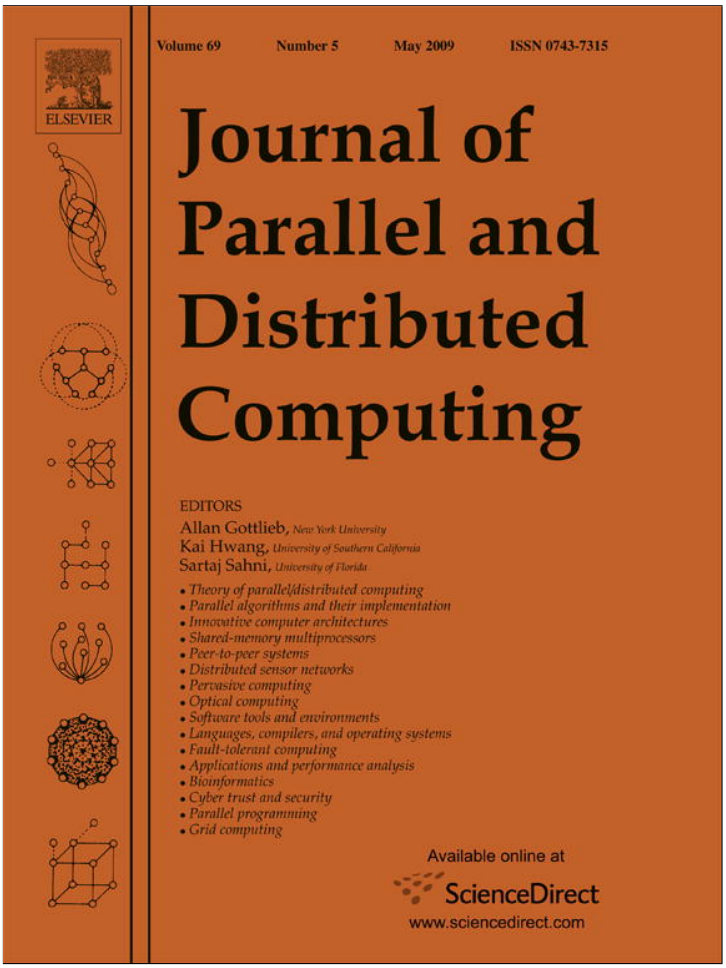

J. Parallel Distrib. Comput. 69 (2009) 470–481

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Solving the flow shop problem by parallel programming

Wojciech Bożejko

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

ARTICLE INFO

Article history:
Received 12 January 2008
Received in revised form
14 January 2009
Accepted 26 January 2009
Available online 11 February 2009

Keywords:
Parallel algorithm
Flow shop problem
PRAM

ABSTRACT

The matter of using scheduling algorithms in parallel computing environments is discussed in this paper. There are proposed methods of parallelizing the criterion function calculations for a single solution and a group of concentrated solutions (local neighborhood) dedicated to being used in metaheuristic approaches. Also a parallel scatter-search metaheuristic is proposed as a multiple-thread approach. Computational experiments are done for the flow shop, the classic NP-hard problem of the combinatorial optimization.

1. Introduction

We take into consideration the permutation flow shop scheduling problem described as follows. A number of jobs is to be processed on a number of machines. Each job must go through all the machines in exactly the same order and the job order must be the same on each machine. Each machine can process at most one job at any point of time and each job may be processed on at most one machine at any time. The objective is to find a schedule minimizing the sum of job's completion times ($F \parallel C_{sum}$ problem) or maximal job completion time ($F \parallel C_{max}$ problem). Both problems are strongly NP-hard. Since the known exact algorithms own, necessarily, exponential computational complexity, then even a significant increase of computer power will result in an incomparably small increase of the size of instances we can solve. Thus, there exist two, not mutually conflicting, approaches which allow one to solve large-size instances in an acceptable time: (1) approximate methods (mainly metaheuristics), (2) parallel methods.

Scheduling and assignment problems are almost always parallelized by multiple-threads methods, such as parallel tabu search of Taillard for the QAP [13] and for the job shop problem [14], parallel scatter-search of James et al. [7] for the QAP and parallel genetic algorithm of Bożejko and Wodecki [3] for the single machine total weighted tardiness scheduling problem. These implementations do not parallelize the cost function – the most expensive element of computations – but execute multiple-working, cooperative or independent, metaheuristic threads (maybe except a parallel genetic algorithm where a process of population evaluation can be also parallelized). The method of cost function computing in parallel is strongly connected with this problem; Steinhöfel et al. [12] propose a method, based on a parallel Floyd–Warshall algorithm, of parallel cost function computing for the job shop problem.

Genuine methods of parallel cost function computing for a single solution and a group of local solutions (neighborhood) are presented in a paper for the flow shop problem with C_{max} and C_{sum} criteria. These methods concern dissimilar techniques of a parallel algorithm's projecting process as well as different necessities of modern algorithms of discreet optimization (analysis of one solution, analysis of a local neighborhood). Efficiency, cost and computation speedup depending on type of the problem, its size and the environment of the parallel system used are considered especially in this part of the paper. Theoretical estimations of properties are derived for particular algorithms, and comparative analysis of the advantages resulting from applications of different approaches has been conducted.

In the matter of computational experiments, dedicated to homogeneous and heterogeneous multiprocessors systems (such as mainframe computers, clusters, and diffuse systems connected by networks), a vector processing based on a MMX instruction set has been projected and researched experimentally in the application of flow shop scheduling problems. Also, a parallel variant of the scatter-search method, one of the most promising current methods of combinatorial optimization, has been projected and researched experimentally, in the application of flow shop scheduling problems with C_{max} and C_{sum} criteria. In some cases the effect of orthodox superlinear speedup has been observed. Although algorithms have not been designed for obtaining extremely good solutions, some new the best solutions have been obtained for the C_{sum} flow shop problem for benchmark instances of Taillard [15].

E-mail address: wojciech.bozejko@pwr.wroc.pl.

0743-7315/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.01.009

2. Problems

The flow shop problem with makespan criterion. We consider the well-known in the scheduling theory strongly NP-hard problem called the permutation flow-shop problem with the makespan criterion denoted by $F \parallel C_{\max}$. Skipping consciously the long list of papers dealing with this subject, we only refer the recent reviews and the best up-to-now algorithms [8,6,9] to the reader.

This problem has been introduced as follows. There are n jobs from a set $J = \{1, 2, \ldots, n\}$ to be processed in a production system having m machines, indexed by $1, 2, \ldots, m$, organized in the line (sequential structure). A single job reflects one final product (or sub-product) manufacturing. Each job is performed in m subsequent stages, in a common way for all tasks. Stage i is performed by machine i, $i = 1, \ldots, m$. Each job $j \in J$ is split into a sequence of m operations $O_{1j}, O_{2j}, \ldots, O_{mj}$ performed on machines in turn. Operation O_{ij} reflects processing job j on the machine i with the processing time $p_{ij} > 0$. Once started a job cannot be interrupted. Each machine can execute at most one job at a time; each job can be processed on at most one machine at a time.

The sequence of loading jobs into a system is represented by a permutation $\pi = (\pi(1), \ldots, \pi(n))$ on the set J. The optimization problem is to find the optimal sequence π^* so that

$$C_{\max}(\pi^*) = \min_{\pi \in \Pi} C_{\max}(\pi) \tag{1}$$

where $C_{\max}(\pi)$ is the makespan for permutation π and Π is the set of all permutations. Denoting by C_{ij} the completion time of job j on the machine i we have $C_{\max}(\pi) = C_{m,\pi(n)}$. Values C_{ij} can be found by using either the recursive formula

$$C_{i\pi(j)} = \max\{C_{i-1,\pi(j)}, C_{i,\pi(j-1)}\} + p_{i\pi(j)}, \quad i = 1, 2, \ldots, m, j = 1, \ldots, n, \tag{2}$$

with initial conditions $C_{i\pi(0)} = 0$, $i = 1, 2, \ldots, m$, $C_{0\pi(j)} = 0$, $j = 1, 2, \ldots, n$, or the non-recursive one

$$C_{i\pi(j)} = \max_{1=j_0 \le j_1 \le \cdots \le j_i = j} \sum_{s=1}^{i} \sum_{k=j_{i-1}}^{j_i} p_{s\pi(k)}. \tag{3}$$

The computational complexity of (2) is $O(mn)$, whereas for (3) is $O(\binom{j+i-2}{i-1}(j+i-1)) = O(\frac{(n+m)^{n-1}}{(n-1)!})$. In practice the former formula has been commonly used.

It should be noted that the problem of transforming the sequential algorithm for scheduling problems into a parallel one is nontrivial because of the strongly sequential character of computations carried out by (2) and by other known scheduling algorithms.

The flow shop problem with C_{sum} criterion. The objective is to find a schedule minimizing the sum of a job's completion times. The problem is indicated by $F \parallel C_{\text{sum}}$. There are plenty of good heuristic algorithms for solving the $F \parallel C_{\max}$ flow shop problem, with the objective of minimizing a maximal job's completion time. For the sake of special properties (blocks of critical path, [6]) it is recognized as an easier one than a problem with objective C_{sum}. Unfortunately, there are no similar properties (which can speed up computations) for the $F \parallel C_{sum}$ flow shop problem. Constructive algorithms (LIT and SPD from [16]) possess low efficiency and can only be applied to a limited range. There is a hybrid algorithm in [11] consisting of elements of tabu search, simulated annealing and path relinking methods. The results of this algorithm, applied to Taillard benchmark tests [15], are the best known ones in the literature nowadays.

The flow shop problem along with the sum of a job's completion time criterion can be formulated, taking notations from the previous paragraph, in the following way: we wish to find a permutation $\pi^* \in \Pi$ that:

$$C_{\text{sum}}(\pi^*) = \min_{\pi \in \Pi} C_{\text{sum}}(\pi), \quad \text{where } C_{\text{sum}}(\pi) = \sum_{j=1}^{n} C_{m\pi(j)},$$

where $C_{i\pi(j)}$ is the time required to complete the job j on the machine i in the processing order given by the permutation π. The completion time of job $\pi(j)$ on the machine m can be found by using the same formulas (2) or (3) as in the problem with a makespan criterion.

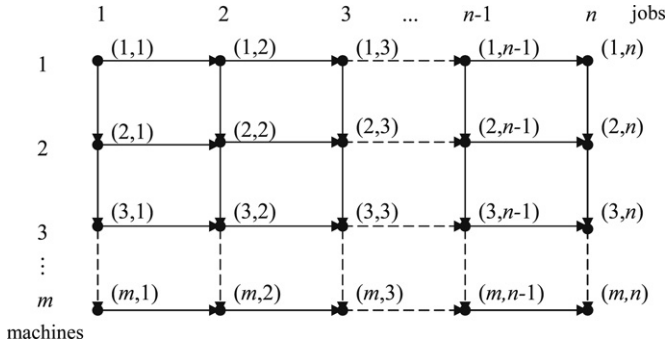

Fig. 1. Graph $G(\pi)$.

2.1. Graph model

Values C_{ij} from Eqs. (2) and (3) can be also determined using a graph model of the flow shop problem. For a given sequence of a job's execution $\pi \in \Pi$ we create a graph $G(\pi) = (M \times N, F^0 \cup F^*)$, where $M = \{1, 2, \ldots, m\}, N = \{1, 2, \ldots, n\}$.

$$F^0 = \bigcup_{s=1}^{m-1} \bigcup_{t=1}^{n} \{((s, t), (s+1, t))\} \tag{4}$$

is a set of technological arcs (vertical) and

$$F^* = \bigcup_{s=1}^{m} \bigcup_{t=1}^{n-1} \{((s, t), (s, t+1))\} \tag{5}$$

is a set of sequencing arcs (horizontal). Arcs of the graph $G(\pi)$ have no weights, but each vertex (s, t) has as weight $p_{s,\pi(t)}$. A time C_{ij} of finishing a job $\pi(j)$, $j = 1, 2, \ldots, n$ on the machine i, $i = 1, 2, \ldots, m$ equals the length of the longest path from the vertex (1,1) to the vertex (i, j), including the weight of the last one. For the $F \parallel C_{\max}$ problem, the value of the criterion function for fixed sequence π equals the length of the critical path in the graph $G(\pi)$ (see Fig. 1). For the $F \parallel C_{\text{sum}}$ problem the value of the criterion function is the sum of length of the longest paths which begins from the vertex (1, 1) and ends on vertexes $(m, 1), (m, 2), \ldots, (m, n)$.

3. Single-thread search

We consider an algorithm which uses a single process (thread) to guide the search. The thread performs in a cyclic way (iteratively) two leading tasks: (A) goal function evaluation for single solution or a set of solutions, (B) management, e.g. solution filtering and selection, collection of history, updating. The part (B) takes statistically 1%–3% total iteration time, then its acceleration is useless. The part (A) can be accelerated in a parallel environment in various manners — our aim is to find either a *cost optimal* method or a non-optimal one in the cost sense offering *the shortest running time*.

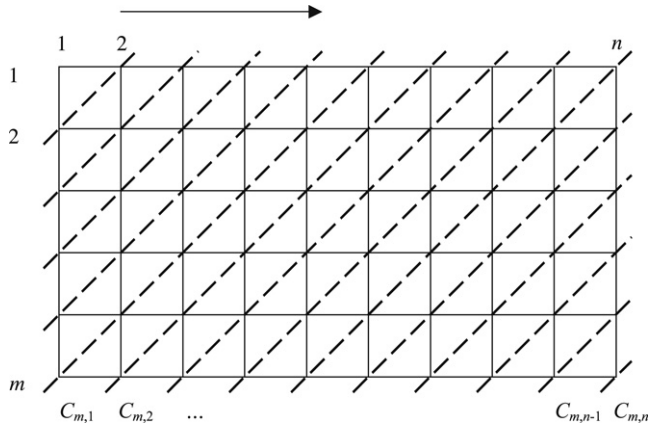

Fig. 2. Sequence of calculations for C_{ij} in Proof of Theorem 1.

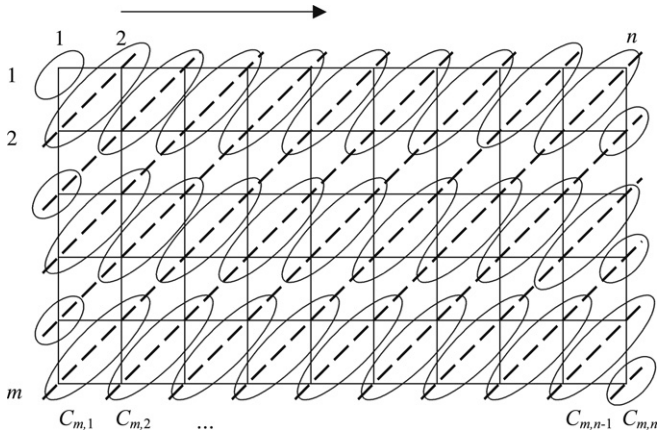

Fig. 3. Sequence of calculations for C_{ij} in Proof of Theorem 2. Calculations assigned to set of p processors have been clustered.

3.1. Single solution

In each iteration we have to find a goal function value for a *single* fixed π. Calculations can be spread into parallel processors in a few ways.

Theorem 1. *For a fixed π the value of criterion function for problems $F \parallel C_{\max}$ and $F \parallel C_{\text{sum}}$ can be found on the CREW PRAM machine in time $O(n+m)$ by using m processors.*

Proof. Without the loss of generality one can assume that $\pi = (1, 2, \ldots, n)$. Calculations of $C_{i,j}$ by using (2) have been clustered. Cluster k contains values C_{ij} such that $i+j-1 = k, k = 1, 2, \ldots, n+m-1$ and requires at most m processors. Clusters are processed in an order $k = 1, 2, \ldots, n+m-1$. The cluster k is processed in parallel on at most m processors. The sequence of calculations is shown in Fig. 2 on the background of a grid graph commonly used for the flow shop problem. Values linked by dashed lines constitute a single cluster. The value of $C_{\max}$ criterion is simple $C_{m,n}$. To calculate $C_{\text{sum}} = \sum_{j=1}^{n} C_{m,j}$ we need to add n values $C_{m,j}$, which can be performed sequentially in n iterations or in parallel by using m processors with the complexity $O(n/m+\log m)$. Finally the computational complexity of determining the criterion value for $F \parallel C_{\max}$ and $F \parallel C_{\text{sum}}$ problems is $O(n+m)$ by using m processors. ■

Fact 1. The speedup of the method from Theorem 1 is $O(\frac{nm}{n+m})$, efficiency is $O(\frac{n}{n+m})$.

The presented method is not cost optimal. Its efficiency slowly decreases along with increasing m. For example for $n = 10, m = 3$ efficiency is about 77%, whereas for $m = 10$ is 50%. [1] Clearly, for $n \gg m$ efficiency tends to 100%, and for $n \ll m$ quickly decreases to 0. This method requires fixing an a priori number of processors $p = m$, which does not seem to be troublesome since usually in practice $m \leq 20$. Emulation of calculations by using $p < m$ processors increases computational complexity to $O((n+m)m/p)$, although the construction of a proper algorithm remains open. If $p \geq m$, then $(p-m)$ processors will be unloaded.

Theorem 2. *For a fixed π the value of criterion function for problems $F \parallel C_{\max}$ and $F \parallel C_{\text{sum}}$ can be found on the CREW PRAM machine in the time $O(n+m)$ by using $O(\frac{nm}{n+m})$ processors.*

Proof. Without loss of generality one can assume that $\pi = (1, 2, \ldots, n)$. It is based on the scheme of calculations shown in Fig. 3. Let $p \leq m$ be the number of processors used. The calculation process will be carried out for *levels* $k = 1, 2, \ldots, d, d = n+m-1$ in this order. On the level k we perform a calculation of n_k values $C_{i,j}$ such that $i+j-1 = k$, $\sum_{k=1}^{d} n_k = nm$.

We cluster n_k elements on the level k into $\left\lceil \frac{n_k}{p} \right\rceil$ groups; first $\left\lfloor \frac{n_k}{p} \right\rfloor$ groups contain p elements each, whereas the remaining elements (at most p) belong to the last group. Fig. 3 shows the structure of groups on successive levels. Parallel computations on level k are performed in time $O\left(\left\lceil \frac{n_k}{p} \right\rceil\right)$. The total calculation time is equal to the sum over all levels and is of order

$$\sum_{k=1}^{d} \left\lceil \frac{n_k}{p} \right\rceil \leq \sum_{k=1}^{d} \left(\frac{n_k}{p} + 1 \right) = \frac{nm}{p} + d = \frac{nm}{p} + n + m - 1. \quad (6)$$

We are seeking the number of processors p, $1 \leq p \leq m$, for which the efficiency of parallel algorithm is $O(1)$ — this ensures the cost optimality of the method. Value p can be found from the following condition

$$\frac{1}{p} \frac{nm}{\frac{nm}{p} + n + m - 1} = c = O(1) \quad (7)$$

for some constant $c < 1$. After a few simple transformations of (7) we obtain

$$p = \frac{nm}{n+m-1} \left(\frac{1}{c} - 1 \right) = O\left(\frac{nm}{n+m} \right). \quad (8)$$

Setting $p = O(\frac{nm}{n+m})$, we get that the total calculation time of C_{ij} values equals

$$O\left(\frac{nm}{p} + n + m - 1 \right) = O\left(\frac{nm}{\frac{nm}{n+m}} + n + m \right) = O(n+m). \quad \blacksquare \quad (9)$$

Fact 2. The speedup of method based on Theorem 2 is $O(\frac{nm}{n+m})$, the cost is $O(nm)$.

This method is cost-optimal and allows one to control the efficiency as well as speed of calculations by choosing the number of processors and adjusting the parameters of calculations to the real number of parallel processors existing in the system. Besides, Theorem 2 provides the 'optimal' number of processors that ensures cost-optimality of this method. This number can be set by a flexible adaptation of the number of processors to both sizes of the problem, namely n and m simultaneously. For example, for

[1] Evaluation is true with a certain constant multiplier.

$n \gg m$ we have $p \approx m$, for $n \ll m$ we have $p \approx n \ll m$, whereas for $n \approx m$ we have $p \approx n/2$.

Observe that both Theorems 1 and 2 own the same bound $O(n+m)$ on the computational complexity, which is the natural consequence of the sequential structure of formula (2). In order to obtain a higher speedup we need to give up the scheme (2). On the current state of knowledge there remains only a non-recursive scheme (3) of a very high computational complexity.

Theorem 3. *For a fixed π the value of criterion function for problems $F \parallel C_{\max}$ and $F \parallel C_{\sum}$ can be found on the CREW PRAM machine in time $O(m + \log n)$ by using $O(\frac{(n+m)^{n-1}}{m(n-1)!})$ processors.*

Proof. Without loss of generality one can assume that $\pi = (1, 2, \ldots, n)$. We use the formula (3) which can be re-written in the form of

$$C_{ij} = \max_{1=j_0 \le j_1 \le \cdots \le j_i = j} \sum_{s=1}^{i} \left(P_{s,j_s} - P_{s,j_{s-1}-1} \right), \tag{10}$$

where $P_{s,t} = \sum_{k=1}^{t} p_{sk}$ is the prefix sum, $t = 1, 2, \ldots, n$. For a fixed s the value of $P_{s,t}$ can be found in time $O(\log n)$ on $O(n/\log n)$ processors, $t = 1, 2, \ldots, n$. Thus, all $P_{s,t}$ for $t = 1, 2, \ldots, n$, $s = 1, 2, \ldots, m$ can be found by using $O(mn/\log n)$ processors in time $O(\log n)$ once at the beginning. For the goal function we need C_{mn}. The number of all subsequences $(j_0, j_1, \ldots, j_m)$ satisfying the condition $1 = j_0 \le j_1 \le \cdots \le j_m = n$ corresponds one-to-one to the number of combinations of $m-1$ elements with repetitions on the $(n-2)$th element set and is equal $\binom{n+m-2}{m-1}$. The method of generating such subsequences in time $O(m)$ by using $\binom{n+m-2}{m-1}$ processors one can find in the Appendix. Next, by using $\binom{n+m-2}{m-1}$ processors one can find sequentially all sums $\sum_{s=1}^{m} \left(P_{s,j_s} - P_{s,j_{s-1}-1} \right)$ from the formula (10) for all subsequences in time $O(m)$. To find value C_{mn} we have to find the maximum (for $C_{\max}$ criterion), or the sum (for $C_{\sum}$) of $\binom{n+m-2}{m-1}$ calculated sums which can be found in time $O(\log \binom{n+m-2}{m-1})$ by using $O(\binom{n+m-2}{m-1} / \log \binom{n+m-2}{m-1})$ processors.

The computational complexity of this step through the inequality,

$$\binom{n+m-2}{m-1} = \frac{m(m+1)\ldots(n+m-2)}{(n-1)!} \le \frac{(n+m)^{n-1}}{(n-1)!} \tag{11}$$

and well-known equation $\log(n!) = \Theta(n \log n)$ is equal to

$$O\left(\log \frac{(n+m)^{n-1}}{(n-1)!} \right) = O\left(n \log\left(1 + \frac{m}{n} \right) \right). \tag{12}$$

Note that (12) implies

$$\begin{aligned} n \log\left(1 + \frac{m}{n} \right) &\le \lim_{n\to\infty} \left(n \log\left(1 + \frac{m}{n} \right) \right) \\ &= \log \lim_{n\to\infty} \left(1 + \frac{m}{n} \right)^{n} = \log e^{m} = m \log e \end{aligned} \tag{13}$$

which is $O(m)$. Since the computational complexity of the remaining algorithm steps is not greater than $O(\max(m, \log n)) = O(m + \log n)$ it is also the final computational complexity of this method. The number of processor used is

$$O\left(\max\left(\frac{mn}{\log n}, \frac{\binom{n+m-2}{m-1}}{\log \binom{n+m-2}{m-1}} \right) \right) = O\left(\frac{(n+m)^{n-1}}{m(n-1)!} \right). \quad \blacksquare$$

Fact 3. The speedup of the method from Theorem 3 is $(\frac{mn}{m+\log n})$.

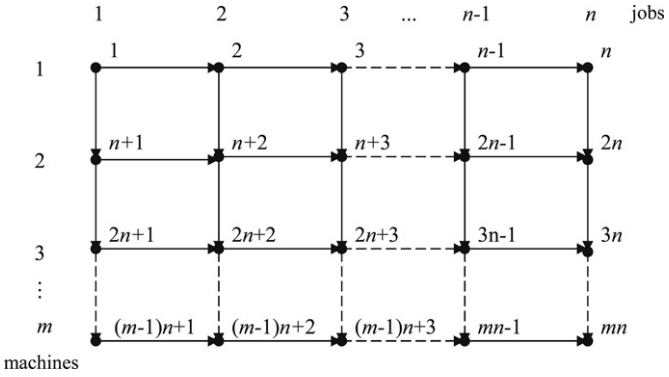

Fig. 4. Graph $G^*(\pi)$.

The number of processors grows exponentially along with increasing n, simultaneously decreasing the efficiency very quickly. Thus, the result has a theoretical meaning rather than a practical one.

The next two theorems also deal with the problem of determining a goal-function for flow shop problems with $C_{\max}$ and $C_{\sum}$ criteria. The lower computational complexity (logarithmic) can be obtained at the expense of increasing the number of processors and losing the of cost-optimality property using different techniques of calculation.

Theorem 4. *For a fixed π the value of criterion function for the problem $F \parallel C_{\sum}$ can be determined in a time $O(\log(n+m)\log(nm))$ by using $O(\frac{(nm)^3}{\log(nm)})$ processors of the CREW PRAM machine.*

Proof. We will propose a parallel method based on sequential Floyd–Warshall algorithm [4] dedicated to determining the shortest path in a graph. Without losing generality we can assume that $\pi = (1, 2, \ldots, n)$. Determining C_{ij} values will be done by using a mesh graph $G(\pi)$ described in Section 2.

To obtain a better clearness of notations an original graph $G(\pi)$ was transformed into an equivalent graph $G^*(\pi)$ by re-enumerating vertexes. We set numbers $1, 2, \ldots, nm$ to successive vertexes as in Fig. 4 to obtain the homogenous enumeration. Therefore, we define a new graph $G^*(\pi)$ with a set of vertexes

$$W = \{u : u = 1, 2, \ldots, nm\}. \tag{14}$$

A vertex (i, j) in the graph $G(\pi)$ is equivalent to mutually unambiguous transformation to a vertex

$$u = (i-1)n + j \tag{15}$$

of a new graph $G^*(\pi)$. A vertex $u \in W$ is equivalent to a vertex (i, j) of a graph $G(\pi)$ such that

$$i = \left\lfloor \frac{u-1}{n} \right\rfloor + 1, \qquad j = u - \left\lfloor \frac{u-1}{n} \right\rfloor n. \tag{16}$$

By such a transformation we obtain a graph

$$G^*(\pi) = (W, E^0 \cup E^*), \tag{17}$$

where sets of vertical and horizontal arcs are the following:

$$E^0 = \bigcup_{u=1}^{nm-n} \{(u, u+n)\}, \qquad E^* = \bigcup_{k=1}^{m} \bigcup_{u=(k-1)n}^{kn-1} \{(u, u+1)\}. \tag{18}$$

A vertex $u \in W$ obtains a weight (described as p_u) equals to p_{ij} of an equivalent vertex from a graph $G(\pi)$. A graph $G^*(\pi)$ is presented in the Fig. 4.

For a graph $G^*(\pi)$ we introduce a distance matrix $A = [a_{u,v}]$ with a size $nm \times nm$, where $a_{u,v}$ is the length of the longest path between vertexes u and v. Values $a_{u,v}$ we initiate as follows:

$$a_{u,v} = \begin{cases} p_u & \text{if } (u,v) \in E^0 \cup E^* \\ 0 & \text{if } (u,v) \notin E^0 \cup E^*. \end{cases} \tag{19}$$

The matrix A will be used to determine the length of the longest path in a graph $G^*(\pi)$ which also is the length of the longest path in a graph $G(\pi)$. Initial values of a matrix A can be determined in a time $O(1)$ by using $(nm)^2$ processors, because this operation depends on executing $(nm)^2$ independent instructions of assignment.

The problem of determining a value of the goal function for the flow shop problem $F \parallel C_{\text{sum}}$ demands finding lengths of the longest paths from the vertex $1 \in W$ to vertexes $(m-1)n+1, (m-1)n+2, \ldots, mn$ (which is equivalent to determining the following values of times of jobs finishing: $C_{m,1}, C_{m,2}, \ldots, C_{m,n}$). To determine the lengths of paths it is enough to execute $\lceil \log(n+m-1) \rceil$ parallel steps, because in each step $k = 1, 2, \ldots, \lceil \log(n+m-1) \rceil$ the algorithm described below actualizes lengths of the longest paths between vertexes with a distance (between them, in the sense of a number of vertexes) equal to $1, 2, 4, 8, \ldots, 2^{\log(n+m-1)}$ (see also [12]). After having executed $\lceil \log(n+m-1) \rceil$ steps the matrix A contains information about the lengths of paths between vertexes remote from (in the sense of a number of vertexes) $2^{\log(n+m-1)} = (n+m-1)$, therefore between *all* the vertexes: because the number of vertexes on the longest (in the sense of a number of vertexes) paths from vertex 1 to nm, equals $(n+m-1)$.

For the need of an algorithm an additional three-dimension table $T = [t_{u,w,v}]$, with a dimensions $nm \times nm \times nm$, is defined to compute the transitive closure of lengths of paths in a graph $G^*(\pi)$. An algorithm needs to execute the following identical steps $\lceil \log(n+m) \rceil$ times:

(1) actualization $t_{u,w,v}$ for all triplets (u, w, v) due to the constraint $t_{u,w,v} = a_{u,w} + a_{w,v}$,
(2) actualization $a_{u,v}$ for all pairs (u, v) due to the constraint $a_{u,v} = \max\{a_{u,v}, \max_{1 \le w \le nm} t_{u,w,v}\}$.

The step 1 executed on $(nm)^3$ processors can be carried out in a time $O(1)$. On $\lceil (nm)^3 / \log(nm) \rceil$ processors the calculations should be performed $\lceil \log(nm) \rceil$ times, therefore the computational complexity of the Step 1 is $O(\log(nm))$.

The step 2 consists of determining the maximum of $nm+1$ values which can be done on $O(nm/\log(nm))$ processors in a time $O(\log(nm))$ Because such a maximum should be determined for $(nm)^2$ pairs $(u, v) \in W$ $(|W| = nm)$, and these calculations are independent, it should be repeated $\lceil \log(n+m) \rceil$ times, and thus all the method needs is the following number of processors

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

The computational complexity of the described above fragment of the algorithm is

$$\lceil \log(n+m) \rceil O(\log(nm)) = O(\log(n+m)\log(nm)).$$

Finally, for the calculation of the criterion value it is necessary to sum n values C_{mj}, where C_{mj} is a length between a vertex $1 \in W$ and a vertex $(m-1)n+j$, $j = 1, 2, \ldots, n$. It can be done in a time $O(\log n)$ by using $O(n/\log n)$ processors which preserves the computational complexity $O(\log(n+m)\log(nm))$ of all described methods and the number of processors $O(\frac{(nm)^3}{\log(nm)})$. ■

Fact 4. For the method based on Theorem 4 we have a speedup $O(\frac{nm}{\log(n+m)\log(nm)})$ and efficiency $O(\frac{1}{(nm)^2 \log(n+m)})$.

The efficiency of this method is very far from the optimal $O(1)$. A profit is only connected with a considerable reduction of computational complexity, from linear to logarithmic (to the square) with reference to the size of the problem. Unfortunately, it is done at the expense of a significant increase in the number of processors used. The analysis of profits leads to surprising results. For example, for $n = 10$, $m = 3$ the efficiency of the method is about 0.03% with a speedup of 1.65. For $n = 20$, $m = 3$ the efficiency is below 0.01%, but the speedup is 2.25. Using the proof of Theorem 4 one can formulate a fully analogical theorem for the $C_{\max}$ class of the flow shop problem.

Theorem 5. *For a fixed permutation π value of the criterion function for a problem $F \parallel C_{\max}$ can be determined in a* time $O(\log(n+m)(\log(nm)))$ *by using $O((nm)^3/\log(nm))$ processors CREW PRAM machine.*

The proof results from a proof of the Theorem 4.

Theorems 3 and 5 ruin, in some sense, the intuition which says that if for the problem $F^* \parallel C_{\max}$ there are $(n+m-1)$ vertexes on the longest (critical) path, connected by recurrent relation, then one should execute at least $O(n+m)$ iterations. It occurs that it is possible to obtain a computational complexity of a lower order, namely $O(\log(n+m)\log(nm))$ or $O(m+\log n)$. For a fixed number of machines m it is possible to obtain the computational complexity $O(\log^2 n)$ (method from Theorem 5) or even $O(\log n)$ (method from Theorem 3), respectively. The problem, if it is a limit value, is open.

3.2. The API neighborhood

A neighborhood based on an adjacent pairwise interchange (API) of elements in permutation is the simplest and commonly used one. Sequential algorithms searching API use the so called *accelerator* to speedup the run by suitable decomposition and aggregation of computations for relative solutions: see [8]; this can be applied only for the problem $F \parallel C_{\max}$. Since some further theorems refer to this concept we will introduce it briefly.

Let π be the permutation that generates neighborhood API and $v = (a, a+1)$ be the pair of adjacent positions such that their interchange in π leads us to the new solution π_v. At first for the permutation π we calculate

$$r_{st} = \max\{r_{s-1,t}, r_{s,t-1} + p_{s\pi(t)}\}, \quad t = 1, 2, \ldots, n, s = 1, 2, \ldots, m, \tag{20}$$

$$q_{st} = \max\{q_{s+1,t}, q_{s,t+1} + p_{s\pi(t)}\}, \quad t = n, \ldots, 2, 1, s = m, \ldots, 2, 1, \tag{21}$$

where $r_{0t} = 0 = q_{m+1,t}$, $t = 1, 2, \ldots, n$, $r_{s0} = 0 = q_{s,n+1}$, $s = 1, 2, \ldots, m$. $C_{\max}(\pi_v)$ for a single interchange $v = (a, a+1)$ can be found in time $O(m)$ from equations

$$C_{\max}(\pi_v) = \max_{1 \le s \le m} (e_s + q_{s,a+2}), \tag{22}$$

$$e_s = \max\{e_{s-1}, d_s\} + p_{s,\pi(a)}, \quad s = 1, 2, \ldots, m, \tag{23}$$

$$d_s = \max\{d_{s-1}, r_{s,a-1}\} + p_{s\pi(a+1)}, \quad s = 1, 2, \ldots, m. \tag{24}$$

Initial conditions are as follows : $e_0 = 0 = d_0$, $r_{s0} = 0 = q_{s,n+2}$, $s = 1, 2, \ldots, m$. The neighborhood API contains $n-1$ solutions π_v, $v = (a, a+1)$, $a = 1, 2, \ldots, n-1$, and is searched conventionally in time $O(n^2 m)$. By using the *sequential accelerator* for API we can do it in time $O(nm)$.

Theorem 6. *For a fixed π the neighborhood API for $F \parallel C_{\max}$ and $F \parallel C_{\text{sum}}$ problems can be searched on the CREW PRAM machine in time $O(n+m)$ by using $O(\frac{n^2 m}{n+m})$ processors.*

Proof. Skipping solution affinity we allocate for each π_v the number $O(\frac{nm}{n+m})$ of processors which allows us to find all $C_{\max}(\pi_v)$ in time $O(n+m)$, see Theorem 2. The best solution in the neighborhood can be found in time $O(n)$ by using a single processor. ■

There is a dilemma to which version of sequential algorithm should be compared with the parallel method — with or without the sequential accelerator? If we take the best one (with accelerator) then we have the following evaluation.

Fact 5. Speedup of the method from Theorem 6 is $O(\frac{nm}{n+m})$, efficiency is $O(\frac{1}{n})$.

The presented method is not cost-optimal, its efficiency quickly decreases along with growing n. Note that if the sequential accelerator cannot be applied (as for the $F \parallel C_{\text{sum}}$ problem), this method is cost-optimal with efficiency $O(1)$. Employing knowledge about relationship among solutions in the neighborhood one can prove a significantly stronger result.

Theorem 7. *For a fixed π the neighborhood API for $F \parallel C_{\max}$ problem can be searched on the CREW PRAM machine in time $O(n+m)$ by using $O(\frac{nm}{n+m})$ processors.*

Proof. Let $v = (a, a+1)$. We design a parallel algorithm using the sequential accelerator for API. Values r_{st}, q_{st} are generated once: at the beginning of the search, in time $O(n+m)$ using $O(\frac{nm}{n+m})$ processors in a way analogous to that from the proof of Theorem 2. This is a cost-optimal method. The process of overlooking of the API neighborhood has been split into groups of cardinality $\left\lceil \frac{n}{p} \right\rceil$ each, where $p = \left\lceil \frac{nm}{n+m} \right\rceil$ is the number of processors used. Computations in each group are performed independently. Processor $k = 1, 2, \ldots, p$ serves the group defined by v

$$v = \left((k-1)\left\lceil \frac{n}{p} \right\rceil + a, (k-1)\left\lceil \frac{n}{p} \right\rceil + a + 1 \right), \quad a = 1, 2, \ldots, \left\lceil \frac{n}{p} \right\rceil \tag{25}$$

for $k = 1, 2, \ldots, p-1$, and

$$v = \left((p-1)\left\lceil \frac{n}{p} \right\rceil + a, (p-1)\left\lceil \frac{n}{p} \right\rceil + a + 1 \right), \tag{26}$$

$$a = 1, 2, \ldots, n - (p-1)\left\lceil \frac{n}{p} \right\rceil - 1$$

for $k = p$. The last group can be incomplete. Since the computational complexity of finding $C_{\max}(\pi_v)$ in single group equals

$$\left\lceil \frac{n}{p} \right\rceil O(m) = O\left(\frac{nm}{p}\right) = O\left(\frac{nm}{\frac{nm}{n+m}}\right) = O(n+m),$$

then all $C_{\max}(\pi_v)$ can be found in the same time. Each processor, while sequentially calculating its portion of $C_{\max}(\pi_v)$ values, can simultaneously store the best solution in the group. To achieve this aim it additionally makes

$$\left\lceil \frac{n}{p} \right\rceil - 1 = O\left(\frac{n}{p}\right) = O\left(\frac{n}{\frac{nm}{n+m}}\right) = O\left(\frac{n+m}{m}\right) \tag{27}$$

comparisons to the best solution which has no influence on the earlier provided computational complexity. Choosing the best solution, among the whole API neighborhood, requires p comparisons of best values found for all groups. This can be done in time $O(\log p)$ by using $p = O(\frac{nm}{n+m})$ processors. The last fact follows from the following sequence of inequalities

$$\begin{aligned} \log p &= \log\left\lceil \frac{nm}{n+m} \right\rceil < \log\left(\frac{nm}{n+m} + 1\right) \\ &= \log\left(\frac{nm+n+m}{n+m}\right) = \log\left(\frac{(n+1)(m+1)-1}{n+m}\right) \\ &= (\log((n+1)(m+1)-1)) - \log(n+m) \\ &< \log((n+1)(m+1)) \\ &= \log(n+1) + \log(m+1) < n+1+m+1. \quad \blacksquare \end{aligned} \tag{28}$$

Fact 6. Speedup of the method from Theorem 7 is $O(\frac{nm}{n+m})$, efficiency is $O(1)$.

3.3. The INS neighborhood

The neighborhood INS based on an insertion of elements in the permutation has the computational complexity $O(n^3m)$ for the searching. For the INS and the problem $F \parallel C_{\max}$ there is the *sequential accelerator*, see e.g. [8] which reduces this complexity to $O(n^2m)$. We will show a stronger result for a parallel algorithm.

Theorem 8. *For a fixed π the neighborhood INS for the $F \parallel C_{\max}$ problem can be searched on the CREW PRAM machine in time $O(n+m)$ by using $O(\frac{n^2m}{n+m})$ processors.*

Proof. Let $v = (a, b)$ define the INS neighborhood for a permutation π as follows: the job $\pi(a)$ has been removed from its position and then it is inserted so that its new position in the resulting permutation π_v becomes b; $a, b \in \{1, \ldots, n\}$, $a \neq b$. Let r_{st}, q_{st}, $s = 1, 2, \ldots, m$, $t = 1, 2, \ldots, n-1$, be values found by (20) and (21) for a permutation obtained from π by removing the job $\pi(a)$. For each fixed $a = 1, 2, \ldots, n$ values r_{st}, q_{st} can be found in time $O(n+m)$ by using $O(\frac{nm}{n+m})$ processors in a way analogous to Theorem 2. Employing $O(\frac{n^2m}{n+m})$ processors we can perform such calculations in time $O(n+m)$ for all permutations obtained from π by removing the job $\pi(a)$, $a = 1, 2, \ldots, n$. For each fixed a values $C_{\max}(\pi_{(a,b)})$, $b = 1, 2, \ldots, n$, $b \neq a$ can be found using (22) in time $O(m)$. We split the whole computation process on $p = \left\lceil \frac{n^2m}{n+m} \right\rceil$ groups, each of which is assigned to a separate processor. Since the INS neighborhood contains $(n-1)^2 = O(n^2)$ solutions, all $C_{\max}(\pi_v)$ can be found in time $\left\lceil \frac{(n-1)^2}{p} \right\rceil O(m) = O(n+m)$. The best solution in the neighborhood can be found in time $O(\log(n^2)) = O(2\log n) = O(\log n)$ by using n processors. The whole method possesses a complexity $O(n+m+\log n) = O(n+m)$ and employs $O(\frac{n^2m}{n+m})$ processors. ■

Fact 7. Then speedup of the method from Theorem 8 is $O(\frac{n^2m}{n+m})$, efficiency is $O(1)$.

3.4. The NPI neighborhood

The neighborhood is generated by swapping any pair of jobs $\pi(a)$, $\pi(b)$, for $a, b \in \{1, 2, \ldots, n\}$, $a \neq b$. We start from the description of a sequential accelerator [8], used in the parallel version presented below. The direct method of searching the neighborhood NPI possesses the computational complexity $O(n^3m)$. The sequential accelerator for NPI reduces this complexity to $O(n^2m)$.

Let $v = (a, b)$, $a \neq b$ define the move that generates a new permutation π_v. Without loss of generality we can assume that

$a < b$, due to the symmetry. Next, let $r_{st}, q_{st}, s = 1, 2, \ldots, m, t = 1, 2, \ldots, n$ be values found by (20) and (21) for π. Denote by D_{st}^{xy} the length of the longest path between nodes (s, t) and (x, y) in the grid graph $G(\pi)$, [8]. The method of calculating $C_{\max}(\pi_v)$ can be decomposed into the following steps. At the beginning we calculate the length of the longest path which goes to the node (s, a), and joins the job $\pi(b)$ dislocated by v on the position a

$$d_s = \max\{d_{s-1}, r_{s,a-1}\} + p_{s,\pi(b)}, \quad s = 1, 2, \ldots, m, \tag{29}$$

where $d_0 = 0$. Then, we calculate the length of the longest path going to the node $(s, b-1)$, joining the part of $G(\pi)$ located between jobs on positions from $a+1$ to $b-1$, invariant for $G(\pi)$

$$e_s = \max_{1 \le w \le s}(d_w + D_{w,a+1}^{s,b-1}), \quad s = 1, 2, \ldots, m. \tag{30}$$

In the successive step we calculate the length of the longest path going to the node (s, b), joining the job $\pi(a)$, put by v on position b

$$f_s = \max\{f_{s-1}, e_s\} + p_{s,\pi(a)}, \quad s = 1, 2, \ldots, m, \tag{31}$$

where $f_0 = 0$. Finally we obtain

$$C_{\max}(\pi_v) = \max_{1 \le s \le m}(f_s + q_{s,b+1}). \tag{32}$$

The value of $C_{\max}(\pi_v)$ can be found if we have a suitable D_{st}^{xy}. These values can be calculated recursively for the fixed t and $y = t+1, t+2, \ldots, n$, by using this equality

$$D_{st}^{x,y+1} = \max_{s \le k \le x}\left(D_{st}^{ky} + \sum_{i=k}^{x} p_{i\pi(y+1)}\right) \tag{33}$$

where $D_{st}^{xt} = \sum_{i=s}^{x} p_{i\pi(t)}$. The formula (33) can be re-written in the form of

$$D_{st}^{s,t+1} = D_{st}^{st} + p_{s,\pi(t+1)}, \qquad D_{st}^{x0} = D_{st}^{0y} = 0, \tag{34}$$

$$D_{st}^{x,y+1} = \max\{D_{st}^{xy}, D_{st}^{x-1,y}\} + p_{x,\pi(y+1)}, \tag{35}$$

$x = 1, 2, \ldots, m$, $y = 1, 2, \ldots, n$, which allows us to find all D_{st}^{xy}, $x = 1, 2, \ldots, m$, $y = 1, 2, \ldots, n$ for the fixed (s, t) in time $O(nm)$. Finally, the sequential calculation of all $O(n^2)$ values $C_{\max}(\pi_v)$ (including all $D_{st}^{xy}, x, s = 1, 2, \ldots, m, y, t = 1, 2, \ldots, n$) can be processed in time $O(n^2 m)$.

Theorem 9. *For a fixed π the neighborhood NPI for the $F \parallel C_{\max}$ problem can be searched on the CREW PRAM machine in time $O(nm)$ by using $O(n^2 m)$ processors.*

Proof. We employ the parallel counterpart of a sequential accelerator. Let each of $\frac{n(n-1)}{2}$ elements of the neighborhood be associated with stakes of $O(m)$ processors. For the fixed (s, t), and all $x = 1, 2, \ldots, m$, $y = 1, 2, \ldots, n$, values D_{st}^{xy} can be found sequentially in time $O(nm)$. Using $O(nm)$ processors we can calculate D_{st}^{xy} for all $x, s = 1, 2, \ldots, m$, $y, t = 1, 2, \ldots, n$ in time $O(nm)$ once at the beginning. Let us analyze a calculation of $C_{\max}(\pi_v)$ for fixed v. We have to compute the values d_s in (29) sequentially in time $O(m)$. The maximum among m values in (30) can be performed in parallel, for all s, by using $O(m)$ processors in time $O(m)$. The formula (31) we calculate sequentially, for each s, in time $O(m)$. A single value $C_{\max}(\pi_v)$ in (32) requires m independent adding operations and then finding the maximum among m numbers. We carry this out sequentially in time $O(m)$. Finally, parallel computations of $O(n^2)$ values $C_{\max}(\pi_v)$ can be performed in time $O(m)$ by using $O(n^2 m)$ processors. Since the process of generating D_{st}^{xy} had the complexity $O(nm)$, this is the final complexity of the whole method. ■

Fact 8. The speedup of the method from the Theorem 9 is $O(nm)$, efficiency is $O(\frac{1}{n})$.

4. Empirical tests

Computational experiments have been developed in two directions: (1) determining the real speedup that is possible to obtain using theorems from previous sections and (2) applying these methods as an element of parallel metaheuristics.

4.1. Parallel cost function calculations

Theoretical results presented in the previous section assume using practically a non-realizable CREW PRAM machine. It is because of the synchronic nature of approaches proposed in Theorems 1–9 that SIMD is the best model of real parallel architecture. Vector-processing using the MMX instruction set, as an implementation of SIMD, can be easily used here to check the real speedup of the proposed methods.

Almost all PC computers exploited are furnished with processors equipped with the so-called extended instruction set MMX. These special instructions allow one to make identical operations simultaneously (therein mathematical) on the 2, 4 or 8 pairs of arguments occupied accordingly by 4, 2, 1 bytes. The potential profits can be evaluated as follows. Assuming that data and results are integers represented by k bytes, we can perform $8/k$ math operations (e.g. sum, max) simultaneously for MMX processors. The theoretical speedup of calculations can be found from the formula:

$$s_{\text{theor}} = \frac{nm}{\lceil mk/8 \rceil (n + 8/k)}. \tag{36}$$

Let us see that for each vertex of the graph $G(\pi)$ the calculation processing comes in the same way accordingly to formulae 2. Furthermore, the simple interchangeable enumeration $e(i, j) = (i + j - 1)m + j$, $i = 0, \ldots, m$, $j = 0, \ldots, n$ assigns the consecutive integers (indexes of the vector) to the elements of the same cluster. To verify this let us take into account the mth cluster as an example. This cluster consists of the following (i, j) pairs : $(1, m), (2, m-1), \ldots (m-1, 2), (m, 1)$ and the following enumeration: $m^2 + m, m^2 + m - 1, \ldots, m^2 + 2, m^2 + 1$.

Two procedures, Schedule (connected with Theorem 2) and API (connected with Theorem 4), were coded in Visual C++ 2008 Express Edition. The first procedure determines the completion times for the natural job order $\pi = \{1, \ldots, n\}$ and the second procedure $C_{\max}$ values for the API neighborhood. The procedures were coded in the sequential version NPR and parallel version PR. The MMX intrinsics were called by a suitable C++ function. The compilation was optimized to obtain the maximum speed by Visual C++ compiler. The computation runs on a PC machine with Intel Core 2 Duo 2.66 GHz processor and Windows XP Professional operating system with 1GB RAM. Processing and completion times were coded as 16-bit integer values, which allows performing parallel synchronic instructions on 4 data.

The first test was provided on the 5 groups of 10 instances. All instances had the same number of jobs $n = 100$. The groups varied in the number of machines $m \in \{4, 8, 12, 16, 20\}$. For each group of instances the mean calculation time (CPUs), speedup ratio (s_{empiric}) and theoretical speedup ratio ((s_{theor}), given by formulae (36)) were calculated. They are shown in Table 1 and Fig. 5. Also, the method of the API neighborhood searching based on the method from the proof of Theorem 7 has been tested. The results of the empirical speedup obtained is shown on Fig. 6.

The second test was conducted on the first 9 groups of benchmark instances of the flow shop problem provided by Taillard [15]. Each group $n \times m$: 20×5, 20×10, 20×20, 50×5, 50×10, 50×20, 100×5, 100×10, 100×20 consists of 10 hard instances selected from a large number of randomly generated problems. The computational results are shown in the Table 2 and on Figs. 7 and 8.

Table 1
Times of 10^6 natural schedule calculations.

Group	Schedule				API		
	CPUs		Speedup	Theoretical	CPUs		Speedup
	NPR	PR	$s_{empiric}$	speedup s_{theo}	NPR	PR	$s_{empiric}$
100×4	2.6	1.2	2.17	3.85	20.4	8.5	2.40
100×8	5.8	2.1	2.76	3.85	45.2	17.4	2.60
100×12	9.6	3.5	2.74	3.85	70.7	25.6	2.76
100×16	13.0	4.5	2.89	3.85	95.3	35.4	2.69
100×20	16.7	5.5	3.04	3.85	121.1	45.5	2.66

Table 2
10^6 schedule calculations times for the Taillard's benchmarks.

Group	Schedule				API		
	CPUs		Speedup	Theoretical	CPUs		Speedup
	NPR	PR	$s_{empiric}$	speedup s_{theo}	NPR	PR	$s_{empiric}$
20×5	0.7	0.5	1.40	2.08	5.5	4.8	1.15
20×10	1.6	0.8	2.00	2.78	11.8	6.9	1.71
20×20	3.4	1.3	2.62	3.33	24.3	13.3	1.83
50×5	1.8	1.1	1.64	2.31	13.5	9.7	1.39
50×10	4.2	1.7	2.47	3.09	29.6	14.0	2.11
50×20	8.6	2.8	3.07	3.70	61.8	24.7	2.50
100×5	3.3	2.1	1.57	2.40	25.9	17.3	1.50
100×10	8.0	3.5	2.29	3.21	58.0	25.7	2.26
100×20	16.7	5.5	3.04	3.85	121.1	45.6	2.66

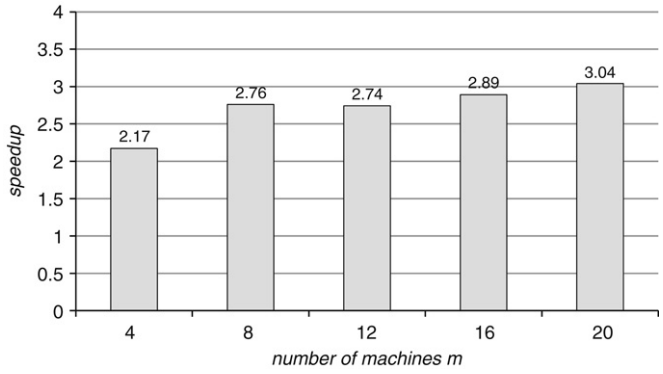

Fig. 5. Empirical speedup obtained for the cost function's computation of the flow shop C_{max} instances $n \times m$ with constant $n = 100$ and $m = 4, 8, 12, 16, 20$. Theoretical speedup was 3.85 for all these instances.

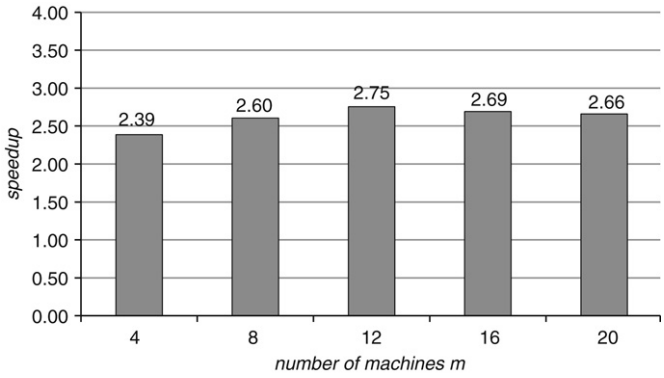

Fig. 6. Empirical speedup obtained for the API neighborhood searching for the flow shop C_{max} instances $n \times m$ with constant $n = 100$ and $m = 4, 8, 12, 16, 20$. Theoretical speedup was 3.85 for all these instances.

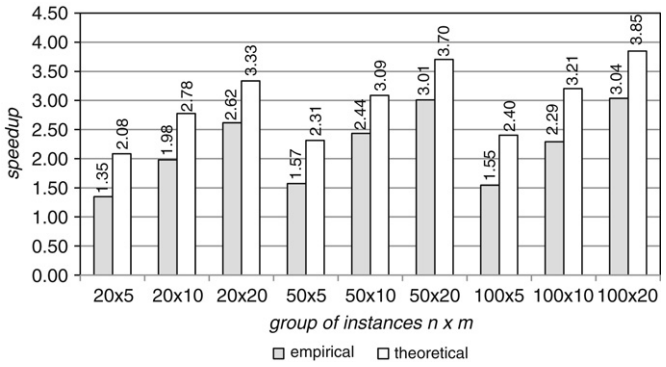

Fig. 7. Empirical and theoretical speedup obtained for the cost function's computation of the flow shop C_{max} instances $n \times m$ with $n = 20, 50, 100$, $m = 5, 10, 20$.

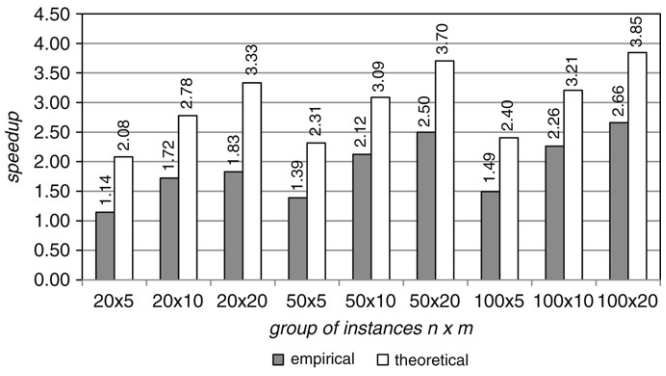

Fig. 8. Empirical and theoretical speedup obtained for parallel API neighborhood searching of the flow shop C_{max} instances $n \times m$ with $n = 20, 50, 100, m = 5, 10, 20$.

As we can observe in Table 2, the results of empirical speedup are greater for the large m values. The average relative percentage deviation of the empirical speedup to the theoretical one $\left((s_{theo} - s_{empiric})/s_{empiric} \cdot 100\%\right)$ equals 29% for the cost function calculation (function Schedule) and 32% for the API neighborhood search. For a constant number of jobs $n = 100$ (Table 1) these values equal 26% and 36%, which means that the constant value which is hidden in the $O(m + n)$ description is relatively small.

4.2. Application: Scatter-search method

Methods proposed in previous sections were used as an element of the local search metaheuristics, namely the scatter-search method. The main idea of the scatter-search method is presented in [7]. The algorithm is based on the idea of evaluation of the so-called starting solutions set. In the classic version a linear combination of the starting solution is used to construct a new solution. In the case of a permutational representation of the solution, the usage of a linear combination of permutations gives us an object which is not a permutation. Therefore, in this paper a path relinking procedure is used to construct a path from one solution of the starting set to another solution from this set. The best element of such a path is chosen as a candidate to add to the starting solution set.

4.2.1. Path-relinking

The base of the path-relinking procedure, which connects two solutions $\pi_1, \pi_2 \in \Pi$, is a multi-step crossover fusion (MSXF) described by Reeves and Yamada [11]. Its idea is based on a stochastic local search, starting from the π_1 solution, to find a new good solution where the other solution π_1 is used as a reference point.

The API neighborhood $N(\pi)$ of the permutation (individual) π is used and it is searched in parallel by the method from the Theorem 7. The distance measure $d(\pi, \sigma)$ is defined as a number of adjacent pairwise exchanges needed to transform the permutation π into the permutation σ. Such a measure is known as Kendall's τ measure.

Algorithm 2 (*Path-relinking Procedure*).

Let π_1, π_2 be reference solutions. Set $x = q = \pi_1$;
repeat
  For each member $y_i \in N(\pi)$, calculate $d(y_i, \pi_2)$;
  Sort $y_i \in N(\pi)$ in ascending order of $d(y_i, \pi_2)$;
  repeat
    Select y_i from the $N(\pi)$ with a probability inversely proportional to the index i; Calculate $F(y_i)$; $F \in \{C_{\max}, C_{\text{sum}}\}$;
    Accept $y_i$with probability 1 if $F(y_i) \leq F(x)$,
    and with probability $P_T(y_i) = \exp(\frac{F(x)-F(y_i)}{T})$
    otherwise (T is the temperature);
    Change the index of y_i from i to n and the indices of $y_k, k = i+1, \ldots, n$ from k to $k-1$;
  until y_i is accepted;
  $x \leftarrow y_i$;
  if $F(x) < F(q)$ **then** $q \leftarrow x$;
until *some termination condition is satisfied*;
return q {q is the best solutions lying on the path from π_1 to π_2}.

The termination condition consisted in exceeding 100 iterations by the path-relinking procedure, or achieving solution π_2.

4.2.2. Parallel scatter-search algorithm

The parallel algorithm was designed to be executed on two machines:

- the cluster of 152 dual-core Intel Xeon 2.4 GHz processors connected by Gigabit Ethernet with 3Com SuperStack 3870 switches (for the $F \parallel C_{\text{sum}}$ problem),
- Silicon Graphics SGI Altix 3700 Bx2 with 128 Intel Itanium2 1.5 GHz processors and cache-coherent Non-Uniform Memory Access (cc-NUMA), craylinks NUMAflex4 in fat tree topology with the bandwidth 4.3 Gbps (for the $F \parallel C_{\max}$ problem),

installed in the Wrocław Center of Networking and Supercomputing. Both supercomputers have a distributed memory, where each processor has its local cache memory (in the same node) which is accessible in a very short time (compared to the access time to the memory in another node). Taking into consideration this type of architecture we choose a client-server model for the scatter-search algorithm proposed here, where calculations of path-relinking procedures are executed by processors on local data and communication takes place rarely to create a common set of new starting solutions. The process of communication and evaluation of the starting solutions set S is controlled by the processor number 0. We call this model *global*.

For comparison a model without communication was also implemented in which independent scatter-search threads are executed in parallel. The result of such an algorithm is the best solution from solutions generated by all the searching threads. We call this model *independent*.

Both machines have processors with an extended instruction set MMX, so it was possible to apply the Theorem 2 and the Theorem 7 to parallelize cost function calculations and neighborhood searching on the low level of computations as parallel calculations inside each processor.

Algorithms were implemented in the C++ language using MPI (mpich 1.2.7) library and executed under the OpenPBS batching system which measures times of a processor's usage.

Algorithm 3 (*Parallel Scatter-Search Algorithm for the SIMD Model Without Shared Memory*).

par for $p := 1$ **to** number_of_processors **do**
  for $i := 1$ **to** *iter* **do**
    Step 1. if $(p = 0)$ **then** {only processor number 0}
      Generate a set of unrepeated starting solutions S, $|S| = n$.
      Broadcast a set S among all the processors.
    else {other processors}
      Receive from the processor 0 a set of starting solutions S.
    end if;
    Step 2. For randomly chosen $n/2$ pair from the S apply path relinking procedure to generate a set S'- of $n/2$ solutions which lies on paths.
    Step 3. Apply local search procedure to improve value of the cost function of solutions from the set S'.
    Step 4. if $(p \neq 0)$ **then**
      Send solutions from the set S' to processor 0
    else {only processor number 0}
      Receive sets S' from other processors and add its elements to the set S
    end if;
    Step 5. Leave in the set S at most n solutions by deleting the worst and repeated solutions.
    if$|S| < n$ **then**
      Add a new random solutions to the set S such, that elements in the set S does not duplicate and $|S| = n$.
    end if;
  end for;
end parfor.

4.2.3. Computer simulations

Tests were based on 50 instances with $100, \ldots, 500$ operations ($n \times m = 20\times5, 20\times10, 20\times20, 50\times5, 50\times10$) due to Taillard [15], taken from the OR-Library [10]. The results were compared to the

Table 3
Values of APRD for parallel scatter-search algorithm for the $F \parallel C_{\max}$ problem (global model). The sum of iterations number for all processors is 9600.

$n \times m$	Processors				
	1 *iter* = 9600	2 2 *iter* = 4800	4 *iter* = 2400	8 8 *iter* = 1200	16 *iter* = 600
20 × 5	0.000%	0.000%	0.000%	0.000%	0.096%
20 × 10	0.097%	0.060%	0.072%	0.131%	0.196%
20 × 20	0.039%	0.035%	0.061%	0.062%	0.136%
50 × 5	0.007%	−0.001%	−0.015%	−0.001%	0.007%
50 × 10	0.345%	0.104%	0.113%	0.123%	0.272%
Average	**0.098%**	**0.029%**	**0.046%**	**0.063%**	**0.142%**
t_{total} (h:min:s)	30:04:40	15:52:13	7:40:51	3:35:47	1:42:50
t_{cpu} (h:min:s)	30:05:02	31:44:21	30:41:54	28:45:30	27:24:58

Table 4
Values of APRD for parallel scatter-search algorithm for the $F \parallel C_{\max}$ problem (independent model). The sum of iterations number for all processors is 9600.

$n \times m$	Processors				
	1 *iter* = 9600	2 2 *iter* = 4800	4 *iter* = 2400	8 8 *iter* = 1200	16 *iter* = 600
20 × 5	0.000%	0.000%	0.000%	0.000%	0.096%
20 × 10	0.097%	0.080%	0.066%	0.039%	0.109%
20 × 20	0.039%	0.062%	0.048%	0.031%	0.031%
50 × 5	0.007%	0.000%	0.007%	0.007%	0.000%
50 × 10	0.345%	0.278%	0.148%	0.238%	0.344%
Average	**0.098%**	**0.084%**	**0.054%**	**0.063%**	**0.097%**
t_{total} (h:min:s)	30:04:40	14:38:29	6:58:59	3:15:34	1:32:46
t_{cpu} (h:min:s)	30:05:02	29:16:14	27:54:19	26:03:33	24:41:24

best known, taken from [10] for the $F \parallel C_{\max}$ and from [11] for the $F \parallel C_{\text{sum}}$.

For each version of the scatter-search algorithm (global or independent), the following metrics were calculated:

- ARPD — Average Percentage Relative Deviation to the benchmark's cost function value where

$$PRD = \frac{F_{\text{ref}} - F_{\text{alg}}}{F_{\text{ref}}} \cdot 100\%,$$

  where F_{ref} is the reference criterion function value from [10] for the $F \parallel C_{\max}$ and from [11] for the $F \parallel C_{\text{sum}}$ and F_{alg} is the result obtained by a parallel scatter-search algorithm. There were no situations where $F_{\text{ref}} = 0$ for the benchmark tests.
- t_{total}(in seconds) — real time of executing the algorithm for 50 benchmark instances from [15],
- t_{cpu}(in seconds) — the sum of times consumed on all processors for 50 benchmark instances from [15].

Flow shop problem with makespan $C_{\max}$ criterion. Tables 3 and 4 present results of computations of the parallel scatter-search method for the number of iterations (as a sum of iterations on all the processors) equals 9600. The cost of computations, understood as a sum of the time consumed on all the processors, is about 7 h for all 50 benchmark instances of the flow shop problem, so we have about 8 min per instance for 1-processor implementation, which is the acceptable time of solving big flow shop instances. Let us note that the 16-processor version executes 600 iterations per processor, which is not a lot. If we used a smaller number of iterations it would be too few to make a parallel program effective, after a number of iterations for calculation stabilization, which number is similar for the sequential and parallel algorithm.

The 2-processors version of the global model of the scatter-search algorithm, with communication, has the best results (average percentage deviations to the best known solutions), which are 70.4% better comparing to the average 1-processor implementation (0.029% vs 0.098%).

Flow shop problem with C_{sum} criterion. A similar situation takes place for the parallel scatter-search algorithm tests for the $F \parallel C_{\text{sum}}$ problem. Tables 5 and 6 present results of computations, for the global and independent model, for the number of iterations (as a sum of iterations on all the processors) equals 16 000, so the 16-processor implementation executed 1000 iterations per each processor. The best results are achieved for the 8-processors version of the global model version of scatter-search and they are 52.3% better than the results of the sequential scatter-search algorithm (0.173% vs 0.363%).

The two new best-known solutions have been found for the flow shop problem with C_{sum} criterion during computational experiments. The new upper bound for the `tai42` instance is 83 145 (the previous one was 83 157, from [11]) and for `tai50` instance the new one is 88 106 (was 88 215, [11]).

4.2.4. Speedup calculations

As we do not know the best algorithm for the flow shop instances, it is impossible to use the *strong speedup* definition, i.e. comparing the parallel run-time against the best-so-far sequential algorithm. Therefore, we have to use the weak definition of speedup. We cannot compute speedup against a sequential scatter-search algorithm, since we compare different algorithms. Hence, we turn to compare the same parallel scatter-search algorithm on 1 versus p processors. Such a speedup is known as the orthodox speedup (see Alba [1]).

Several authors reported superlinear speedup [2,5] cause of the following sources:

- implementation source — the sequential algorithm is inefficient, i.e. uses data structures which can be managed faster by the parallel algorithm,
- numerical source — the parallel algorithm finds a good solution more quickly because it changes the order in which solution space is searched compared to sequential algorithm,
- physical source — the parallel algorithm has more than a simple increase in the computational power of CPUs, i.e. other resources as the total size of a fast cache memory.

In this paper we observe a situation where the work performed by parallel and sequential algorithms is different.

Table 5
Values of APRD for parallel scatter-search algorithm for the $F \parallel C_{sum}$ problem (independent model). The sum of iterations number for all processors is 16 000.

$n \times m$	Processors				
	1 *iter* = 16 000	2 2 *iter* = 8000	4 *iter* = 4000	8 8 *iter* = 2000	16 *iter* = 1000
20 × 5	0.000	0.007	0.000	0.006	0.016
20 × 10	0.000	0.000	0.000	0.000	0.000
20 × 20	0.000	0.000	0.000	0.000	0.000
50 × 5	0.904	1.037	0.906	0.903	0.933
50 × 10	0.913	0.986	1.033	0.989	1.110
Average	**0.363**	**0.406**	**0.388**	**0.380**	**0.412**
t_{total} (h:min:s)	75:27:40	37:40:08	18:38:23	9:06:24	4:28:57
t_{cpu} (h:min:s)	75:25:48	75:02:51	74:10:18	72:19:26	70:57:24

Table 6
Values of APRD for parallel scatter-search algorithm for the $F \parallel C_{sum}$ problem (global model). The sum of iterations number for all processors is 16 000.

$n \times m$	Processors				
	1 *iter* = 16 000	2 2 *iter* = 8000	4 *iter* = 4000	8 8 *iter* = 2000	16 *iter* = 1000
20 × 5	0.000	0.000	0.000	0.008	0.007
20 × 10	0.000	0.000	0.000	0.004	0.000
20 × 20	0.000	0.000	0.000	0.000	0.000
50 × 5	0.993	0.677	0.537	0.449	0.764
50 × 10	1.103	0.648	0.474	0.404	0.734
Average	**0.419**	**0.265**	**0.202**	**0.173**	**0.301**
t_{total} (h:min:s)	75:23:44	41:19:51	23:28:19	14:30:03	7:23:50
t_{cpu} (h:min:s)	75:20:42	77:57:57	75:46:07	74:38:51	73:13:35

Flow shop with C_{max} criterion. As the time consumed on all the processors is a little bit longer than the time of the sequential version, we can say that the speedup of this version of the algorithm is almost-linear (see Tables 3 and 4). For the 4 and 8-processors implementation of the global model and for 2,4 and 8-processors implementations of the independent model the average results of ARPD are better than ARPD of the 1-processors versions, but the time consumed on all the processors (t_{cpu}) is *shorter*. So these algorithms obtain better results with a smaller cost of computation - the orthodox speedup is superlinear.

Flow shop with C_{sum} criterion. Also here the orthodox superlinear speedup effect has been observed for the 8 and 16-processors implementations of the global model of parallel scatter-search (see Tables 5 and 6). The total time consumed from this implementations for all 50 instances (74:38:51 and 73:13:35, h:min:s) was smaller than the total time of sequential algorithm execution (75:20:42). Such a situation takes place only for the global model of the scatter-search algorithms: independent searches are not so effective, both in results (ARPD) and speedup.

The superlinear speedup anomaly obtained here has a numerical source and it can be understood as the situation where the sequential algorithm may have to search a large portion of solutions before finding a good one. A parallel algorithm may find the solution of similar quality more quickly due to the change in the order in which the space is searched. This situation can be interpreted in terms of diversification versus intensification of the search in the solution space — a parallel algorithm can achieve better solutions faster than a sequential algorithm as a result of the searching process diversification in the first phase of the algorithm's work (due to the multiple-walk strategy) and intensification in the second phase after finding a 'good' region by the one of multiple walking parallel searching threads.

5. Conclusions

The general approach to parallelization of the scheduling algorithms for the flow shop problem has been described here. In the single-thread the single-solution methods parallelization derived from basic recursive formula led us to cost optimal algorithms; other approaches own low efficiency although offer high speed. Some results obtained in this section can be extended to the EREW PRAM model. This observation follows from the fact that problem data can be copied n times (this can be done in time $O(\log n)$ using $O(n/\log n)$ processors in the initial phase), therefore it is easy to modify algorithms in proofs of theorems to obtain versions of the theorems for the EREW model.

In the single-thread neighborhood-search methods the whole neighborhood can be searched in time of the same order what for a single solution along with sufficiently increased number of processors. This computational complexity appears to be an obvious bound of the neighborhood analysis time.

In a multiple-thread search, represented by a parallel scatter-search here, parallelization increases the quality of obtained solutions keeping comparable costs of computation. The orthodox superlinear speedup is observed in the global (cooperative) model of parallelism.

Acknowledgments

The author wishes to thank referees for their detailed comments and constructive criticisms of the initial draft.

Appendix. Subsequence generating

Lemma 1. *All subsequences $(j_0, j_1, \ldots, j_m)$ satisfying the condition $1 = j_0 \le j_1 \le \cdots \le j_m = n$ can be generated in time $O(m)$ by using the $\binom{n+m-2}{m-1}$ processor CREW PRAM machine.*

Proof. The number of all such subsequences equals the number of $m-1$-element combinations with repetition from the $n-2$-element set and it equals $\binom{n+m-2}{m-1}$. All subsequences can be generated on the CREW PRAM in time $O(m)$ by using a number of processors equals to the number of subsequences, applying the tree-generating scheme. At the beginning n processors generate n subsequences with the length 2: $(1, 1), (1, 2), \ldots, (1, n)$. Next, on the base of just-generated subsequences, n processors generate subsequences with the length 3

in the form of: $(1, 1, 1), (1, 1, 2), \ldots, (1, 1, n)$; $n - 1$ processors generate subsequences with the length 3 in the form of: $(1, 2, 2), (1, 2, 3), \ldots, (1, 2, n)$; $n - 2$ processors generate subsequences with the length 3 which begins from (1,3), etc. On the next level (in the second iteration) $\binom{n+3-2}{3-1} = \binom{n+1}{2} = \frac{n(n+1)}{2}$ processors are used. Subsequences are stored in the memory as lists with pointers, so the generation of the next element of the list does not need to copy previous elements but generating the next element only and add the pointer to the previous element in time $O(1)$. Following such a method, after $m-1$ iterations $\binom{n+m-2}{m-1}$ subsequences with a length m will be generated and connected with $\binom{n+m-2}{m-1}$ processors which generate them (each processor connected with the last generated element from the list with the pointer to the previous element of the list). In the $m + 1$-step each processor will generate the last element of the subsequence, identical for all the subsequences, $j_m = n$. ■

References

[1] E. Alba, Parallel Metaheuristics. A New Class of Algorithms, Wiley, 2005.
[2] E. Alba, A.J. Nebro, J.M. Troya, Heterogeneous computing and parallel genetic algorithms, Journal of Parallel and Distributed Computing 62 (2002) 1362–1385.
[3] W. Bożejko, M. Wodecki, Parallel genetic algorithm for minimizing total weighted completion time, in: Lecture Notes in Computer Science, vol. 3070, Springer, 2004, pp. 400–405.
[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press and McGraw-Hill, 1990.
[5] T.G. Crainic, M. Toulouse, Parallel strategies for meta-heuristics, in: F. Glover, G. Cochenberger (Eds.), Handbook of Metaheuristics, Kluwer Academic Publishers, Norwell, MA, 2003, pp. 475–514.
[6] J. Grabowski, J. Pempera, New block properties for the permutation flow shop problem with application in tabu search, Journal of Operational Research Society 52 (2000) 210–220.
[7] T. James, C. Rego, F. Glover, Sequential and parallel path-relinking algorithms for the quadratic assignment problem, IEEE Intelligent Systems 20 (4) (2005) 58–65.
[8] E. Nowicki, C. Smutnicki, A fast tabu search algorithm for the permutation flow shop problem, European Journal of Operational Research 91 (1996) 160–175.
[9] E. Nowicki, C. Smutnicki, Some aspects of scatter search in the flow-shop problem, European Journal of Operational Research 169 (2006) 654–666.
[10] OR-Library: http://people.brunel.ac.uk/~mastjjb/jeb/info.html.
[11] C.R. Reeves, T. Yamada, Genetic algorithms, path relinking and the flowshop sequencing problem, Evolutionary Computation 6 (1998) 45–60.
[12] K. Steinhöfel, A. Albrecht, C.K. Wong, Fast parallel heuristics for the job shop scheduling problem, Computers and Operations Research 29 (2002) 151–169.
[13] E. Taillard, Robust taboo search for the quadratic assignment problem, Parallel Computing 17 (1991) 443–455.
[14] E. Taillard, Parallel taboo search techniques for the job shop scheduling problem, ORSA Journal on Computing 6 (1994) 108117.
[15] E. Taillard, Benchmarks for basic scheduling problems, European Journal of Operational Research 64 (1993) 278–285.
[16] C. Wang, C. Chu, J. Proth, Heuristic approaches for $n/m/F/\Sigma C_i$ scheduling problems, European Journal of Operational Research (1997) 636–644.

Wojciech Bożejko, assistant professor at Wroclaw University of Technology. He obtained M.Sc. in University of Wroclaw, Institute of Computer Science in 1999 and Ph.D. at Wroclaw University of Technology, Institute of Computer Engineering, Control and Robotics in 2003. He is an author of over 70 papers in journals and conference proceedings from the field of parallel processing, scheduling and optimization. He is also a reviewer of some journals in this field. He is interested in parallel algorithms, discrete optimization, scheduling and group theory. He is also a qualified musician. He graduated from the Academy of Music in Wroclaw in a specialization of the piano.