Parallel Branch and Bound and Anomalies

Harry W.J.M. Trienekens

Erasmus University Rotterdam

ABSTRACT

In this paper we present a classification of parallel branch and bound algorithms and investigate the anomalies which can occur during the execution of such algorithms. We develop sufficient conditions to prevent deceleration anomalies from degrading the performance. Such conditions were already known for some synchronous cases. It turns out that these conditions can be generalized to arbitrary cases. Finally we develop necessary conditions for acceleration anomalies to improve upon the performance.

1980 Mathematical Subject Classification: 90C27, 68Q10, 68R05.

Key Words & Phrases: parallel computer, branch and bound, nondeterminism, asynchronicity, anomalies.

Note: This paper will be submitted for publication elsewhere.

1. INTRODUCTION

There always is a need for more powerful computers. On one hand we want to solve problems faster than can be done now, on the other hand we want to solve problems which are too big for the computers we have today.
Under the traditional model of computation, in which a single processing element performs all tasks sequentially, it is no longer possible to realize significantly more powerful computers. This is due to the fact that in order to execute an instruction, data have to be transferred from one place in the computer to another. Such a transfer cannot be done at a speed greater than the speed of light. Some of today's computers are approaching this upper bound.
To be able to build still more powerful computers, a new model of computation had to be developed. In this new model several processing elements cooperate in executing a single task. The idea behind this model is that a task can be split into subtasks which are independent of each other, and therefore can be executed in parallel. This type of computer is called a parallel computer.

As long as it is possible to split a task into independent subtasks in such a way that the length of the biggest subtask continues to decrease, and as long as there are enough processing elements to execute all these subtasks in parallel, the computing power of a parallel computer is unlimited.
However, in real life these two conditions cannot be fulfilled. Firstly, it is impossible to divide each task in an arbitrary number of independent subtasks. There will always be a number of dependent subtasks which must be executed sequentially. Hence the time needed for executing a task in parallel has a lower bound. Secondly, each parallel computer which has actually been built has a fixed number of processing elements. As soon as the number of tasks exceeds the number of processing elements, some of the subtasks have to be executed sequentially and the parallel computer can realize at the most a speedup by a constant factor.

Report EUR-CS-89-01
Erasmus University Rotterdam, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

Branch and bound algorithms solve discrete optimization problems by partitioning the solution space. The problem to be solved is decomposed into smaller subproblems by splitting the solution space into smaller subspaces. Each subproblem thus generated is either solved or proved not to yield an optimal solution to the original problem and eliminated. If, for a given subproblem, neither of these two possibilities can be realized immediately, the subproblem is decomposed into smaller subproblems again. This process continues until all subproblems generated are either solved or eliminated.

For most algorithms the work involved in executing the algorithm is fairly well known in advance. The exact work to be done depends only in a minor way on the particular problem instance on hand. There are even algorithms for which the work to be done is completely independent of the problem instance on hand, for example, adding or multiplying two fixed length integers.
Unfortunately, for branch and bound algorithms the work to be done is heavily influenced by the particular problem instance on hand. Without carrying out the actual execution, it is often impossible to obtain a good estimate of the work involved.
Next to that, the way the work is organized also influences the work to be done. Each successive step to be performed during the execution of a branch and bound algorithm depends on the knowledge obtained thus far. The use of another search strategy, or the branching from several subproblems in parallel, can result in different knowledge obtained, and thus in a different order in which the subproblems are branched from. Without making additional assumptions, nothing can be said about how these changes will influence the actual execution. The total amount of work done can even increase or decrease by an arbitrary factor [Ibaraki 1976, Lai & Sahni 1984].
Note that in the sequential model of computation an increase in computing power only influences the speed of the execution of a branch and bound algorithm. The work involved remains exactly the same, it will only be done faster. However, if the computing power of a parallel computer is increased by adding additional processing elements, the execution of the branch and bound algorithm (i.e., the order in which the subproblems are branched from) changes implicitly. Therefore, solving discrete optimization problems as fast as possible with a parallel computer is not just a case of increasing the computing power of the parallel computer, but also of developing good parallel algorithms.

In this paper we develop a classification of parallel branch and bound algorithms and investigate the anomalies that can occur during the execution of parallel branch and bound algorithms. In section 2 we present this classification and present an example of a parallel branch and bound algorithm. Section 3 is devoted to the investigation of anomalies that can occur during execution. We define the anomalies that can occur and investigate what causes them. We develop sufficient conditions to prevent deceleration anomalies from degrading the performance. Such conditions were known already for some synchronous parallel branch and bound algorithms [Li & Wah 1984]. It turns out that these conditions can be generalized to arbitrary parallel branch and bound algorithms. Finally we develop necessary conditions for acceleration anomalies to improve upon the performance.

2. PARALLEL BRANCH AND BOUND

2.1. The Branch and Bound Algorithm

Branch and bound algorithms solve discrete optimization problems by partitioning the solution space. Throughout this paper, we will assume that all optimization problems are posed as minimization problems, and that solving a problem means finding a feasible solution with minimal value. If there are many such solutions, it does not matter which one is found.
A branch and bound algorithm can be characterized by four rules [Mitten 1970]: a *branching rule* defining how to split a problem into subproblems, a *bounding rule* defining how to compute a lower bound on the optimal solution of a subproblem, a *selection rule* defining which subproblem to branch from next, and an *elimination rule* stating how to recognize and eliminate subproblems which cannot yield an optimal solution to the original problem.

Let P_0 be the minimization problem to be solved. The way P_0 is repeatedly decomposed by the branching rule into smaller subproblems can be represented as a finite rooted tree $B = (P, A)$, where P is the set of nodes, and A is the set of arcs. This tree is called a *search tree*. There is a one-to-one

correspondence between the nodes of the search tree and the subproblems generated by decomposition. The *root* of the search tree is P_0. If subproblem P_j is generated by decomposition from subproblem P_i, then $(P_i, P_j) \in A$.
The *level* of a node in the search tree is equal to the number of arcs between this node and the root. The root is at level 0.
Let $f(P)$ be the optimal solution of subproblem P, and let subproblem P be decomposed into $P_1, \ldots, P_k$. The function f satisfies:

(2.1) $$f(P) = \min_{j=1,\ldots,k} \{ f(P_j) \}.$$

Let $g(P)$ be a lower bound to subproblem P computed by the bounding rule, and let T be the set of subproblems which can be solved without decomposition, i.e., the leaves of the search tree. We postulate the following properties of the lower bound function g:

(2.2) $g(P_i) \leq f(P_i)$ for $P_i \in P$

(2.3) $g(P_i) = f(P_i)$ for $P_i \in T$

(2.4) $g(P_i) \leq g(P_j)$ for $(P_i, P_j) \in A$

These 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 increase.
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 a heuristic search a *heuristic function h* is defined on the set of subproblems. This function governs the order in which the subproblems are branched from. The algorithm always branches from the subproblem with the smallest heuristic value.
The elimination rule can consist of three tests for eliminating subproblems.
Firstly, the *feasibility test*: a subproblem can be eliminated if it can be proven not to have a feasible solution.
Secondly, the *lower bound test*: a subproblem can be eliminated if its lower bound is greater than or equal to the value of a known feasible solution. If L denotes the lower bound test, then we use the notation $P_i L P_j$, which means that subproblem P_i can eliminate subproblem P_j by a lower bound test, i.e., P_i is a feasible solution (i.e., a subproblem whose optimal solution value is known) and $f(P_i) \leq g(P_j)$.
Finally, the *dominance test*: a subproblem which is dominated by another subproblem can be eliminated. If a subproblem P_i dominates a subproblem P_j this implies that $f(P_i) \leq f(P_j)$, or in other words, for each feasible solution of the dominated subproblem there is at least one feasible solution of the other subproblem with a smaller or equal solution value. The dominance test is based upon a dominance relation D. The fact that the dominance relation holds for a given pair of subproblems P_i and P_j, denoted by $P_i D P_j$, implies that P_i dominates P_j. A dominance relation D satisfies the following properties [Ibaraki 1977]:

(2.5) $P_i D P_j$ implies that P_j is not a proper descendant of P_i.

(2.6) D is a partial ordering (i.e., reflexive, antisymmetric, and transitive).

(2.7) $P_i D P_j$ and $P_i \neq P_j$ imply $P_i D P_{j'}$ for any descendant $P_{j'}$ of P_j.

Note that 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 subproblems are incomparable. A subproblem P_i is said to be a *currently dominating subproblem* if it has been generated and has not been dominated so far.

A possible sequential implementation of a branch and bound algorithm can be described as follows. An *active subproblem* is a subproblem which is generated and hitherto neither branched from nor eliminated. Note that a subproblem which is currently being branched from is still an active subproblem. In each stage of the computation there exists an *active set*, i.e., the set containing all active subproblems which are not being branched from at that moment. There is a main loop in which the following steps are repeatedly executed. Using the selection rule, one of the subproblems in the active set is chosen to branch from. This subproblem is extracted from the active set and decomposed into smaller subproblems using the branching rule. For each of the subproblems thus generated the bounding rule is used to calculate a lower bound. If during the computation of this bound the subproblem is solved, i.e., the optimal solution to the

subproblem has been found, the value of the best known solution is updated. This value is an upper bound on the value of the optimal solution to the original problem. If the subproblem is not solved during computation of the bound, the subproblem is added to the active set. Finally the elimination rule is used to prune the active set. The computations continue until there are no more active subproblems.
To use dominance tests, the complete set of currently dominating subproblems has to be stored. In general, the set of active nodes is not sufficient to determine this set, because a subproblem that has been decomposed might still be a currently dominating subproblem, i.e., $P_i DP_j$ does not necessarily imply that there exists a proper descendant $P_{i'}$ of P_i such that $P_{i'} DP_j$.

During execution of a branch and bound algorithm *knowledge* is continually generated and collected about the problem instance to be solved. This knowledge consists of all subproblems generated, branched from, currently dominating and eliminated, lower bounds on the value of the optimal solution, and the feasible solutions and upper bounds found. The decisions on what to do next, for example, the choice of the next subproblem to branch from or the elimination of a subproblem, are based upon this knowledge.
Depending on the exact characteristics of a branch and bound algorithm, part of the knowledge generated can be redundant, and therefore need not be preserved. For example, if a branch and bound algorithm uses an elimination rule which does not involve dominance tests, there is no need to keep track of the subproblems branched from or eliminated. Once such a subproblem is branched from or eliminated, there is no way in which the algorithm can use this subproblem again. Hence such an algorithm only has to keep track of the active subproblems and the best solution found hitherto. However, if the elimination rule involves dominance tests, a currently dominating subproblem, although already branched from or eliminated by a lower bound test, can still dominate other subproblems. Hence such a branch and bound algorithm has to keep track of all currently dominating subproblems, regardless of whether these subproblems are still active or not.

2.2. Parallel Branch and Bound Algorithms

Algorithms can be parallelized at a *low* or at a *high* level. In case of *low level parallelization*, the sequential algorithm is taken as a starting point, and only part of this algorithm is parallelized in such a way that the interactions between the parallelized part and the other parts of the algorithm do not change. For example, in case of branch and bound algorithms, the computation of the lower bound, the selection of the subproblem to branch from next, or the application of the elimination rule, could be performed by several processes in parallel.
Because the interaction between the various parts of the algorithm is not changed, low level parallelism does not have consequences for the algorithm as a whole. The overall behavior of the thus created parallel algorithm resembles the behavior of the original sequential algorithm (i.e., in case of branch and bound algorithms, it will branch from the same subproblems in the same order). This implies that there is no need to study the parallel algorithm as a whole. The only thing which must be studied is the part of the algorithm which was actually parallelized. Once the effects of this parallelization are known, the behavior of the parallel algorithm can be completely predicted (in terms of the parallelization effects on the behavior of the sequential algorithm).

In case of *high level parallelization*, the effects and consequences of the parallelism introduced are not restricted to a particular part of the algorithm, but influence the algorithm as a whole. The parallel algorithm is essentially different. The work performed by the parallel algorithm need not be equal to the work performed by the sequential algorithm. The order in which the work is performed can differ, and it is even possible that some parts of the work performed by the parallel algorithm are not performed by the sequential algorithm or vice versa.
For example, in branch and bound algorithms several iterations of the main loop can be performed in parallel. Because the iterations of this loop are fairly independent of each other, rearranging the order in which these iterations are performed, or performing several iterations in parallel, does not affect the correctness of the algorithm. For all the active subproblems it still holds that they have neither been branched from nor eliminated.

In this paper we study the particular idea of parallelizing branch and bound algorithms on a high level, viz., to perform several iterations of the main loop in parallel. In doing so, there are some decisions

to be made which we will now discuss.
To arrive at precise insights in the consequences of these decisions is the prime goal of our research.

2.2.1. Parameters of parallel branch and bound algorithms

It is impossible to get an accurate estimate of the work involved in executing a branch and bound algorithm without carrying out the actual computations. Therefore, the only way to achieve an equitable division of the work among the various processes is to divide the work as it is generated, i.e., dynamically during execution. To be able to divide the work, a basic unit of work has to be chosen. The units of work still to be completed are stored upon generation in pools of outstanding work. Each time a process becomes idle, it accesses one of the pools, and extracts a unit of work.
During execution of a parallel branch and bound algorithm the processes interact with each other and with the pools via the knowledge generated and exchanged. New knowledge can be used in two different ways. Firstly, it can affect the work still to be started by pruning units of work in the pools. Secondly, it can influence the work currently under progress by preempting processes working on units of work that are of low priority or redundant.
Finally it also has to be specified what happens when a process completes the unit of the work it is currently working upon. There are two extremes: before starting with its next unit of work the process can wait for all other processes to complete their unit of the work, or, the process can start working upon a new unit of work immediately without waiting for fellow processes to complete their unit.

Summarizing, parallel branch and bound algorithms can be classified according to the way in which the available work is divided among the various processes, the interaction between the processes and the pools, and the synchronicity of the processes.
In the next sections we will elaborate on these parameters.

2.2.1.1. Dividing the work

In order to divide the work among the processes some unit of work as well as some mechanism for dividing these units among the processes have to be defined.
The *unit of work* can be arbitrarily chosen. In practice however, the unit of work interacts with the communication complexity of the resulting parallel branch and bound algorithm. Each time a process completes its current unit of work, it has to communicate to get its next unit of work. If the unit is too small, the communication network can become saturated. Examples of units of work are the branching from a single subproblem [Trienekens 1989, Kindervater & Trienekens 1988], the optimal solving of a subproblem [Kindervater 1989], or the computing of lower bounds to a subproblem generated by decomposition [Kindervater 1989].
Dividing the work among the processes is carried out by using *pools of outstanding work*. New work created by the processes is added to the pools, and each time a process becomes idle, it obtains new work out of one of the pools. The two extremes are a single central pool accessed by all processes, and a private decentral pool for each process. In case a pool dries up, there is the option to refill it from one of its fellow pools.
Just as a branch and bound algorithm has a selection rule to determine the next subproblem to branch from, an idle process must have a selection rule to determine the unit of work to extract from a pool. Again the framework of heuristic search can be used by defining a heuristic function on the units of work: the unit of work to be extracted is the unit of work with minimal heuristic value.
For example, suppose that we choose the basic unit of work to correspond to branching from a single subproblem including the computation of the lower bounds on the subproblems thus generated. Suppose furthermore that we use a single central pool containing the active set, i.e., the work to be divided among the processes, and that the heuristic function defined on the units of work is equal to the heuristic function defined on the subproblems. Each time a process becomes idle, it extracts the subproblem with minimal heuristic value from the central pool. All subproblems generated are added to the pool. Each time a process adds a subproblem to the pool, the process prunes the pool by applying the elimination rule.

The advantage of a single central pool is that it provides a good overall picture of the work still to be done. This makes it easy to provide each process with a good unit of work (e.g., good subproblems to branch from), and to prune the active set. However, the disadvantage is that accessing the pool tends to be

a bottleneck because the pool can only be accessed by one process at a time.
The advantage and disadvantage of decentralized pools are just the opposite. The bottleneck of all processes accessing the same pool is avoided, but some of the processes might be working on bad units of work (e.g., branching from subproblems with low priority) simply because there happened to be no good unit in their pool. Apart from that, it is hard to eliminate subproblems by dominance tests because the subproblems are scattered all around.

The choice of the number of pools to use depends on the frequency with which the processes access these pools and the number of processes accessing the same pool. The access frequency in turn can be influenced by the unit of work chosen as the basic one.

2.2.1.2. Interaction

There are two aspects with respect to interaction, namely the exchange of the knowledge generated and the actions to be taken upon arrival of new knowledge. New knowledge can be used in two different ways. Firstly, it can be used for pruning the pools of work. Secondly, it can indicate that the unit of work currently being worked upon by a particular process has a lower priority than a unit of work just generated, or even that the former unit is redundant and superfluous with respect to the solution process of the problem instance on hand. In case of the unit of work being of lower priority, the process could suspend the part of the work to be completed until a later point in time (i.e., preempt this work and resume it at a later point in time), and start working upon a unit of work with higher priority. In case of the work being redundant, the part of the unit of work to be completed could be cancelled, i.e., the unit of work could be eliminated.
Information exchange can only be realized by means of communication. The best possible information exchange is obtained if each process broadcasts all knowledge it generates immediately once generated. This broadcasting, however, increases the communication complexity of the parallel algorithm, and thus reduces the possible advantage of fewer units of work to be performed (e.g., subproblems to be branched from). It is clear that there exists a tradeoff between the number of units of work eliminated, and the amount of communication carried out.
It also has to be decided what actions a process takes upon arrival of new knowledge. Again there are two extremes: The process can take the new knowledge into account immediately by preempting the work it is currently performing, or, the process can neglect the new knowledge until it has to make its next decision. Taking the new knowledge into account immediately by preempting can have the effect that the unit of work currently being worked upon is suspended until a later point in time or even eliminated. If the new knowledge does not indicate the current unit of work to be of low priority or being redundant, the work on this unit continues. Therefore preemption tends to reduce the total amount of work executed by the parallel branch and bound algorithm. Again it is clear that there exists a tradeoff between the total reduction in the work to be performed, and the work involved in the preemption.

We say that a parallel branch and bound algorithm has *perfect information exchange* if all the knowledge generated is available to all the processes immediately once it is generated. Note that the mere availability of new knowledge does not imply that this knowledge is actually used immediately. It is possible that a process first completes the work it is currently performing before it takes a look at the new knowledge.
A process of a parallel branch and bound algorithm *preempts upon arrival of new knowledge* if, each time new knowledge arrives, the process preempts the work it is currently performing, and takes the new knowledge into account immediately. As a consequence of the new knowledge just arrived, the process might decide to suspend or cancel the part of the current unit of work not yet completed, or to continue with it.
A parallel branch and bound algorithm has *perfect interaction* if it has perfect information exchange, if the processes preempt upon arrival of new knowledge, and if this preemption can be performed in zero time. Remember that preempting includes taking decisions based upon the new knowledge. Due to the perfect information exchange, such an algorithm preempts its work and decides whether to continue on the part not yet completed or to suspend or cancel it, each time new knowledge is generated.
Note that perfect information exchange and perfect interaction are theoretical concepts which will be used in analyzing the behavior of parallel branch and bound algorithms. In real life, branch and bound algorithms cannot attain perfect information exchange or perfect interaction because communication cannot be

done in zero time.

As mentioned before, applying dominance tests can be very hard if several decentral pools are used for dividing the work. Because a subproblem can be dominated by an arbitrary other subproblem, the process pruning a pool has to have complete knowledge about all currently dominating subproblems, even those stored in other pools and those branched from or eliminated by lower bound tests. The only way to accomplish this is by broadcasting all subproblems once they are generated. This however severely increases the communication complexity of the algorithm.
Instead of applying dominance tests at a global level by checking all pairs of subproblems generated, these tests can also be applied at a local level by checking only pairs of subproblems in the same pool. This way some of the advantages of dominance tests can be preserved without an increase in communication complexity.

What constitutes a good tradeoff between communication complexity and computation complexity depends on the characteristics of the problem instance to be solved, as well as on the characteristics of the parallel algorithm and the parallel computer system used.

2.2.1.3. Synchronicity

The last parameter of a parallel branch and bound algorithm to be specified is what happens when a process completes the unit of the work it is working upon. Again there are two extremes: before accessing a pool of outstanding work and starting with its next unit of work the process can wait for all other processes to complete their unit of the work, or, the process can access a pool and start working upon a new unit of work immediately without waiting for fellow processes to complete their unit.
The fact whether or not the processes perform their work in a synchronized way has consequences for the utilization of the various processes as well as for the communication complexity of the parallel algorithm.
For synchronized processes it holds that if the tasks to be performed are not of equal size, computation power is lost while waiting for other processes to complete their task. In addition, synchronization involves communication because one way or another, the processes must find out whether the other processes have completed their tasks or not. Therefore synchronization tends to increase the communication complexity of the algorithm. An advantage of synchronization is that dividing the work among the various processes tends to be easy, because it is known when the processes will be ready to start their next task.
The characteristics of asynchronous processes are exactly the opposite.

Note that if a parallel algorithm is synchronous, this does not imply that all processes are always executing the same instruction at the same point in time.

The synchronicity of the processes has also consequences for the *determinism* of the resulting parallel branch and bound algorithm. If the processes are not synchronized, the division of the work among the processes, and the exchange of the knowledge generated, introduce *nondeterminism* in the resulting algorithm. This is due to the fact that variations can occur with respect to the exact order in which the extractions, additions, and eliminations in the pools of work are carried out by the various processes, or with respect to the point in time a process will be notified of, for example, an update of the upper bound. Interchanging an addition to a pool by a process with an extraction from the same pool by another process might cause the algorithm to follow a different path, resulting in a different solution. Even if two consecutive executions of the algorithm yield the same solution, the work carried out during these executions might be completely different.
The nondeterminism does not change the properties of the subproblems in the active set. For all these subproblems it still holds that, as long as they are in a pool, they are neither branched from nor eliminated. Therefore the solution yielded by the asynchronous parallel algorithm is always a correct one.

2.2.2. An example of a parallel branch and bound algorithm

For our research, we are interested in the consequences of high level parallelism introduced by branching from several subproblems in parallel using the knowledge generated in an optimal way. Here ‘optimal’ means that during execution of the parallel branch and bound algorithm we try to branch only from those subproblems which the sequential algorithm would also branch from. We have developed an asynchronous

parallel branch and bound algorithm which uses a single central pool for dividing the work among the processes. The basic unit of work is branching from a single subproblem, which includes computing the lower bounds to the subproblems thus generated. Because the unit of work is the branching from a single subproblem, the heuristic function defined on the units of work is the same as the heuristic function defined on the subproblems.
The algorithm features a master process and several slave processes (cf. figure 2.1). The master takes all important decisions, and the slaves do as they are told by the master. In particular, the master maintains the active set (i.e., the pool of outstanding work) and decides which subproblems to branch from and when. The slaves perform the actual branchings and compute lower bounds to the subproblems thus generated.
In order to supervise everything properly, the master collects all knowledge generated so far by the slaves (the subproblems generated, the feasible solutions found, etc.). Interpreting this knowledge, the master maintains the active set. Each time the master is aware of a slave becoming idle, he extracts the subproblem with the smallest heuristic value from this set, and sends this problem to the slave to branch from. If there are no active subproblems, the slave is temporarily put into a queue of idle slaves until new subproblems become available. The execution of the branch and bound algorithm is terminated as soon as all slaves are in the idle queue. Each time the upper bound is updated, the master sends the new upper bound to all slaves. This enables the slaves to perform lower bound tests on the subproblems they generate. The slaves do not preempt upon arrival of a new upper bound.
A slave branches from the subproblem it receives from the master. The subproblems thus generated are sent back to the master. After completing the branching, the slave requests new work from the master by notifying the master that it has become idle.

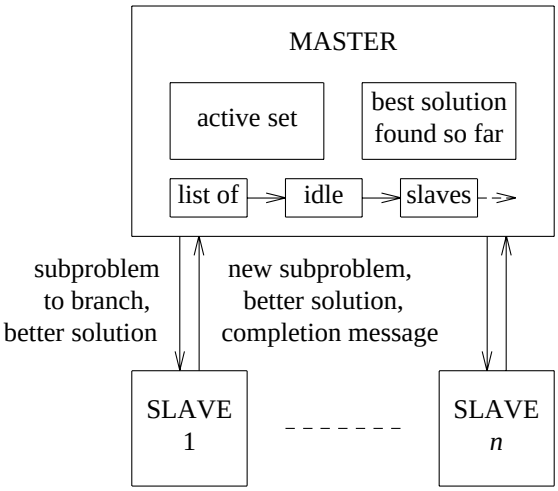

Figure 2.1. The Boulder algorithm

At first sight it may seem strange that a slave sends all the subproblems it generates to the master, and then asks the master for a subproblem to branch from, instead of branching from one of the subproblems it just generated itself. This is done because a slave does not have overall knowledge. The subproblems just generated might not be good ones. By letting a slave ask the master for new work, the amount of superfluous work done is minimized.
We call this parallel branch and bound algorithm the Boulder algorithm. The name is derived from the fact that we first tested this algorithm while visiting the University of Colorado at Boulder, Colorado. For a complete description of the algorithm, and for computational experiments with the algorithm, we refer to [Trienekens 1989].

3. ANOMALIES

3.1. Introduction

It is known for some time that the parallel execution of a branch and bound algorithm can result in unreasonable speedups or even slowdowns. In the literature these phenomena are described as anomalies [Li & Wah 1984]. The execution of deterministic parallel branch and bound algorithms can suffer from acceleration anomalies, deceleration anomalies, and detrimental anomalies. Next to suffering from these anomalies, the execution of non deterministic parallel branch and bound algorithms can also suffer from fluctuation anomalies.

Let the *corresponding sequential branch and bound algorithm* be defined as the parallel algorithm executed by a single process. Let I denote a problem instance to be solved. Let $T_n(I)$ denote the amount of time needed to solve problem instance I by a parallel branch and bound algorithm executed by n processes. Let $T_1(I)$ denote the amount of time needed to solve this problem instance by the corresponding sequential algorithm.
An unsuspecting reader might expect the following relation to hold:

$$T_n(I) = \frac{T_1(I)}{n}. \tag{3.1}$$

I.e., a decrease in execution time which is proportional to the number of processes the parallel branch and bound algorithm is executed by. However, a few computational experiments reveal that most of the time

$$\frac{T_1(I)}{n} \leq T_n(I) < T_1(I). \tag{3.2}$$

Sometimes it even happens that

$$T_n(I) < \frac{T_1(I)}{n} \tag{3.3}$$

or

$$T_n(I) \geq T_1(I). \tag{3.4}$$

The anomalies described by formulaes 3.2 - 3.4 are called respectively a detrimental anomaly, an acceleration anomaly, and a deceleration anomaly. In case of a *detrimental anomaly* the parallelism speeds up the execution, although not to the extent expected, in case of an *acceleration anomaly* an unreasonable speedup occurs, whereas in case of a *deceleration anomaly* the parallelism slows down the solution process of the problem instance on hand.
It is also possible that consecutive runs of the same parallel algorithm solving the same problem instance need substantially different amounts of time. In that case, a *fluctuation anomaly* occurs.
The first three kinds of anomalies all compare the execution of a parallel branch and bound algorithm with the execution of the corresponding sequential branch and bound algorithm. Note that none of these anomalies states something about what happens if there is a change in the number of processes executing the parallel branch and bound algorithm.

An example of a fluctuation anomaly can be demonstrated from the problem instance displayed in figure 3.1. In this problem instance, each subproblem decomposes into two smaller subproblems. Branching from a subproblem involves eight units of work. The branching from the first subproblem generated yields a feasible solution which can eliminate all other subproblems by a lower bound test. All subproblems generated by branching from the second subproblem do not have a feasible solution, but this fact cannot be proven without complete decomposition.
Suppose this problem instance is solved by a parallel branch and bound algorithm which has perfect interaction. The parallel algorithm is executed by two processes, *A* and *B*. Process *A* can perform four units of work in one unit of time, whereas process *B* can only perform one unit of work per unit of time. Process *A* branches from the original problem. As soon as this branching is completed, the two processes start branching from the two subproblems thus generated. The following two divisions of work are possible: firstly, process *A* gets subproblem 1 and process *B* gets subproblem 2, or secondly, process *A* gets

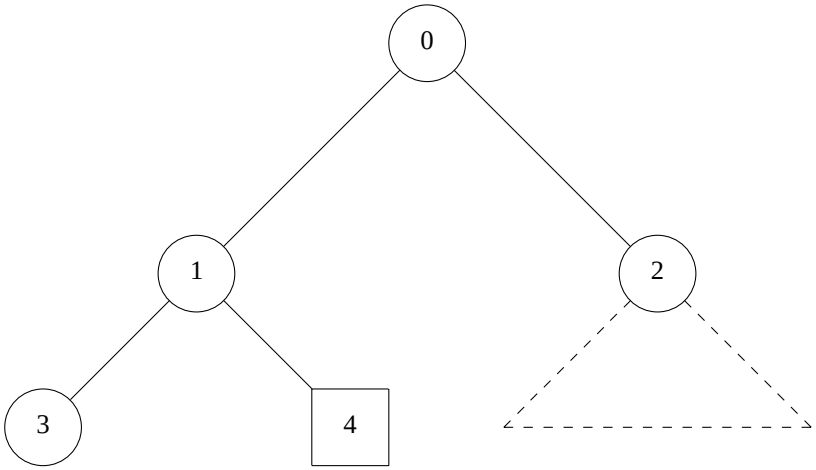

Figure 3.1. An example of a fluctuation anomaly

subproblem 2 and process *B* gets subproblem 1. In the first case, the feasible solution eliminating all other subproblems is generated after four units of time. In the second case however, it takes ten units of time.
Note that consecutive runs can only differ in execution time if there is a nondeterministic component in the parallel branch and bound algorithm. A deterministic algorithm always takes the same decisions, and therefore divides exactly the same work in exactly the same way among the processes. Hence the execution times will always be the same. Due to the nondeterminism introduced by the asynchronicity, it is impossible to guard asynchronous parallel branch and bound algorithms against fluctuation anomalies.
For examples of the first three kinds of anomalies we refer to [Li & Wah 1984] and [Lai & Sahni 1984]. Although the examples in these papers are for synchronous parallel branch and bound algorithms, they can be adapted in a straightforward manner to asynchronous parallel branch and bound algorithms.

Note that it is very difficult to determine all anomalies which occurred during execution of a parallel branch and bound algorithm, or to verify that some kind of anomaly did not occur. For example, the mere fact that the execution of the parallel algorithm takes more time than the execution of the corresponding sequential algorithm does not imply that only a deceleration anomaly occurred. It is possible that some part of the execution was improved upon by an acceleration anomaly, whereas other parts suffered from detrimental and deceleration anomalies. In the end, the negative effects caused by the deceleration anomaly were dominating.

Of course one would like to preserve acceleration anomalies, and avoid deceleration and detrimental anomalies. Next to that, one hopes to be lucky in that fluctuation anomalies will improve upon the execution of the parallel branch and bound algorithm. However, detrimental and deceleration anomalies cannot be completely eliminated. Detrimental anomalies can sometimes be unavoidable because they can be inherent to the particular problem instance to be solved. For example, it can happen that all the subproblems branched from by the sequential algorithm are descendants of each other. On top of that, deceleration anomalies can occur because a parallel branch and bound algorithm has to spend effort on interactions between processes and/or synchronizations, whereas the corresponding sequential algorithm does not.

Before we can start our investigation of anomalies we need a closer look at the information used by the various rules and tests of a branch and bound algorithm.

3.2. The Information Used by Rules and Tests

During execution, a branch and bound algorithm repeatedly applies rules and tests to sets of objects. For example, the lower bound rule is used to compute a lower bound to a given subproblem, the branching rule is used to split a given subproblem into smaller subproblems, the selection rule is used to select a subproblem from a given set of subproblems, and the feasibility test is used to determine whether or not it can be proven that a given subproblem does not have a feasible solution.

The precise set of objects a rule or test is applied to, depends upon this particular rule or test. The bounding and branching rules, and the feasibility test, are all applied to singleton sets, i.e., sets consisting of a single subproblem. A set of objects the selection rule is applied to consists of the subproblems in the active set. A set of objects the lower bound and dominance tests are applied to consists of all subproblems to be checked for elimination and the current upper bound, respectively all currently dominating subproblems.

The outcome of the application of a given rule or test to a given set of objects depends on the knowledge available to and used by the rule or test. For example, the computation of a lower bound can depend on the current value of the upper bound, the outcome of a lower bound test depends on the value of the best feasible solution found hitherto, and the outcome of the branching rule can depend on a branching scheme which was found to be successful in other parts of the search tree.
The information that can be used by a rule or a test when it is applied to a set of objects can be divided into two categories: controllable information and non controllable information. *Controllable information* consists of the information generated by the algorithm itself, for example, all feasible solutions found. *Non controllable information* consists of the information not generated by the algorithm itself, but by the environment the algorithm is executing in. For example, while solving integer programming problems, the operator could be asked for the next cutting plane to be used [Grötschel 1980], or some test used by the algorithm could base its outcome on the number of free blocks on the computer's disk or the cpu time used hitherto.
Controllable information can be subdivided into local information and global information. *Local information* consists of all information derivable from the set of objects the rule or test is applied to. For example, the information used by a selection rule which selects the subproblem with the smallest heuristic value can be determined completely from the set of subproblems this rule is applied to. *Global information* consists of all information derivable from all currently existing sets of objects. For example, global information is used when a lower bound rule improves upon its initial bound by applying a Lagrangean technique which uses the current value of the upper bound [Shapiro 1979]. Or, a feasibility test might be able to prove that the given subproblem does not have a feasible solution because somewhere in the past a similar subproblem has been proven by complete decomposition to not have a feasible solution. Note that local information is a subset of global information.

In theory, global information is under control of the algorithm because the algorithm decides whether and when to create an object. However, as soon as the order in which the various objects are generated changes, rules and tests based upon global information can yield different outcomes when applied to the same set of objects. Hence it is very hard to compare consecutive executions of a branch and bound algorithm which uses rules and tests based upon global information. The same holds for rules and tests based upon non controllable information.
To be able to compare the execution of several invocations of the same branch and bound algorithm solving the same problem instance, it is needed that the application of the same rule or test to the same set of objects always yields the same result. Therefore, for the remainder of this paper, we will assume that all rules and tests only use local information.

3.3. Anomalies and Perfect Interaction

In this section we will investigate the behavior of parallel branch and bound algorithms which have perfect interaction, and in which the basic unit of work is branching from a single subproblem including the computation of lower bounds to the subproblems thus generated.

All anomalies during execution of a parallel branch and bound algorithm are caused by the branching from several subproblems in parallel. Due to these branchings in parallel, the knowledge generated at a particular point during the execution of the parallel branch and bound algorithm (i.e., the subproblems generated, branched from, eliminated, and currently dominating, the lower and upper bounds, and the feasible solutions found) can differ from the knowledge generated at the corresponding point during execution of the sequential algorithm. This different knowledge can result in a different order in which the subproblems are branched from, and thus in a different search tree generated.
The subproblems branched from by the sequential branch and bound algorithm can be seen as a set of travel instructions to get to the optimal solution. The effort involved in traveling to the optimal solution

using this set of instructions is proportional to the number of instructions, i.e., the number of subproblems branched from. However, just as there are multiple ways to reach a given city, there are multiple sets of travel instructions to get to the optimal solution. There even might be multiple optimal solutions. It is possible that some processes of a parallel branch and bound algorithm neglect this set of travel instructions, and try other sets of instructions instead. Because a priori nothing is known about these sets of travel instructions, nothing is known about the effort needed to get to an optimal solution using these instructions.
The execution of a parallel branch and bound algorithm suffers from an anomaly as soon as some of the processes neglect the travel instructions used by the sequential algorithm. As long as at least one of the processes is proceeding according to these travel instructions, the time needed for executing the parallel algorithm will never be greater than the time needed by the sequential algorithm. However, as soon as all the processes neglect this set of travel instructions, there is the possibility that the algorithm might suffer from a deceleration anomaly.
Acceleration anomalies can be seen as shortcuts. It could be the case that the sequential set of travel instructions happened to be a detour. The parallel branch and bound algorithm might be lucky enough to encounter a shortcut. Hence, as soon as some process neglects the sequential set of travel instructions, there is the probability of this process finding a shortcut, and the execution being improved upon by an acceleration anomaly.

We start our investigation by developing sufficient conditions to prevent deceleration anomalies from degrading the performance of parallel branch and bound algorithms whose elimination rule involves only feasibility and lower bound tests. It turns out that the sufficient conditions already known for synchronous branch and bound algorithms with a single central pool for dividing the work [Li & Wah 1984] can be generalized for all synchronous and asynchronous parallel branch and bound algorithms.
Next we investigate the consequences of adding a dominance test to the elimination rule. Again we develop sufficient conditions for preventing deceleration anomalies.
Finally we develop necessary conditions for acceleration anomalies to improve upon the performance of the algorithm.

Before we can start our investigation of anomalies, we need some more definitions.

A *basic subproblem* is a subproblem which is branched from during execution of the corresponding sequential branch and bound algorithm. Note that a basic subproblem can only be generated by branching from a basic subproblem.

The *sequential solution* is defined as the optimal solution found by the corresponding sequential algorithm.

The process of extracting a subproblem to branch from from a pool, branching from this subproblem, adding the descendants created to a pool, pruning the pools, and broadcasting the knowledge just generated, is called *working upon a subproblem*.

Let, at a given point in time during execution of a parallel branch and bound algorithm, $P_1, \ldots, P_n$ be the series of active subproblems. Let P_k be the basic subproblem with minimal heuristic value in this series, i.e.,

$$k = \min \{ i \mid P_i \text{ is a basic subproblem and } h(P_i) = h_{\min},\ i = 1, \ldots, n \}$$

with $h_{\min} = \min \{ h(P_j) \mid P_j \text{ is a basic subproblem, } j = 1, \ldots, n \}$. The *corresponding point* during the execution of the corresponding sequential algorithm is defined as the branching from P_k if P_k exists; otherwise, the corresponding point is undefined.

The *path number* of a node in the search tree is a sequence of $d+1$ integers, where d is the level of the node in the search tree. This number uniquely represents the path from the root to the node. The root has path number 1. All other path numbers $e = e_0 e_1 e_2 \cdots e_n$, $n = 1, \ldots, d$, are defined recursively. A node P_{i_j} on level k that is the j'th son (counting from the left) from node P_i with path number $e(P_i) = e_0 e_1 e_2 \cdots e_{k-1}$ has path number $e(P_{i_j}) = e_0 e_1 e_2 \cdots e_{k-1} j$. Figure 3.2 shows an example of how to number the nodes of a search tree.
Path numbers can be ordered lexicographically. Two path numbers $e = e_0 e_1 \cdots e_m$ and $\hat{e} = \hat{e}_0 \hat{e}_1 \cdots \hat{e}_n$

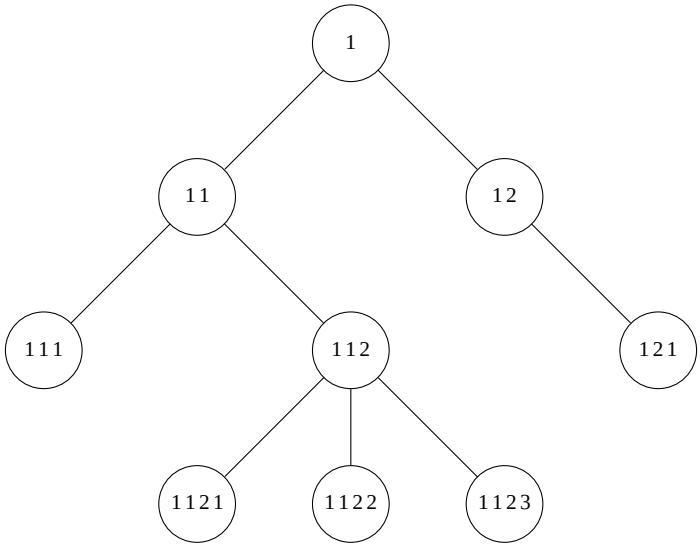

Figure 3.2. Dewey decimal notation

are said to be equal (denoted by $e = \hat{e}$) if they are identical sequences of numbers, i.e., $m = n$ and $e_i = \hat{e}_i$ for $i = 0, \ldots, m$. e is said to be smaller than $\hat{e}$ (denoted by $e < \hat{e}$) if its sequence is lexicographically smaller, i.e., either there exists an $i \leq min(m,n)$ such that $e_j = \hat{e}_j$ for $j = 0, \ldots, i-1$, and $e_i < \hat{e}_i$, or $e_j = \hat{e}_j$ for $j = 0, \ldots, m$ and $m < n$.
This method of numbering the nodes of a tree is known as *Dewey Decimal Notation* [Knuth 1973].

A heuristic function h is *injective* if it assigns a unique value to each subproblem:

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

Injectivity is no severe restriction because each non injective heuristic function can be easily transformed into an injective one by extending a heuristic value with the unique path number of its argument (which corresponds to a node of the search tree) as an additional component, and by defining a lexicographic ordering on these tuples. Due to the fact that all path numbers are unique, all heuristic values will be unique.
Note that an injective heuristic function induces a complete ordering on the subproblems through their heuristic values.

A heuristic function h is *non misleading* if it never provides misleading information about a subproblem. The heuristic values assigned by the function must be such that no subproblem has a smaller heuristic value than the subproblem it was generated from by decomposition has, i.e.,

$$h(P_i) \leq h(P_j) \quad \text{if } P_j \text{ is a descendant of } P_i.$$

A dominance relation D is said to be *consistent* with a lower bound function g if the fact that P_i dominates P_j implies that $g(P_i) < g(P_j)$ for all subproblems P_i and P_j.

A heuristic function h is said to be *consistent* with a lower bound function g if the fact that $h(P_i) < h(P_j)$ implies that $g(P_i) < g(P_j)$ for all subproblems P_i and P_j.
Note that if a heuristic function is injective, consistency with the lower bound function implies that the lower bound function too is injective.

The theorems to follow will use the following property of a parallel branch and bound algorithm which has perfect interaction.

Lemma 1:
A parallel branch and bound algorithm which has perfect interaction will always be working upon the active subproblem with the smallest heuristic value. I.e., at each point in time during execution of the algorithm, the active subproblem with the smallest heuristic value will be worked upon by some process.

Proof (by contradiction):
Consider the first time during execution that the parallel branch and bound algorithm is not working upon the active subproblem with the smallest heuristic value.
This fact implies that the processes accessing the pool of work this active subproblem is in, are all working upon active subproblems with higher heuristic values.
This, however, contradicts perfect knowledge exchange and preemption upon arrival of new knowledge. □

3.3.1. Preventing deceleration anomalies

Deceleration anomalies can be prevented if two conditions can be met. Firstly, the knowledge generated and used during execution of the parallel branch and bound algorithm must always be a superset of the knowledge generated and used during execution of the corresponding sequential algorithm. Or more precisely: each time the parallel algorithm takes a decision, the knowledge on which this decision is to be based must include all the knowledge used by the corresponding sequential algorithm at the corresponding point during its execution. Secondly, at each point in time, at least one of the processes of the parallel algorithm must be working upon a basic subproblem.

3.3.1.1. Parallel branch and bound with lower bound tests only

Now we are ready to consider the case of parallel branch and bound algorithms that have elimination rules which do not involve dominance tests. In this section we will investigate how to prevent deceleration anomalies from degrading the performance of such parallel branch and bound algorithms.
The first condition of the previous section can be met by accomplishing that for each point in time during execution of the parallel algorithm, all the subproblems the corresponding sequential algorithm branches from until reaching the corresponding point during its execution are either branched from by the parallel algorithm, or are eliminated by it. To accomplish this, all subproblems must be uniquely identifiable by their heuristic value. Hence the heuristic function h used by the selection rule must be injective.
The second condition can be met by requiring the heuristic function used to be non misleading. A non misleading heuristic function precludes the generation of children of non basic subproblems which have a higher priority to be branched from than some of the basic subproblems.
We start by proving a theorem which yields an upper bound on the work to be done during execution of a parallel branch and bound algorithm. The theorem states that if during execution of the algorithm a point is reached at which there exist no more basic subproblems (i.e., all basic subproblems have either been branched from or have been eliminated) enough knowledge has been generated to solve the problem instance on hand. Next we will prove a theorem which ensures that always at least one of the basic subproblems is being worked upon. Thus, in the worst case, all basic subproblems are branched from sequentially by the least powerful process. After these two theorems, we will give some corollaries which define upper bounds on the complexity of synchronous and asynchronous parallel branch and bound algorithms.

Theorem 1:
If during execution of a parallel branch and bound algorithm which uses an elimination rule involving only feasibility and lower bound tests, a point is reached where there exist no more active basic subproblems, enough knowledge has been generated to solve the problem instance on hand. I.e., a feasible solution has been generated which can eliminate all remaining active subproblems by a lower bound test.

Proof:
By definition, the execution of the corresponding sequential algorithm is completed after branching from all basic subproblems. All non basic subproblems can be eliminated by a lower bound test by the sequential solution.
Note that, due to the fact that the lower bound of a descendant can never be smaller than the lower bound of an ancestor, all the subproblems not generated by the sequential algorithm can be eliminated by a lower

bound test by the sequential solution.
A parallel branch and bound algorithm starts by working upon the original problem. Therefore, at the beginning of the execution, there is an active basic subproblem.
If during execution of the parallel algorithm a point is reached at which there are no more active basic subproblems, the parallel algorithm has either branched from all basic subproblems, or eliminated some basic subproblems. Because basic subproblems are descendants of basic subproblems, no more basic subproblems can be generated any more.
The parallel algorithm has either generated the sequential solution, or it has eliminated one of the ancestors of this subproblem.
If the sequential solution has been generated, this solution can eliminate all active non basic subproblems by a lower bound test.
If an ancestor of the sequential solution has been eliminated, this ancestor must have been eliminated by a lower bound test because another feasible solution was found. This other feasible solution must have the same value as the sequential solution, and therefore can eliminate all active non basic subproblems. □

Deceleration anomalies degrade the performance of a parallel branch and bound algorithm by performing work which is not performed by the corresponding sequential algorithm, i.e., by branching from non basic subproblems. Deceleration anomalies can be prevented from degrading the performance by ensuring that always at least one of the processes is following the same travel instructions as the corresponding sequential algorithm, i.e., is working upon a basic subproblem. One way to realize the above is by ensuring that the active basic subproblem with the smallest heuristic value is always being worked upon. Note that this condition is sufficient but not necessary. The time 'lost' during execution of a parallel algorithm whilst all processes were following other travel instructions could be easily compensated for by the time 'gained' when several processes were concurrently following the sequential travel instructions.
To ensure that always at least one process is following the sequential travel instructions, it is necessary that the subproblems can be identified uniquely. Therefore the heuristic function used must be injective. Next to that, the heuristic function must also be non misleading. Otherwise some subproblems might look whether they are part of the travel instructions, whereas in reality they are not. Or, in other words, the branching from some non basic subproblem might generate subproblems with a lower heuristic value, and thus a higher priority to be branched from than some of the basic subproblems.

Theorem 2:
At each point in time during execution of a parallel branch and bound algorithm which has perfect interaction, an elimination rule involving only feasibility and lower bound tests, and a heuristic function which is injective and non misleading, some process will be working upon the active basic subproblem with the smallest heuristic value.

Proof (by contradiction):
Consider the first time during execution of the parallel branch and bound algorithm that the active basic subproblem with the smallest heuristic value is not being worked upon. This fact together with the fact that a parallel branch and bound algorithm with perfect interaction always works upon the active subproblem with the smallest heuristic value (cf. lemma 1), imply that at least one of the processes is working upon a non basic subproblem with a smaller heuristic value.
Let $P_0, P_1, \ldots, P_n$ be the series of basic subproblems as branched from by the corresponding sequential algorithm. This series is ordered according to increasing heuristic value. Let P_k be the active basic subproblem with the smallest heuristic value which is not chosen to work upon by the parallel algorithm, and let $\hat{P}$ be the non basic subproblem with smaller heuristic value which is chosen instead.
The knowledge generated and collected by the corresponding sequential algorithm at the corresponding point during its execution is obtained by branching from the basic subproblems $P_0, \ldots, P_{k-1}$. The fact that $\hat{P}$ is not a basic subproblem implies the existence of a subproblem P_j, $0 \leq j \leq k-1$, whose branching from generated a feasible solution P' which eliminated $\hat{P}$, or one of its ancestors, by a lower bound test.
Now we return to the parallel algorithm. Note that due to the perfect interaction, all knowledge generated is immediately available to, and taken into account by, all the processes and pools.

Because P_k is the active basic subproblem with the smallest heuristic value, the basic subproblems $P_0, \ldots, P_{k-1}$ are not active at that point in time (i.e., they are either generated and branched from, generated and eliminated, or, not yet generated). Furthermore, because the heuristic function used is non misleading and injective, none of the subproblems P_i, $0 \leq i \leq k-1$, can be generated any more. Hence P_j must have been branched from, or P_j (or one of its ancestors) must have been eliminated by the parallel algorithm.
Suppose P_j has been branched from. In this case P' has been generated, and $\hat{P}$ can be eliminated by this subproblem.
Suppose P_j or one of its ancestors has been eliminated. In that case a feasible solution P'' has been generated which eliminated P_j or one of its ancestors. Because the lower bound of a descendant is never smaller than the lower bound of an ancestor, P'' can also eliminate $\hat{P}$ by a lower bound test.
Hence the parallel algorithm would not branch from $\hat{P}$. □

Note that the theorem is independent of the way the work is divided among the various processes and of the relative computing power of the various processes. If there are several pools of work, it does not matter which pool the basic subproblem with the smallest heuristic value is in, to which pool subproblems created are added, how many processes access a particular pool, or whether work is exchanged between the various pools.

Using the two theorems just proven it is easy to get at a worst case complexity of synchronous and asynchronous parallel branch and bound algorithms.

Corollary 1:
The number of iterations needed for solving a given problem instance by a synchronous parallel branch and bound algorithm which has perfect interaction, an elimination rule involving only feasibility and lower bound tests, and a heuristic function which is injective and non misleading, is less than or equal to the number of iterations needed by the corresponding sequential algorithm.

Proof:
Theorem 1 states that once there are no more basic subproblems, there is enough knowledge generated to solve the problem instance on hand. Due to the perfect interaction, this knowledge is available to all processes and pools. Theorem 2 states that at each point in time during execution the basic subproblem with the smallest heuristic value will be worked upon. Hence each iteration at least one active basic subproblem will be being branched from. In the worst case, this is the only basic subproblems branched from per iteration. □

Let $T_p(I)$ be the time needed for solving problem instance I on a given parallel computer system with a parallel branch and bound algorithm whose elimination rule involves only feasibility and lower bound tests, and whose heuristic function is injective and non misleading. Let $T_s(I)$ be the time needed by the least powerful processing element of this parallel system for solving problem instance I by executing the corresponding sequential algorithm. Let N be the number of basic subproblems, and let m be the number of processes the parallel algorithm is executed by.
We assume that pools of outstanding work are entities which can handle the following kinds of requests: add subproblems to the pool, extract the subproblem with smallest heuristic value from the pool, and prune the pool. Furthermore we assume that all these requests can be handled in constant time c, and that the requests are served using a first come, first served policy.

Corollary 2:
For the execution of an asynchronous parallel branch and bound algorithm which has perfect interaction, an elimination rule involving only feasibility and lower bound tests, and a heuristic function which is injective and non misleading, it holds that

$$T_p(I) \leq T_s(I) + 3.N.(m-1).c$$

Proof:
Theorem 1 states that once there are no more basic subproblems, the problem instance has been solved. Theorem 2 states that at each point in time the active basic subproblem with the smallest heuristic value will be being worked upon. In the worst case, all basic subproblems are worked upon sequentially by the process executing on the least powerful processing element.
The difference between the corresponding sequential and the parallel algorithm is that in the parallel case several processes are interacting with the pools. It is possible that if a process sends a request to a pool, the pool first has to serve pending requests from other processes. Per subproblem branched from, a process sends at the most 3 requests to the pools: 1 request for getting the subproblem, 1 request for adding its descendants to the pools, and 1 request for pruning the pools. Because there are m processes, there are at the most m-1 pending requests when a process sends a request to a pool. Serving these requests takes at the most $(m-1).c$ time. Hence the additional time needed by the parallel algorithm is equal to the product of the number of basic subproblems, the maximum number of requests per basic subproblem, and the maximum waiting time per request. □

Note that corollary 2 does not state anything about the total number of subproblems branched from.

3.3.1.2. Parallel branch and bound with lower bound and dominance tests
The combined use of lower bound and dominance tests in a parallel branch and bound algorithm can cause additional anomalies because this combined use need not be transitive. For example, if subproblem P_i can eliminate subproblem P_j by a lower bound test, and if subproblem P_j can eliminate subproblem P_k by a dominance test, it does not follow automatically that P_i can eliminate P_k by either lower bound or dominance test.
This non transitivity can cause all kinds of anomalies if a parallel algorithm applies these tests in a different order than the corresponding sequential algorithm does. For example, suppose again P_i can eliminate P_j by a lower bound test, and P_j can eliminate P_k by a dominance test. Suppose the order in which the corresponding sequential algorithm generates these subproblems is P_k, P_j, and P_i, whereas the parallel algorithm generates them in order P_j, P_i, and P_k. Hence the corresponding sequential algorithm will eliminate both P_j and P_k, whereas the parallel algorithm will only eliminate P_j. Therefore the parallel algorithm has to branch from P_k (and its descendants), which might involve lots of work.

Because nothing can be said about the exact order in which the various subproblems are generated by the parallel algorithm, it is impossible to solve this non transitivity problem by ensuring that the tests are always performed in the same order as the corresponding sequential algorithm performs them. However, for certain classes of dominance relations it holds that dominance and lower bound tests are transitive.
Once again we will prove two theorems. The first theorem states that if during the solution process of a problem instance which has a unique optimal solution, a point is reached at which there exist no more active basic subproblems, there is enough knowledge generated to solve this problem instance. The second theorem states sufficient conditions to ensure that always at least one of the basic subproblems is being worked upon. Using these theorems, corollaries stating upper bounds on the complexity of synchronous and asynchronous parallel branch and bound algorithms can be formulated in a straightforward manner.

Theorem 3:
If the problem instance to be solved has a unique solution, then if during execution of a parallel branch and bound algorithm which uses an elimination rule involving feasibility and lower bound tests, and dominance tests based upon a dominance relation which is consistent with the lower bound function, a point is reached where there are no more active basic subproblems, then enough knowledge is generated to solve the problem. I.e., a feasible solution has been generated which can eliminate all remaining active subproblems by a lower bound test.

Proof:
The subproblems can be divided into three disjoint classes:

$C_1 = \{ P_i \mid P_i$ is generated by and branched from by the sequential algorithm $\}$

$C_2 = \{ P_i \mid P_i$ is generated and eliminated by the sequential algorithm $\}$

$C_3 = \{ P_i \mid P_i$ is not generated by the sequential algorithm $\}$

Note that the class C_1 consists of the basic subproblems and that the class C_2 consists of the subproblems eliminated and the subproblem yielding the optimal solution. Note next that for each subproblem $P_i \in C_2$ (except the subproblem yielding the optimal solution) there exists a subproblem P_j, $P_j \in C_1$ or $P_j \in C_2$, which can eliminate P_i by a lower bound or a dominance test. P_j can also eliminate all descendants P_k of P_i by the same kind of test. By definition, $P_k \in C_3$.
Because the problem instance to be solved has a unique solution, neither the subproblem yielding this solution nor one of its ancestors can be eliminated by a lower bound or a dominance test. Therefore, once there are no more basic subproblems, the subproblem yielding the unique solution must have been generated, and the optimal solution has been generated.
Now this solution has to be proven optimal by eliminating all active subproblems. Because subproblems belonging to the class C_3 can be eliminated by the same subproblem that can eliminate their ancestor in the class C_2, it is sufficient to prove that there is enough knowledge generated to eliminate all subproblems belonging to the class C_2.
Let P_i be an arbitrary member of the class C_2, but not the subproblem yielding the optimal solution. Because $P_i \in C_2$ there exists a subproblem P_j, $P_j \in C_1$ or $P_j \in C_2$, which can eliminate P_i. The subproblem P_j has either been generated by the parallel algorithm or not.
Suppose P_j has been generated. In that case P_i can be eliminated by P_j.
Suppose P_j has not been generated. The fact that P_j is a basic subproblem or a direct descendant of a basic subproblem, together with the fact that there are no more basic subproblems, imply that the parallel algorithm generated a subproblem P_k which eliminated P_j or one of its ancestors. If P_k eliminated an ancestor of P_j, then by definition P_k can also eliminate P_j by the same kind of test (because descendants can be eliminated by the same kind of test that eliminated their ancestor). So, without loss of generality, it can assumed that P_k eliminated P_j. Now there are four possibilities:

i. P_j eliminated P_i by a lower bound test, and P_k eliminated P_j by a lower bound test. Because lower bound tests are transitive, P_k can eliminate P_i by a lower bound test.

ii. P_j eliminated P_i by a lower bound test, and P_k eliminated P_j by a dominance test. Because the optimal solution has been generated, P_i can be eliminated by a lower bound test by this solution.

iii. P_j eliminated P_i by a dominance test, and P_k eliminated P_j by a dominance test. Because dominance tests are transitive, P_k can eliminate P_i by a dominance test.

iv. P_j eliminated P_i by a dominance test, and P_k eliminated P_j by a lower bound test. The fact that the dominance relation is consistent with the lower bound function implies that $g(P_j)$ is smaller that $g(P_i)$. Hence P_k can eliminate P_i by a lower bound test.
Therefore P_i can be eliminated by P_k. □

The above mentioned conditions together with a heuristic function which is injective and non misleading, are not strong enough to ensure that the active basic subproblem with the smallest heuristic value is always being worked upon. The problem lies with a dominance test followed by a lower bound test. These two tests are not transitive in this order, i.e., if P_k can eliminate P_j by a dominance test and if P_j can eliminate P_i by a lower bound test, this does not imply that P_k can eliminate P_i by either one of these tests.
In the proof of theorem 3 we could bypass this obstacle due to the fact that once there existed no more basic subproblems, the optimal solution had been generated, and P_i could be eliminated by a lower bound test.
However, because nothing can be said about the particular point in time the optimal solution will be generated by the parallel algorithm, this method cannot be used to guarantee that the active basic subproblem with the smallest heuristic value is always being worked upon. If there exist non basic subproblems which have a smaller heuristic value than some of the basic subproblems, and which also have a lower bound

higher than the optimal solution, it is possible that all processes will be working upon these subproblems because they cannot be eliminated yet by a lower bound test (for example, because the parallel algorithm eliminated by a dominance test a subproblem whose branching from by the sequential algorithm yielded a non optimal solution).
In essence, the problem lies in the fact that some non basic subproblems have a higher lower bound value but a smaller heuristic value than some of the basic subproblems, i.e., in the heuristic function not being consistent with the lower bound function. Hence a heuristic function which is consistent with the lower bound function should be a sufficient condition to prevent deceleration anomalies.

Theorem 4:
At each point in time during execution of a parallel branch and bound algorithm which has perfect interaction, an elimination rule involving feasibility and lower bound tests, and a dominance test based upon a dominance relation which is consistent with the lower bound function, and a heuristic function which is injective, non misleading and consistent with the lower bound function, some process will be working upon the active basic subproblem with the smallest heuristic value.

Proof (by contradiction):
Consider the first time that during execution of the parallel branch and bound algorithm the active basic subproblem with the smallest heuristic value is not being worked upon. This fact together with the fact that a parallel branch and bound algorithm with perfect interaction always works upon the active subproblem with the smallest heuristic value (cf. lemma 1), imply that at least one of the processes accessing the pool of work this basic subproblem is in, has chosen a non basic subproblem with a smaller heuristic value to work upon.
Let $P_0, P_1, \ldots, P_n$ be the series of basic subproblems as branched from by the corresponding sequential algorithm. This series is ordered according to increasing heuristic value. Let P_k be the active basic subproblem with the smallest heuristic value which is not chosen to work upon by the parallel algorithm, and let $\hat{P}$ be the non basic subproblem with smaller heuristic value which is chosen instead.
The knowledge collected by the corresponding sequential algorithm at the corresponding point during its execution is obtained by branching from the basic subproblems $P_0, \ldots, P_{k-1}$. The fact that $\hat{P}$ is not a basic subproblem implies the existence of a subproblem P_j, $0 \leq j \leq k-1$, whose branching from generated a subproblem P' which eliminated $\hat{P}$ or one of its ancestors. The fact that the heuristic function is injective, non misleading, and consistent with the lower bound function together with the fact that $\hat{P}$ has a smaller heuristic value than P_k imply that the lower bound of $\hat{P}$ is smaller than the lower bound of P_j. Therefore $\hat{P}$ can never be eliminated by a lower bound test. Hence $\hat{P}$ must be eliminated by a dominance test by P'.
Now return to the parallel algorithm. Because P_k is the active basic subproblem with the smallest heuristic value, the basic subproblems $P_0, \ldots, P_{k-1}$ are not active at that point in time (i.e., they are either generated and branched from, generated and eliminated, or, not yet generated). Furthermore, because the heuristic function used is non misleading and injective, none of the subproblems P_i, $0 \leq i \leq k-1$, can be generated any more. Hence P_j must have been branched from, or P_j (or one of its ancestors) must have been eliminated by a dominance test by the parallel algorithm (basic subproblems can never be eliminated by a lower bound test because their heuristic values are smaller than the heuristic value of the optimal solution, hence their lower bounds are also smaller than the optimal solution).
Due to the perfect interaction, all knowledge generated hitherto is available to and taken into account by all the processes. So, if P_j has been branched from, P' has been generated, and $\hat{P}$ can be eliminated by dominance test by this subproblem. Otherwise, if P_j or one of its ancestors has been eliminated, this implies that a subproblem P'' has been generated which eliminated P_j by a dominance test. Because dominance tests are transitive, this solution can also eliminate $\hat{P}$ by a dominance test. Hence the parallel algorithm would not branch from $\hat{P}$. □

Together the conditions mentioned in theorems 3 and 4 are sufficient conditions to prevent deceleration anomalies because these conditions also guarantee that the problem instance to be solved has a unique solution, as is shown by the next lemma.

Lemma 2:
A problem instance solved by a branch and bound algorithm which uses a heuristic function which is injective and consistent with the lower bound function used, has a unique solution.

Proof:
An injective heuristic function assigns different heuristic values to all subproblems. Because the heuristic function is consistent with the lower bound function, it holds that

$$h(P_i) < h(P_j) \text{ implies } g(P_i) < g(P_j).$$

Because all heuristic values are different, all lower bounds must be different too. The fact that the lower bound is equal to the optimal solution if a subproblem can be solved without decomposition (cf. equation 2.3) implies that all solutions have different values. Hence the problem instance has a unique solution. □

Corollaries stating that the performance of the above described parallel branch and bound algorithms cannot be degraded by deceleration anomalies can now be easily formulated and proven.

3.3.2. Allowing acceleration anomalies

Now for a necessary condition to let acceleration anomalies improve upon the performance of a parallel branch and bound algorithm whose elimination rule involves only feasibility and lower bound tests. The only way an acceleration anomaly can improve upon the performance of a parallel branch and bound algorithm is by eliminating some of the basic subproblems instead of branching from them.

Theorem 5:
The performance of a parallel branch and bound algorithm which uses an elimination rule involving only feasibility and lower bound tests and a heuristic function which is injective, non misleading, and consistent with the lower bound function used, cannot be improved upon by acceleration anomalies.

Proof:
Let $P_0, P_1, \ldots, P_n$ be the series of basic subproblems as branched from by the corresponding sequential algorithm. Because the heuristic function is injective and non misleading, this series is strictly ordered according to increasing heuristic value.
The fact that the heuristic function is injective and consistent with the lower bound function implies that all lower bounds are different. Next to that, it implies that the series $P_0, P_1, \ldots, P_n$ is also ordered according to increasing lower bound value. Therefore the lower bound of basic subproblem P_n must be greater than the lower bounds of all other basic subproblems P_i, $i = 1, \ldots, n-1$.
The fact that P_n has been branched from by the corresponding sequential algorithm implies that this subproblem cannot be eliminated by a lower bound test. Therefore the lower bound of P_n must be smaller than the value of the optimal solution. In turn this implies that the lower bounds of all other basic subproblems are smaller than the value of the optimal solution. Therefore no basic subproblem can be eliminated by a lower bound test. Hence, the execution cannot be improved upon. □

Now it is easy to formulate a necessary condition for acceleration anomalies to occur: the heuristic function is not injective, or the heuristic function is not non misleading, or the heuristic function is not consistent with the lower bound function.

4. CONCLUSIONS AND ONGOING RESEARCH

In this paper we presented a classification of parallel branch and bound algorithms, and investigated the performance of parallel branch and bound algorithms which attain perfect interaction and use as the unit of work the branching from a single subproblem. We presented sufficient conditions to prevent deceleration anomalies from degrading the performance of these parallel branch and bound algorithm as well as a necessary condition to allow acceleration anomalies to improve upon the performance.
In real life perfect interaction is impossible to obtain. In a forthcoming paper we will investigate the consequences for the performance of a parallel branch and bound algorithm of dropping some features of the perfect interaction.

Acknowledgments
The author would like to thank Arie de Bruin, Gerard Kindervater, and Alexander Rinnooy Kan for their help during preparation of this paper.

References

M. Grötschel (1980). On the symmetric travelling salesman problem: Solution of a 120-city problem. *Mathematical Programming Study, No. 12.*

T. Ibaraki (1976). Theoretical comparisons of search strategies in Branch-and-bound algorithms. *Int'l Jr. of Comp. and Info. Sci., Vol. 5, No. 4.*

T. Ibaraki (1977). On the computational efficiency of branch-and-bound algorithms. *Journal of the Operations Research Society of Japan, Vol. 20, No 1.*

G.A.P. Kindervater (1989). *Exercises in Parallel Combinatorial Computing.* Thesis, Centre for Mathematics and Computer Science, Amsterdam.

G.A.P. Kindervater, H.W.J.M. Trienekens (1988). Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research, Vol. 33, No. 1.*

D.E. Knuth (1973). *Fundamental Algorithms: The Art of Computer Programming 1.* Addison-Wesley, Reading, Mass.

T.H. Lai, S. Sahni (1984). Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM, Vol. 27, No. 6.*

G. Li, B.W. Wah (1984). *Computational efficiency of parallel approximate branch-and-bound algorithm.* School of Electrical Engineering, Purdue University, Report TR-EE 84-6.

L.G. Mitten (1970). Branch-and-bound Methods: General Formulation and Properties. *Operations Research 18.*

J.F. Shapiro (1979). A survey of Lagrangean techniques for discrete optimization. *Annals of Discrete Mathematics 5.*

H.W.J.M. Trienekens (1989). *Computational Experiments with an Asynchronous Parallel Branch and Bound Algorithm.* Report EUR-CS-89-02, Department of Computer Science, Erasmus University, Rotterdam.