Parallel Simulated Annealing for the Job Shop Scheduling Problem

Wojciech Bożejko*, Jarosław Pempera, and Czesław Smutnicki

Institute of Computer Engineering, Control and Robotics
Wrocław University of Technology
Janiszewskiego 11-17, 50-372 Wrocław, Poland
wojciech.bozejko@pwr.wroc.pl, jaroslaw.pempera@pwr.wroc.pl,
czeslaw.smutnicki@pwr.wroc.pl

Abstract. This paper describes two parallel simulated annealing algorithms for the job shop scheduling problem with the sum of job completion times criterion. Some properties of the problem associated with the *block theory* have been presented and discussed. These properties allow us to introduce the effective neighborhood based on the adjacent swap type moves. In this paper, an original method for parallel calculation of optimization criterion value for set of solutions, recommended for the use in metaheuristics with single- and multiple- search trajectories is proposed. Additionally, the vector calculation method, that uses multiple mathematical instructions MMX supported by suitable data organization, is presented. Properties of parallel calculations are empirically verified on the PC with Intel Core 2 Duo processor on Taillard's benchmarks.

Keywords: parallel metaheuristics, scheduling, optimization, job shop, simulated annealing.

1 Introduction

Job shop scheduling problems follow from many real cases, which means that they own good practical applications as well as the industrial significance. Because of NP-hardness of the problem, despite the criteria value form, heuristics and metaheuristics are recommended as "the most reasonable" solution methods. The majority of these methods refers to the makespan minimization. We mention here, as an example, a few recent studies: Jain, Rangaswamy, and Meeran [7]; Pezella and Merelli [11]; Grabowski and Wodecki [4]; Nowicki and Smutnicki [10].

The job shop problem with general regular criteria is commonly regarded as *harder* than the job shop problem with the makespan criterion, mainly because of the lack of special properties that would reinforce the solution algorithm. Moreover, till now, there have not been discovered for the problem any *sequential accelerators* (properties that can speed up computations throughout skillful aggregation and decomposition of calculations). In this context, parallelization

* Corresponding author.

techniques are the only methods allowing ones, practical instances in reasonable time to be solved. Therefore they are especially desirable.

Some heuristics algorithms based on dispatching rules for the considered problem are presented in papers of Holthaus and Rajendran [6], Bushee and Svestka [3]. For the other regular criteria such as the total tardiness there are proposed metaheuristics based on various local search techniques: simulated annealing [5], [14], tabu search [2] and genetic search [8].

In this paper we propose a genuine method of cost function computing in parallel by using multi-processor system as well as a single-processor system with multiple mathematical instructions MMX. The obtained results can be applied directly to modern PCs equipped with a few processors, multi-core processors or processors with multiple mathematical instructions.

2 The Problem

We consider a manufacturing system with any structure consisting of m machines of a unit capacity given by the set $M = \{1, \dots, m\}$. In the system, there are processed n jobs given by set $J = \{1, 2, ..., n\}$. The job j-th requires the sequence of n_j operations indexed consecutively $(l_{j-1} + 1, ..., l_{j-1} + n_j)$, where $l_j = \sum_{i=1}^{j} n_i$, is the total number of operations of the first j jobs, $j = 1, 2, ..., n$, $(l_0 = 0)$, and $o = \sum_{i=1}^{n} o_i$. The operation x is to be processed on the machine $\mu_x \in M$ during an uninterrupted processing time $p_x > 0$, $x \in O = \{1, 2, \dots, o\}$. Our aim is to find the schedule under the following constraints: (1) each machine can process at most one product at a time, (2) each product can be processed by at most one machine at a time, (3) operations cannot be preempted.

The set of operations O can be decomposed into subsets $O_k = \{x \in O : \mu_x = k\}$, each of them contains operations to be processed on the machine k, $k \in M$. Let the permutation π_k defines the processing order of operations from the set O_k on machine k, and let Π_k be the set of all permutations on O_k. The processing order of all operations on machines is determined by the m-tuple $\pi = (\pi_1, \pi_2, ..., \pi_m)$, where $\pi \in \Pi_1 \times \Pi_2 \times ... \times \Pi_m$.

For any operation $j \in O$ and processing order given by π we define the machine predecessor/successor $\underline{s}_j$, $\overline{s}_j$ as well as the technological predecessor/successor $\underline{t}_j$, $\overline{t}_j$, according to the expressions below

$$\underline{s}_{\pi_i(j)} = \begin{cases} 0 & j = 1 \\ \pi_i(j-1) & j = 2, \dots, n_j, \end{cases} \qquad \underline{s}_{\pi_i(j)} = \begin{cases} \pi_i(j+1) & j = 1, \dots, n_j - 1, \\ 0 & j = n_j, \end{cases}$$

$$\overline{t}_{\pi_i(j)} = \begin{cases} 0 & j = l_{i-1} + 1, i \in J \\ j - 1 & otherwise, \end{cases} \qquad \overline{t}_{\pi_i(j)} = \begin{cases} 0 & j = l_i, i \in J \\ j + 1 & otherwise, \end{cases}$$

Note that if the index of predecessor of the operation j is 0, then j is the first operation of proper job or/and the first operation processed on the proper machine. Similarly, if the index of successor of the operation j is 0 then j is the last operation of proper job or/and the last operation processed on the proper machine. The machine predecessor/successor depends on π, but for simplicity in a notation we will not express it explicitly.

The schedule for the fixed processing order π is described by event vectors $S = (S_1, \ldots, S_o)$ and $C = (C_1, \ldots, C_o)$, where values S_j and C_j denote the starting time of operation j and its completion time. The schedule has to satisfy the following constraints

$$C_{\underline{t}_j} \leq S_j \qquad \underline{t}_j \neq 0, \;\; j \in O, \tag{1}$$

$$C_{\underline{s}_j} \leq S_j \qquad \underline{s}_j \neq 0, \;\; j \in O, \tag{2}$$

$$C_j = S_j + p_j \qquad j \in O. \tag{3}$$

Because of equation (3), the schedule can be represented by a single event vector, and we use C to this aim. The schedule C, for the fixed π, is feasible if it satisfies conditions (1)–(3). The constraint (1) follows from the technological processing order of operations inside job, whereas (2) from the unit capacity of machines. Our aim is to find the feasible processing order $\pi^* \in \Pi$, so that

$$C_{sum}(\pi^*) = \min_{\pi \in \Pi} C_{sum}(\pi), \tag{4}$$

where $C_{sum}(\pi) = \sum_{i \in O^L}^{n} C_i$ is the sum of jobs completion times and $O^L = \{i : i = l_i, \; i \in O\}$ is the set of the last operations of jobs.

It is convenient to represent the processing order π and the schedule C by using the following direct graph $G(\pi) = (O, R \cup E(\pi))$ with a set of nodes O and a set of arcs $R \cup E(\pi)$, where $R = \{(\underline{t}_j, j) : \underline{t}_j \neq 0, j \in O\}$ $E(\pi) = \{(\underline{s}_j, j) : \underline{s}_j \neq 0, j \in O\}$. The node $j \in O$ represents the j-th operation of a certain job and has the weight p_j. Arcs from the set R represent the processing order of operations in jobs and correspond to the constraint (1), whereas arcs from set $E(\pi)$ represent the processing order of operations on machines and correspond to the constraint (2). All arcs from both subsets have the weight zero. Let us start from some well-known facts.

Property 1. *The processing order π is feasible if $G(\pi)$ does not contain a cycle.*

Consider the graph $G(\pi)$ for a feasible π. Denote by U_i the longest path (i.e. sequence of nodes) going to node i and by d_i the length (including p_i) of U_i, $i \in O$.

Property 2. *For each feasible processing order π, there exists a feasible schedule C of the problem, such that $C_i = d_i$ and each C_i is as small as possible, $i \in O$.*

The path U_i can be written as $U_i = (u_i(1), u_i(2), \ldots, u_i(w_i))$, where $u_i(x) \in O$, $1 \leq x \leq w_i$, and w_i is the number of nodes in this path. Clearly, $u_i(w_i) = i$. Each path U_i, $i \in O$ can naturally be decomposed into several specific subpaths, so that each subpath contains nodes linked by the same type of arcs. We define *block* as the maximal subsequence $B_i^* = (u_i(g_i^*), ..., u_i(h_i^*))$ of U_i such that $\mu_{u_i(g_i^*)} = \mu_{u_i(g_i^*+1)} = \ldots = \mu_{u_i(h_i^*)}$ and $(u_i(j), u_i(j+1)) \in E(\pi)$ for all $i = g_i^*, \ldots, h_i^* - 1$,

$g_i^* < h_i^*$. A block corresponds to a sequence of operations (jobs) processed on the same machine without inserted an idle time. In further considerations we will be interested only in *non-empty* blocks, i.e. containing at least two operations.

Based on the properties of the job shop problem with the makespan criterion [4] one can prove the following properties of the problem.

Theorem 1. *Let $B_i^1, B_i^2, \ldots, B_i^{r_i}$ be decomposition of U_i, $i \in O^L$ for the acyclic graph $G(\pi)$. If the acyclic graph $G(\alpha)$ has been obtained from $G(\pi)$ through the modifications of π so that $C_{sum}(\alpha) < C_{sum}(\pi)$, then in $G(\alpha)$ at least one operation $x \in B_i^k$ is executed on earlier position than an original one in permutation π_{μ_x}, for some $k \in \{1, 2, ..., r_i\}$ and some $i \in O^L$.*

Property 3. *Let $B_i^k = (u_i(g_i^k), ..., u_i(h_i^k))$, $k \in \{1, 2, ..., r_i\}$, $i \in O^L$ for the acyclic graph $G(\pi)$. If the graph $G(\alpha)$ has been obtained from $G(\pi)$ through the interchanging two successive operations of B_i^k that $G(\alpha)$ is acyclic.*

Property 4. *All paths U_i, $i \in O$ and their lengths can be found in the time $O(o)$.*

3 Simulated Annealing

Simulated annealing (SA) method applies an analogy to the thermodynamic cooling process to avoid local minima and escape from them. The search trajectory is guided through the set of solution Π in a "statistically suitable" direction. The SA application to our problem is described briefly as follows. In each iteration the new processing order π' is selected randomly among those from the neighborhood $N(\pi)$ of current processing order π. This processing order (solution) can provide either $C_{sum}(\pi') \leq C_{sum}(\pi)$ or $C_{sum}(\pi') > C_{sum}(\pi)$. In the former case π' is accepted immediately as the new solution for the next iteration, i.e. $\pi = \pi'$. In the latter case π' is accepted as the new solution with the probability $\exp(\Delta/T)$, where $\Delta = C_{sum}(\pi') - C_{sum}(\pi)$ - and T is a parameter called a temperature at iteration. The temperature T is getting changed along iterations by the use of a cooling scheme. A number of iterations, say m, is performed at the fixed temperature. Although the cooling should be carried out very slowly, most of the authors consider the change of the temperature at every iteration ($m = 1$). Two schemes of the temperature modification are commonly used: geometric $T_{i+1} = \lambda_i T_i$ and logarithmic $T_{i+1} = 1/(1 + \lambda_i T_i)$, $i = 1, \ldots, N$, where N is the total number of iterations, λ_i is a parameter, and T_0 is an initial temperature.

Aarts and van Laarhooven [1] have also made several suggestions concerning the choice of initial solution π^0, initial temperature T_0, λ_i, m and the stopping criterion. They have proposed to select π^0 at random, which helps with randomizing the search and removing solution dependence on π^0. The initial temperature is set to be $k = 10$ times the maximum value $\Delta_{\max} = max_{1 \leq i \leq k}(C_{sum}(\pi^i) - C_{sum}(\pi^{i-1}))$ between any two successive perturbed solutions when both are accepted, where π^i is the current solution in i-th iteration of algorithm. The initial temperature is set $T_0 = -\Delta_{\max}/\ln(p)$, where $p = 0.9$, the logarithmic cooling scheme parameter $\lambda_i = ln(1+\delta)/3\sigma_i$ where δ is the parameter of closeness to the equilibrium (0.1 - 10.0) and σ_i is the standard deviation of $C_{sum}(\pi)$ for all π generated at the temperature T. The value m equals the number (or its fraction) of different solutions that can be reached from the given one by introducing a single perturbation.

3.1 Neighborhood

The neighborhood $N(V,\pi)$ of a solution π is defined as a set of new solutions generated by the set of moves $V(\pi)$. The move $v \in V(\pi)$ transforms (perturbs) solution $\pi \in \Pi$ into another one $\pi^{(v)} \in \Pi$. One of the well-known transition operators for the job shop problem is the swap operator which takes two adjacent operation x and y and insert the operation y into the original position of operation x and x into the original position of operation y. The swap move can be unambiguously described by the pair of adjacent operation $v = (x, y)$. The set of moves $V(\pi)$ consists of all such moves that can be applied to π. For each machine $i \in M$ there are $n_k - 1$ possible swap moves. Thus, the size of this set and neighborhood is $o - m = \sum_{k=1}^{m}(n_k - 1)$. Unfortunately, $N(V,\pi)$ contains a huge number of unfeasible solutions as well as a quite large number of solutions worse than π.

Based on the Theorem 1 and Property 3 we can reduce the set $V(\pi)$ to the set $X(\pi) \subset V(\pi)$ which consists of only feasible and perspective moves. Formally, the set $X(\pi)$ is defined by the following formula

$$X(\pi) = \{(x,y) \in V(\pi) : \; x = u_i(j), \; y = u_i(j+1), \\ j = g_i^k, \ldots, h_i^k - 1, \; k = 1, \ldots, r_i, \; i \in O^L\}. \tag{5}$$

The size of $X(\pi)$ strongly depends on the distribution of blocks in π.

3.2 The Representative Neighborhood

As already mentioned, in each iteration, the SA algorithm selects in $N(X,\pi)$ randomly a single solution neighboring to π. We consider also an alternative method of selecting neighbors which refers to the idea of representatives, see Nowicki and Smutnicki, [9], applied to the tabu search method for the permutation flow shop problem with a makespan criterion. This method has been also successively applied by Yamada and Reeves [12] for the permutation flow shop problem with the total completion time criterion.

In the *representative neighborhood* the large original neighborhood $N(X,\pi)$ is shared into small subsets (clusters). The representative of the cluster is the best solution in this cluster. Selection is made among representatives. Notice that selection of representatives requires significantly greater computational effort than for the conventional SA method. Hence, this method is useful for problems having effective *accelerators* and/or effective methods of parallel computing.

4 Parallel Computation of the Objective Function

In this section we propose the original method of parallel computation of C_{sum} criterion for the given set of neighbors $N(\pi)$ of the current solution π. At the begin, we will analyze properties of π and $\pi^{(v)}$, where v is a move. In the description we refer to the well known method of computing C_i, $i \in O$ values.

Fact 1. Values C_i, $i \in O$, can be found by using the recursive formula

$$C_i = \max(C_{\underline{s}_i}, C_{\underline{t}_i}) + p_i, \text{ where } C_0 = 0. \tag{6}$$

The application of (6) is correct if nodes of the graph are revised in a suitable order. Let $T_\pi = (t_1, \ldots, t_o)$ be a topological order of nodes in graph $G(\pi)$. Note that T_π can be perceived as a permutation of elements from the set O.

Fact 2. The topological order T_π, for the fixed feasible π, can be found in the $O(o)$ time.

Fact 3. Values C_i, $i \in O$, for the fixed feasible π, can be found by running (6) for $i = t_1, \ldots, t_o$. It requires the $O(o)$ time.

Using Facts (1)–(3) one can propose the following Procedure C of calculating C_i, $i \in O$, for the fixed π. The computational complexity of this procedure is $O(o)$.

PROCEDURE C

Step 1. Find the topological order T_π. If it does not exist return the *unfeasible solution.*

Step 2. Calculate values C_i, $i \in O$, by using (6) for $i = t_1, \ldots, t_o$, where $(t_1, \ldots, t_o) = T_\pi$.

Let us analyze the quick method of obtaining $T_{\pi^{(v)}}$ and C_i, $i \in O$, after the swap move $v = (x, y)$ made from π. Let T_π^{-1} be the inverse permutation to T_π. The element $T_\pi^{-1}(x)$ denotes the position of x in T_π. Clearly, we have $T_\pi^{-1}(x) < T_\pi^{-1}(y)$ for move $v = (x, y)$. It is easy to verify that $T_{\pi^{(v)}}$ can be obtained by reordering in T_π elements from position $T_\pi^{-1}(x)$ to position $T_\pi^{-1}(y)$. It takes $O(T_\pi^{-1}(y) - T_\pi^{-1}(x) + 1)$ time. For frequently met case $T_\pi^{-1}(y) = T_\pi^{-1}(x) + 1$ the computation complexity is $O(1)$. Unfortunately, regarding to C_i, the change of completion time of two swapped operations should be broadcasted to all successive operations; then updating of C_i from the position $T_\pi^{-1}(x)$ to position o, in $T_{\pi^{(v)}}$ obtained by reordering T_π, requires $O(o - T_\pi^{-1}(x) + 1)$ time.

Now, we are ready to present our parallel computing method dedicated to the fast calculation of objective function values for a set of neighbors. Among the known parallel computation models we select the vector calculations which are the most promising now and easy in hardware implementation. The selected model is the special case of the single instruction multiple data (SIMD) model.

Assume that the vector processor operates on vectors consisting of s elements. Let $V_s = \{v_1, \ldots, v_s\}$ be a subset of parallel computed neighbors and let

$\overline{C}_i = (\overline{C}_i^1, \ldots, \overline{C}_i^s)$, where $\overline{C}_i^k$, $k = 1, \ldots, s$, denote completion times of operations $i \in O$ calculated for the k-th neighbor. In a similar way we define the vector of processing times $\overline{P}_i = (\overline{P}_i^1, \ldots, \overline{P}_i^s)$, obviously $\overline{P}_i^1 = \overline{P}_i^2 =, \ldots, = \overline{P}_i^s = p_i$. For each $v = (x, y) \in V_s$ we define three positions in T_π : (i) $f_{T_\pi}(v) = T_\pi^{-1}(x)$ the first position of updating T_π, (ii) $l_{T_\pi}(v) = T_\pi^{-1}(y) = T_\pi^{-1}(\overline{s}_x)$ the last position of updating T_π, (iii) $l_C(v)$ the last position of updating $\overline{C}^k$. The $l_C(v) = T_\pi^{-1}(y)$ if operation y is the last operation executed on the proper machines and $l_C(v) = T_\pi^{-1}(\overline{s}_y)$ in the opposite case. Finally, we reorder moves from the set V_s according to the non-decreasing value of $l_C(v)$. The proposed method is outlined in the Procedure P.

PROCEDURE P

```
Step 0: set $\overline{C}_0 = 0$.
Step 1: for $i = 1$ to $o$ do
Step 1.1: Calculate values $\overline{C}_{t_i}$ by using (6).
Step 1.2: for each $k$ such that $l_C(v_k) = i$ do
Step 1.2.1: execute move $v_k$ in $\pi$
Step 1.2.2: reorder $T_\pi$ from position $f_{T_\pi}(v_k)$ to $l_{T_\pi}(v_k)$
Step 1.2.3: calculate values $\overline{C}_{t_i}^k$ from position $f_{T_\pi}(v_k)$ to $l_C(v_k)$.
Step 1.2.4: restore $\pi$ and $T_\pi$
```

The initial Step 0 is clear. In the Step 1.1 all values of the vector $\overline{C}_{t_i}$ are calculated in parallel. This step is performed for $t_1, \ldots, t_o$ and takes $O(o)$ time. It is easy to verify that computations in Step 1.1 are performed according to order T_π determined by π. Therefore, for each π_v, $v \in V_s$ it is necessary to recalculate the completion times for all operations whose position have changed, i.e. from the position $f_{T_\pi}(v_k)$ to $l_{T_\pi}(v_k)$ and additionally for the successor of the operation y.

The computational complexity of the Step 1.2.1 is $O(1)$, for Step 1.2.2 and 1.2.4 is $O(l_{T_\pi}(v_k) - l_{T_\pi}(v_k))$. Step 1.2.3 is the most time consuming and takes $O(l_C(v_k) - l_{T_\pi}(v_k))$ time. The total time required by Step 1.2 is $O(\sum_{v \in V_s}(l_C(v) - f_{T_\pi}(v)))$. In the optimistic case, namely $l_C(v_k) - l_{T_\pi}(v_k) = 2$ for all $k = 1, ..., s$, the computational complexity of the algorithm is $O(o + s)$.

4.1 Multiple Mathematical Instructions

Contemporary used PCs are equipped with processors having the extended instruction MMX set. These special instructions allow one to make vectoring computations. The single instruction operates in a single processor cycle on extended registers (8 bytes). In the MMX set, it can be distinguished three groups of vector instructions operating on vector 8×1, 4×2, 2×4, where the former number denotes the vector size and the latter number – the data size. Since for the tested instances all performed values (C_i, p_i) can be coded on two bytes, we can perform vectoring operation on the vector of a size equals 4.

The seven main steps of parallel computation of expression (6) using MMX intrinsics are shown in Fig. 1. In the steps 1 and 2, the MMX registers mm0 and mm1 are loaded by data following from the vectors $\overline{C}_{s_i}$ and $\overline{C}_{t_i}$ respectively. The calculation of value $\max(\overline{C}_{s_i}, \overline{C}_{t_i})$ are decomposed into two steps: 3 and 4. At first, the value of $\max(0, \overline{C}_{s_i} - \overline{C}_{t_i})$ are calculated by using subtracts with saturation MMX intrinsic (step 3), the result is stored into the $mm0$ register. Afterwards the content of the $mm0$ register is increased by $\overline{C}_{t_i}$ (step 4). The values of vector $\overline{P}_i$ are stored in the $mm1$ register (step 5) and added to the contents of the $mm0$ register (step 6). The final results (contents of the $mm0$ register) are stored into memory in the step 7.

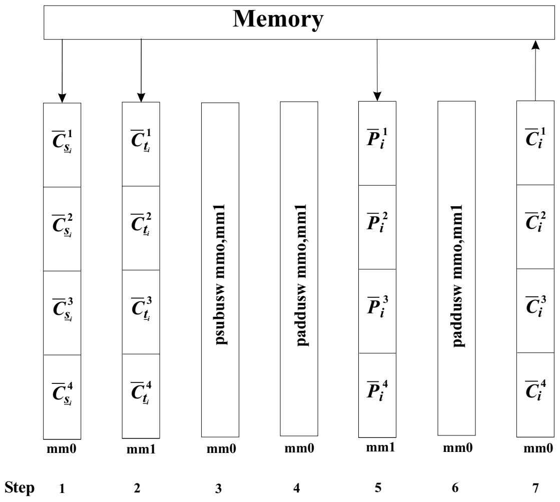

Fig. 1. Parallel computation of expression (6) for the given operation i

5 Computational Experiments

We have implemented three algorithms based on the simulated annealing method. In the first algorithm PSA-R we have used the representative-based neighborhood. From the neighborhood $N(X, \pi)$ we can select randomly $s = 4$ neighbors and we compute in parallel the value of the objective function. From the set obtained now we select the best solution. In the second PSA algorithm, for each solution $s = 4$ moves from $X(\pi)$ are generated at random. Moves are applied in turn to order π until a new solution is accepted. The objective function for all the solutions is computed in parallel. It is easy to observe that PSA emulates a traditional one-thread simulated annealing (SA) algorithm. The classic SA algorithm is the third from the implemented and tested algorithms.

Algorithms were coded in Visual C++ 2008 Express Edition, ran on a PC with Intel Core 2 Duo 2.66 GHz processor and the Windows XP operating system, and tested on 50 benchmark instances provided by Taillard [13]. The benchmark set contains 5 groups of hard instances in different sizes. For each size (group) $n \times m$: $15 \times 15, 20 \times 15, 20 \times 20, 30 \times 15, 30 \times 20$ a set of 10 instances was provided. In our tests all the values of tuning parameters for algorithms are found in the automatic way described in the previous section. Each algorithm was terminated after performed 1000 iterations. At the fixed temperature, the SA-R algorithm performs n iteration, whereas SA and PSA algorithms $4n$ iteration, i.e. all algorithms calculated the objective function value for the same number of solutions.

For each test instance and for each run of algorithm we have collected the following values: π^{ref} – reference solution – the best solution found in all runs of algorithms, $PRD = 100 \cdot (C_{sum}(\pi) - C_{sum}(\pi^{ref}))/C_{sum}(\pi^{ref})$ – the value of the percentage relative difference between C_{sum} function values for the solution π and reference solution π^{ref}, CPU – total computations time (in seconds). For each instance and for each algorithm based on 10 solutions generated during each of 10 runs we have calculated the following values: $MPRD$ – minimal PRD value, $APRD$ – average PRD value, $ACPU$ – average computation time (in seconds).

Table 1. Computational results of PSA-R, PSA and SA for PRD and CPU values

Group	PSA-R		PSA (SA)		ACPU time			Speedup ratio	
	MPRD	APRD	MPRD	APRD	PSAR	PSA	SA	PSAR	PSA
15×15	1.27	2.72	0.11	1.07	3.9	10.3	18.7	4.7	1.8
20×15	1.38	3.10	0.11	1.21	6.7	19.3	34.2	5.1	1.8
20×20	0.51	1.72	0.23	1.43	11.3	33.6	61.6	5.4	1.8
30×15	0.60	3.02	0.42	1.45	14.8	46.9	81.2	5.5	1.7
30×20	0.20	2.23	1.11	1.97	26.1	83.9	147.2	5.6	1.8

Table 1 shows results of computational experiments. The main observation is that the proposed method of SA algorithm parallelization significantly reduces the computation time. The speedup values are from 4.7 to 5.6 and increase with the increasing number of machines. It is easy to notice that the speedup is greater than the theoretical one (≤ 4) (we are obtaining superlinear speedup). The additional speedup is obtained due to the MMX instruction utilization, which eliminates branches in the executing code. The branches are essential for implementation of max function in x86 set of instruction. Comparing computations time of PSA and SA one can observe that the speedup is significantly smaller (1.8). With respect to PRD values it has been pointed that PSA provides better results than PSA-R for the majority of groups of instance. For the first five groups the MPRD values are from 0.5 to 1.4 for PSA-R, whereas from 0.1 to 0.4 for PSA (SA). In the last group of instances, conversely PSA-R provides better results (MPRD=0.2) than PSA-R (MPRD=0.2). It can be notice that for this group of instances we observe the highest speedup value.

6 Conclusions

To the best of our knowledge, this is the first paper which deals with the small-grain parallelization of algorithms for the job shop problem. A special attention has been paid to the kind of parallelism which can be easily applied to the new generation of processors installed in PCs, with multiple cores (dual, quad, etc.) as well as with the extended set of instructions (such as MMX and SSE2). The proposed methods have been applied successfully to various simulated annealing metaheuristics for the job shop problem with C_{sum} criterion.

References

1. Aarts, E.H.L., van Laarhoven, P.J.M.: Simulated annealing: a pedestrain review of the theory and some aplications. In: Deviijver, P.A., Kittler, J. (eds.) Pattern Recognition and Applications. Springer, Berlin (1987)
2. Armentano, V.A., Scrich, C.R.: Tabu search for minimizing total tardiness in a job shop. International Journal of Production Economics 63(2), 131–140 (2000)
3. Bushee, D.C., Svestka, J.A.: A bi-directional scheduling approach for job shops. International Journal of Production Research 37(16), 3823–3837 (1999)
4. Grabowski, J., Wodecki, M.: A very fast tabu search algorithm for job shop problem. In: Rego, C., Alidaee, B. (eds.) Metaheuristic optimization via memory and evolution. Tabu search and scatter search, vol. 30, pp. 117–144. Kluwer Academic Publ., Boston (2005)
5. He, Z., Yang, T., Tiger, A.: An exchange heuristic embedded with simulated annealing for due-dates job-shop scheduling. European Journal of Operational Research 91, 99–117 (1996)
6. Holthaus, O., Rajendran, C.: Efficient jobshop dispatching rules: further developments. Production Planning and Control 11, 171–178 (2000)
7. Jain, A.S., Rangaswamy, B., Meeran, S.: New and stronger job-shop neighborhoods: A focus on the method of Nowicki and Smutnicki (1996). Journal of Heuristics 6(4), 457–480 (2000)
8. Mattfeld, D.C., Bierwirth, C.: An efficient genetic algorithm for job shop scheduling with tardiness objectives. European Journal of Operational Research 155(3), 616–630 (2004)
9. Nowicki, E., Smutnicki, C.: A fast tabu search algorithm for the permutation flow shop problem. European Journal of Operational Research 19(1), 160–175 (1996)
10. Nowicki, E., Smutnicki, C.: An advanced tabu search algorithm for the job shop problem. Journal of Scheduling 8, 145–159 (2005)
11. Pezzella, F., Merelli, E.: A tabu search method guided by shifting bottleneck for the job-shop scheduling problem. European Journal of Operational Research 120, 297–310 (2000)
12. Reeves, C.R., Yamada, T.: Solving the Csum Permutation Flowshop Scheduling Problem by Genetic Local Search ICEC 1998. In: IEEE International Conference on Evolutionary Computation, pp. 230–234 (1998)
13. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64, 278–285 (1993)
14. Wang, T.Y., Wu, K.B.: An eficient configuration generation mechanism to solve job shop scheduling problems by the simulated annealing. International Journal of Systems Science 30(5), 527–532 (1999)