Parallel Calculating of the Goal Function in Metaheuristics Using GPU

Wojciech Bożejko, Czesław Smutnicki, and Mariusz Uchroński

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

Abstract. We consider a metaheuristic optimization algorithm which uses single process (thread) to guide the search through the solution space. Thread performs in the cyclic way (iteratively) two main tasks: the goal function evaluation for a single solution or a set of solutions and management (solution filtering and selection, collection of history, updating). The latter task takes statistically 1-3% total iteration time, therefore we skip its acceleration as useless. The former task can be accelerated in parallel environments in various manners. We propose certain parallel small-grain calculation model providing the *cost optimal* method. Then, we carry out an experiment using Graphics Processing Unit (GPU) to confirm our theoretical results.

1 Introduction

Almost all combinatorial optimization tasks formulated for scheduling problems are strongly NP-hard. Currently known exact solution algorithms (dedicated to find global optimum) own exponential computational complexity which causes unacceptable long solution time for instances that come from practice. In this context one can propose two, not mutually conflicted, approaches, which allow one to solve large-size instances in acceptable time: (1) approximate methods (chiefly metaheuristics), (2) parallel methods. The best hybrid combination of both is the one which we really needed.

The most promising metaheuristic algorithms search solution space in a certain intelligent way. Quality of the best solutions generated by these algorithms strongly depends on the number of analyzed solution, and thus on the running time. Time and quality have opposing tendency in such a sense, that finding a better solution requires a significant computation time growth. Through the parallel processing one can increase the number of checked solutions (per time unit). In this paper there are proposed several solutions algorithms dedicated to a single solution analysis employed in widely used metaheuristics.

In the scope of a single-thread search, dedicated fundamentally for uniform multiprocessor system of small granularity, one can distinguish a few parallel approaches taking into account various design technologies and different needs applied by modern discrete optimization algorithms, namely: (a) single solution

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 1014–1023, 2009.

analysis (dedicated for simulated annealing SA, simulated jumping SJ, random search RS), (b) local neighborhood analysis (for tabu search TS, adaptive memory search AMS, descending search DS), (c) analysis of population of distributed solutions (for genetic approach GA, scatter search SS). In each case special attention should be paid to efficiency, cost and speedup of methods depending on the used parallel computing environment. For each algorithm, theoretical evaluation of its numerical properties as well as comparative analysis of potential benefits from proposed approaches are expected. In this paper we deal chiefly with the approach (a).

In this paper we use the following notions which are fundamental in the parallel computing area, see e.g. [4]: theoretical parallel architectures, theoretical models of parallel computations, granularity, threads, cooperation, speed up, efficiency, cost, cost optimality, computational complexity, real parallel architectures and parallel programming languages.

This work constitutes the continuation of authors research on constructing efficient algorithms applied to solve hard combinatorial problems ([2,5,6]).

2 Permutation Flow Shop Problem

We consider, as the test case, the well-known in the scheduling theory, strongly NP-hard problem, called the permutation flow-shop problem with the makespan criterion and denoted by $F||C_{max}$. Skipping consciously the long list of papers dealing with this subject we only refer the reader to the recent reviews and the best up-to-now algorithms [5,6].

The problem has been introduced as follows. There is 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. The 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. The operation O_{ij} reflects processing of job j on the machine i with the processing time $p_{ij} > 0$. Once started 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 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 the recursive formula

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

with initial conditions $C_{i\pi(0)} = 0$, $i = 1, 2, \ldots, m$, $C_{0\pi(j)} = 0$, $j = 1, 2, \ldots, n$. Computational complexity of (2) is $O(mn)$.

Values C_{ij} from the equation (2) can be also determined by using a graph model of the flow shop problem. For a given sequence of jobs execution $\pi \in \Pi$ we create a graph $G(\pi) = (M \times N,\ F^0 \cup F^*)$, where $M = \{1, 2, \ldots, m\}$, $N = \{1, 2, \ldots, n\}$, $F^0 = \bigcup_{s=1}^{m-1} \bigcup_{t=1}^{n} \{((s,t),(s+1,t))\}$ is a set of technological arcs (vertical) and $F^* = \bigcup_{s=1}^{m} \bigcup_{t=1}^{n-1} \{((s,t),(s,t+1))\}$ is a set of sequencing arcs (horizontal). Arcs of the graph $G(\pi)$ have no weights, but each vertex (s,t) has weight $p_{s,\pi(t)}$. The time C_{ij} of completing job $\pi(j)$, j = 1,2,...,n on the machine i, i = 1,2,...,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 $F||C_{max}$ problem the value of the criterion function for the fixed sequence π equals the length of the critical path in the graph $G(\pi)$.

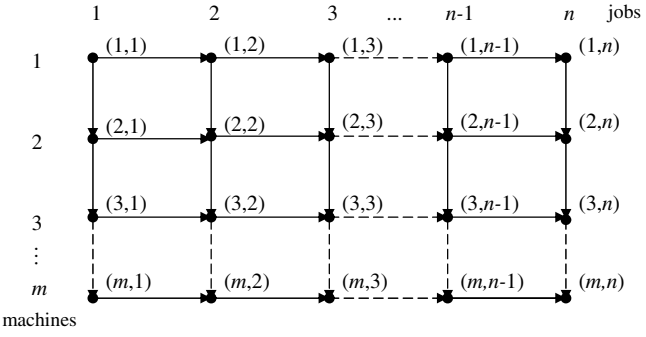

Fig. 1. Graph $G(\pi)$

3 Searching

We consider a solution method which uses only one thread to manage the search process. The process executes cyclic iterations consisting of: (1) numerical calculations (i.e. the goal function value determination), (2) managing functions (i.e. solution selection, calculations memory realization, solution acceptation). Activities connected with managing take statistically 1-3% of the iteration's time. Therefore, its acceleration by using parallel environment gives us insignificant benefit. From the other side, the numerical calculations acceleration by implementing in the parallel or distributed architecture may significantly improve the efficiency of the solution space's search algorithm.

We take advantage of the following well-known facts for the PRAM parallel computer model:

Fact 1. *Sequence of prefix sums* $(y_1, y_2, ..., y_n)$ *of input sequence* $(x_1, x_2, ..., x_n)$ *such, that*

$$y_k = y_{k-1} + x_k = x_1 + x_2 + ... + x_k \text{ for } k = 2, 3, ..., n$$

where $y_1 = x_1$ *can be calculated in time* $O(\log n)$ *on the EREW PRAM machine with* $O(n/\log n)$ *processors.*

From what we have stated above we can assume that the sum of n values can be calculated in time $O(\log n)$ on $O(n/\log n)$ – processors EREW PRAM machine.

Fact 2. *The minimal and the maximal value of input sequence* $(x_1, x_2, ..., x_n)$ *can be determined in the time* $O(\log n)$ *on the EREW PRAM machine with* $O(n/\log n)$ *processors.*

Fact 3. *The value of* $y = (y_1, y_2, \ldots, y_n)$ *where* $y_i = f(x_i)$, $x = (x_1, x_2, \ldots, x_n)$ *can be calculated on the CREW PRAM machine with* n *processors in a time* $O(c) = O(1)$, *where* c *is a time needed to calculate the single value of* $y_i = f(x_i)$.

Fact 4. *The problem formulated in the previous fact can be calculated in the time* $O(\log n)$ *on* $O(n \log n)$ *processors.*

If we do not have such a big number of processors we can use such a fact to keep the same cost:

Fact 5. *If the algorithm* A *works on* p *– processors PRAM in the time* t, *then for every* $p' < p$ *exists an algorithm* A' *for the same problem which works on* p' *– processors PRAM in time* $O(pt/p')$.

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* $F||C_{max}$ *and* $F||C_{sum}$ *can be found on the CREW PRAM machine in the 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 the order $k = 1, 2, \ldots, n+m-1$. The cluster k is processed in parallel on at most m processors. The calculations sequence is shown in Fig. 2 on the background of the 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 $C_{sum} = \sum_{j=1}^{n} C_{m,j}$ we need to add n values $C_{m,j}$, which can be done 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 $F||C_{max}$ and $F||C_{sum}$ problems is $O(n + m)$ by using m processors.

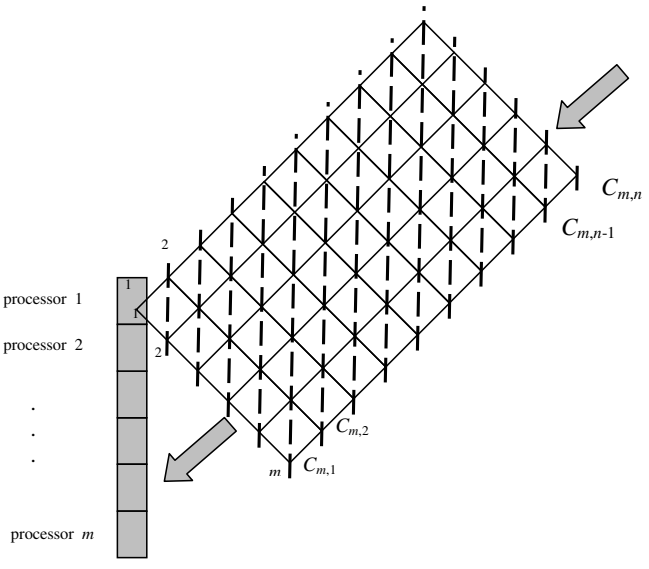

Fig. 2. Computing C_{ij} order using m threads

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

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

Proof. Without the loss of generality one can assume that $\pi = (1, 2, \ldots, n)$. We based on the scheme of calculations shown in Fig. 2. Let $p \leq m$ be the number of used processors. 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. Parallel computations on the level k are performed in the time $O(\left\lceil \frac{n_k}{p} \right\rceil)$. The total calculation time is equal to the sum over all levels and is of order

$$\sum_{k=1}^{d} \left\lceil \frac{n_k}{p} \right\rceil \leq \sum_{k=1}^{d} \left(\frac{n_k}{p} + 1 \right) = \frac{nm}{p} + d = \frac{nm}{p} + n + m - 1. \tag{3}$$

We are seeking for the number of processors p, $1 \leq p \leq m$, for which efficiency of parallel algorithm is $O(1)$, which ensures cost optimality of the method. The

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

value p can be found from the following condition

$$\frac{1}{p}\,\frac{nm}{\frac{nm}{p}+n+m-1}=c=O(1) \tag{4}$$

for some constant $c<1$. After a few simple transformations of (4) we get

$$p=\frac{nm}{n+m-1}\left(\frac{1}{c}-1\right)=O(\frac{nm}{n+m}). \tag{5}$$

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

$$O(\frac{nm}{p}+n+m-1)=O(\frac{nm}{\frac{nm}{n+m}}+n+m)=O(n+m). \tag{6}$$

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

The method is cost optimal and allows one to control 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 the 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.

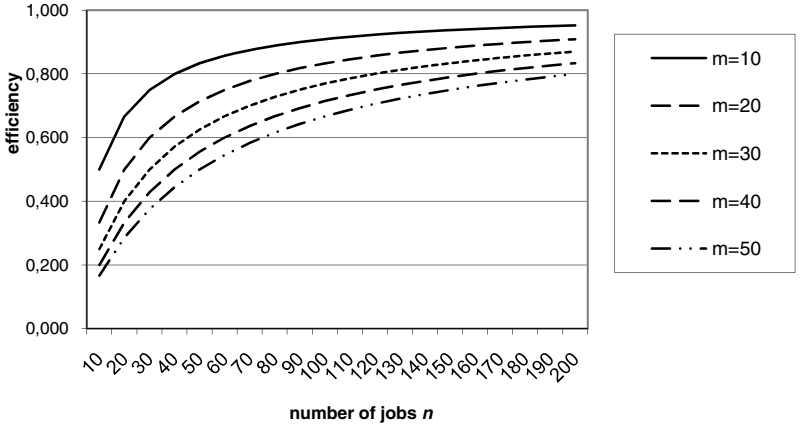

Fig. 3. Efficiency of the method from Theorem 2

4 Experimental Results

The parallel algorithm for the considered problem of calculating makespan in permutation flow shop problem was coded in C (CUDA) for GPU, ran on the

Tesla C870 GPU (512 GFLOPS) with 128 streaming processor cores and tested on the benchmark problems of Taillard. The benchmark set contains 120 particularly hard instances of 12 different sizes. For each size (group) $n \times m$: 20×5, 20×10, 20×20, 50×5, 50×10, 50×20, 100×5, 100×10, 100×20, 200×10, 200×20, 500×20, a sample of 10 was provided. The considered algorithm showed as Algorithm 1 uses m GPU processors for calculating makespan. Its sequential version is obtained by assigning $p = 1$ and it is also executed on GPU. Algorithm 2 presents details of the parallel method coded in CUDA.

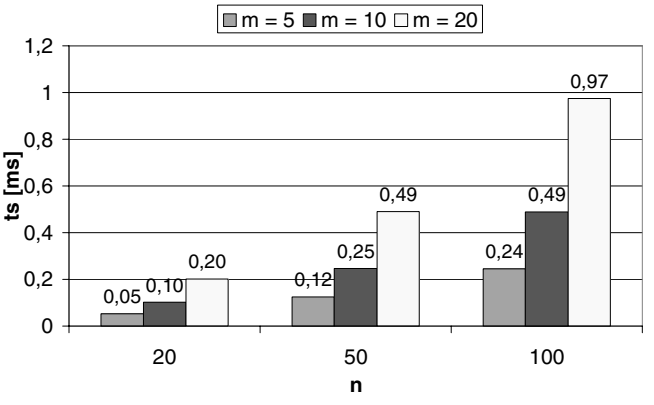

Fig. 4. The one thread algorithm computational times

Algorithm 1. Parallel algorithm of the makespan calculating

parfor $i = 1$ to p (for each processor)
  for $j = 1$ to $n + m - 1$ **do**
    $x = j - i + 1$
    if $x \geq 1$ and $x \leq n$ **then**
      $C_{x,i} = \max\{C_{x-1,i}, C_{x,i-1}\} + p_{x,i}$
    end if
  end for
end of parfor.

Each multiprocessor of the TESLA C870 GPU has on-chip memory of the four following types:

- one set of local 32-bit *registers* per processor,
- a parallel data cache or *shared memory* that is shared by all scalar processors cores and is where the shared memory space resides,
- a read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory,
- a read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory.

Algorithm 2. Parallel algorithm for makespan calculating coded in CUDA

```
__global__ void cmax(int *c, int n, int m, int *cmax) {
  int idy;
  int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
  if(idx<=m) {
    for(int j=1;j<m+n;j++) {
      idy = j - idx + 1;
      if(idy>=1&&idy<=n) {
        c[idx*(n+1)+idy]=max(c[(idx-1)*(n+1)+idy],
        c[idx*(n+1)+(idy-1)]) + tex2D(tex,idy ,idx);
        if(idx==m&&idy==n) cmax[0]=c[idx*(n+1)+idy];
} } } }

int main() { //Kernel invocation
  int blockSize = 16;
  int nBlocks = M/blockSize + (M%blockSize == 0?0:1);
  cmax<<< nBlocks, blockSize >>> (devC, N, M, devCmax);
}
```

There is no need to copy table with processing times before each calculating of makespan so this table was copied once to very fast read-only texture memory. Therefore timings are measured without this preparing time. Texture memory is cached. Table C was allocated in global memory. After calculations makespan on GPU value of C_{max} is copied to the CPU. This operation is very fast (only one-element table is copied).

The sequential algorithm using one GPU processor was coded with the aim of determining the speedup value which can be obtained by a parallel algorithm. Table 1 shows computational times for the sequential and the parallel algorithm as well as speedup. The value of relative speedup s can by found by the following

Table 1. Experimental results for Taillard's instances

$n \times m$	p	t_p $[ms]$	t_s $[ms]$	speedup s
20×5	5	0.0271	0.0526	1.94
20×10	10	0.0309	0.1022	3.31
20×20	20	0.0386	0.2014	5.22
50×5	5	0.0480	0.1244	2.59
50×10	10	0.0518	0.2469	4.76
50×20	20	0.0601	0.4909	8.16
100×5	5	0.0835	0.2449	2.93
100×10	10	0.0874	0.4885	5.59
100×20	20	0.0968	0.9740	10.06
200×10	10	0.1582	0.9716	6.14
200×20	20	0.1697	1.9403	11.43
500×20	20	0.3919	4.8392	12.35

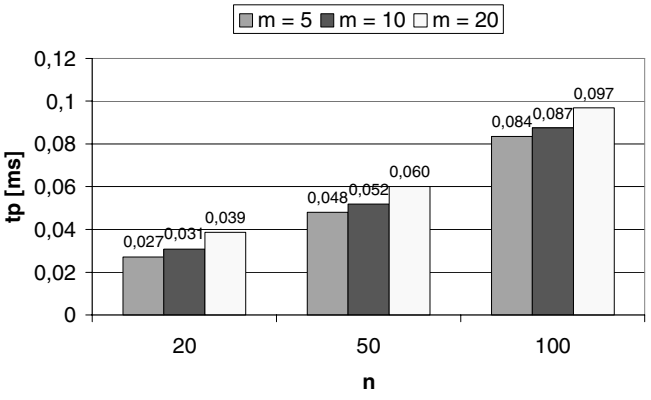

Fig. 5. The parallel algorithm computational times

expression $s = \frac{t_s}{t_p}$, where t_s constitutes the computational time of sequential algorithm and t_p - computational time of parallel algorithm. Figure 4 shows computational times of the sequential algorithm for different sizes of problem. The algorithm computational time increases with the size of a problem. For the fixed number of jobs 100% increasing of the number of machines results in 100% increasing computational time.

Figure 4 shows computational times of the parallel algorithm for different sizes of the problem. For the fixed number of jobs the increase of the number of machines results in a small computational time increase. Increase size of problem results in increasing the speedup of parallel algorithm in comparison with the sequential algorithm. Figure 6 confirms theoretical effectiveness on the basis of Theorem 2 shown on Figure 3. Obtained experimental results are fully common with the theoretical analysis.

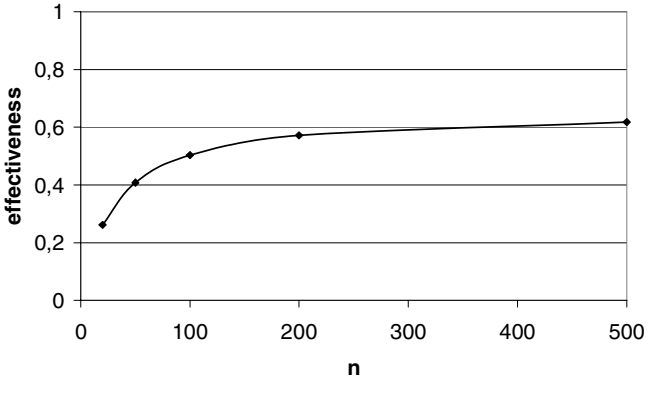

Fig. 6. Experimental effectiveness

5 Conclusions

We propose taking advantage of modern many-core computing processors GPU to accelerate local search algorithm's work. Results, for a classic flow shop scheduling problem, allow us to obtain a speedup which is proportional to the number of machines m from the problem definition. Theorems presented in this paper can be easily extended to the EREW PRAM model, with exclusive read, which requires an additional $O(\log n)$ time, however the architecture of the used GPU allows us to implement CREW algorithms (with a possibility of concurrent read). On the other side the shared memory usage is connected with a huge time latency (400-600 cycles of the clock) comparing to the local memory of a processor or the textures (constant) memory. Therefore, a further acceleration of the algorithm is possible under condition of the methods adaptation to the GPU memory access specific.

References

1. Aarts, E.H.L., Verhoeven, M.: Local search. In: Dell'Amico, M., Maffioli, F., Martello, S. (eds.) Annotated bibliographies in Combinatorial Optimization, Wiley, pp. 163–180. Wiley, Chichester (1997)
2. Bożejko, W.: Parallel scheduling algorithms, PhD Thesis, Report 29/2003, Institute of Engineering Cybernetics, Wrocław University of Technology 1–205 (2003)
3. Fiechter, C.N.: A parallel tabu search algorithm for large traveling salesman problems. Discrete Applied Mathematics 51, 243–267 (1994)
4. Grama, A., Kumar, V.: State of the Art in Parallel Search Techniques for Discrete Optimization Problems. IEEE Transactions on Knowledge and Data Engineering 11, 28–35 (1999)
5. Nowicki, E., Smutnicki, C.: A fast tabu search algorithm for the permutation flow shop problem. European Journal of Operational Research 91, 160–175 (1996)
6. Nowicki, E., Smutnicki, C.: Some aspects of scatter search in the flow-shop problem. European Journal of Operational Research 169, 654–666 (2006)
7. Porto, S.C., Ribeiro, C.C.: Parallel tabu search message passing synchronous strategies for task scheduling under precedence constraints. Journal of Heuristics 1, 207–223 (1995)
8. Porto, S.C., Ribeiro, C.C.: A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. International Journal of High Speed Computing 7, 45–71 (1995)
9. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64, 278–285 (1993)