# Computational Experiments with an Asynchronous Parallel Branch and Bound Algorithm

*Harry W.J.M. Trienekens*

Erasmus University Rotterdam

*ABSTRACT*

In this paper we present an asynchronous branch and bound algorithm for execution on an MIMD system, state sufficient conditions to prevent the parallelism from degrading the performance of this algorithm, and investigate the consequences of having the algorithm executed by nonhomogeneous processing elements.
We introduce the notions of perfect parallel time and achieved efficiency to empirically measure the effects of parallelism, because the traditional notions of speedup and processor utilization are not adequate for fully characterizing the actual execution of an asynchronous parallel branch and bound algorithm.
Finally we present some computational results obtained for the symmetric traveling salesman problem.

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

*Key Words & Phrases*: parallel computer, MIMD, loosely coupled system, branch and bound, traveling salesman problem, nondeterminism, asynchronicity.

*Note*: This paper will be submitted for publication elsewhere.

## 1. INTRODUCTION

Enumerative methods, especially branch and bound algorithms, are essentially the only tools available to solve hard combinatorial problems to optimality. The worst case exponential behaviour of these algorithms [Garey & Johnson 1979], and the anomalies that can happen during execution of these algorithms [Ibaraki 1977], are well known facts of life. In theory, these hard problems can be solved by a parallel algorithm in polynomial time with an exponential number of processors [Goldschlager 1982]. In practice however, it is impossible to attain this result because, given a parallel computer system, for problem instances that are sufficiently large, the number of processors needed is greater than the number of processors available. Hence, small problem instances might be solvable in polynomial time, but the solving of large problem instances can only be speeded up by a constant factor.
In [Trienekens 1989] we presented a theoretical analysis of parallel branch and bound algorithms. That paper contains a taxonomy of parallel branch and bound algorithms, an investigation into the causes of anomalies during the execution of parallel branch and bound algorithms, and sufficient conditions to prevent deceleration anomalies from degrading the performance of a parallel branch and bound algorithm.
In this paper we discuss a particular asynchronous parallel branch and bound algorithm. In section 2 we introduce this particular algorithm. In section 3 we propose a way to measure the effects of the use of parallelism, and look at the anomalies that can occur during parallel execution. We present sufficient conditions to prevent deceleration anomalies from degrading the performance of our parallel branch and bound algorithm. In section 4 we report on our experiments solving symmetric traveling salesman problems. For

Report EUR-CS-89-02
Erasmus University Rotterdam, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

a limited number of processors it turned out to be possible to get a nearly perfect speedup.

## 2. A PARALLEL BRANCH AND BOUND ALGORITHM

### 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.
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.

A branch and bound process generates a *search tree*. The root of this tree $P_0$ corresponds to the original problem, whereas all other nodes $P_i$ correspond to subproblems generated by decomposition. If subproblem $j$ is generated by decomposition from subproblem $i$, there is an arc between $P_i$ and $P_j$, the two corresponding nodes in the search tree.

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. The subproblem in the active set with the smallest heuristic value is extracted from the active set, and decomposed into smaller subproblems using the branching rule. (If there are many such subproblems, the tie is broken arbitrarily). 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., an optimal solution to the subproblem is 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.

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 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.

### 2.2. The Boulder Algorithm

For our research, we are interested in the consequences of the 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. For a complete description of how to parallelize branch and bound algorithms, and for a taxonomy of parallel branch and bound algorithms, we refer to [Trienekens 1989].

Our 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 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 parallel 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, and returning the subproblems thus generated, the slave requests new work from the master by notifying the master that it has become idle.
We chose for an asynchronous algorithm (i.e., if a slave process completes working upon a subproblem, it can start working upon a new subproblem immediately without waiting for all the other slaves processes to complete their subproblems) because we did not want to waste computing power in case the effort needed for branching from a subproblem depends on the particular subproblem on hand. Apart from that, because we intended to run our algorithm on a loosely coupled system where communications are slow and tedious, we did not want to spend time on communications to synchronize. Note that due to the asynchronicity the algorithm is nondeterministic (i.e., consecutive executions of the algorithm solving the same problem instance may yield different solutions) [Trienekens 1989].
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.

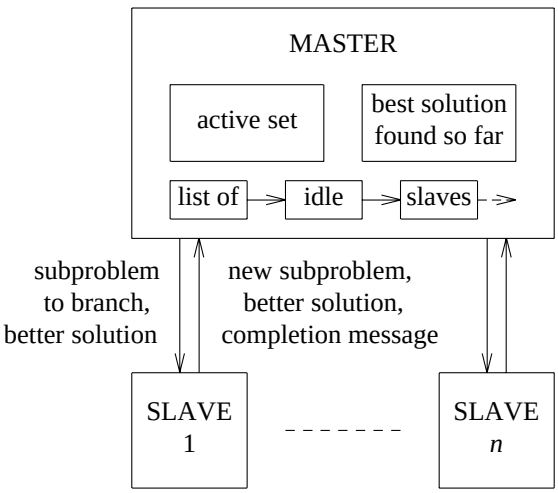


Figure 2.1. The Boulder algorithm

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.

The master process is not strictly necessary. It is also possible to let the slave processes themselves take all decisions by storing the active set and the knowledge obtained so far in some kind of common memory. This however introduces all kinds of problems if several slaves try to access and update this information at the same time. These problems are problems around concurrency control, and can be solved in various ways [Andrews & Schneider 1983].

In our algorithm we do not make any assumptions about the computing power of the various slave processes. We only postulate that a subproblem given to a slave will be branched from eventually. If all slaves are not equally powerful, the queue of idle slaves becomes a priority queue in which the slaves are ordered according to decreasing computing power. As a subproblem becomes available, it is given to the most powerful idle slave.

Because parallel computer systems tend to be complex entities, there is always the possibility of one or more processing elements or communication channels suddenly failing. It is possible to build a certain level of fault tolerance and correction into our parallel branch and bound algorithm. By letting the master keep track of the orders given to the various slaves, it is possible for him to give the same order to another slave if he detects that a slave has gone dead. However, a failure in the master process is still lethal.

To execute the Boulder algorithm we would like to have a system in which it is possible for a process to send a message to another process without these two processes becoming synchronized and with the capability to buffer messages. A process must be able to continue its computations immediately after having sent its message, without having to wait for the message to arrive. This type of communication is to be preferred because, for example, each time a slave obtains new knowledge it has to send this knowledge to the master. Immediately thereafter the slave wants to continue with the rest of its work. There is no need (or use) for a slave to wait until the master has received the message. The system must buffer the messages because it is possible that a process sends a second message before the other process has processed the first one.
If the system used does not provide the interprocess communication described above, it is still capable of executing the algorithm, although computing power will be lost due to the unnecessary synchronizations.

## 3. COMPLEXITY, SPEEDUP, AND ANOMALIES

In order to define the time complexity of sequential branch and bound algorithms some unit of time, i.e., some granularity of time, has to be defined with which this complexity can be measured. Defining a granularity involves abstracting from the precise work performed in executing a branch and bound algorithm.
We can define three kinds of granularities here. We talk about a *coarse granularity* is when the unit of work is defined in terms of the problem to be solved, for example, as the branching from a single subproblem or the computing of a lower bound. A *fine granularity* is when the unit of work is defined in terms of processor operations, for example, the number of machine instructions performed while executing the algorithm. A *very fine granularity* is when the complexity is defined in terms of the time needed to execute the instructions of the branch and bound algorithm on a given machine. In essence a coarse granularity defines the complexity in terms of the search tree generated, whereas a fine granularity defines the complexity in terms of the effort involved in traversing the search tree.
Coarse and fine granularities are roughly spoken equivalent if the algorithm is executed on a single processing element. Due the the fact that all the work to be performed is executed by a single processing element, there is a straightforward relation between the number of machine instructions and the time involved in executing them on one hand, and a single coarse unit of work on the other hand. However, if there are several processing elements of varying computing power, this relation might be lost. For example, one of the processing elements could be an order of magnitude more powerful than the other processing elements. Hence, coarse and fine grain granularities need not be equivalent any more once parallelism is introduced in a branch and bound algorithm.

The overall complexity of parallel branch and bound algorithms cannot be defined solely in terms of the work involved in executing the various steps of the algorithm, i.e., the *computation complexity*. The complexity also has to account for the effort and time involved in the interactions between the various processes, i.e., the *communication complexity*. The ultimate complexity of the execution of a parallel algorithm (i.e., the total time involved in executing the algorithm) is a function of the computation and communication complexities of the algorithm, as well as of the characteristics of the particular processing elements the processes of the algorithm are executing on.
We can make a few remarks with regard to the complexity of parallel algorithms. Coarse granularity completely neglects the effort involved in the interactions between processes. Using this kind of granularity one only looks at the problem to be solved, and from the problem in isolation nothing can be inferred about

the parallelism used and the interaction between the various processes. Fine granularity takes part of the effort and time involved in interacting into account. It accounts for the instructions involved in interacting, but it does not account for the waiting until the actual interaction occurs. For example, if a process sends a message to another process and receives an answer, than fine granularity accounts for the effort involved in sending and receiving the message, but not for the waiting time in case the other process does not react immediately. Only very fine granularity accounts for the full consequences of the interactions.
Therefore coarse grained complexity notions for asynchronous parallel branch and bound algorithms are adequate only within certain limits. If the characteristics of the processing elements executing the processes of the parallel algorithm vary heavily, the relation between the work performed and the time needed to perform the work can become very processing elementing element dependent. Hence such a complexity notion is of little use for predicting the time needed for executing the algorithm. Note that this remark only holds for asynchronous algorithms. For synchronous algorithms it holds that, due to the synchronizations, the effort and/or time involved in executing the work in between two synchronizations is equal to the effort and/or time needed by the process running on the least powerful processing element.

The effects of the use of parallelism on the time needed for executing a synchronous parallel branch and bound algorithm have been described using the notions of speedup and processor utilization, and by the way these notions change as a function of the number of processing elements [Kindervater & Lenstra 1988]. The *speedup* measures the reduction in total execution time due to the parallelism, and is defined as the number of steps performed by a single processing element to execute the best sequential algorithm, divided by the number of steps performed by a parallel computer to execute the parallel algorithm. The *processor utilization* provides information on the quality of this speedup compared to an ideal speedup, and is defined as the speedup divided by the number of processing elements used.

However, the notions of speedup and processor utilization described above are not adequate for characterizing the execution of asynchronous parallel algorithms. Firstly, they do not take into account the time needed for synchronizing. Secondly, they postulate that the time needed for a single communication is a constant. In reality, these times not only depend on the number of communications and/or synchronizations, but also on the particular parallel computer system the algorithm is executed on. Thirdly, these notions postulate that all processing elements used are identical, whereas for many parallel computer systems, especially loosely coupled systems, the processing elements are different. For a synchronized parallel algorithm this variation in processing power does not matter because the time needed for parallel execution is the time needed by the least powerful processing element. For asynchronous algorithms however, there need not be a direct link between the work performed by the various processing elements.

To be able to describe the execution of asynchronous parallel algorithms the above described notions have to be generalized. We therefore introduce the notions of perfect parallel time and achieved efficiency. A parallelization is *perfect* if the use of parallelism does not influence the work to be done, and if the work to be done is divisible among the processing elements in such a way that, firstly, each processing element needs the same amount of time to complete its part of the work, and, secondly, all parts are independent of each other. More powerful processing elements thus will take a larger share of the work, whereas weak processing elements will take a smaller share. Note that the first assumption implies that interactions between processing elements do not take time.
The *perfect parallel time* is the time needed for the perfect parallel execution of an algorithm. Let $W$ be the total number of units of work to be performed by the processing elements during the perfect parallel execution. Because this number is independent of the number of processing elements used, $W$ is equal to the number of units of work performed by the *corresponding sequential algorithm*, i.e., the parallel algorithm using only a single process for branching from subproblems. Let $S_i$ be the speed of processing element $i$, $i = 1, \ldots, N$, i.e., the number of units of work processing element $i$ can perform per unit of time. The perfect parallel time is defined as follows:

$$PPT = W / \sum_{i=1}^{N} S_i. \tag{3.1}$$

The *achieved efficiency* is the quotient of perfect parallel time and the actual time needed for parallel execution. As can be seen from this definition, the achieved efficiency takes the effort and the time involved in communication and synchronization into account.

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 $n$ processing elements. Let $T_1(I)$ denote the amount of time needed to solve this problem instance by the corresponding sequential algorithm executed by processing element 1.
An unsuspecting reader might expect the following relation to hold:

$$T_n(I) = \frac{S_1.T_1(I)}{\sum_{i=0}^{N} S_i} = PPT. \tag{3.2}$$

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

$$PPT \leq T_n(I) < T_1(I). \tag{3.3}$$

This is due to the fact that the assumptions on divisibility, independency, and interaction on which our definition of perfect parallelism is based, are unrealistic in real life. Sometimes it even happens that

$$T_n(I) < PPT \tag{3.4}$$

or

$$T_n(I) \geq T_1(I). \tag{3.5}$$

The anomalies described by formulae 3.3 - 3.5 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 on the same parallel computer system 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 three anomalies states something about what happens if there is a change in the number of processes executing the parallel branch and bound algorithm.
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 branch and bound algorithms, they can be adapted in a straightforward manner to asynchronous branch and bound algorithms. For an example of a fluctuation anomaly we refer to [de Bruin, Rinnooy Kan & Trienekens 1988].

### 3.1. Preventing Deceleration Anomalies

In this section we will state sufficient conditions to prevent deceleration anomalies from degrading the performance of the Boulder algorithm. The already known sufficient conditions for synchronous parallel branch and bound algorithms [Li & Wah 1984] are also sufficient in our case. For a more general analysis of the preventing of anomalies, and for the proofs of the lemmas, we refer to [Trienekens 1989].

First we need some 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 process of extracting a subproblem to branch from from the active set, branching from this subproblem, adding the descendants created to the active set, and pruning the active set, is called *working upon a subproblem*.

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

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

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.,

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

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 algorithm can differ from the knowledge generated at the corresponding point during the 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.
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.
The first condition 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 then some of the basic subproblems.

**Lemma 1**:
If during execution of the Boulder algorithm 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.

**Lemma 2**:
During execution of a Boulder algorithm which uses an elimination rule involving only lower bound tests and a heuristic function which is injective and non misleading, at each point in time some process will be working upon the active basic subproblem with the smallest heuristic value.

For the proofs of these lemmas we refer to [Trienekens 1989]. As can be seen in that paper, these lemmas are valid in a much more general setting.
The first lemma states an upper bound on the work to be done while executing the Boulder algorithm. The second lemma together with the fact that a lower bound function is non misleading, state that the branching from basic subproblems always continues. Together these two lemma's imply that the execution of the Boulder algorithm will not suffer from deceleration anomalies.

Now for an upper bound on the time complexity of the Boulder algorithm. Let $T_p(I)$ be the time needed for solving problem instance $I$ on a given parallel computer system with the Boulder algorithm. 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 generated by the corresponding sequential algorithm. Let $m$ be the number of slave processes the algorithm is executed by. Finally, let $b$ be the time needed by a slave process to interact with the master process.

We assume that the master can add a subproblem to the active set and prune this set (i.e., removing all subproblems with a higher or equal lower bound), or extract a subproblem from the active set, in constant time $c$. The master serves the requests of the slaves using a first come, first served policy.

**Theorem**:
For the execution of a Boulder algorithm which uses an elimination rule involving only lower bound tests and a heuristic function which is injective and non misleading, it holds that

$$T_p(I) \leq T_s(I) + 2 \,.\, N \,.\, (b + (m-1).c).$$

**Proof**:
Lemma 1 states that once there are no more basic subproblems, the problem instance has been solved. Lemma 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 term $2.N.(b+(m-1).c)$ stems from the fact that apart from the computations on the branchings, there is also work involved in interactions between the master and the slaves. The additional time needed by a slave process for subtracting a subproblem from the active set is the sum of the time needed for the interaction with the master process and the waiting time in case of the master process first having to server requests of other slaves. The same holds for the addition of the descendants created by branching and the subsequent pruning of the active set. Because there are m slave processes, there are at the most m-1 slave processes already trying to interact with the master process. The requests by these processes takes at the most $(m-1).c$ time. Because these interactions are not needed by the corresponding sequential algorithm, we have to account for it. □

Note that neither the theorem nor lemma 2 state anything about the total number of subproblems branched from.

## 4. COMPUTATIONAL EXPERIMENTS

As mentioned before, for our research we were amongst others interested in the consequences of the branching from several subproblems in parallel with the best possible use of the knowledge generated so far. By ‘best possible use of the knowledge generated so far’ we mean that during execution of a parallel branch and bound algorithm we try to branch only from basic subproblems. The branching from non basic subproblems is considered to be superfluous work which unnecessarily slows down the execution. With this in mind, we developed the Boulder algorithm described in section 2.2.

We executed the Boulder algorithm using the *Distributed Processing Utilities Package* (DPUP) of the Department of Computer Science of the University of Colorado at Boulder, Colorado, United States [Gardner et al. 1986]. This department owns several Vaxes, Pyramids, and Suns. DPUP is a distributed programming tool box which allows the user to use the machines on the net as a loosely coupled system
DPUP enables a process to create a remote process and establish a communication link with it. The machine on which the remote process is created can be determined either by the creating process or by DPUP itself. In the latter case, DPUP starts the remote process on the machine with the lowest load in the network. The machine the remote process is created on can be the same as the machine the original process is running on. Of course DPUP enables a process to communicate with a remote process created previously by this process. DPUP also enables a process to kill a remote process it started, and to discard the corresponding communication link. Finally, DPUP enables different remote processes to communicate directly with each other. While communicating, the sending process does not become synchronized with the receiving process. DPUP stores the message in an internal buffer and the sending process can continue directly after having sent the message. The receiving process can empty the buffer as it is ready to do so. It is possible to interrupt a remote process. The fact that DPUP enables a process to start another process is very elegant because it allows a user to initiate the execution of a parallel program by just starting a single process (which in turn will start all other processes).
Because the machines are connected through an Ethernet, only one process at a time can send a message. Therefore all communications have to be handled sequentially. Fortunately an Ethernet has a very high

bandwidth, i.e., a very high communication capacity. So, as long as there are not too many big messages or too many processes, the probability of two messages colliding or of the Ethernet becoming a bottleneck is small.

We tested the Boulder algorithm by solving some random generated instances of the symmetric traveling salesman problem.
Let $G = (V,E)$ be an undirected graph with node set $V = \{1, \ldots, N\}$ and edge set $E$. Each edge $e_{i,j} \in E$ has weight $c_{i,j}$. The *symmetric traveling salesman problem* is to find a hamiltonian circuit on $G$ of minimum total weight.

The sequential branch and bound algorithm we started from has been developed by Jonker & Volgenant. We will give a brief description of this algorithm. For a more detailed description we refer to [Jonker & Volgenant 1982].
The lower bound function used by the algorithm is based on a *minimal 1-tree relaxation*. A 1-tree of G consists of a spanning tree on the nodes $V\backslash\{1\}$ combined with two edges incident to node 1. A minimal 1-tree is a 1-tree of minimum total weight. Clearly, a hamiltonian circuit is a special case of a 1-tree. Therefore the weight of a minimal 1-tree is a lower bound on the weight of a hamiltonian circuit, and thus on the weight of the minimal hamiltonian circuit. Computing a minimal spanning tree is very easy to do using the greedy algorithm due to Kruskal [1956]. The major part of the work involved in executing this algorithm is the sorting of the edges according to increasing weight. Given a minimal spanning tree, a minimal 1-tree can be constructed by adding to the spanning tree the two edges incident to node 1 with minimal weight.
Obviously a weight transformation $\hat{c}_{i,j} = c_{i,j} + \pi_i + \pi_j$ has no influence on the minimal hamiltonian circuit. Because in a hamiltonian circuit there are exactly two edges incident to each node, the above transformation will add a constant to the weight of a hamiltonian circuit, and therefore will not change the minimal hamiltonian circuit. However, the minimal 1-tree will generally be changed. Using the above described weight transformation as Lagrange multipliers, the quality of the lower bound can be improved upon by a heuristic iterative process which tries to maximize the weight of a minimal 1-tree.
The branching rule decomposes each problem into three subproblems by requiring and/or forbidding some of the edges of $E$ to be in the 1-tree. The new required and forbidden edges are chosen in such a way that they try to break the cycle in the minimal 1-tree, thereby forcing the 1-tree towards a hamiltonian circuit. The choice of these edges therefore depends on the 1-tree generated during the lower bound process.

As the heuristic function guiding the selection of the subproblem to branch from next, we chose the lower bound computed on a subproblem, i.e., our algorithm performs a best bound first search. We chose for this selection strategy because, in a sequential branch and bound algorithm, it minimizes the number of subproblems branched from [Fox, Lenstra, Rinnooy Kan & Schrage 1978]. We did not enforce that all heuristic values would be distinct, because due to the use of the Lagrange multiplier technique, the probability of two lower bounds being equal is very small.
As soon as it can be deduced that a subproblem cannot yield a feasible solution, this subproblem is eliminated. For example, if a subproblem contains three or more required or $(m-1)$ or more forbidden edges incident to a particular node (where $m$ is the number of edges incident to this node), then this subproblem does not have a feasible solution.

### 4.1. Implementation

We coded the Boulder algorithm in C [Kernighan & Ritchie 1978], the same language DPUP is written in. This way it was easy to link the DPUP routines into the program.
The program is organized in such a way that the user only has to start the master process. The master process in turn starts and kills all slave processes.
As mentioned earlier, DPUP provides the user the opportunity to run several processes in parallel, but some of these processes might be running on the same processing element. Running more than one process on a single processing element introduces all kinds of overhead like process control, swapping etc. Therefore we intended to run all our processes on different processing elements. However, we had to make an exception for the master process. It turned out that this process had hardly any work to do, so it would have been a waste to dedicate a processing element to it. We ran the master process together with one of the slave processes on the same processing element. However, because the work to be done by the master is more

important than the work to be done by the slave, the master process was given a higher priority.

While implementing the Boulder algorithm, and debugging the program, we encountered some problems caused by the nondeterminism of the algorithm and the particular computer system used for executing the algorithm.

The first problem was the nondeterminism of the algorithm. Due to this nondeterminism, consecutive executions of the program could be completely different. Therefore it could be very hard to reproduce the errors occurred in a given execution. The problems of debugging a nondeterministic program were partly bypassed by first executing the program using only a single slave process. Because the interactions between the master and a particular slave are well defined, such a program is deterministic. (The nondeterminism is introduced only if several slaves want to interact simultaneously with the master.) Only after this program executed flawlessly, we started to use more slave processes.
Even using the above described method of program development, debugging a parallel program turned out to be very cumbersome. The main obstacle was that it was very hard to see what exactly happened inside a slave process. Because a slave process is started by the master process and not by the user, it could not be run under control of a debugger.

The second problem was due to the floating point arithmetic of the machines used. While implementing the Boulder algorithm and debugging the program, it turned out that not all the machines on the net performed their floating point arithmetic in exactly the same way. The outcome of a floating point operation depended on the machine on which the operation was performed. The differences between the various machines were minor (in the same order as rounding errors). However, these minor differences could accumulate and influence the way a subproblem was decomposed into smaller subproblems. (The way a subproblem is decomposed depends on the minimal 1-tree generated, which in turn depends on the weights of the various edges. Due to the fact that these weights were computed using floating point arithmetic, they differed on the various machines. These different weights could result in a different ordering of the edges, and hence in different 1-trees generated.) Therefore the subproblems generated by decomposition did not only depend on the branching rule but also on the machine the slave process branching from this subproblem was running on.
These different decompositions did not affect the correctness of our program. However, they made it very difficult to compare consecutive executions of the parallel algorithm, or to compare the execution of the parallel algorithm with the execution of the corresponding sequential algorithm. Because we were interested in studying the effects of parallelism, we decided to eliminate this variation in decomposition into subproblems. The only way to do this was by changing all our floating point arithmetic to integer arithmetic. Although this change decreased the accuracy of the Lagrange multipliers and thereby the quality of the 1-trees generated, it speeded up the overall execution of the algorithm, especially if the algorithm was executed on the Suns. This speedup was due to the fact that integer arithmetic is much easier to perform than floating point arithmetic. The extraordinary speedup on the Suns was due to the fact that those machines did not perform their floating point operations in hardware but had to simulate them in software.

The third problem had to do with the communication between processes running on machines of different kind. In essence, communications through DPUP correspond to shipping bit patterns from one process to another. Therefore processes on different types of machines can communicate using DPUP if they use the same interpretation of bit patterns. If two processes use a different internal representation of bit patterns, one of the two processes has to convert the representation of the message to the representation used by the other process. Currently the user himself has to arrange for this conversion process.
It turned out that the various C compilers running on the Pyramids and the Suns did not all use the same internal representation of records, i.e., data structures containing various fields of nonhomogeneous data. The compilers differed in the way they packed the various fields in a record. By adding dummy fields to a record, it could be easily arranged that all compilers used the same representation.

The last problem was caused by fact that there were other users on the system. It turned out that a parallel program executed using DPUP was heavily influenced by other activities occurring in the computer system. The fact that a machine is executing other processes apart from the processes of the parallel program amounts to a decrease in computing power of the processes running on this machine. Next to that, if there is much traffic on the Ethernet connecting the various machines, the communications between the

various processes of the program can be interfered with. The real problem in this respect is that all these circumstances cannot be controlled by the program. Therefore, each time a parallel program is executed, the environment differs, and the effects caused by the different environment mingle with the effects caused by the parallelism.
The only way to acquire reliable test results was by using a dedicated system, i.e., a system whose machines are doing nothing besides the execution of the parallel program. Because we were running our tests between Christmas and New Year we could obtain exclusive access to all machines during the night.

### 4.2. Computational Results

We tested the Boulder algorithm by applying it to two sets of randomly generated symmetric traveling salesman problems.
To study the effects of the number of processes used in parallel and of processes of varying computing power, we ran our program on different combinations of machines. Each machine, except the most powerful machine, executed a single slave process. The most powerful machine executed next to a slave process also the master process. We started by using two Pyramids, and than repeatedly added an additional Pyramid until in the end we used 5 Pyramids. The computing power ratio of these Pyramids was approximately 1.00 : 0.96 : 0.96 : 0.80 : 0.62 (depending on the size of the problem instance to be solved). The Pyramids were added in order of decreasing computing power.
Finally we included two Sun-2's in our tests to study the effects of adding a weak processing element to the system. The computing power of a Sun-2 is approximately 4 times less than the power of the most powerful Pyramid. In these experiments, the master process was executed by one of the Sun-2's, whereas each Pyramid and the other Sun-2 were each executing a single slave process.

The first set of test problems consisted of euclidean problems. Using a two dimensional uniform distribution we generated points in a two dimensional space. The weight of an edge between two nodes is equal to the euclidean distance in the plane between the two corresponding points. The second set of test problems consisted of random problems, in which all weights were drawn at random from a uniform distribution.
The random set consisted of five instances of 50 nodes, two instances of 75 and two instances of 100 nodes, whereas the euclidean set consisted of five instances of 50 nodes and a single instance of 75 nodes. All test problems (except the 75 node euclidean problem) were relative easy to solve. We have chosen to use these easy test problems because we were primarily interested in the effects and the consequences of the use of parallelism, and not in solving problem instances as big as possible as fast as possible. We wanted to study the behavior of a parallel branch and bound algorithm with increasing number of processes. We were especially interested in whether we could keep all processes busy all the time.

As mentioned earlier, asynchronous parallel branch and bound algorithms are nondeterministic. While solving our test problems, the nondeterminism only occasionally resulted in different answers yielded for different executions. The division of the work among the processes however varied heavily for different runs, and thus the time needed for execution of our parallel program. In order to get reliable results, we had to make several test runs for each problem instance and average the results.
The small variety in answers yielded is due to the fact that the cardinality of the set of optimal feasible solutions of a traveling salesman problem tends to be small. Apart from that, if a problem instance has several optimal solutions, the effort involved in generating these solutions must be nearly comparable, i.e., the effort involved in traveling in the search tree from the root to these problems must be comparable.

Figure 4.1 shows the average achieved efficiency and the average number of subproblems branched from as a function of the number of processes used. The number of subproblems the corresponding sequential algorithm branches from is shown underneath the problem name. Remember that due to the fact that not all machines used are equally powerful, there is no direct link between the achieved efficiency and the total time needed for executing the parallel program. In most cases, this time decreased as the number of processes used increased. In a few cases, this time suffered from fluctuation anomalies caused by the varying computing power of the machines used to execute our processes. We will return to these anomalies in our experiments with the Sun-2's.
The results show that apparently there exists only a limited amount of parallelism within a problem

| problem | bounding root node and branching | | | | branching only | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 pyramids | 3 pyramids | 4 pyramids | 5 pyramids | 2 pyramids | 3 pyramids | 4 pyramids | 5 pyramids |
| *e50a* | 0.93 | 0.90 | 0.68 | 0.57 | 1.00 | 1.11 | 0.85 | 0.72 |
| (19) | (20.0) | (19.0) | (20.0) | (21.0) | (20.0) | (19.0) | (20.0) | (21.0) |
| *e50b* | 0.88 | 0.65 | 0.46 | 0.39 | 0.98 | 0.75 | 0.54 | 0.44 |
| (11) | (11.0) | (14.5) | (18.8) | (21.3) | (11.0) | (14.5) | (18.8) | (21.3) |
| *e50c* | 0.84 | 0.60 | 0.42 | 0.34 | 0.91 | 0.68 | 0.47 | 0.37 |
| (11) | (12.0) | (16.0) | (21.0) | (23.0) | (12.0) | (16.0) | (21.0) | (23.0) |
| *e50d* | 0.88 | 0.68 | 0.52 | 0.45 | 0.95 | 0.76 | 0.60 | 0.53 |
| (16) | (18.0) | (21.0) | (24.3) | (26.3) | (18.0) | (21.0) | (24.3) | (26.3) |
| *e50e* | 0.89 | 0.72 | 0.50 | 0.41 | 0.98 | 0.86 | 0.60 | 0.48 |
| (16) | (17.0) | (17.0) | (18.0) | (18.0) | (17.0) | (17.0) | (18.0) | (18.0) |
| *e75b* | 0.97 | 0.94 | 0.95 | 0.93 | 0.99 | 0.97 | 1.00 | 0.99 |
| (260) | (260.0) | (260.0) | (260.0) | (260.0) | (260.0) | (260.0) | (260.0) | (260.0) |
| *r50a* | 0.97 | 0.85 | 0.68 | 0.57 | 1.06 | 1.01 | 0.86 | 0.72 |
| (22) | (22.0) | (22.0) | (22.0) | (23.0) | (22.0) | (22.0) | (22.0) | (23.0) |
| *r50b* | 1.02 | 0.92 | 0.80 | 0.72 | 1.09 | 1.04 | 0.95 | 0.91 |
| (33) | (33.0) | (34.0) | (33.0) | (33.5) | (33.0) | (34.0) | (33.0) | (33.5) |
| *r50c* | 0.85 | 0.61 | 0.46 | 0.39 | 0.95 | 0.71 | 0.55 | 0.46 |
| (11) | (11.0) | (11.5) | (12.8) | (14.3) | (11.0) | (11.5) | (12.8) | (14.3) |
| *r50d* | 1.05 | 0.92 | 0.76 | 0.68 | 1.12 | 1.03 | 0.89 | 0.81 |
| (36) | (36.0) | (36.3) | (37.0) | (37.3) | (36.0) | (36.3) | (37.0) | (37.3) |
| *r50e* | 1.09 | 0.98 | 0.84 | 0.77 | 1.14 | 1.07 | 0.94 | 0.88 |
| (62) | (62.0) | (62.0) | (62.3) | (62.3) | (62.0) | (62.0) | (62.3) | (62.3) |
| *r75b* | 0.82 | 0.72 | 0.71 | 0.63 | 0.90 | 0.84 | 0.90 | 0.81 |
| (34) | (34.0) | (34.0) | (37.0) | (39.0) | (34.0) | (34.0) | (37.0) | (39.0) |
| *r75c* | 0.90 | 0.83 | 0.78 | 0.73 | 0.96 | 0.93 | 0.92 | 0.89 |
| (64) | (64.0) | (64.3) | (65.0) | (65.3) | (64.0) | (64.3) | (65.0) | (65.3) |
| *r100a* | 0.86 | 0.74 | 0.66 | 0.60 | 0.97 | 0.90 | 0.88 | 0.82 |
| (37) | (38.0) | (39.0) | (39.5) | (41.0) | (38.0) | (39.0) | (39.5) | (41.0) |
| *r100c* | 0.84 | 0.65 | 0.54 | 0.48 | 1.01 | 0.85 | 0.75 | 0.68 |
| (22) | (22.0) | (23.0) | (24.0) | (25.0) | (22.0) | (23.0) | (24.0) | (25.0) |

average achieved efficiency
(average nr of subproblems branched)

Figure 4.1

instance. This can be derived from the fact that the achieved efficiency drastically decreases as soon as the number of processes used exceeds a certain maximum number (which depends on the particular problem instance on hand). Until that moment, there is enough parallelism in the problem instance to keep all processes busy most of the time.

The Boulder algorithm starts by computing a lower bound to the original problem. These computations are performed by a single process whilst all other processes are idle. As can be easily deduced, this has bad effects on the achieved efficiency. If we discount for the time needed to compute the lower bound to the original problem, the achieved efficiencies are much higher. The discounted achieved efficiencies are also shown in figure 4.1.

Some of our test runs show an achieved efficiency of more than 100 %, whereas the work done during parallel execution exactly equaled the work done during sequential execution (i.e., the search trees generated were completely identical). The only way we can explain this unexpected outcome is by looking at the various tricks a computer uses to speed up the execution of a program. Firstly, a compiler tries to generate code that is as efficient as possible. This is easier to do for smaller programs than for bigger programs. The parallel program consists of a set of smaller programs whereas the sequential program is one big program. Secondly, the amount of memory needed by a slave process for storing its data is smaller than the amount of memory used by the sequential program. Therefore a greater part of the slave process fits into the cache (a kind of very fast memory, which is only limitedly available on a computer). Finally, a slave process can store all its data in static variables, whereas the sequential program has to store its data in dynamic variables. Again the slave process can be executed faster because accessing static variables is faster than accessing dynamic variables.

The addition of a slave process running on a Sun-2, even though increasing the power of the system as a whole, needed not decrease the time needed for executing a parallel program. If a slave process on a Sun-2 was added, the execution of a program normally ended with all other slave processes being idle and waiting for the slave process running on the Sun-2 to complete its last task. The speedup caused by this greater computing power could be completely dominated by the waiting in the end, and execution could take a longer time even though the work to be done was completely equal (i.e., identical search trees were generated). Especially problem instances with a small amount of intrinsic parallelism suffered from this increase in execution time.
Sometimes even the number of subproblems branched from drastically increased. This increase can be explained by assuming that the least powerful slave process was branching from a basic subproblem at a time that there were no more basic subproblems available for branching from by the more powerful slave processes which just completed branching from their subproblem. Hence these other slave processes started branching from non basic subproblems which would have been eliminated otherwise. As the least powerful slave process completed branching from its subproblem, the subproblems thus generated were given to the more powerful slave processes to branch from, and the execution left the wrong track.
Figures 4.2 and 4.3 show minimum and maximum execution times and the number of subproblems branched from while solving two specific test problems as a function of the machines used. These figures are characteristic for problem instances with a great and a small amount of intrinsic parallelism. Figure 4.2 displays information about problem *e75b*, a huge problem with lots of intrinsic parallelism. Therefore most of the time there was a basic subproblem available for an idle slave process running on a Pyramid and the speedup at the start of the execution dominated the waiting at the end. Figure 4.3 displays data about problem *e50c*, a problem with limited intrinsic parallelism. The waiting time at the end of the execution completely dominated the speedup at the start and the slave processes running on Pyramids started branching from non basic subproblems. Next to that, the execution suffered from fluctuation anomalies.

### 4.2.1. Branching in parallel from the original problem

As already mentioned, the Boulder algorithm has one inherent sequential part: a single slave process computes the lower bound to the original problem and branches from this problem. Until this slave process has completed these computations, all other slave processes are idle and thus wasting their computing power.
Computing a lower bound to the original problem tends to be a much heavier task than computing a lower bound to a subproblem generated by branching. In computing a bound to the latter problem often

| slave processes running on | fastest time | nr of nodes | slowest time | nr of nodes |
|---|---|---|---|---|
| 2 pyramids | 29:39 | 260 | 29:49 | 260 |
| 3 pyramids | 20:21 | 260 | 20:24 | 260 |
| 4 pyramids | 15:49 | 260 | 15:51 | 260 |
| 5 pyramids | 13:56 | 260 | 14:03 | 260 |
| 2 pyramids + sun | 26:47 | 260 | 27:02 | 260 |
| 3 pyramids + sun | 18:54 | 260 | 19:02 | 260 |
| 4 pyramids + sun | 15:01 | 260 | 15:06 | 260 |
| 5 pyramids + sun | 13:18 | 260 | 13:45 | 260 |

Figure 4.2. Problem *e75b*

| slave processes running on | fastest time | nr of nodes | slowest time | nr of nodes |
|---|---|---|---|---|
| 2 pyramids | 0:49 | 12 | 0:51 | 12 |
| 3 pyramids | 0:48 | 15 | 0:53 | 14 |
| 4 pyramids | 0:43 | 20 | 0:54 | 17 |
| 5 pyramids | 0:49 | 24 | 0:53 | 24 |
| 2 pyramids + sun | 0:55 | 13 | 1:42 | 25 |
| 3 pyramids + sun | 0:49 | 17 | 1:03 | 19 |
| 4 pyramids + sun | 0:58 | 21 | 1:11 | 29 |
| 5 pyramids + sun | 0:48 | 26 | 1:11 | 27 |

Figure 4.3. Problem *e50e*

knowledge is used which has been obtained while computing the bound on the original problem. A good initial bound tends to lead to good bounds on the subproblems which can also be computed efficiently. Therefore often a more complex lower bound function is used in branching from the original problem.
It might be possible to speed up the execution of the Boulder algorithm by parallelizing the computation of the lower bound to the original problem. However, parallelizing this lower bound computation, although decreasing the computation complexity, increases the communication complexity of the algorithm. Therefore whether such a parallelization is worthwhile depends on the lower bound function used and the particular system which the algorithm is executed on.

As one of our computational experiments we tried to speed up the execution of the Boulder algorithm by parallelizing the computation of the lower bound to the original problem. Most of the effort involved in computing this bound is spent in generating a good 1-tree. This process consists of a main loop, which repeatedly updates the weights of the edges and constructs a minimal spanning tree and a 1-tree. Because the way the weights are updated depends on the 1-tree generated in the previous iteration, the iterations of this main loop are inherently sequential. Therefore the only part fit for parallelizing is the computation of the spanning tree. The greedy algorithm used for generating the minimal spanning tree is difficult to parallelize [Anderson & Mayr 1984]. Because we are not primarily interested in parallelizing spanning tree computations, we tried to speed up this computation by simply parallelizing the sorting of the edges while constructing the spanning tree. We divided the edges into sets and gave each slave process its own set of edges to sort. After that, the master process collected the sorted sets and merged them.
It turned out that the increase in communication complexity completely dominated the decrease in computation complexity. Only when we ran our program solely on the Suns, and still used floating point arithmetic, it turned out to be faster.

### 4.3. CONCLUSION

Our experiments showed that it is possible to get high achieved efficiencies while executing the Boulder algorithm on a loosely coupled system with a limited number of processing elements. If the effort needed for branching from a single subproblem is big, the frequency with which the various slave processes access

the active set is low, and accessing the set is not a bottleneck.
A problem instance contains only a limited amount of parallelism. There exists an upper bound on the number of processes which can be used successfully in parallel to decrease the execution time.
Adding a weak processing element to the system turned out to be dangerous, especially if the intrinsic parallelism in the problem instance was small with respect to the number of processes used. Most of the time such an addition increased the execution time.

**Acknowledgments**
The author would like to thank Bobby Schnabel for his help and for making the computational experiments possible, and Arie de Bruin, Gerard Kindervater, and Alexander Rinnooy Kan for their help during preparation of this paper.

**References**
R. Anderson, E. Mayr (1984). *Parallelism and Greedy algorithms.* Computer Science Dept, Stanford University, Report STAN-CS-84-1003.
G.R. Andrews, F.B. Schneider (1983). Concepts and Notations for concurrent Programming. *Computing Surveys, Vol 15, Nr 1.*
A. de Bruin, A.H.G. Rinnooy Kan, H.W.J.M. Trienekens (1988). A simulation tool for the performance evaluation of parallel branch and bound algorithms. *Mathematical Programming, Vol. 42.*
B.L. Fox, J.K. Lenstra, A.H.G. Rinnooy Kan, L.E. Schrage (1978). Branching from the largest upper bound. Folklore and facts. *European Journal of Operational Research 2.*
T.J. Gardner, I.M. Gerard, C.R. Mowers, E. Nemeth, R.B. Schnabel (1986). *DPUP : A Distributed Processing Utilities Package.* University of Colorado Department of Computer Science Technical Report CU-CS-337-86.
M.R. Garey, D.S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco.
L.M. Goldschlager (1982). A universal connection pattern for parallel computers. *Journal of the ACM, No. 29.*
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.*
R. Jonker, T. Volgenant (1982). A branch and bound algorithm for the symmetric travelling salesman problem based on the 1-tree relaxation. *EJOR (1982).*
B.W. Kernighan, D.M. Ritchie (1978). *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
G.A.P. Kindervater, J.K. Lenstra (1988). Parallel Computing in Combinatorial Optimization. *Annals of Operations Research, No. 14.*
J.B. Kruskal (1956). On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. *Proc. Amer. Math. Soc., No. 7.*
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.*
H.W.J.M. Trienekens (1989). *Parallel branch and bound and anomalies.* Report EUR-CS-89-01, Department of Computer Science, Erasmus University, Rotterdam.