

Grid Branch-and-Bound for Permutation Flowshop

Maciej Drozdowski, Paweł Marciniak, Grzegorz Pawlak, Maciej Płaza

Institute of Computing Science, Poznań University of Technology,
Piotrowo 2, 60-965 Poznań, Poland
{Maciej.Drozdowski,Grzegorz.Pawlak}@cs.put.poznan.pl
Maciej.Plaza@gmail.com

Abstract. Flowshop is an example of a classic hard combinatorial problem. Branch-and-bound is a technique commonly used for solving such hard problems. Together, the two can be used as a benchmark of maturity of parallel processing environment. Grid systems pose a number of hurdles which must be overcome in practical applications. We give a report on applying parallel branch-and-bound for flowshop in grid environment. Methods dealing with the complexities of the environment and the application are proposed, and evaluated.

Keywords: branch-and-bound, flowshop, grid computing.

1 Introduction

Solving a hard combinatorial optimization problem on the grid is considered in this paper. Flowshop is a classic **NP**-hard combinatorial optimization problem. A set of test instances has been proposed for this problem [16]. Despite 20-year attempts, some of the instances remain unsolved. Thus, flowshop became a benchmark problem in combinatorial optimization, and deterministic scheduling.

Though many methods have been proposed to solve combinatorial optimization problems, Branch-and-Bound (BB) remains the main algorithm delivering guaranteed optimum solutions. BB partially enumerates the solutions, and this process often can be envisioned as searching a tree. Various approaches are used to prune the tree, but still search spaces of combinatorial problems remain huge. Therefore, parallel branch-and-bound (PBB) is used to reduce the running time [1, 3]. BB parallelization introduces a host of new complications [3, 7]. Overcoming them requires making design decisions which influence performance of PBB.

By combining computational resources of many institutions Grid environments provide computational power not available to any single institution separately. Therefore, grid is a very attractive computing platform for combinatorial optimization applications. Yet, grid has a number of inherent peculiarities such as resource heterogeneity and volatility which must be dealt with when designing a robust application.

Overall, the three elements: benchmark hard combinatorial problem, parallel branch-and-bound, the grid environment make the implementation practically

hard. Therefore, these three elements can serve as a benchmark of *maturity* and *robustness* in parallel processing. With such a goal 2nd Grid Plugtests Flowshop Challenge was organized by European Telecommunications Standards Institute in 2005 [4]. A testbed for solving benchmark hard problems on the grid was provided to verify usability and maturity of the computational platform and the programming environment. The code presented in this paper scored 1st prize in the above competition. This paper is dedicated to the grid parallel processing, and the challenges which must be faced by application designers, rather than to solving flowshop problem itself. Flowshop serves as a benchmark here.

The rest of this paper is organized as follows. In the next section we define flowshop problem. Section 3 is dedicated to grid middleware, and the computational platform peculiarities. In Section 4 parallel branch-and-bound algorithm is described. In Section 5 we report on the results obtained by our implementation of PBB.

2 Flowshop

Flowshop (FS) problem is defined as follows. Sets $\{M_1, \dots, M_m\}$ of m dedicated machines, and $\mathcal{J} = \{J_1, \dots, J_n\}$ of n jobs are given. Each job J_j consists of a sequence of m operations $(O_{j1}, O_{j2}, \dots, O_{jm})$. Operations O_{ji} are executed on machine M_i , for all $j = 1, \dots, n$. Consequently, operations $O_{j1}, O_{j2}, \dots, O_{jm}$ proceed through machines M_1, M_2, \dots, M_m , as for example, cars on the production line. Execution time of operation O_{ji} is a non-negative integer p_{ji} . Only active schedules are allowed, which means that operations are started as soon as it is possible. The problem consists in finding the shortest schedule. We will denote schedule length by C_{max} .

In general, the operations of different jobs can be executed on a given machine in an arbitrary order. This results in at most $(n!)^m$ possible solutions of the problem. In *permutation flowshop* jobs proceed through the machines in the same order. It means that the sequence of operations from different jobs is the same on all the machines. In this paper we consider permutation flowshop. Though there are 'only' $n!$ solutions for the permutation flowshop, the number of possible solutions is still very big in practice. Flowshop problem is polynomially solvable for $m = 2$ [8], and strongly **NP**-hard for $m \geq 3$ [5].

A set of test instances is known [16] for flowshop problem. In the following we refer to Taillard's instances [17] which sizes are from $(n \times m)$: 20×5 to 500×20 . Over the years, flowshop has been used as a benchmark problem to test new methods in combinatorial optimization [6, 9, 14].

3 The Test Platform

In this section we characterize grid environment in the 2nd Grid Plugtests Flowshop Challenge. The information mentioned here follows [4]. Initial runs of the algorithm were also executed on an SMP SunFire 6800 machine with 24 CPUs and 96GB RAM in Poznań Supercomputing and Networking Center.

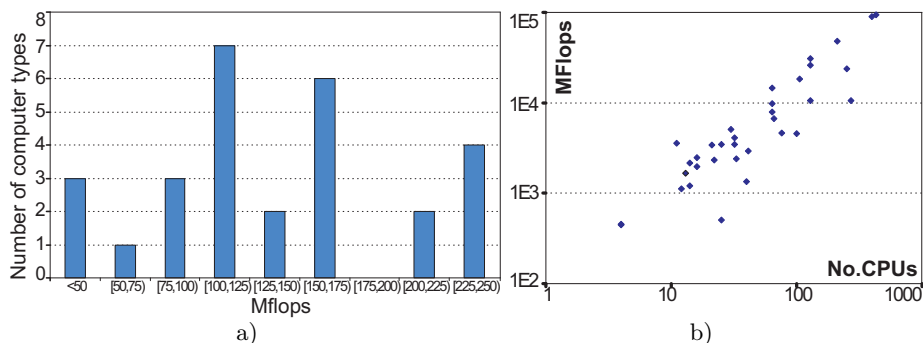


Fig. 1. a) Distribution of computer performance score. b) Datacenter performance vs. CPU number. On the basis of [4].

The test grid consisted of 2700 CPUs distributed in 40 locations (datacenters) in 13 countries on 5 continents [4]. The computing platform comprised 5 different operating systems, 10 job submission and deployment systems, 5 Java Virtual Machine types. The total performance of the grid was estimated at 450GFlops according to SciMark benchmark [4]. Fig.1a depicts distribution of the CPU speeds, and Fig.1b the spread of datacenter CPU numbers and the total performance. It should be noted that the computers were not continuously and exclusively available. Since the computing platform lacked sufficient reliability mechanisms, such mechanisms had to be implemented by the application.

Developing an application for such a diverse computing platform was possible thanks to Proactive middleware [15]. Proactive is a Java programming library providing active objects, asynchronous communication between the objects, their deployment and method execution. An active object has its own thread of execution. For example, it can be executed in a loop actively pooling conditions and reacting to them. Proactive provides uniform view of the application memory space. Therefore, methods of a remote active object can be called by other objects in the same way as in a sequential program. For deployment of the active objects Proactive used XML Deployment Descriptors comprising information on: the addresses of available computers, process initiation, communication and file transfer protocols. Thus, a programmer was separated from the actual grid hardware and referred to an active object in its very own code rather than to a description of a process on a remote computer.

4 Parallel Branch-and-Bound for Grid

In this section we give a general description of the PBB, our implementation of PBB for flowshop, and the above computing platform. An interested reader will find description of BB and PBB in, e.g., [2, 3, 7, 10].

4.1 PBB in General

The BB search for the optimum solution may be understood as a recursive analysis of the set of solutions in divide-and-conquer manner. The initial set of solutions is divided into subsets. A subset is fathomed by either eliminating it, as not containing optimum solution, or by further sub-dividing it. The recursion stops if a subset comprises only one solution. This process of creating and eliminating subsets can be viewed as constructing and searching of a tree. Thus, names sub-tree or search tree node can be used when referring to a subset of solutions. We will use abbreviation *BBTnode* for Branch and Bound tree node.

Search trees for hard combinatorial problems have exponential size in the length of the input (here the number of jobs n). Since it is not desirable, various methods are applied to limit the search tree. We will be calling a BBTnode *active* if it has neither been branched into offspring, nor eliminated. The active nodes can be eliminated on the following basis (e.g.): 1) all the solutions represented by the subtree are infeasible, 2) all the solutions represented by the subtree are dominated, 3) all the solutions represented by the subtree are not better than some already known solution. If it is known that the optimum solution has certain feature, then the analyzed BBTnode a is *dominated* and can be eliminated if it lacks such a feature. Suppose we solve minimization problem. In the third case a lower bound $LB(a)$ on the objective function is calculated for all the solutions represented by a . Suppose some solution b is known with objective value $C_{max}(b) \leq LB(a)$, then node a is pruned. The best known solution b establishes an upper bound UB . The more the search tree is cut, the smaller the set of visited nodes and the faster the algorithm is. Yet, sometimes shallower cuts but faster to calculate give faster BB than deep time consuming cuts.

Parallelization is a natural step to speed up BB. Yet, PBB has to deal with several problems. A common approach is to distribute the search tree between several computers. Since the search tree is instance-dependent, its structure is unknown before the runtime. Consequently, the tree cannot be partitioned statically because some computers would quickly run out of work, while the other would be overloaded. However, an optimistic scenario is also possible. When several computers search the solution space simultaneously, then a good upper bound UB may be found earlier than in the sequential run. As a result, some parts of the search tree which would have been visited in a sequential run, may be pruned in a parallel run. Such phenomena are known as performance anomalies in PBB [11, 12]. In distributed systems other difficulties arise. For example, termination of the computation may be erroneously declared when some BBTtree node is lost. This may happen due to errors in communication, or as a result of transition state when some tree nodes are in the communication network, while computers have nothing to process.

Thus, the following issues must be taken into account in PBB:

- 1) *load balancing* is necessary to avoid idling or overloading the processors,
- 2) upper bounds must be globally communicated to prune unneeded nodes,
- 3) deploying and quick initiation of computation to avoid idling of the computers,
- 4) reliable termination of computation.

4.2 Branching Scheme

In the following subsections we outline our PBB implementation. The branching scheme can be seen as construction of all possible job permutations. Let AS be the set of already sequenced jobs. The jobs in set $TS = \mathcal{J} - AS$ remain to be sequenced. Suppose we have two sequences π, σ partitioning AS , i.e., job sets in π, σ are disjunctive and their sum is equal to set AS . Initially $\pi = \sigma = ()$, $AS = \emptyset$, and $TS = \mathcal{J}$. Pair (π, σ) is a BBTnode representing all the schedules starting with sequence π and finishing with sequence σ . Let $|x|$ denote the number of jobs in sequence x . *Height* of a BBTnode is the height of the subtree it represents, i.e. height of (π, σ) is $n - |\pi| - |\sigma|$.

Branching BBTnode (π, σ) consists in inserting unassigned jobs $J_j \in TS$ between sequences π and σ . Jobs J_j may be attached either to the end of π or to the beginning of σ , but only one option can be used to avoid generating the same sequences many times. Both assignments are verified for each job in TS , but only one assignment type is used in all the successors of (π, σ) . The assignment which results in a greater number of the offspring nodes with their lower bounds smaller than the current upper bound UB is selected. If both choices are equivalent, then the assignment giving the smallest lower bound for any new node is chosen. Otherwise, all the jobs are arbitrarily attached at the beginning of σ .

The branching process is finished when a *complete* solution is achieved, i.e. when $TS = \emptyset$ or equivalently $|\pi| + |\sigma| = n$. Observe that the search method is exhaustive, i.e. all possible sequences may be enumerated (unless they are pruned), and no sequence is generated twice.

The search tree was explored in the Depth-First Least Lower Bound (DF/LLB) order, i.e. the newly branched BBTnodes (π', σ') were sorted according to non-decreasing values of lower bounds $LB(\pi', \sigma')$ and analyzed in this order before the older BBTnodes.

4.3 Bounding Techniques

Let us remind that BBTnodes with lower bound greater than or equal to the upper bound are not expanded. The upper bound UB is the schedule length of the best known solution of the problem. The initial UB value is the length of the schedule built by NEH heuristic [13]. NEH first sorts the jobs in the order of nonincreasing total processing times $\sum_{i=1}^m p_{ji}$. Then the first two jobs are scheduled for the minimum schedule length. Thus, a sequence of $l = 2$ jobs is constructed. For the given sequence of l jobs schedules with the $(l + 1)$ th job inserted between all the jobs in the sequence of length l , including the starting and the ending positions, are verified. The best schedule for $l + 1$ jobs is chosen and l is increased. This procedure is repeated until inserting all jobs, i.e. until $l = n$. The value of UB is updated and the new solution is recorded when a better solution is found in a leaf of the search tree.

Now we proceed to the methods of lower bound calculation. A BBTnode consists of two sequences (π, σ) . The unscheduled tasks from set TS shall be

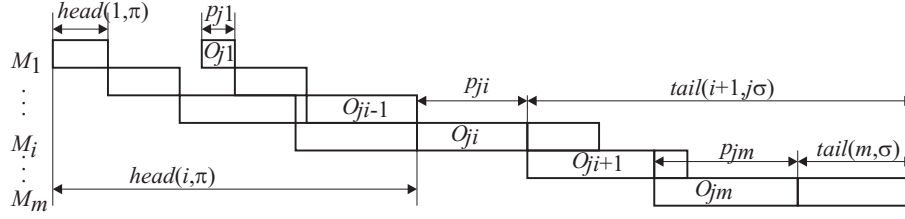


Fig. 2. Calculation of $head(i, \pi_j)$ and $tail(i, j\sigma)$.

either appended to π , or attached at the beginning of σ . Suppose job J_j is inserted at the beginning of sequence σ , and the offspring node is $(\pi, j\sigma)$. Consider operation O_{ji} which is immediately followed by the operations in σ on machines M_i, \dots, M_m (cf. Fig.2). Operation O_{ji} and its successors in $j\sigma$ will be executed in time at least $tail(i, j\sigma)$. Let $tail(m+1, j\sigma) = tail(m, \sigma)$. The value of $tail(i, j\sigma)$ is calculated using the following formula:

$$tail(i, j\sigma) = \max\{p_{ji} + tail(i+1, j\sigma), tail(i, \sigma)\} \text{ for } i = m, \dots, 1 \quad (1)$$

Note that $tail(i, j\sigma)$ is tabulated after $tail(i+1, j\sigma)$. Analogously, assume $J_j \in TS$ is to be appended to π , and (π_j, σ) is the offspring. Operation O_{ji} is preceded by the operations in π on machines M_1, \dots, M_i . Let $head(i, \pi_j)$ denote the minimum time it takes to execute these operations. Let $head(0, \pi_j) = head(1, \pi)$. The value of $head(i, \pi_j)$ can be tabulated using the following formula:

$$head(i, \pi_j) = \max\{p_{ji} + head(i-1, \pi_j), head(i, \pi)\} \text{ for } i = 1, \dots, m \quad (2)$$

Again $head(i, \pi_j)$ is calculated after $head(i-1, \pi_j)$. Let π', σ' be the offspring sequences. If J_j is appended to π , then $\pi' = \pi_j, \sigma' = \sigma$, otherwise J_j immediately precedes σ and $\pi' = \pi, \sigma' = j\sigma$. The first lower bound was calculated in $O(m)$ time from the formula

$$LB_1(\pi', \sigma') = \max_{i=1}^m \{head(i, \pi') + \sum_{j \in TS - \{J_j\}} p_{ij} + tail(i, \sigma')\} \quad (3)$$

Also a second lower bound was initially used. It was based on the above defined $head(k, \pi), tail(l, \sigma)$, and the length of the schedule for tasks in set TS executed on any pair of machines M_k, M_l only. Though this lower bound effectively reduced search tree, it had higher complexity $O(m^2)$. Consequently, despite optimizations speeding computation of the second lower bound, it did not reduce overall running time.

4.4 Parallelizing for the Grid

In this section we describe control mechanisms and load balancing. Active nodes are stored in queues at each of the computers. An active BBTnode is a unit of load balancing.

Control Architecture. The control structure is a three-tier tree comprising a master server, slave servers and clients. There is one master server in the tree root. To avoid a communication bottleneck on the master server monitoring the whole computation, an intermediate layer of slave servers was introduced. A slave server manages a set of its clients. Except for the computation initiation the servers perform communications and load balancing. Clients compute and shift the load. All communications are performed on the paths up and down the tree.

The number of clients is equal to the number of CPU cores. Master server deploys a slave server for each 8 clients, and at least one slave server in each geographic location. The number of 8 clients per slave server was established experimentally in the preliminary runs on the SMP machine. The slave servers deploy their clients.

Both the master server, and the slave servers verify if the subordinate machines are *alive*. This is done by ping-like function during the BBTnode harvesting procedure (see the next paragraph). Let us observe that the timeout for a ping was 15s which is quite long period. A slave server or a client can be eliminated from the computing pool also if it causes a communication error, e.g., when a controlling computer tries to send to its subordinate some BBTnodes. If a computer does not respond or causes a communication error, then it is declared *dead*, removed from the computer pool and no longer used. The BBTnodes assigned to such a computer are moved back by the supervising server to the queue of active nodes to be reassigned. Computations finish when there are no BBTnodes to branch (see the next paragraph for details).

Load Balancing. At the start of the computation the master server branches the BB tree to the depth of two levels, creating $n(n-1)$ active nodes of height $n-2$, and sends one node to each slave server. Each slave server expands the received BBTnode by two additional levels, creating $(n-2)(n-3)$ new BBTnodes of height $n-4$, and sends one node to each of its clients. The servers record information on the forwarded nodes and their destinations. A client expands the received node to a full depth of the BB tree. If a client achieves a complete solution b which is better than the current upper bound, i.e. $C_{max}(b) < UB$, then b, UB are sent to the slave server. If the received upper bound UB is better than the old one, then the slave server sends it up to the master server, and down to its other client computers. Analogously, the master server sends a new better UB to the other slave servers, which forward it to their clients. A client or a slave server ignore the received upper bounds if these bounds are worse than the bounds known to the client or the slave server. A BBTnode a which has $LB(a) \geq UB$ (because a was enqueued when UB was bigger) is discarded when pulled from the queue for branching or load balancing.

If a client exhausts all its BBTnodes, then it requests one node from its slave server. The slave server records that the node previously sent to the client is fathomed, and assigns to the client a new BBTnode from its queue. If no active nodes are available at the slave server, then the slave server requests a node from the master server. The master server acts similarly if it has a BBTnode

available. If it has no BBTnodes available then the initial BBTnodes of height $n - 2$ are distributed, and the computation is in the final stage. Then, the master server starts a harvesting procedure.

In the first phase of harvesting the master server requests half of the nodes from the slave servers. A slave server returns at most half of its BBTnodes but only if they have sufficient height. If the request is successful, then the received BBTnodes are sent by one to the waiting slave servers. If the request is not successful, then the second phase of harvesting is initiated. Namely, the master server sends to the slave servers requests for half of BBTnodes from the clients. If a client has BBTnodes of sufficient height then it sends them to its slave server. The slave server keeps half of the received nodes, and the other half is transferred to the master server, provided that they have sufficient height.

The height of the BBTnodes returned to the servers in the harvesting procedure is important for the PBB performance. On one hand, low BBTnodes allow for fine granularity load balancing because they represent smaller subtrees. On the other hand, low BBTnodes represent smaller computational demands. They can be fathomed faster, and clients return for new BBTnodes earlier than for high BBTnode. Hence, use of high BBT nodes reduces the number of communications. In the tests on an SMP machine the minimum heights of the returned nodes were set to 10 for slave servers, and 8 for clients. In the grid, the number of processors was bigger and the load transfers were too frequent overloading the computers with handling communications. Therefore, BBTnodes lower than 12 at the slave servers or 10 at the clients, were not returned in the grid runs.

If the master server request for BBTnodes at the second phase of harvesting fails, then it means that only low BBTnodes are held by the clients and slave servers (if any). In such a case a request to confirm the completion of the computations is issued by the master server to all live slave servers. The slave servers wait for a confirmation of the completion of the computation at the live clients. When a client exploits all its BBTnodes, then the confirmation is sent. If all its clients confirmed then also the slave server confirms completion to the master server. Finally, computations stop if all live slave servers confirm completion of the computation to the master server.

We observed that in the final stage of the computation the number of exchanged messages was intensively increasing. This was a result of exchanging control messages to achieve load balance with a small number of final BBTnodes. As they had low subtrees, clients quickly fathomed them and returned requests for new BBTnodes. Thus, the load balancing method needs better tuning for the final stage of computation.

5 Experiments

Running a PBB was not always as smooth as one could expect. Application deployment was time consuming and not always successful. To speed up the deployment process our PBB used parallel deployment of the clients by the slave servers. Not always have the machines declared available successfully deployed

Table 1. Example of the infrastructure deployed for instances Taillard 21,28,29.

Location	No. of Computers	CPUs per computer
Amsterdam	20	2 x 1GHz
Supelec	33	2 x 3GHz
Lifl	53	2 x 2.4GHz
Inria Sophia	16	2 x 2GHz
Inria Sophia	16	2 x 933MHz
Inria Nef	32	2 x 2GHz

Table 2. Runtimes for Taillard 20×20 instances.

Taillard instance	Runtime (grid) [s]	CPUs (grid)	Runtime (SMP) [s]
No. 21	435	370	1285
No. 22	219	361	499
No. 23	1746	352	16627
No. 24	234	361	502
No. 25	351	361	925
No. 26	515	361	1486
No. 27	607	352	2119
No. 28	148	370	102
No. 29	187	370	234
No. 30	139	361	108

the code. Consequently, our application quickly discarded slow and unreliable computers. For example, out of 2300 computers declared available, instances Taillard 21, 28, 29 were solved on grid shown in Table 1. Hence, our attempt in heterogeneous computing immediately reduced the heterogeneity of the platform for the benefit of speed and reliability.

It was possible to solve all the submitted 20×20 FS instances to optimality within the 1 hour limit imposed by the competition rules. The running times and numbers of CPUs are shown in Table 2. The last column of Table 2 give running times on SMP SunFire machine. The computing infrastructure comprised moderate number of processors from close locations. This allowed for using fewer but more reliable CPUs. The optimum solutions in Table 2, were obtained despite small number of CPUs, because the FS part in PBB was adequately implemented. It can be observed [4] that other teams of the Flowshop Challenge used a strategy oriented toward employing big number of CPUs. This strategy, however, turned out futile because using more CPUs did not result in solving the instances.

6 Conclusions

We presented PBB for permutation flowshop as a benchmark of maturity of grid computing and the supporting middleware in executing applications with unpredictable resource demands. Both the platform, and the middleware excelled well. Furthermore, qualitative conclusions can be drawn. Three elements constitute the parallel application described above: flowshop algorithm, PBB, and grid interaction algorithms. All the three elements had to be adequately addressed. It is not possible, for example, to ignore flowshop domain issues and rely solely on the sheer parallel processing power to solve the problem because the number of possible solutions is anyway too big. Yet, too complex domain-specific solutions turned out counterproductive. PBB must actively shift the work between the machines to account for irregular and unpredictable load distributions. The

grid part must monitor resource availability to avoid stalls in the computation. Though the techniques we presented are not new, only combined together were they able to produce a successful parallel application for hard problem on a difficult computing platform.

Acknowledgments. Research partially supported by Polish National Science Center grant N519 643340.

References

1. Bąk, S., Błażewicz, J., Pawlak, G., Płaza, M., Burke, E., Kendall, G.: A parallel branch-and-bound approach to the rectangular guillotine strip cutting problem. *INFORMS J.on Computing* 23, 15-25 (2011)
2. Clausen, J.: Branch and bound algorithms - principles and examples, Technical Report, Department of Computer Science, University of Copenhagen (1999)
3. Crainic, T., Le Cun, B., Roucairol, C.: Parallel Branch-and-Bound Algorithms. In: Talbi, E.-G. (ed.), *Parallel Combinatorial Optimization*, pp.1-28. John Wiley & Sons (2006)
4. ETSI: 2nd Grid Plugtests Report (2006), <http://www.etsi.org/website/document/plugtestshistory/2005/2ndgridplugtestsreport.pdf>
5. Garey, M., Johnson, D., Sethi, R.: The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research* 1, 117-129 (1976)
6. Hejazi, S., Saghafian, S.: Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research* 43, 2895-2929 (2005)
7. Horn, J.: Bibliography on parallel branch-and-bound algorithms (1992), <http://liinwww.ira.uka.de/bibliography/Parallel/par.branch.and.bound.html>
8. Johnson, S.M.: Optimal two-and-three-stage production schedules with set-up times included. *Naval Research Logistics Quarterly* 1, 61-68 (1954)
9. Iyer, S., Saxena, B.: Improved genetic algorithm for the permutation flowshop scheduling problem, *Computers & Operations Research* 31, 593-606 (2004)
10. Kohler, W., Steiglitz, K.: Enumerative and iterative computational approaches. In: Coffman Jr., E.G. (ed.): *Computer and Job-Shop Scheduling Theory*, pp.229-287. Wiley, New York (1976)
11. Lai, T.-H., Sahni, S.: Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM* 27, 594-602 (1984)
12. Lai, T.-H., Sprague, A.: Performance of Parallel Branch-and-Bound Algorithms. *IEEE Transactions on Computers* 34, 962-964 (1985)
13. Nawaz, M., Enscore, E., Ham, I.: A heuristic algorithm for the m-machine, n-job flowshop sequencing problem, *Omega* 11, 91-95 (1983)
14. Reeves, C., Yamada, T.: Genetic algorithms, path relinking, and flowshop sequencing problem, *Evolutionary Computation* 6, 45-60 (1998)
15. ProActive - Professional Open Source Middleware for Parallel, Distributed, Multi-core Programming, <http://proactive.inria.fr/>
16. Taillard, E.: Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64, 278-285 (1993)
17. Taillard, E.: Scheduling instances (2008), <http://mistic.heig-vd.ch/taillard/ Problemes.dir/ordonnancement.dir/ordonnancement.html>,