8.94	0.70	0.07	0.07	0.70	0.03	0.03
50×20	4	16.55	4.56	0.00	0.00	0.00	0.00	0.00	0.00
100×5	4	15.01	4.65	0.56	0.13	0.13	0.56	0.01	0.01
100×10	4	18.56	5.99	1.05	0.01	0.01	1.05	0.01	0.01
100×5	4	14.81	4.27	0.13	0.00	0.00	0.13	0.00	0.00
200×10	4	15.05	3.43	0.29	0.00	0.00	0.29	0.00	0.00
200×20	4	12.56	1.11	0.00	0.00	0.00	0.00	0.00	0.00
Average		18.29	6.56	1.06	0.23	0.12	1.06	0.12	0.02

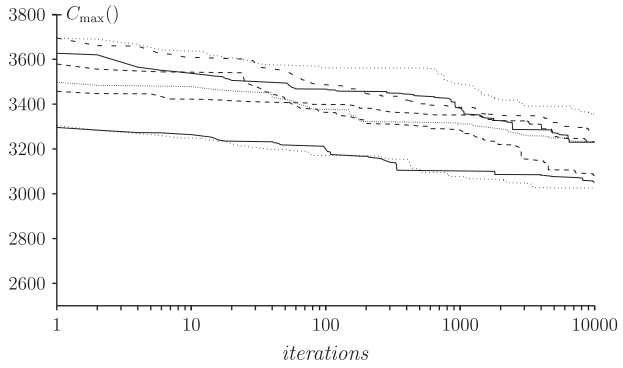

Fig. 6. Typical run of eight unrelated treads of algorithm.

than 10% for L (LPT) and 1% for L (NEH) and for some groups the PRD values achieve even 30% L (LPT) and 17% L (NEH). It shoud be noticed, however, that the L (NEH) algorithm generates significantly better solution than the L (LPT) algorithm.

The most significant part of the improvement occurs within the first 1000 iterations. The PRD values, for this number of iterations, is not greater than 4% for all groups of instances and 1.5% for the instances of practical number of jobs, i.e. 50 and more. The multiple-walk version of *TS* algorithm outperforms single-walk version. The advantages of the cooperation of many threads of *mpTS* algorithm are particulary evident for the groups of instances with small number of jobs.

Researching the convergence of the method proposed, one can observe a typical work of the parallel *mpTS* algorithm which is presented on the Fig. 6. It is visible that proceeding of parallel working threads (as minimum of values of obtained by each thread) almost always gives much better result than the execution of single searching thread. Moreover, such a parallel algorithm is a proof against starting solution quality, what is the main disadvantage of sequential tabu search algorithms.

6. Conclusions

The new small-grain parallel tabu search algorithm for the flexible flow shop problem was proposed in this paper. We designed genuine problem properties which make it possible to obtain good efficiency and speedup of the algorithm. The proposed kind of parallelism could be easily applied in the new generation of multi-core processors as well as in vector processors (e.g. with the extended set of instructions, such as MMX and SSE2).

A natural direction of the future work could be focused on multiple-walk parallel metaheuristics (see Alba, 2005; James, Rego, & Glover, 2005), which executes a number of cooperating algorithms. In this way one could create an efficient distribution of algorithms designed for execution in large clusters or grids equipped with GPUs and multi-core processors.

Acknowledgement

The work was supported by the OPUS grant DEC-2012/05/B/ST7/00102 of Polish National Centre of Science.

References

Alba, E. (2005). *Parallel metaheuristics: A new class of algorithms*. Wiley & Sons Inc..
Azizoğlu, M., Çakmak, E., & Kondakci, S. (2001). A flexible flowshop problem with total flow time minimization. *European Journal of Operational Research, 132*, 528–538.
Bożejko, W. (2012). On single-walk parallelization of the job shop problem solving algorithms. *Computers & Operations Research, 39*, 2258–2264.
Bożejko, W. (2009). Solving the flow shop problem by parallel programming. *Journal of Parallel and Distributed Computing, 69*, 470–481.
Bożejko, W., Pempera, J., & Smutnicki, C. (2009). Parallel simulated annealing for the job shop scheduling problem. *Lecture Notes in Computer Science* (Vol. 5544, pp. 631–640). Springer.
Grabowski, J., Skubalska, E., & Smutnicki, C. (1983). On flow shop scheduling with release and due dates to minimize maximum lateness. *Journal of the Operational Research Society, 34*, 7, 615–62.
James, T., Rego, C., & Glover, F. (2005). Sequential and parallel path-relinking algorithms for the quadratic assignment problem. *IEEE Intelligent Systems, 20*(4), 58–65.
Khademi Zare, H., & Fakhrzad, M. B. (2011). Solving fexible fow-shop problem with a hybrid genetic algorithm and data mining: A fuzzy approach. *Expert Systems with Applications, 38*, 7609–7615.
Nowicki, E., & Smutnicki, C. (1998). The flow shop with parallel machines: A tabu search approach. *European Journal of Operational Research, 106*, 226–253.
Ruiz, R., & Vázquez-Rodríguez, J. A. (2010). The hybrid flow shop scheduling problem. *European Journal of Operational Research, 205*(1), 1–18.
Sawik, T. (1993). A scheduling algorithm for flexible flow lines with limited intermediated buffers. *Applied Stochastic Models and Data Analysis, 9*, 127–138.
Sawik, T. (1999). *Production planning and scheduling in flexible assembly systems*. Berlin: Springer.
Shahvari, O., Salmasi, N., Logendran, R., & Abbasi, B. (2012). An efficient tabu search algorithm for flexible flow shop sequence-dependent group scheduling problems. *International Journal of Production Research, 50*(15), 4237–4254.
Steinhöfel, K., Albrecht, A., & Wong, C. K. (2002). Fast parallel heuristics for the job shop scheduling problem. *Computers and Operations Research, 29*, 151–169.
Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research, 64*, 278–285.
Wittrock, R. (1988). An adaptable scheduling algorithm for flexible flow lines. *Operations Research, 36*, 445–453.