Asynchronous Parallel Branch and Bound and Anomalies

A. de Bruin, G.A.P. Kindervater, H.W.J.M. Trienekens

Abstract

The parallel execution of branch and bound algorithms can result in seemingly unreasonable speedups or slowdowns. Almost never the speedup is equal to the increase in computing power. For synchronous parallel branch and bound, these effects have been studied extensively. For asynchronous parallelizations, only little is known.

In this paper, we derive sufficient conditions to guarantee that an asynchronous parallel branch and bound algorithm (with elimination by lower bound tests and dominance) will be at least as fast as its sequential counterpart. The technique used for obtaining the results seems to be more generally applicable.

The essential observations are that, under certain conditions, the parallel algorithm will always work on at least one node, that is branched from by the sequential algorithm, and that the parallel algorithm, after elimination of all such nodes, is able to conclude that the optimal solution has been found.

Finally, some of the theoretical results are brought into connection with a few practical experiments.

Note: This paper will be presented at 'Irregular 1995', September 4–6, 1995, Lyon, France.

1 Introduction

In many cases, enumeration is the last (and only) resort to obtain a solution to hard problems that do not contain much internal structure upon which a solution method can capitalize. Branch and bound implements a type of enumeration, whereby it is tried to avoid to devote energy to regions of the solution space to be searched, that turn out to be irrelevant for obtaining the desired solution. Due to the nature of the problems in question, it is hard, if not impossible, to predict which direction the search should take in order to reduce as much as possible the work to be done.

The fact that the problems under investigation are hard, suggests to use parallelization. However, as parallel versions of enumerative methods like branch and bound show

Report EUR-CS-95-05
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

an irregular behavior, unexpected effects with respect to the efficiency of the parallelizations have been detected. One would expect (or could hope) that attacking a problem with p processors instead of one, results in a speedup close to p, but in the examination of parallelizations of branch and bound algorithms, so called anomalies have been observed: *detrimental* anomalies (speedup less than 1), and *acceleration* anomalies (speedup greater than p).

For synchronous parallel branch and bound, this type of anomalous behavior has been extensively theoretically studied [Burton, Huntbach, McKeown & Rayward-Smith, 1983; Lai & Sahni, 1984; Lai & Sprague, 1985, 1986; Li & Wah, 1984, 1986]. Typical results are properties that the rules applied by the branch and bound algorithm should have in order to ensure that no detrimental anomaly can occur, while leaving open the possibility of an acceleration anomaly.

For the asynchronous case, we only know of work by Corrêa & Ferreira [1995a, 1995b], who have obtained interesting results for a very broad class of branch and bound algorithms. The effect is, that their analysis leads to rather loose upper bounds on the execution time needed.

In this paper, we investigate asynchronous parallelizations, where the algorithm exploits a global active set (i.e., a pool of nodes to be investigated for finding an optimal solution). We do not only cover the case where elimination is implemented by lower bound tests, we also treat dominance. Our results are obtained by a new technique for analyzing branch and bound executions, of which we expect that it can be applied to other kinds of parallelizations as well.

In our analysis, parallel executions are compared with the corresponding sequential algorithm. To do so, we need the notion of *primary nodes*, i.e., nodes that are branched from by the sequential algorithm. We will postulate properties of the rules and tests applied by the branch and bound algorithm, from which we can derive results like the following:

– If during execution of the parallel algorithm no primary nodes are around any more, the algorithm must have obtained an optimal solution, and may terminate.

– At each point during execution of the parallel algorithm, at least one primary node is being handled.

– The elimination of primary nodes is possible during the parallel execution of the algorithm.

The first two results ensure that the parallel algorithm cannot run slower than the sequential algorithm executed on the slowest processor in the parallel system. In the last situation, acceleration anomalies may be observed, since such an anomaly can only occur if primary nodes are eliminated.

The rest of the paper is laid out as follows. In Section 2, we present the standard rules and ideas of (sequential) branch and bound, together with some definitions that we will need in the sequel. Moreover, we will define the class of parallelizations that we will consider. In Section 3, we will derive the results as sketched above. Section 4 relates the results to preventing or enabling anomalies. One of the observations is that it might be dangerous to extend a parallel system with a processor, that is appreciably less powerful than the other ones, because it increases the theoretical upper bound on the execution time. That this effect is not only of theoretical nature is illustrated in Section 5, where an experiment is described in which adding a slow processor results in a significant slowdown. In the last section, we present our conclusions.

2 The branch and bound algorithm

Branch and bound algorithms solve optimization problems by partitioning the solution space, i.e., the original problem is repeatedly decomposed into smaller subproblems until a solution has been obtained. Throughout the paper, we will assume that all optimization problems are posed as minimization problems, and that solving a problem is tantamount to finding a feasible solution with minimal value. If there are several such solutions, it does not matter which one is found.

Let P_0 be the minimization problem to be solved. The way P_0 is repeatedly decomposed into smaller subproblems can be represented by a finite rooted tree, called *problem tree*. We denote the problem tree by $T = (P, A)$, where P is the set of nodes, and A is the set of arcs. There is a one-to-one correspondence between the nodes of the problem tree and the subproblems generated. The *root* of the problem tree is P_0. If node P_j is generated by decomposition from node P_i, then $(P_i, P_j) \in A$. We say that P_i is the *parent* of P_j, and that P_j is a *child* of P_i. The part of the problem tree that is actually generated by the branch and bound algorithm is called the *search tree*.

Branch and bound algorithms can be characterized by four rules [Mitten, 1970; Ibaraki, 1976]: a *branching rule* stating how nodes can be decomposed, a *bounding rule* for the computation of a lower bound on the optimal solution value of a node, a *selection rule* defining which node to branch from next, and an *elimination rule* stating how to recognize and eliminate nodes that cannot yield an optimal solution to the original problem. We will call these four rules the *basic rules* of the branch and bound algorithm.

In the following, we will discuss the basic rules in detail, and identify important properties of the rules with respect to termination and correctness of branch and bound algorithms. We treat the rules in the same order as their definition.

Let $f(P_i)$ be the optimal solution value of node P_i, and let P_i be decomposed into $P_{i_1}, P_{i_2}, \ldots, P_{i_m}$ by the branching rule. Then, we must have:

$$f(P_i) = \min_{k=1,\ldots,m} f(P_{i_k}).$$

In other words, the optimal solution value of a node can be obtained by evaluating its children. In practice, branching rules are such that each feasible solution to a parent node is also a solution of at least one of its children.

Let $g(P_i)$ be a lower bound on the optimal solution value of node P_i computed by the bounding rule, and let L be the set of leaf nodes that can be solved without decomposition. We postulate the following properties of the lower bound function:

$$g(P_i) \leq f(P_i), \text{ for } P_i \in P,$$

$$g(P_i) = f(P_i), \text{ for } P_i \in L, \text{ and}$$

$$g(P_i) \leq g(P_j), \text{ for } (P_i, P_j) \in A.$$

The properties state respectively that g is a lower bound estimate of f, that g is exact when P_i can be solved without decomposition, and that lower bounds never decrease when traversing down the problem tree.

The concept of *heuristic search* provides a framework to compare all kinds of selection rules, for example, depth first, breadth first, or best bound [Ibaraki 1976]. In heuristic search, a *heuristic search function h* is defined on the set of nodes. It governs the order in which the nodes are branched from. The branch and bound algorithm always branches from the node with the smallest heuristic value.

A heuristic search function h is *injective* if and only if it assigns a unique value to each node, i.e.,

$$h(P_i) \neq h(P_j), \text{ if } P_i \neq P_j.$$

Injectivity is no severe restriction because each noninjective heuristic search function can be easily transformed into an injective one by extending a heuristic value with the unique path number of its argument as an additional component, and by defining a lexicographic order on these tuples. As all path numbers are unique, all heuristic values of the nodes will be unique. Notice that an injective heuristic search function induces a complete order on the nodes.

A heuristic search function h is *nonmisleading* if and only if the heuristic value of a node is not less than the heuristic value of its parent, i.e.,

$$h(P_i) \leq h(P_j), \text{ for } (P_i, P_j) \in A.$$

Note that the most common search strategies, like depth first and best bound search, are nonmisleading by nature.

The last rule to be considered is the elimination rule. It consists of three types of tests for eliminating nodes.

Firstly, the *feasibility test*: a node can be discarded from further examination if it can be proven not to have a feasible solution.

Secondly, the *lower bound test*: a node, of which the lower bound is no less than the value of a known feasible solution, cannot produce a better solution, and can be eliminated. The *incumbent* represents the best solution obtained so far, together with its value.

Finally, a node may be discarded by the *dominance test*. We say that a node P_i dominates a node P_j if it can be proven that $f(P_i) \leq f(P_j)$. If P_j has a feasible solution, P_i must produce as least as good a solution as P_j. However, if P_j is infeasible, P_i may be infeasible as well. In both cases, P_j does not have to be investigated any more. The dominance test is based upon a dominance relation D. If the dominance relation holds for a given pair of nodes P_i and P_j, denoted by $P_i D P_j$, then P_i dominates P_j. According to Ibaraki [1977], a dominance relation D should satisfy the next properties:

D is a partial order (a reflexive, antisymmetric, and transitive relation),

$P_i D P_j \Rightarrow P_j$ is not a proper descendant of P_i, and

$(P_i D P_j$ and $P_i \neq P_j) \Rightarrow P_i D P_{j^*}$, for all descendants P_{j^*} of P_j.

In this paper, we assume that all dominance relations possess the above properties. Because the dominance relation is a partial order, it is possible that for some P_i and P_j neither $P_i D P_j$ nor $P_j D P_i$ holds. If a dominance relation is *weak*, most of the nodes are incomparable. A node P_i is said to be a *currently dominating node* if it has been generated and has not been dominated so far. Due to the transitivity of the dominance relation, for each dominated node there exists a currently dominating node at any point in time of the execution of the branch and bound algorithm. So, for efficiently implementing the dominance test, the branch and bound algorithm only has to keep track of all currently dominating nodes. Note that the lower bound test can be viewed upon as a special case of the dominance test.

At this point, we will introduce some notions and definitions, which we need in the remainder of the paper.

A function f is said to be *consistent* with a function f^*, if

$f(P_i) \leq f(P_j) \Rightarrow f^*(P_i) \leq f^*(P_j)$, for all $P_i, P_j \in P$.

A dominance relation D is said to be *consistent* with a function f, if

$P_i D P_j \Rightarrow f(P_i) \leq f(P_j)$, for all $P_i, P_j \in P$.

An *active node* is a node that has been generated and hitherto neither has been completely branched from nor eliminated. The active nodes can be divided into two categories: those that are currently being branched from, and the others. In each stage of the computation, there exists an *active set*, i.e., the set containing all active nodes that are not being branched from at that moment.

A *critical node* is a node with lower bound less than the optimal solution value.

2.1 A sequential implementation

In sequential branch and bound algorithms, there is a main loop in which the basic rules are repeatedly applied in some predefined order. We consider the following implementation of the loop.

Using the selection rule, a node in the active set is chosen to branch from. The node is extracted from the active set, and decomposed according to the branching rule. For each of the children thus generated the bounding rule is applied to calculate a lower bound. If during the computation of the bound the child node is solved, i.e., the optimal solution to the node is found, the incumbent will be updated, depending on whether the newly found solution value is better than the one found so far. If a child node is not solved during computation of the bound, it is added to the active set. Finally, the elimination rule is used to prune the active set.

The computation continues until the active set is empty. If so, the incumbent represents an optimal solution to the original problem.

2.2 A parallel implementation

In this paper, we will focus on parallelizations where iteration steps of the same type as in the sequential algorithm are executed concurrently and asynchronously by several processes, called workers. There is one global data structure, the *extended active set*, which is the active set, augmented by the incumbent. If no confusion can arise, we will drop the adjective extended.

The data structure can be considered as an abstract data type, on which two operations are defined: `insert` and `select`. The `insert` operation takes as parameters a possibly empty set of nodes, and an optional solution. The effect of the operation will be equivalent to an insertion of the given nodes into the active set, followed by a replacement of the incumbent by the newly offered solution, if the latter one is better, and by an elimination of those nodes in the active set that can be removed by the elimination rule. The `select` operation will only succeed if the active set contains at least one node, in which case it will extract a node using the selection rule. If there is no node in the set, a special value `empty` is delivered.

The above description does not prescribe that the active set is actually implemented as indicated. It only formulates the effect of the operations. Furthermore, we do not indicate whether the active set must be implemented as a separate process, or in a distributed fashion, e.g., distributed over the worker processes.

The `insert` and `select` operations can be executed concurrently. We demand from the active set that the operations are implemented as atomic transactions. Executing the operations in parallel must have the same effect as executing an interleaving of the operations, i.e., a sequential execution in some unspecified order. Consequently, the active set maintains the following invariant, that is true before and after each operation (i.e., in the interleaving): the active set contains, in addition to the incumbent, only nodes that have been submitted to the `insert` operation, it contains no nodes that can be removed by the elimination rule, and if the set is nonempty, the `select` operation will succeed in extracting a node from it.

The worker processes execute a loop. Each iteration starts with a `select` operation on the active set, which will be repeated until the result is not `empty`, i.e., a node is extracted from the active set. Having obtained a node, the worker branches from it, and performs an `insert` operation on the results of the branching.

Finally, we will have to deal with termination. The parallel computation will be finished when all workers receive only `empties` on their `select` operations. In that case, the active set will contain an optimal solution to the original problem. Termination detection can be implemented by letting the active set keep track of the number of workers trying to extract a node from the active set. If this number equals the total number of workers, the active set can notify the workers that the computation has completed.

For the proofs to follow, we will have to specify the notions of *being branched from*, *active node*, and *active set*, in order to avoid ambiguity in the parallel case. A node is being branched from, if it has been extracted by the `select` operation, and the `insert` operation on the results of the branching has not completed yet. An active node is a node that is either being branched from, or that has been inserted in the active set, and has not yet been selected or eliminated. The active set contains the set of active nodes that are currently not being branched from.

As a final remark, note that a parallelization with only one worker is equivalent to the sequential algorithm described in the previous subsection. We will call this algorithm the *corresponding sequential branch and bound algorithm*. The *sequential solution* is defined as the optimal solution yielded by the corresponding sequential branch and bound algorithm.

3 Properties of primary nodes

In this section, we investigate the properties of primary nodes. We will prove the results as indicated in the introduction. The first theorem deals with the elimination of nodes.

Theorem 1. Let D be a dominance relation, consistent with the lower bound function. If during execution of the parallel algorithm, an optimal solution is known, and there are no active primary nodes around any more, all active nodes can be eliminated.

Proof. Consider an active node. As there are no primary nodes around any longer, it must be a nonprimary node. In the sequential case, the node itself or one of its ancestors may have been eliminated, for two reasons. In both situations, we will show that the active node can be eliminated by the parallel algorithm too.

First, if the active nonprimary node, or one of its ancestors, was killed by a lower bound test, it can now be eliminated by a lower bound test as well, as an optimal solution is known.

Second, if the active nonprimary node, or one of its ancestors, was eliminated by a dominance test, the sequentially dominating node, or one of its ancestors, must have been eliminated, because it cannot have not been generated (the active nonprimary node would then have been eliminated) and will not be generated in the future (its father must be a primary node, and there are no more primary nodes nor can they be generated any more). Again, there are two cases. If the sequentially dominating node, or one of its ancestors, has been eliminated by a lower bound test, the active nonprimary node can be eliminated by a lower bound test as well, due to the consistency of D with the lower bound function. Similarly, if the sequentially dominating node, or one of its ancestors, has been eliminated by a dominance test, the active nonprimary node can also be eliminated by a dominance bound test due to the transitivity of D and Ibaraki's third condition on dominance relations. □

The consequence of Theorem 1 is that the parallel algorithm, after having obtained an optimal solution, will terminate when all primary nodes have been branched from or eliminated. The next theorem states sufficient conditions to ensure that the parallel algorithm will actually have found an optimal solution when no primary nodes are around any more, and that during the process at least one primary node is being branched from.

Theorem 2. Let D be a dominance relation, consistent with the lower bound function, such that primary nodes are not dominated by any other node, and let h be an injective, nonmisleading heuristic search function. If during execution of the parallel algorithm, no active primary nodes are around any more, an optimal solution must have been found. Furthermore, if during execution of the parallel algorithm a nonprimary node is being selected, some worker process is currently branching from a primary node.

Proof. We start with the first part of the theorem. If there are no primary nodes around, new primary nodes cannot be generated any more. It follows that either the corresponding sequential solution has been generated, or that it, or one of its ancestors has been eliminated by a lower bound test. The last part is due to the assumptions on D that guarantee that the nodes on the path from to root to the sequential solution (they are all primary nodes) cannot be eliminated by a dominance test. Hence, either the sequential solution or an alternative one has been found.

The second part of the theorem is a bit more complicated. Suppose that there is currently no primary node being branched from, and let nonprimary node P_{np} be the node with lowest heuristic value in the active set. In the corresponding sequential algorithm P_{np} has not been branched from, and, therefore, P_{np}, or one of its ancestors, must have been eliminated. Let P_{np}^{seq} be the node that eliminates P_{np}, or one of its ancestors, either by a lower bound or a dominance test, and let P_f be the father of P_{np}^{seq}.

As P_{np}^{seq} has been generated in the sequential algorithm, P_f must be a primary node. Because P_{np}^{seq} eliminates P_{np}, P_f must have been branched from before the sequential algorithm would select P_{np}. Due to the fact that the heuristic search function h is injective and nonmisleading, the sequential algorithm branches from nodes in increasing order of their h-values, which implies that $h(P_f) < h(P_{np})$. We have to consider three situations with respect to the parallel algorithm.

First, P_f has been branched from, and P_{np}^{seq} has been generated. If P_{np}^{seq} has eliminated P_{np} by a lower bound test, the current incumbent, which must be at least as good as P_{np}^{seq}, can eliminate P_{np}. On the other hand, if P_{np}^{seq} has eliminated P_{np} by a dominance test, the node that currently dominates P_{np}^{seq} also dominates P_n.

Second, P_f, or one of its ancestors, has been eliminated. Let g be the lower bound function, and let v_{inc} denote the value of the incumbent. We now have that $v_{inc} \leq g(P_f) \leq g(P_{np}^{seq}) \leq g(P_{np})$, where the first inequality stems from the fact that primary nodes can only be eliminated by a lower bound test, and the last inequality is true irrespective whether P_{np}^{seq} eliminates P_{np} by a lower bound or a dominance test. Hence, P_{np} can be eliminated by a lower bound test.

Third, because of the assumption that currently no primary node is being branched from, the last possibility is that P_f, or one of its ancestors, is a member of the active set. If this is the case, $h(P_f)$ must be greater than $h(P_{np})$, because P_{np} has the lowest h-value in the active set. We have, however, $h(P_f) < h(P_{np})$, proving that P_f, or one of its ancestors, cannot be a member of the active set. □

The last theorem of this section is concerned with the elimination of primary nodes.

Theorem 3. There exist heuristic search functions and dominance relations such that primary nodes may be eliminated by the parallel algorithm.

Proof. Assume that no primary nodes are eliminated by a dominance test. We have to make a distinction between critical and noncritical nodes. Critical nodes cannot be eliminated by a lower bound test. Consequently, if the parallel algorithm is able to eliminate primary nodes, the sequential algorithm must branch from noncritical nodes. Almost every heuristic search function may allow for branching from noncritical nodes.

Some examples are depth first search, breadth first search, but even best bound search (if there are nodes with lower bound equal to the optimal solution value [Fox, Lenstra, Rinnnooy Kan & Schrage, 1978]). □

4 Exploiting properties of primary nodes

While designing parallel branch and bound algorithms, one wishes to prevent detrimental anomalies, while leaving open the possibility of acceleration anomalies. We will show how the theorems from the previous section can be used to reach this goal.

In our presentation, we will not consider the time needed for communication and the like. We will only take into account the effect of parallelism on the search tree generated. Detrimental and acceleration anomalies are defined by comparing the execution of the parallel algorithm with the execution of the corresponding sequential algorithm. The sequential algorithm is continuously branching from primary nodes. Hence, the anomalies may only happen if the search tree in the parallel case is different from the search tree generated by the sequential algorithm. In other words, an anomaly is induced by the (in)efficiency of either version of the algorithm. In the next subsection, however, we will show that certain inefficiencies are harmless, and do not cause detrimental anomalies to happen.

Unproportional acceleration or deceleration of a parallel branch and bound algorithm may also be observed when extending a parallel architecture with an additional processor. We do not deal with this situation from a theoretical point of view, but only mention an example from Lai & Sahni [1984], in which almost doubling the number of processors still leads to a slowdown of the parallel algorithm, where the search tree remains the same.

4.1 Preventing detrimental anomalies

Under the conditions of Theorems 1 & 2, the parallel algorithm will always work on a primary node, and it terminates as soon as there are no primary nodes around any more. Hence, there will be no detrimental anomaly.

Without dominance relations, the use of an injective and nonmisleading heuristic search function will be sufficient. The presence of dominance relations complexifies the situation. It is most of the time impossible to check whether or not a primary node can be eliminated by a dominance test in the parallel algorithm. However, we know of two examples from the literature which state conditions that prevent detrimental anomalies. We will show that they ensure that no primary nodes are eliminated by a dominance test, i.e., they can be seen as special cases of our theorems.

Li & Wah [1984]. In their paper, in which they develop a synchronous version of the global active set branch and bound algorithm, Li & Wah show that no detrimental anomaly occurs if (a) the heuristic search function is injective, nonmisleading, and consistent with the lower bound function, and (b) the dominance relation is consistent with the heuristic search function.

Theorem 4. Let the heuristic search function h be injective, nonmisleading, and consistent with the lower bound function g, and let the dominance relation D be consistent with h. Under these conditions, no primary node can be eliminated by a dominance test.

Proof. Let primary node P_1 be dominated by node P_2. We have that

$$P_2 D P_1 \Rightarrow h(P_2) \leq h(P_1) \Rightarrow g(P_2) \leq g(P_1), \text{ i.e.,}$$

D is consistent with g. Consider the sequential algorithm. Since $h(P_2) \leq h(P_1) \Rightarrow h(P_2) < h(P_1)$, the father of P_2, with an even a smaller h-value than P_2, cannot be a primary node. Otherwise, P_2 would have been generated and would have caused an elimination of primary node P_1. In the same way as in Theorem 2, we can now show that the elimination of the father of P_2, or one of its ancestors, leads to an elimination of P_1 as well, which is a contradiction since P_1 is assumed to be primary node. $\Box$

Trienekens [1990]. The conditions posed by Trienekens are more or less the same as those from Li & Wah. Trienekens shows that no detrimental anomaly occurs if (a) the heuristic search function is injective, nonmisleading, and strictly consistent with the lower bound function (i.e., $\leq$ is replaced by $<$), and (b) the dominance relation is consistent with the lower bound function.

Theorem 5. Let the heuristic search function h be injective, nonmisleading, and strictly consistent with the lower bound function g, and let the dominance relation D be consistent with g. Under these conditions, no primary node can be eliminated by a dominance test.

Proof. We will prove the theorem by showing that the dominance relation D is consistent with the heuristic search function h, in which case we can apply the previous theorem. Let node P_1 be dominated by a node P_2. It follows that $g(P_2) \leq g(P_1)$. However, if $h(P_1) < h(P_2)$, then $g(P_1) < g(P_2)$, which is in contradiction with the previous statement. Hence, $h(P_2) \leq h(P_1)$, and the dominance relation D is consistent with the heuristic search function h. Now, we can apply Theorem 4 for our proof. $\Box$

We will end our discussion on the prevention of detrimental anomalies by giving an upper bound on the execution time needed by the parallel algorithm, if Theorems 1 & 2 can be applied.

Let $T_{par}(I)$ denote the time needed to solve problem instance I with the given parallel branch and bound algorithm on the given machine, let $T_{seq}(I)$ denote the time needed to solve problem instance I with the corresponding sequential branch and bound algorithm on the least powerful processing element of the given parallel machine, and let T_{max} be an upper bound on the time needed to branch from a single node by the least powerful processing element of the given parallel machine.

Theorem 6. If the dominance relation is consistent with the lower bound function, such that primary nodes are not dominated by any other node, and if the heuristic search function is injective and nonmisleading, then $T_{par}(I) \leq T_{seq}(I) + T_{max}$.

Proof. Trivial. □

The bound on T_{par} is, or course, rather loose, because it assumes that all essential work is performed by the slowest processor element. It is, however, all that can be guaranteed.

4.2 Allowing acceleration anomalies

Acceleration anomalies may only occur if the search tree generated by the parallel algorithm is smaller than the one generated by the corresponding sequential algorithm. Consequently, they only happen if primary nodes are eliminated.

First, suppose primary nodes may be eliminated by a dominance test. If so, the Theorems 1 & 2 cannot be applied, and allowing for acceleration anomalies in this way may result in the opposite: a slowdown of the algorithm.

Second, suppose primary nodes may not be eliminated by a dominance test. For an acceleration anomaly to occur, it is necessary that primary nodes are eliminated by a lower bound test. As critical nodes have to be branched from at all times, the sequential algorithm must branch from other nodes as well. In Lai & Sahni [1984], examples are given that show that the heuristic search strategies indicated in Theorem 3 indeed may effect an acceleration anomaly. Unfortunately, we also have the next theorem.

Theorem 7. Let the heuristic search function h be injective, nonmisleading, and consistent with the lower bound function g, and let the dominance relation be such that it cannot cause elimination of primary nodes. If the lower bound cannot attain the optimal solution value, except for nodes representing an optimal solution, no primary nodes can be eliminated at all.

Proof. The sequential algorithm branches from nodes in increasing order of their h-value, and due to the consistency of the heuristic search function with the lower bound function, in nondecreasing order of their g-value. Hence, the sequential algorithm only branches from critical nodes. Because these critical nodes cannot be eliminated by a dominance test, the parallel algorithm must branch from them too. □

For counter examples (in the absence of dominance relations) if either one of the conditions is dropped, we refer to Lai & Sahni [1984] and Burton, Huntbach, McKeown & Rayward-Smith [1983].

5 Anomalies in the real world

Acceleration anomalies of parallel branch and bound algorithms have been observed in practice, whereas detrimental anomalies don't seem to happen very often. See, for instance, McKeown, Rayward-Smith & Rush [1992].

In a number of computational experiments, in which we solved the traveling salesman problem (TSP) on a loosely coupled network of workstations with the branch and bound algorithm that we are considering in this paper, we encountered an unexpected behavior of the algorithm. As the results are interesting, we will deal with them briefly. For a complete description of the experiments, we refer to Trienekens [1990].

worker processes on	fastest time (in minutes)	number of nodes	slowest time (in minutes)	number of nodes
2 pyramids	29.65	260	29.82	260
3 pyramids	20.35	260	20.40	260
4 pyramids	15.82	260	15.85	260
5 pyramids	13.93	260	14.05	260
2 pyramids + 1 sun	26.78	260	27.03	260
3 pyramids + 1 sun	18.90	260	19.03	260
4 pyramids + 1 sun	15.02	260	15.10	260
5 pyramids + 1 sun	13.30	260	13.75	260

(a) Results on a 75 city instance.

worker processes on	fastest time (in minutes)	number of nodes	slowest time (in minutes)	number of nodes
2 pyramids	0.82	12	0.85	12
3 pyramids	0.80	15	0.88	14
4 pyramids	0.72	20	0.90	17
5 pyramids	0.82	24	0.88	24
2 pyramids + 1 sun	0.92	13	1.70	25
3 pyramids + 1 sun	0.82	17	1.05	19
4 pyramids + 1 sun	0.97	21	1.18	29
5 pyramids + 1 sun	0.80	26	1.18	27

(b) Results on a 50 city instance.

Figure 1: Results from the parallel algorithm for the TSP.

In the experiments, we repeatedly enlarged the loosely coupled network by adding processing elements. All the processing elements, except one, were of equal processing power. The one exception had significantly less power. Because the asynchronism of the branch and bound algorithm introduces nondeterminism, we solved each instance of the traveling salesman problem several times.

Most of the time, adding a processing element of equal power decreased the time needed to solve the problem instance. Sometimes, the time didn't change significantly, but it never increased noticeably. The situation changed when we added the less powerful processing element. Next to a greater fluctuation in execution times, also deceleration of the algorithm could be observed. Figure 1 shows some typical outcomes. Looking at primary nodes and Theorem 6, we can explain what happened.

If at a given point in time during execution, the current number of processing elements is not large enough to handle all presently active primary nodes at the same time, the addition of a processing element has the effect that more primary nodes can be handled in parallel and, therefore, the time needed to handle all primary nodes will decrease.

However, if there are not enough primary nodes for the available processing elements, the processing elements start branching from nonprimary nodes. As long as the relatively fast processing elements work on primary nodes, and the slow ones on nonprimary nodes, everything is still under control. But if it is the other way around, it may happen that the computation is hold up, because the results from the slow processing elements are delayed. The numbers from Figure 1 show that both situations occur when adding a slow processing element.

We conclude that it can be dangerous to increase the processing power of a system by adding a less powerful processing element. Only if there is enough parallelism in the problem instance to be solved, this will decrease the execution time. Otherwise, the execution time may increase.

6 Conclusions

Our research on the anomalies that can occur during the execution of parallel branch and bound algorithms has yielded sufficient conditions to prevent detrimental anomalies. The conditions are stated as properties concerning the problem instance to be solved, and not in terms of the branch and bound algorithm used for solving the problem instance. Hence, the results are valid for all (parallel) branch and bound algorithms, as long as these algorithms comply to the conditions under which the properties are valid.

The conditions derived contain the ones already known for synchronous branch and bound algorithms as a special case.

The conditions for preventing detrimental anomalies are all worst case conditions. They do not state anything about the average performance of branch and bound algorithms.

Acceleration anomalies may occur, and indeed do occur, under certain conditions. The presence of 'nasty' dominance relations imply that allowing for acceleration anomalies may also imply the occurrence of detrimental anomalies.

In practice, a potential detrimental anomaly may be obscured by an acceleration anomaly, and the other way around. The extension of a parallel architecture with some relatively slow processor has to be done with some care.

References

[1] F.W. Burton, M.M. Huntbach, G.P. McKeown, V.J. Rayward-Smith (1983). *Parallelism in Branch-and-Bound Algorithms*, Report CSA/3/1983, University of East Anglia, Norwich.

[2] R. Corrêa, A. Ferreira (1995a). A distributed implementation of asynchronous parallel branch-and-bound. A. Ferreira & J. Rolim (eds.). *Solving Irregular Problems in Parallel: State of the Art*, Kluwer, Boston, to appear.

[3] R. Corrêa, A. Ferreira (1995b). Modeling parallel branch-and-bound for asynchronous implementations. *1994 DIMACS Workshop on Parallel Processing of Discrete Optimization problems*, DIMACS, Piscataway, to appear.

[4] B.L. Fox, J.K. Lenstra, A.H.G. Rinnnooy Kan, L.E. Schrage (1978). Branching from the largest upper bound: folklore and facts. *European J. Oper. Res. 2*, 191–194.

[5] T. Ibaraki (1976). Theoretical comparisons of search strategies in branch-and-bound algorithms. *Int. J. Comput. Inform. Sci. 5*, 315–344.

[6] T. Ibaraki (1977). The power of dominance relations in branch-and-bound algorithms. *J. Assoc. Comput. Mach. 24*, 264–279.

[7] T.-H. Lai, S. Sahni (1984). Anomalies in parallel branch-and-bound algorithms. *Comm. ACM 27*, 594–602.

[8] T.-H. Lai, A. Sprague (1985). Performance of parallel branch-and-bound algorithms. *IEEE Trans. Comput. C-34*, 962–964.

[9] T.-H. Lai, A. Sprague (1986). A note on anomalies in parallel branch-and-bound algorithms with one-to-one bounding functions. *Inform. Process. Lett. 23*, 119–122.

[10] G.-J. Li, B.W. Wah (1984). *Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms*, Report TR-EE 84-6, Purdue University, West Lafayette.

[11] G.-J. Li, B.W. Wah (1986). Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Trans. Comput. C-35*, 568–573.

[12] G.P. McKeown, V.J. Rayward-Smith, S.A. Rush (1992). Parallel branch-and-bound. L. Kronsjoe, D. Shumsheruddin (eds.) *Advances in Parallel Algorithms*, Advanced Topics in Computer Science 14, Blackwell, Oxford, 111–150.

[13] L.G. Mitten (1970). Branch-and-bound methods: general formulation and properties. *Oper. Res. 18*, 24–34.

[14] H.W.J.M. Trienekens (1990). *Parallel Branch and Bound Algorithms*, Ph.D. thesis, Erasmus University, Rotterdam.

This work was partially supported by the Human Capital and Mobility project SCOOP — Solving Combinatorial Optimization Problems in Parallel — of the European Union.