On the best search strategy in parallel branch-and-bound: Best-First Search versus Lazy Depth-First Search

Jens Clausen[a] and Michael Perregaard[b,★]

[a]*IMM, Department of Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark*

[b]*DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark*

E-mail: jc@imm.dtu.dk;perre@diku.dk

The Best-First Search strategy (BeFS) and the Depth-First Search strategy (DFS) are regarded as the prime strategies when solving combinatorial optimization problems by parallel Branch-and-Bound (B&B) – BeFS because of efficiency with respect to the number of nodes explored, and DFS for reasons of space efficiency.

We investigate the efficiency of both strategies experimentally, and two versions of each strategy are tested: In the first, a B&B iteration for a node consists of bounding followed by branching on the node if necessary. For the second, the order is reversed – first branching takes place, and then each child of the node is bounded and possibly fathomed. The first is called *lazy*, the second *eager*.

The strategies are tested on the Quadratic Assignment Problem and the Job Shop Scheduling Problem. We use parallel codes developed specifically for the solution of the problem in question, and hence containing different heuristic rules and tests to speed up computation. In both cases, we start with an initial solution close to but not equal to the optimal solution.

Surprisingly, the BeFS-based strategies turn out to be inferior to the DFS-based strategies, both in terms of running times and in terms of bound calculations performed. Furthermore, when tested in a sequential setting, DFS turns out to be still superior because pruning and evaluation tests are more effective in DFS due to the presence of better incumbents.

1. Introduction

One of the key issues of search-based algorithms in general and B&B algorithms in particular is the search strategy employed: In which order should the unexplored parts of the solution space be searched? Different search strategies have different properties regarding time efficiency and memory consumption, both when considered in a sequential and a parallel setting.

★ Supported by the EU HCM project SCOOP and the Danish NSF project EPOS.

In parallel B&B, one often regards the Best-First Search strategy (BeFS) and the Depth-First Search strategy (DFS) to be two of the prime candidates – BeFS due to expectations of efficiency and theoretical properties regarding anomalies, and DFS for reasons of space efficiency. However, BeFS requires that the bound for each node be calculated when the node is created, whereas DFS leaves freedom to postpone the bound calculation.

We introduce the concept of *laziness*, i.e. postponing bound calculations as long as possible, and describe traditional (here termed *eager*) and *lazy* versions of both BeFS and DFS. These are all implemented in an existing parallel branch-and-bound framework code running on a 16-processor MEIKO Computing Surface. The code has been tailored to solve two hard combinatorial optimization problems: Quadratic Assignment (QAP) and Job Shop Scheduling (JSS). The tailoring consists of adding bounding functions and different efficiency enhancing tests between branchings to the parallel framework.

Experiments with the different search strategies are then performed based on initial lower bounds close to but not equal to the optimal solution.These reveal that general statements on the efficiency of eager BeFS with respect to minimizing the number of bounds calculated are not valid. The parallelism and the enhancement of a given branch-and-bound algorithm with rules to fix variables between branchings result in superiority of DFS regarding time efficiency (in addition to its superior properties regarding space efficiency). Furthermore, even in the sequential case, eager BeFS is inferior to the other strategies.

To explain the poor results for eager BeFS, we conducted experiments for QAP for different sequential versions of the branch-and-bound algorithm. These versions gradually approach a pure branch-and-bound algorithm, in which the only operations are branching and bounding. We make no use of additonal information, either during branching or during bounding. Surprisingly, BeFS turns out to be superior with respect to efficiency only if pure branch-and-bound is applied, however at a cost of almost tripling the number of bound calculations made to obtain the optimal solution.

The paper is organized as follows. In section 2, we outline the strategies to be tested, and previous results on the behaviour of parallel branch-and-bound algorithms are reviewed in section 3. Section 4 contains descriptions of the two problems on which the experiments were performed. The experimental results constitute section 5, and section 6 is a general discussion of the pros and cons of each strategy in relation to the experiments performed.

2. Search strategies

Recently, two surveys on parallel search methods in combinatorial optimization have been published: one covering specifically branch-and-bound by Gendron and Crainic [10], and one also dealing with search methods in connection with artificial intelligence by Grama and Kumar [11]. Both surveys describe the BeFS and the DFS

search strategy. From these descriptions, it is apparent that branch-and-bound algorithms using different strategies differ slightly with respect to the contents of an iteration of the algorithm.

If BeFS is used, the live subproblems are stored together with their lower bound, and an iteration in a sequential branch-and-bound consists of choosing a live subproblem with least lower bound, performing branching on the subproblem to generate its children in the search tree, and calculating the lower bound for each child. Each child is then either fathomed by comparing the bound with the best solution value yet discovered (called the *incumbent*) or stored in the pool of live subproblems together with its lower bound. We call this processing scheme *eager* (in line with terminology from functional programming), since bound calculation is performed as soon as a subproblem is generated.

When DFS is used, a live subproblem is stored without a bound (or with the bound of the father node in the search tree). An iteration starts with one bound calculation, followed by branching if the subproblem is not fathomed. Finally, the children of the node are stored in a last-in-first-out data structure to facilitate easy access to the next subproblem to be processed, which by the selection strategy is one of those just generated. Hence, all children of a node are considered before any of its siblings. The processing scheme is called *lazy* since the heavy part of the work in processing a subproblem – the bound calculation – is postponed as long as possible (in the hope that it becomes unnecessary).

The virtue of BeFS with regard to the number of subproblems bounded stems from the fact that no subproblem with a lower bound greater than the optimum value is ever branched on – it will simply not be processed before the optimal solution has been discovered. All subproblems with a lower bound less than the optimum value (critical subproblems) will of course be branched on and, in addition, some subproblems with a lower bound equal to the optimum value (semicritical) may be processed. Note that if an optimal solution for the problem at hand is given as the initial solution (for instance, produced by a heuristic), then DFS also processes only critical subproblems. The better an initial solution is, the less one would expect the difference between BeFS and DFS to be (measured in number of nodes bounded).

In a sequential DFS branch-and-bound, it is not possible to find a solution during the search of the subtree of the first child of a node, which enables fathoming of the other children without bounding these. Since each child is stored either without a bound or with the bound of the father node, fathoming of the children requires fathoming of the father node, which obviously cannot be done based on a solution found in one of the child nodes. In a parallel setting, other parts of the search tree are searched concurrently. The situation is therefore different in that a solution may be found during the processing of the first child, which enables fathoming of the parent node and thereby all children. Hence, it is common in the parallel setting to store each node in a DFS branch-and-bound together with the bound of its parent node and check this bound when the subproblem is to be processed – i.e. to use the lazy strategy.

For parallel BeFS branch-and-bound, we observe a similar situation. The existence of several concurrently working processors implies that the following scenario can occur: A processor *p* works with a subproblem *P*, which had the smallest lower bound among the available live subproblems when *p* started processing it, but a subproblem *Q* with a smaller bound has been created by branching at another processor *q* during the processing of *P* by *p*. Hence, parallel BeFS does not correspond exactly to sequential BeFS. In contrast to the sequential case, nodes which are neither critical nor semicritical may be processed in a parallel algorithm. Therefore, it is (as for DFS) possible to take advantage of new good solutions found by other processors, if each subproblem is stored with the lower bound of its parent rather than its own lower bound, and each iteration starts with bounding rather than branching.

Based on the above considerations, we decided to test four different branch-and-bound algorithms, both in the sequential and the parallel case: all combinations of search strategy (BeFS, DFS) and processing strategy (eager, lazy). We decided to use two difficult, but well studied test problems, for which we already had parallel codes. Regarding initial solution, we decided to use good, but not optimal, solutions: If optimal solutions are used, BeFS and DFS show the same sequential behaviour, and if initial values far from the optimum are used, BeFS will run out of memory too early to facilitate any comparison. The measures to be used in the comparisons regarding efficiency are the number of nodes bounded and the running times measured. Since DFS is known to be space efficient compared to BeFS, we expect to be able to solve larger problems using DFS.

3. Theoretical results on search strategies in parallel branch-and-bound

As mentioned previously, parallelism in connection with branch-and-bound introduces the possibility of anomalies, i.e. that increasing the number of processors in the parallel system does not lead to a corresponding increase in speed of computation. If the running time increases with an increase in number of processors, a detrimental anomaly has been observed, whereas if the time decreases by a factor larger than the ratio between the number of processors in the new and the old parallel system, we have an acceleration anomaly. The reason for such behaviour is that the number of nodes developed in the search tree usually varies with the number of processors – the discovery of good or optimal solutions may take place both earlier and later than before. One usually wants to increase the probability of observing acceleration anomalies and decrease the probability of detrimental anomalies.

A number of researchers have addressed the issue of anomalies in parallel branch-and-bound [4,12–15]. In [10], the results are summarized. Essentially, anomalies, when going from 1 to *p* processors, can be avoided for synchronous parallel BeFS branch-and-bound. Here, the available *p* processors synchronously perform an eager branch-and-bound iteration on the *p* nodes of lowest bounds in the search trees, represented by a global data structure, and return the undiscarded children to the data

structure to make ready for the next synchronous iteration. The condition to be met to avoid anomalies is that the value of the bound function does not decrease from a node to each child node. This result is also true for more general BeFS rules based on so-called heuristic selection functions, provided that these satisfy the same condition and, in addition, are one-to-one, i.e. that different nodes have different values of the selection function. However, these results do not hold in general when going from p_1 to p_2 processors.

In [4], similar results are derived in connection with asynchronous parallel branch-and-bound. In addition, algorithms in which dominance tests are performed are dealt with in the paper.

From a practical point of view, the results mentioned above are unfortunately based on implementations of parallel branch-and-bound which differ from practice and/or assumptions which, in general, are not fulfilled. The synchronous parallel branch-and-bound algorithm will, in general, suffer from so-called synchronization loss, since in each synchronous step, all other processors have to wait for the one performing the most time-consuming branch-and-bound iteration. In practice, parallel BeFS is normally implemented asynchronously using a shared global data structure or a procesor, which coordinates the solution process. The technique is known as the master–slave paradigm: The slaves receive subproblems from the master, process these, and communicate newly generated subproblems and new best solutions back to the master. The master is responsible for the implementation of the search strategy. The asynchronous implementation eliminates the synchronization loss and does not, in general, introduce new problems compared to the synchronous implementation.

Regarding the selection function, a common situation is that a large number of nodes have a lower bound equal to the optimal solution value of the problem. In order to apply the theory, one hence has to construct a two- or even three-argument heuristic function based on the lower bound and additional properties of each node (such as path number and level in the search tree, cf. [15]). Nevertheless, the majority of parallel branch-and-bound implementations consider BeFS to be the optimal search strategy and deviate from it only because of memory limitations. Likewise, some of the available branch-and-bound libraries such as, for example, the BOB library [3], invest substantial effort in enabling the user to implement BeFS branch-and-bound regardless of the computational platform used in the implementation.

4. The test problems: Quadratic Assignment and Job-Shop Scheduling

In the following, we briefly describe our test problems. The descriptions follow [8] and [18].

4.1. QAP

Here, we consider the Koopmans–Beckman version of the Quadratic Assignment Problem, which can informally be stated with reference to the following practical

situation: A company is to decide the assignment of n facilities to an equal number of locations and wants to minimize the total transportation cost. For each pair of facilities (i, j), a flow of communication $f(i, j)$ is known, and for each pair of locations (l, k), the corresponding distance $d(l, k)$ is known. The transportation cost between facilities i and j, given that i is assigned to location l and j is assigned to location k, is $f(i, j) \cdot d(l, k)$, and the objective of the company is to find an assignment minimizing the sum of all transportation costs.

Each feasible solution corresponds to a permutation of the facilities, and letting S denote the group of permutations of n elements, the problem can hence be formally stated as

$$\min_{\pi \quad S} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{i,j} \quad d_{\pi(i),\pi(j)}.$$

The matrices of flows and distances are denoted by F and D, respectively. If we introduce binary variables x_{ik}, $i, k \quad \{1,\ldots, n\}$, and define that facility i is assigned to location k if and only if x_{ik} equals 1, the problem can be formulated as an integer programming problem in the following way:

$$QAP(F, D) := \text{minimise} \quad \sum_{i=1}^{n} \sum_{k=1}^{n} \sum_{j=1}^{n} \sum_{l=1}^{n} f_{ij}\, d_{kl} x_{ik}\, x_{jl}$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_{ik} = 1, \quad k \quad \{1, \quad , n\},$$

$$\sum_{k=1}^{n} x_{ik} = 1, \quad i \quad \{1, \quad , n\},$$

$$x_{ik} \quad \{0,1\}, \quad i, k \quad \{1, \quad , n\}.$$

The difficulty of the problem stems from the objective function, which is quadratic. Hence, methods based on linear programming relaxations are not immediately applicable.

4.2. JSS

In the Job-Shop Scheduling (JSS) problem, n jobs are to be processed on m distinct machines. Each job is composed of a set of operations of different length to be processed on the m machines in a predefined order. Each machine is only able to process one operation at a time and once an operation is started, it cannot be interrupted. The problem is to find an optimal schedule on each machine such that the overall processing time is as short as possible.

Formally, we consider a set $\mathcal{J}$ of jobs, a set $\mathcal{M}$ of machines and a set $\mathcal{O}$ of operations. For each operation $o \quad \mathcal{O}$, there is a job $j_o \quad \mathcal{J}$ to which it belongs and a

machine $m_o \in \mathcal{M}$ on which it is to be processed. All the operations belonging to one job must be processed in a predefined order. Thus, each operation o can be identified by the corresponding job j_o and its position within the job, denoted by k_o. Finally, each operation $o \in \mathcal{O}$ has a processing time p_o.

The earliest time an operation o can be started if the precedence constraints of a given schedule are met is called the release time of o and is denoted by r_o. A set of release times for all operations constitutes a potential schedule, in which each operation is started at its release time. The schedule is feasible if (a) all release times are non-negative, (b) if operation o_1 precedes operation o_2 in some job j, then o_2 does not start before o_1 has finished, (c) no machine processes more than one operation at a time, and (d) the schedule is non-preemptive (when started, an operation finishes without interruption). The problem can be stated as follows:

$$
\begin{array}{lll}
\text{minimise} & \left(\underset{o \in \mathcal{O}}{\text{maximise}}\,(r_o + p_o)\right) & \\
\text{subject to} & r_{o_1} + p_{o_1} \le r_{o_2}, & o_1, o_2 \in j \in \mathcal{J},\ k_{o_1} < k_{o_2}, \\
 & r_{o_1} + p_{o_1} \le r_{o_2} \vee r_{o_2} + p_{o_2} \le r_{o_1}, & m_{o_1} = m_{o_2},\ o_1, o_2 \in \mathcal{O}, \\
 & r_o \ge 0, & o \in \mathcal{O}.
\end{array}
$$

The difficulty here is not so much the objective function but the constraints: those regarding non-preemption turn out to be disjunctive, leading to a non-convex feasible region. Again, methods based directly on a linear programming relaxation are of little use because of the severe weakening of the disjunctive constraints imposed by the convexification.

5. Experimental results

Here, we briefly describe the test problems and the implemented branch-and-bound algorithms for QAP and JSS. For each of the algorithms, we describe the bound function, the branching strategy, and the various efficiency enhancing tests included between branchings. However, first we describe the common parallel framework for the parallel versions of the algorithms.

5.1. Parallel branch-and-bound framework

Both of the implemented parallel branch-and-bound algorithms are based on a framework for defining communication protocols and information exchange for an asynchronous distributed branch-and-bound algorithm implemented on a 16-processor Intel i860-based MEIKO Computing Surface. The system is a distributed system with message passing as the communication facility. Each processor has 16 MB local memory and 8 communication channels used to communicate with other processors. Each communication channel is handled by a T800 transputer. Point-to-point

communication between processors is implemented at system level, but no shared memory is available. The algorithm works with distributed pools of live problems, on which each processor works using the sequential branch-and-bound method with the relevant search strategy. The processors are organized in a ring. After 128 iterations, each processor exchanges information regarding pool size with the neighbouring processors in the ring. Then, subproblems are sent in order to equalize the pool size. Finally, a termination test for distributed computation is used in order to detect when the algorithm has finished.

5.2. QAP

The problems we used for the experiments for QAP are from the classical Nugent test set [17]. The results are obtained using an adapted version of the code (described in [9]), where we use the Gilmore–Lawler (G–L) bound function combined with iteratively binding variables based on the information provided by the reduced costs calculated during the actual bound calculation. Each node in the search tree corresponds to a partial solution, in which each location in a subset of the locations has been assigned to a specific facility, leaving a set of free locations to be assigned to a set of free facilities in order to produce a complete solution from the partial solution of the node.

The G–L bound function first calculates, for each combination of an unassigned facility i and a free location k, a lower bound on the increase in objective function value given that i is assigned to k. This is done by sorting the flow coefficients from i to unassigned facilities ascendingly and the distance coefficients from location k to free locations descendingly. The scalar product of these two vectors is a lower bound on the cost incurred from the flow between i and the unassigned facilities when i have been assigned to k. All these lower bounds are joined into a matrix C, for which a linear assignment problem is then solved to produce the final G–L bound value. The result is the G–L bound and reduced cost information $\bar{c}_{jl}$ for all combinations of an unassigned facility j and a free location l. The value of $\bar{c}_{jl}$ equals the increase in the solution of the linear assignment problem, which will be incurred if j is forced to be assigned to l rather than being assigned as indicated by the optimal solution. Hence, $\bar{c}_{jl}$ is a lower bound on the increase in the G–L bound for the child of the current node resulting from a j-to-l assignment. We take advantage of this information in the branching scheme.

We branch on locations, i.e. a location among the free locations is chosen and a child node corresponding to locating each of the free facilities on the chosen location is formed. The scheme was originally proposed in [16]. For each combination of free location j and free facility l, we use the reduced cost $\bar{c}_{jl}$ as a lower bound on the increase in the G–L bound. If this lower bound together with the incumbent is sufficient to rule out the possibility of finding the optimal solution in the subspace corresponding to a j-to-l assignment, this assignment is discarded. The location to

branch on is chosen to be the one with the smallest number of locations remaining feasible candidates after the test just described.

We assign a facility permanently to a location if we discover that all other facilities have been discarded through the testing – this is called a *forced assignment*. Similarly, we may discover that a facility has only one possible location – then a forced assignment also takes place. After a forced assignment has taken place, we recalculate the G–L bound for the new subproblem, again performing the test for forced assignments. Results in [9] show that the effect of forcing assignments is to "push downwards" the nodes of the search tree rather than to reduce their number substantially. Note that the efficiency of the forcing procedure is dependent on the incumbent – the better the incumbent is, the more forced assignments we will make. Hence, the search strategy in this implementation indirectly influences the branching performed.

The aim of our investigation is to shed light on the efficiency of different branching strategies with respect to the size of the search tree. Hence, we do not set the initial solution equal to the optimum value for the given problem, even if we know that for QAP a heuristic such as simulated annealing will produce this value with high probability. Instead, the initial solution is set to 2% above the optimum of the problem. The value of 2% is chosen because the initial values supplied by a heuristic for the JSS test problems deviate by approximately 2% from optimum. No symmetries with respect to solution structure are exploited in the algorithm.

The results (running times in seconds, number of nodes bounded in the search tree, and number of critical nodes bounded) for one processor are given in table 1. For the 16-processor version, we report the same figures in table 2. We also report the relative speedup for the parallel algorithm, i.e. the running time for a sequential version divided by the running time obtained for the parallel algorithm using 16 processors. The speedup results show that the algorithm scales well and that QAP is a good candidate for parallel solution – the results are stable, with no sign of detrimental anomalies.

Comparing the results from tables 1 and 2 regarding the number of nodes bounded, it is obvious that the major reason for the poor behaviour of parallel eager BeFS is not the parallelization itself. The sequential BeFS algorithms are by far inferior to the DFS-based algorithms. The lazy versions have a slight advantage over the eager versions and, surprisingly, this effect is more profound for BeFS than for DFS. To explain these results, we investigated under which circumstances the eager best-first search strategy satisfies that the number of nodes bounded is the smallest among the strategies tested. We tested our sequential code, first without the iterative binding of variables between branchings, but still using the reduced cost information in the branching scheme, and secondly by also disregarding the reduced cost information (i.e. a "pure" branch-and-bound, in which the only operations are branching and bounding). The results are shown in table 3. The pure version shows the expected superior behaviour of BeFS compared to DFS, however, at the cost of a large increase in the

Table 1

Running times in seconds, number of nodes bounded, and number of critical nodes (with G–L lower bound less than optimum) bounded when solving QAPs on one i860 processor.

	Lazy		Eager	
Problem	Best-first	Depth-first	Best-first	Depth-first
	Running times in seconds			
Nugent 10	0.51	0.39	0.47	0.36
Nugent 12	18.19	11.00	16.52	10.04
Nugent 14	137.56	117.05	156.74	108.84
Nugent 15	–	407.16	666.26	381.15
Nugent 16	–	3002.43	–	2767.62
	Number of nodes bounded			
Nugent 10	690	650	819	655
Nugent 12	15592	13948	21447	13936
Nugent 14	93466	107383	144634	107397
Nugent 15	–	318920	518799	319013
Nugent 16	–	206498	–	2064997
	Number of critical nodes			
Nugent 10	200	190	216	195
Nugent 12	4475	4347	4806	4379
Nugent 14	21838	21694	23138	22746
Nugent 15	–	76265	80929	77088
Nugent 16	–	500717	–	502289

number of nodes bounded. The major reason for the degradation of performance is that the reduced cost information is not used during branching. Essentially, a cost-free way of bounding *and discarding* nodes of the search tree is lost.

5.3. JSS

We now turn to the corresponding results for JSS. The test problems considered are taken from the collection electronically available through the OR library [2], and the results are obtained using the code described in [18] combined with the branching strategy recently developed by Caseau and Laburthe [7]. As for QAP, we briefly describe the bound function, the branching strategy and the efficiency enhancing tests employed.

Table 2

Running times in seconds and speedup, number of nodes bounded, and number of critical nodes (having G–L lower bound less than optimum) bounded when solving QAPs on sixteen i860 processors.

	Lazy				Eager			
Problem	Best-first		Depth-first		Best-first		Depth-first	
	Running times in seconds and speedup							
	Time	Sp-up	Time	Sp-up	Time	Sp-up	Time	Sp-up
Nugent 10	0.12	4.3	0.10	3.9	0.14	3.4	0.14	2.6
Nugent 12	1.45	12.5	1.06	10.4	1.31	12.6	1.10	9.1
Nugent 14	11.34	12.1	7.58	15.4	10.61	14.8	6.60	16.5
Nugent 15	43.25	–	28.26	14.4	43.34	15.3	24.74	15.4
Nugent 16	–	–	213.38	14.1	336.05	–	183.24	15.1
	Number of nodes bounded							
Nugent 10	763		680		833		815	
Nugent 12	15325		15375		22548		17680	
Nugent 14	93688		90452		144243		90685	
Nugent 15	319895		299747		518170		311204	
Nugent 16	–		2082710		3596716		2095039	
	Number of critical nodes							
Nugent 10	199		195		213		208	
Nugent 12	4474		4357		4737		4519	
Nugent 14	21838		21523		23080		21818	
Nugent 15	77186		75956		80869		76972	
Nugent 16	–		500628		519025		502184	

A complete schedule for a given JSS problem is modelled as a digraph, with a node for each operation and two additional nodes, a source corresponding to the start of the schedule and a drain corresponding to the completion of the schedule. There is an arc from o_1 to o_2 if o_1 is scheduled before o_2. Furthermore, there are arcs from the source to each starting operation of a job, and from each final operation of a job to the drain. Each node in the graph has an associated cost corresponding to the processing time of the operation – the cost of the source and the drain are 0. The length of the schedule is the cost of a most-expensive (longest) path from the source to the drain, where the cost is calculated as the sum of the costs of the nodes on the path. The initial information for a JSS problem gives rise to those arcs in a final schedule which

Table 3

Running times in seconds, number of nodes bounded, and number of critical nodes bounded when solving QAPs on one i860 processor, first with forced assignments and then with pure G–L branch-and-bound.

	Lazy		Eager	
Problem	Best-first	Depth-first	Best-first	Depth-first
Without forced assignments				
Running times in seconds				
Nugent 10	0.52	0.42	0.50	0.40
Nugent 12	16.12	11.59	16.60	11.03
Nugent 14	138.87	121.76	161.3	115.19
Nugent 15	–	419.94	651.18	400.17
Nugent 16	–	3115.36	–	2876.56
Number of nodes bounded				
Nugent 10	709	719	878	722
Nugent 12	15759	14616	21920	14604
Nugent 14	93561	111071	145538	111084
Nugent 15	–	327422	521022	327512
Nugent 16	–	2122311	–	2122316
Number of critical nodes				
Nugent 10	201	207	219	212
Nugent 12	4479	4446	4818	4480
Nugent 14	21841	21828	23144	22896
Nugent 15	–	76977	90980	77805
Nugent 16	–	502218	–	503786
Pure G–L branch-and-bound				
Running times in seconds				
Nugent 10	1.09	1.01	0.83	0.83
Nugent 12	35.62	25.86	24.01	23.36
Nugent 14	–	268.00	199.60	249.76
Nugent 15	–	956.36	898.82	893.20
Nugent 16	–	6977.70	–	6560.43
Number of nodes bounded				
Nugent 10	1540	1727	1606	1727
Nugent 12	38376	35993	34701	35996
Nugent 14	–	267390	194785	267392
Nugent 15	–	804632	758885	804635
Nugent 16	–	5280080	–	5280086
Number of critical nodes				
Nugent 10	209	209	209	209
Nugent 12	4401	4401	4401	4401
Nugent 14	–	20945	20945	20945
Nugent 15	–	72357	72357	72357
Nugent 16	–	470100	–	470100

correspond to precedence relations within jobs. The problem is now to add arcs to the initial graph which schedule all operations belonging to identical machines, such that the resulting schedule is as short as possible. A partial schedule hence corresponds to a digraph in which a directed edge has been introduced from o_1 to o_2 if they belong to the same machine and their relative sequence has been decided to be o_1 before o_2. The length of a partial schedule is of course a lower bound on the length of any completion, but this bound can be improved. If a partial schedule cannot be discarded by a combination of bound calculations and tests, we perform a branching on a pair of operations, for which the relative order has not been determined. We generate two subproblems corresponding to the two possible relative orderings of the operations.

The bound function for JSS is based on Jackson's Preemptive Scheduling rule for the scheduling of operations with given release times (heads) and post-processing times (tails) on one machine. The JPS schedule is the optimal schedule obtained when preemption is allowed, i.e. when the processing of an operation may be interrupted. At each point in time, the operation processed is the one with the longest tail. Note that rescheduling is only necesary when the current operation finishes or a new operation is released – hence the schedule can be computed very efficiently. Each of the machines in the JSS problem is considered independently. The partial decisions already taken give rise to trivial values for the head and tail of an operation o, namely the length of a longest path from the source node to o resp. from o to the drain node. For each machine, we now calculate the length of a JPS schedule, and the maximum length constitutes our lower bound. As noted in [18], the bound is very weak, but the immediate ideas for improvement (disallowing preemption in the Jackson schedule and scheduling two machines rather than just one) do not work.

The pair of operations to branch on is chosen based on the idea of creating a large difference in the lower bounds of the two new nodes. The scheme resembles that originally proposed by Carlier and Pinson [5], but is a little more sophisticated.

After having calculated the lower bound of a node in the search tree corresponding to a partial schedule, we perform a number of different tests and adjustments. These are a subset of the tests proposed by Carlier and Pinson [5,6]. The idea is that by using the heads and tails of two operations o_1 and o_2, which must be scheduled on the same machine, one may infer their relative schedule in any completion schedule which is shorter than the current best known upper bound UB. If e.g. $head_{o_1} + p_{o_1} + p_{o_2} + tail_{o_2} > UB$, then o_2 must be scheduled first in a completion of the current partial schedule with a value smaller than UB. Based on such tests, we can sometimes fathom a node (none of the children survive the test) and sometimes fix the order between two operations. Essentially, we perform an easy bound calculation. If the tests result in the addition of an edge in the digraph corresponding to our partial solution, we recalculate heads and tails, and test again. Testing and recalculating heads and tails is very expensive in running time compared to the inter-branching tests used in QAP.

For each of the problems, the initial value for the current best solution has been set to the best solution produced by the shifting bottleneck heuristic [1].

Table 4

Running times in seconds and number of nodes bounded when solving JSSs on one i860 processor.

	Lazy		Eager	
Problem	Best-first	Depth-first	Best-first	Depth-first
	Running times in seconds			
LA16	83.84	15.59	155.61	18.98
LA17	55.30	2.00	4.91	1.95
LA18	12.79	7.03	14.33	7.74
LA19	244.35	61.84	320.19	62.77
LA20	279.67	209.10	145.17	126.08
LA22	–	689.05	2379.09	89.88
	Number of nodes bounded			
LA16	3552	681	7622	808
LA17	4178	138	278	112
LA18	482	322	538	292
LA19	9222	2391	10976	2188
LA20	10909	8533	5392	4354
LA22	–	11889	40864	1824

Table 4 reports the running times and the corresponding number of nodes when solving a selected set of problems on a single i860 processor. In table 5, the running times and the corresponding number of nodes when solving the same problems distributed on 16 processors are given for the parallel version of the code. Contrary to the situation for QAP, large differences between the different strategies can be observed, especially between BeFS and DFS (LA17 and LA22 in the sequential version and LA22 in the parallel version). The lazy DFS strategy is here the clear winner with respect to running times, whereas the picture is more unclear with respect to number of nodes bounded. This seems to be due to the complicated tests between branching. The branch-and-bound method for JSS is much more sensitive to the incumbent value, which again can be seen in the speedup results. The speedups are quite unstable, ranging from 0.6 to 29.9 with 16 processors.

6. Discussion and conclusions

The first immediate fact noted from the results is that BeFS in general is inferior to DFS, both in the sequential case and in the parallel case, and both with respect to running times and numbers of nodes bounded. The eager BeFS, i.e. the combination

Table 5

Running times in seconds and number of nodes bounded when solving JSSs on sixteen i860 processors.

	Lazy				Eager			
Problem	Best-first		Depth-first		Best-first		Depth-first	
	Running times in seconds and speedup							
	Time	Sp-up	Time	Sp-up	Time	Sp-up	Time	Sp-up
LA16	8.57	9.8	3.99	3.9	8.16	19.1	7.14	2.7
LA17	2.24	24.7	1.97	1.0	3.58	1.4	3.41	0.6
LA18	2.05	6.2	1.95	3.6	5.97	2.4	5.62	1.4
LA19	15.01	16.3	7.97	7.8	20.65	15.5	10.27	6.1
LA20	15.70	17.8	9.53	21.9	12.00	12.1	10.31	12.2
LA22	144.23	–	23.05	29.9	101.32	23.6	28.23	3.2
	Number of nodes bounded							
LA16	4624		2382		4346		3660	
LA17	1354		923		844		586	
LA18	799		688		1370		1044	
LA19	8300		3487		9582		3208	
LA20	8683		4715		4202		1956	
LA22	38956		5810		25874		5554	

of strategy and processing usually recommended/described, is inferior to all other combinations in the QAP test, both in the sequential and the parallel versions. For the JSS test, eager and lazy BeFS perform comparably, but in general much worse than lazy DFS.

Regarding DFS, there seems to be no clear winner between the lazy and eager strategy, neither in the sequential nor in the parallel setting.

The results are surprising and counter-intuitive. When BeFS in the sequential case is known to process in some sense as few subproblems as possible, why does DFS perform better when we count the number of subproblems processed during the actual solution of problems? According to the results obtained for QAP, the answer lies in the efficiency enhancing tests performed between branchings. In the case of QAP, the assignment of a currently free facility to a free location is excluded if the reduced cost information shows that this will lead to a non-optimal solution, i.e. that any solution with the facility assigned to that particular location will have a cost exceeding the current best solution. Such a decision may give rise to other variable bindings, and the process continues until no more information can be derived. This has a considerable effect, both on the size and on the shape of the search tree compared

to the situation in which the information is not exploited. In the latter, all children of a given partial solution are generated and bounded, even if the reduced cost information shows that this is not necessary.

Similar tests are used in JSS, where the order of two operations in a machine may be fixed, if it is known that the head of one plus the processing times of both plus the tail of the other exceeds the best know solution – then the second of the operations must be scheduled first on the machine in question. We have not, in this case, conducted the experiment of running "pure" branch-and-bound. Our experience, as well as results from others, indicate that a pure branch-and-bound algorithm for JSS is too inefficient to provide any interesting result from a practical perspective – the best bounds currently known are simply too weak.

The superiority of DFS compared to BeFS seems intimately related to the tests between branchings. The tests crucially depend on the value of the incumbent – the better this is, the more effective the performance of the tests. Hence, a search strategy which facilitates early discovery of good solutions is preferable. DFS rapidly generates complete solutions, but may in the process bound non-critical nodes. BeFS bounds only critical nodes, but may spend quite some time processing partial solutions before a good complete solution is identified. The presence of a good incumbent is, however, critical for the efficiency of the tests, and hence these tests perform much better with the DFS strategy, resulting in superior performance. Finally, the interplay between the incumbent and the tests implies that a processing of each node should be postponed as long as possible to allow for the maximum decrease in the incumbent.

Another way of viewing the tests is that these represent ad hoc boundings performed by a bound function different to the one used in general. In this sense, the tests represent parts of a search based on different bounds, in which the search order is not BeFS. The resulting combined search may hence differ substantially from the BeFS aimed at with respect to the general bound function. Therefore, conclusions on the efficiency of search methods in "pure" branch-and-bound are not applicable.

In conclusion, the claimed efficiency of BeFS branch-and-bound has been shown invalid for QAP and JSS; here, DFS branch-and-bound is more efficient. Experiments should be carried out with a broader range of problems before general conclusions are drawn, and the effect of tests between branchings should be investigated in more detail, both theoretically and experimentally.

References

[1] J. Adams, E. Balas and D. Zawack, The shifting bottleneck procedure for job-shop scheduling, Management Science 34(1988)391–401.

[2] J.E. Beasley, Operations research library of problems, available via anonymous FTP from mscmga.ms.ic.ac.uk (155.198.66.4).

[3] M. Benaichouche, V.-D. Chung, S. Dowaji, B.L. Cun, T. Mautor and C. Roucairol, Building a parallel branch and bound library, in: *Solving Combinatorial Optimization Problems in Parallel*, eds. A. Ferreira and P. Pardalos, Lecture Notes in Computer Science 1054, Springer, 1996.

[4] A. de Bruin, G.A.P. Kindervater and H.W.J.M. Trienekens, Asynchronous parallel branch-and-bound and anomalies, in: *Parallel Algorithms for Irregularly Structured Problems*, eds. A. Ferreira and J. Rolim, Lecture Notes in Computer Science 980, Springer, 1995.
[5] J. Carlier and E. Pinson, An algorithm for solving the job-shop problem, Management Science 35(1989)164–176.
[6] J. Carlier and E. Pinson, A practical use of Jackson's preemptive schedule for solving the job-shop problem, Annals of Operations Research 26(1990)269–287.
[7] Y. Caseau and F. Laburthe, Disjunctive scheduling with task intervals, LIENS Technical Report 95-25, Paris.
[8] J. Clausen, Parallel branch-and-bound – principles and personal experiences, in: *Parallel Computing in Optimization*, eds. A. Migdalas, P.M. Pardalos and S. Storøy, Kluwer, 1997, chap. 7.
[9] J. Clausen and M. Perregaard, Solving large quadratic assignment problems in Parallel, Computational Optimization and Applications 8(1997)111–127.
[10] B. Gendron and T.G. Crainic, Parallel branch-and-bound algorithms: Survey and synthesis, Operations Research 42(1994)1042–1066.
[11] A. Grama and V. Kumar, Parallel search algorithms for discrete optimization problems, ORSA Journal of Computing 7(1995)365–385.
[12] T.H. Lai and S. Sahni, Anomalies in parallel branch-and-bound algorithms, Comm. ACM 27(1984) 594–602.
[13] T.H. Lai and A. Sprague, Performance of parallel branch-and-bound algorithms, IEEE Trans. Comput. C-34(1985)962–964.
[14] T.H. Lai and A. Sprague, A note on anomalies in parallel branch-and-bound algorithms with one-to-one bounding functions, Inf. Proc. Lett. 23(1986)119– 122.
[15] G.J. Li and B.W. Wah, Coping with anomalies in parallel branch-and-bound algorithms, IEEE Trans. Comput. C-35(1986)568–573.
[16] T. Mautor and C. Roucairol, A new exact algorithm for the solution of quadratic assignment problems, Discrete Applied Mathematics 55(1994)281–293.
[17] C. Nugent, T. Vollmann and J. Ruml, An experimental comparison of techniques for the assignment of facilities to locations, Operations Research 16(1968)150–173.
[18] M. Perregaard and J. Clausen, Solving large job shop scheduling problems in parallel, Annals of Operations Research 83(1998)137–160.