Parallel Branch and Bound on an MIMD System

Harry W.J.M. Trienekens

Erasmus University Rotterdam

ABSTRACT

In this paper we give a classification of parallel branch and bound algorithms and develop a class of asynchronous branch and bound algorithms for execution on an MIMD system.
We develop sufficient conditions to prevent the anomalies that can occur due to the parallelism, the asynchronicity or the nondeterminism, from degrading the performance of the algorithm. Such conditions were known already for the synchronous case. It turns out that these conditions are sufficient for asynchronous algorithms as well. We also investigate the consequences of nonhomogeneous processing elements in a parallel computer system.
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 efficiency are not capable of fully characterizing the actual execution of an asynchronous parallel 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

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.

Report 8640/A
Erasmus University Rotterdam, Econometric Institute
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

As long as it is possible to split a task into 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 speed up by a constant factor.

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 decomposes the problem into smaller subproblems by splitting the solution space into smaller subspaces. Each subproblem thus generated is either solved or proved not to yield the 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 it holds that 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 during execution 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 a 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 remained exactly the same, it was only 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 class of asynchronous parallel branch and bound algorithms. Section 2 gives a brief description of sequential branch and bound. Section 3 gives a brief description of parallel computers and algorithms and proposes a way to measure the effects of the use of parallelism. Section 4 describes how to parallelize branch and bound algorithms and gives a classification of these parallel algorithms. Section 5 is devoted to the investigation of sufficient conditions to prevent the parallelism from degrading the execution of a parallel branch and bound algorithm. It turns out that the conditions already known for the synchronous case are also sufficient for the asynchronous case. Sections 6 through 9 report on our experiments solving symmetric traveling salesman problems. Sections 6 and 7 describe the computer system we used and the algorithm we developed. Section 8 contains the computational results. Finally, section 9 states our conclusions. It turned out to be possible to get a nearly perfect speedup.

2. Branch and Bound

A branch and bound algorithm can be characterized by four rules: 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 the 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.
The elimination rule can consist of three tests for eliminating subproblems. Firstly, 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. Secondly, the *feasibility test*: a subproblem can be eliminated if it can be proven not to have a feasible solution. Finally, the *dominance test*: a subproblem which is dominated by another subproblem can be eliminated. A subproblem is dominated by another subproblem if a positive outcome of the dominance test applied to these subproblems implies that for each feasible solution of the former problem there is at least one feasible solution of the latter problem with a smaller or equal solution value. A dominance relation is required to be reflexive, antisymmetric and transitive and defines a partial ordering on the subproblems [Ibaraki 1977].
The branch and bound process generates a *search tree*. The root of this tree P_0 corresponds to the original problem, whereas all the other nodes P_i correspond to subproblems generated by decomposition. If subproblem j is generated by decomposition from subproblem i, there is an edge between P_i and P_j, the two corresponding nodes in the search tree. The *level* of a node in the search tree is equal to the number of edges between this node and the root (the root is at level 0).

A possible implementation of branch and bound algorithms can be described as follows. An *active subproblem* is a subproblem which is generated and hitherto neither branched from nor eliminated. 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 repeatedly the following steps are 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 a lower bound is calculated. If during computation of this bound the subproblem is not solved, it is added to the active set and the elimination rule is used to prune this set. If the subproblem is solved during computation of the bound, 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. The computations continue until there are no more active subproblems. The work done during the execution can be represented by the search tree generated.
During execution of a branch and bound algorithm knowledge is continually collected about the problem instance to be solved. This knowledge consists amongst others of all subproblems generated, branched from and eliminated, upper bounds on the value of the optimal solution and the feasible solutions 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 on this knowledge.

3. Parallel computers and algorithms

In this section we give a brief description of parallel computers and parallel algorithms and we propose a method to measure the effects of the use of parallelism.

3.1. Parallel Computers

One of the main models of computation is the *control driven model* [Treleaven, Brownbridge & Hopkins 1982]. In this model the user has to explicitly specify the order in which the operations have to be performed as well as which of these operations can be performed in parallel.
The control driven model can be subdivided using the independency of the processing elements as a criterion [Flynn 1966]. In SISD (*Single Instruction stream, Single Data stream*) computers, there is one

processing element which can perform a single operation at a time. In MIMD (*Multiple Instruction stream, Multiple Data stream*) computers, there are several processing elements, performing in parallel different instructions on different data. In SIMD (*Single Instruction stream, Multiple Data stream*) computers all processing elements have to perform the same instruction at the same time on different data, although it is often possible to mask out processing elements. A masked processing element does not store the result of the operation it just performed. SIMD systems tend to consist of a large number of simple processing elements, whereas MIMD systems usually consist of a few powerful ones.

The class of MIMD systems can be further subdivided using as a criterion the way in which the various processing elements are connected and communicate with each other. If all processing elements are connected to and communicate through a common memory, the system is called a *tightly coupled system*. Another possibility is that the processing elements do not share a common memory but are connected through some kind of network and communicate through message passing on this network. Such a system is called a *loosely coupled system*.
The common memory of a tightly coupled system is both its strongest and its weakest point. It allows for easy and fast sharing of information between various processing elements. Communication boils down to simple read/write operations on the common memory and each processing element can communicate directly with all other processing elements. However, if the number of connected processing elements increases, the common memory becomes a bottleneck. Therefore the number of processing elements that a tightly coupled system can consist of is limited. Due to the fact that all processing elements have to be connected to the common memory, tightly coupled systems are built as a whole and tend to be compact, i.e. the processing elements are packed close to each other in one cabinet. It is not possible to add processing elements to the system.
On the other hand, communication in a loosely coupled system is tedious and slow. The passing of messages requires more time than reading or writing on a common memory would do and it is possible that a processing element is not connected directly to the processing element it wants to communicate with. In such a case, the message has to be sent through other processing elements. In contrast to the compactness of a tightly coupled system, the processing elements of a loosely coupled system might be scattered all around. Therefore the physical distance that a message has to travel can be big. Due to the fact that the processing elements use a protocol to communicate on the network, loosely coupled systems can consist of different types of processing elements and it is possible to add additional processing elements to the system. In general the processing elements are complete computers in themselves.

An example of an SIMD system is the ICL-DAP [ICL 1981]. Due to their capability of performing vector operations at very high speed, pipeline computers resemble SIMD systems very much, although strictly spoken the results are delivered sequentially by the pipeline. Examples of pipeline systems are the CRAY-1 and the CYBER-205. An example of a tightly coupled system is the Denelcor HEP. Examples of loosely coupled systems are the Distributed Processing Utilities Package running on the various computers of the University of Colorado at Boulder [Gardner et al. 1986] and the AMETEK hypercube.
A brief overview of commercially available parallel systems can be found in [Dongarra & Duff 1985].

3.2. Parallel Algorithms

A parallel algorithm consists of subtasks that have to be performed. Some of these subtasks can be executed in parallel, but there also might be subtasks which must be executed sequentially. The execution of a subtask is done by a separate process. Important features of a parallel algorithm are the way these processes interact, whether they are synchronized and whether the algorithm is deterministic.
Two processes interact if (part of) the output of one of these processes is (part of) the input of the other process. The way two processes interact can either be completely specified or not. If an interaction between two processes is completely specified, the two processes can communicate only if both are willing to do so. The sender cannot proceed if the receiver is not yet ready to communicate.
During execution of a synchronous algorithm all processes have to synchronize at various points, amongst others before interacting. Synchronizing means that before starting their next subtask, they have to wait for other processes to complete the subtask they are currently working on. While executing an asynchronous algorithm, processes do not have to wait for each other to finish their tasks. Of course it is possible for an asynchronous algorithm to be partly synchronous.

Note that synchronized parallelism need not imply that all processing elements are always executing the same instruction.
An algorithm is deterministic if each time the algorithm is executed on the same input the executions are identical, i.e. the same instructions are performed in the same order. This will be the case if in each execution each decision to be made will have the same outcome. Subsequent executions of such an algorithm will therefore always yield the same output. In nondeterministic algorithms however, the same decision can have different outcomes, for example the outcome of a decision might depend also on certain environmental factors which are not controlled by the algorithm. Therefore subsequent executions of a nondeterministic algorithm can yield different outputs.
Nondeterminism in a tightly coupled system can occur because a process may read a variable from the common memory exactly at the same time that another process wants to write to this variable. In case of an asynchronous algorithm, nothing can be said about the order in which the read and write will occur. Nondeterminism in a loosely coupled system can occur if the next task to be performed by a process depends on the availability of a message from another process. Again, without synchronizing, nothing can be said about what will happen first: the sending (and arriving) of the message or the check whether there is a message.
The work to be done by a parallel algorithm can be split into computations and communications. Therefore the overall complexity of a parallel algorithm can also be split into a computation complexity and a communication complexity.

Some examples of synchronicity, interaction and determinism can be demonstrated from a fixed point computation in parallel. A fixed point of a function $f: R^n \rightarrow R^n$ is a point x for which $f(x) = x$. During a parallel computation of fixed points, each process repeatedly computes a new estimate of a specific coordinate of x and broadcasts this estimate to the other processes. The estimate is based upon estimates for all coordinates of x. Each time new estimates of the other coordinates become available, a process computes a new estimate of its coordinate and sends this new estimate to the other processes. In a synchronous algorithm, a process does not start to compute its next estimate until it has received exactly one new estimate for each coordinate. In an asynchronous algorithm a process can start the next computation on its coordinate as soon as the old computation is finished and at least one new estimate has arrived [Bertsekas 1983].

Synchronous algorithms tend to divide the work in a more rigid way among the processes than asynchronous algorithms do. In a synchronous algorithm this division tends to depend solely on the kind of problem to solve and not on the particular problem instance on hand. Before the execution of the algorithm starts, it is usually known how the work will be divided among the available processes. In an asynchronous algorithm however, the division of the work can also depend on the particular problem instance on hand. The exact division of the work is therefore made at run time.

Avoiding synchronizations has two main advantages. Firstly, it allows for a higher utilization of the processing elements in the case that the tasks to be executed are not of equal size. Secondly, it makes it easier to deal with algorithms in which the work to be done is not completely known beforehand. It is difficult to divide an unknown quantity in an equitable way.

3.3. Complexity and Speedup

One would like to know and measure the effects of the use of parallelism on the time needed for executing an algorithm.
In case of synchronous parallel algorithms these effects can be described using the notions of *speedup* and *efficiency* and by the way these notions change as a function of the number of processing elements used. The speedup measures the reduction in total execution time due to the parallelism and is defined as the number of instructions needed by a single processing element to execute the best sequential algorithm, divided by the number of instructions needed by a parallel computer to execute the parallel algorithm. The efficiency 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 above described notions of speedup and efficiency are not capable of characterizing the execution of an asynchronous parallel algorithm adequately. Firstly, they do not take into account the time needed for synchronization. Secondly, they postulate that the time needed for a single communication is 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 running on. Thirdly, these notions postulate that all processing elements used are identical, whereas for many parallel computer systems, especially loosely coupled systems, the various 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 done by the various processing elements. It is possible that a redistribution of the work among the processing elements can result in a lower execution time.

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 share of the work and, secondly, all shares 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 number of instructions to be performed by the corresponding sequential algorithm and let S_i be the speed of processing element i, $i = 1, \ldots, N$, i.e. the number of instructions processing element i can perform per unit of time. The *corresponding sequential algorithm* is defined as the parallel algorithm executed by a single process. The perfect parallel time is defined as follows:

$$PPT = W / \sum_{i=0}^{N} S_i. \tag{3.1}$$

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 time required for communication and synchronization into account.

Because the above assumptions on divisibility, independency and interacting are unrealistic in real life, the perfect parallel time is a lower bound on the time needed for parallel execution on the given parallel computer system.

4. Parallel Branch and Bound

There are several ways to introduce parallelism in a branch and bound algorithm. Again, we can distinguish between synchronous and asynchronous parallelism. In the synchronous case the execution of the main loop of the branch and bound algorithm is divided into subtasks which are executed sequentially, although within the execution of such a subtask there may be parallelism. In the asynchronous case, an execution can no longer be divided this way. Subtasks can correspond to one or more iterations of the main loop or conversely, an iteration of the main loop can consist of several subtasks.
The parallelism in an asynchronous parallel algorithm tends to be at a higher algorithmic level than the parallelism in a synchronous algorithm.

4.1. Synchronous Parallel Branch and Bound

It is very easy to introduce synchronous parallelism in a branch and bound algorithm. Each part of the algorithm in which many steps have to be performed is a natural candidate for such a parallelization. Promising candidates are:

(a) *Lower bound calculation*. Some branch and bound algorithms use lower bound functions which are hard to compute. Dependent on the particular lower bound function used, parallelism might be introduced in this part of the algorithm. There are however lower bound algorithms that are intrinsically

sequential, for example bounds based on the greedy algorithm [Anderson & Mayr 1984].

(b) *Selection*. At some stages during the execution the number of subproblems in the active set can be very large. Therefore, selecting the next subproblem to branch from and extracting this subproblem from the active set can involve a large amount of work.

(c) *Elimination*. Testing whether the bound of a subproblem is still better than the hitherto best known feasible solution is easy. However, applying dominance tests and checking whether a subproblem still can produce a feasible solution can be very hard. The dominance test involves the comparison of the subproblem just generated with all other subproblems generated and not dominated so far. It might be possible to perform (part of) these tests in parallel.

(d) *Branching*. Instead of selecting and branching from a single subproblem at a time, it is possible to select several subproblems at once and branch from them in parallel. In order to have a good processing element utilization, the number of active subproblems must at least be equal to the number of processing elements.

The parallelism thus created is a parallelism on a higher algorithmic level than the parallelism created by (a), (b) and (c).

Not all of the above mentioned opportunities to introduce synchronous parallelism leave the execution of the branch and bound algorithm unchanged. The work done while branching from several subproblems in parallel might differ from the work involved in executing the corresponding sequential algorithm, i.e. the search trees generated might be different. This is due to the fact that it is possible that some of the subproblems branched from in parallel would have been eliminated or even never been generated in the sequential case.

Branching from several subproblems in parallel in essence amounts to a change the search strategy. Due to the fact that several subproblems are branched from in parallel, the knowledge obtained at a certain point during execution can differ from the knowledge obtained in the sequential case. This in turn can influence the selection of the subproblems to branch from next. The consequences of such a change are not clear in advance. All kinds of anomalies can occur, creating unreasonable speed ups or slow downs [Li & Wah 1984, Lai & Sahni 1984].

The resulting synchronous parallel algorithm is deterministic. Due to the synchronizations, the knowledge obtained will always be combined in the same manner. So consecutive executions of the algorithm will always yield the same answer.

In case of several subproblems being branched from in parallel, it can be very hard to organize the work in such a way that the various processing elements are always executing the same instruction at the same time. If the processing elements only infrequently execute different instructions and if the number of different instructions at a time is small, this can be arranged for by masking. This technique however has the disadvantage that it increases the overall complexity of the algorithm and obscures its essence. Not surprisingly then, earlier experiments showed that SIMD(like) systems are very unsuitable for the execution of synchronous parallel branch and bound algorithms [Kindervater & Trienekens 1985].

4.2. Asynchronous Parallel Branch and Bound

Asynchronous parallelism can only be introduced by parallelizing the branch and bound algorithm as a whole. If only part of the algorithm would be parallelized, all processes would have to synchronize after this part.

The iterations of the main loop of a branch and bound algorithm are fairly independent of each other. Rearranging the order in which these iterations are performed does not affect the correctness of the algorithm. For all active subproblems it still holds that, until that point of time, they are neither branched from nor eliminated. Therefore the preconditions on these subproblems remain valid. However, a rearrangement of the ordering can result in a different search tree generated. (In essence, such a rearrangement is a change in selection rule used.)

The basic idea behind asynchronous parallelism is that several iterations of the main loop are performed in parallel. Each process is given its own set of iterations to perform. As soon as a process has finished its iterations, it starts executing a new set of iterations without waiting for other processes to finish their set.

As said before, it is impossible to get a good impression of the work involved in executing a branch and bound algorithm without carrying out the actual execution. 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.
During execution of a parallel branch and bound algorithm the processes have a need to exchange information. For example, knowledge obtained by a particular process might be used by another process to eliminate some of its active subproblems. Information exchange can only be realized by means of communication.
Therefore asynchronous parallel branch and bound algorithms can be classified according to the way the available work is divided among the various processes, the unit of work chosen as the basic one and the way the knowledge obtained by the various processes is exchanged.

Dividing the work among the processes is done by creating *pools of work still to be done*. New work created by the processes is added to the pools and each time a process becomes idle, it obtains new work out of the pools. Each pool is accessed by a subset of the processes. 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. For example, suppose we chose the branching from a single subproblem and the computing of the lower bounds to the subproblems thus generated as the basic unit of work and use a single central pool, containing the active set, to divide the work among the processes. Each time a process becomes idle, a subproblem is extracted from this pool and given to this process to branch from. All subproblems generated are added to the pool. Of course the pool is pruned regularly by 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 give each process a good subproblem to branch from and to prune the set of active subproblems. 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 decentral pools are just the opposite. The bottleneck of all processes accessing the same pool is avoided, but some of the processes might not be branching from good subproblems simply because there happened to be no good subproblems in their pool. Apart from that, it is hard to eliminate subproblems by dominance tests because the subproblems are scattered all around the place.
The choice of the number of pools to use depends on the frequency with which the processes access these pools. This frequency can be influenced by the unit of work chosen as the basic one.

While executing an asynchronous parallel branch and bound algorithm, the processes have a need to communicate. For example, knowledge obtained by a particular process might be used by another process to eliminate some of its active subproblems. However, the broadcasting of all knowledge obtained increases the communication complexity of the parallel algorithm, and thus reduces the gain of fewer subproblems to solve. As can easily be seen, there exists a tradeoff between the number of subproblems eliminated and the number of communications performed.
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, each pool has to have complete knowledge about all the subproblems in the other pools generated thus far. The only way to accomplish this is by broadcasting all subproblems 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 kept without encountering the increase in communication complexity.
What constitutes a good tradeoff between communication complexity and computation complexity depends on the characteristics of the problem that has to be solved as well as on the characteristics of the parallel algorithm and the parallel computer system used.

Dividing the work among the processes and exchanging the knowledge obtained introduce nondeterminism in an asynchronous parallel branch and bound algorithm. Due to the asynchronicity, nothing can be said about the exact order in which the extractions, additions and eliminations on the pools of work are done by the various processes or when exactly 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 take another course, resulting in a different solution. Even if two consecutive executions of the algorithm yield the same solution, the work done 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. The solution yielded is always a correct one. Because we are only interested in finding a solution yielding the optimal solution value, the fact that different solutions are found is no drawback.
Nondeterminism however has one practical disadvantage: debugging a parallel branch and bound program becomes complex because it is very hard to trace what has happened during execution. Logical errors which caused run time errors and wrong answers during a particular execution might never occur again, simply because the program does not take the same course again.

5. Anomalies

During execution synchronous parallel branch and bound algorithms can suffer from *acceleration anomalies*, *deceleration anomalies* and *detrimental anomalies* [Li & Wah 1984]. Apart from these three anomalies, the execution of an asynchronous parallel branch and bound algorithm can suffer from *fluctuation anomalies*.
An acceleration anomaly occurs if the time needed for parallel execution is smaller than the perfect parallel time. A deceleration anomaly occurs if the time needed for parallel execution is larger than the time needed for executing the corresponding sequential algorithm. A detrimental anomaly occurs if the time needed for parallel execution is smaller than the time needed by the corresponding sequential algorithm, but greater than the perfect parallel time. Finally, a fluctuation anomaly occurs if another division of the work among the processes would result in a change in execution time.

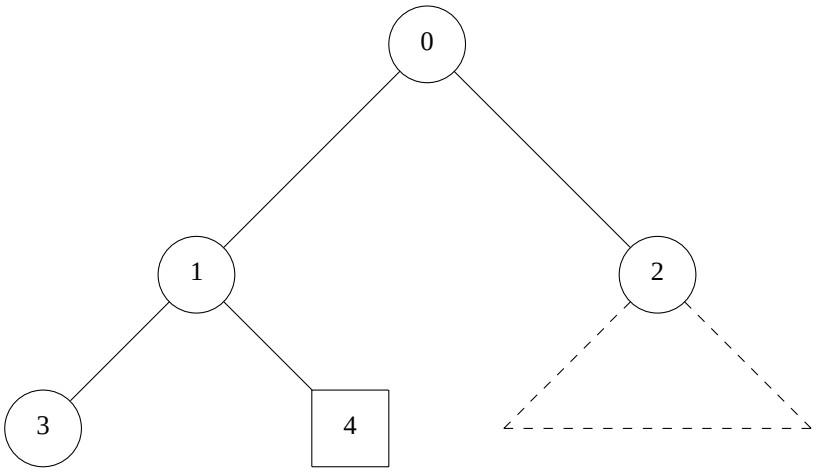

Figure 1.

An example of a fluctuation anomaly can be demonstrated from the problem instance displayed in figure 1. In this problem instance, each subproblem decomposes into two smaller subproblems. The branching from the first subproblem generated yields a feasible solution which can eliminate all other subproblems. All subproblems generated by branching from the second subproblem do not have a feasible solution, but this fact cannot be proven without complete decomposition. All decompositions take eight units of work to perform, four units for each subproblem to be generated.
Suppose this problem is solved using two processing elements, *A* and *B*. Processing element *A* can perform four units of work in one unit of time, whereas processing element *B* can only perform one unit of work per unit of time. Processing element *A* branches from the original problem. As soon as this branching is completed, the two processing elements start branching from subproblems. Amongst others, the following two divisions of work are possible: Firstly, processing element *A* gets subproblem 2 and processing element *B* gets subproblem 1. Or secondly, processing element *A* gets subproblem 1 and processing element *B* gets

subproblem 2. In the first case, the feasible solution eliminating all other subproblems is generated after nine units of time. In the second case however, it takes only four units of time.
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.

Fluctuation anomalies are likely to occur if a subproblem which must be branched from in order to (eventually) generate the optimal solution, is being branched from by the process running on the least powerful processing element, at a time that there are not enough promising subproblems available for branching by the other processes. In this case, the other processes start performing superfluous work by branching from subproblems which otherwise would have been eliminated in a later stage of the execution.
Clearly fluctuation anomalies can only occur while executing asynchronous parallel branch and bound algorithms on a set of processing elements which are not equally powerful.
As a consequence of fluctuation anomalies, the addition of a very powerful processing element to a system, although increasing the computing power of the system as a whole, need not decrease the time needed for executing a parallel algorithm. Adding a weak processing element to a system can even increase the total time needed for execution: the additional power of the system might be used only to perform additional superfluous work, whereas all the 'useful' work is performed by the weak processing element.

Naturally one would like to preserve acceleration anomalies, avoid deceleration and detrimental anomalies and use fluctuation anomalies in such a way that the time needed for executing a parallel branch and bound algorithm is minimized.
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 on hand. For example, it can happen that all the subproblems which the corresponding sequential algorithm would branch from are descendants of each other. Deceleration anomalies can always occur because a parallel branch and bound algorithm does have to spend effort on communications and/or synchronizations, whereas the corresponding sequential algorithm does not have to spend effort on these subjects.

In the following sections we will develop sufficient conditions to prevent deceleration anomalies from degrading the performance of a parallel branch and bound algorithm in which the basic unit of work is the branching from a single subproblem and the computing of lower bounds to the subproblems thus generated. Within the above framework, we will develop necessary conditions to allow acceleration anomalies from improving the performance of the algorithm.
It turns out that the already known sufficient conditions for synchronous branch and bound algorithms with a single central pool for dividing the work [Li & Wah 1984] are sufficient for all synchronous and asynchronous branch and bound algorithms.

For the lemma's and theorems following, 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 another basic subproblem.

The *path number* of a node in the search tree is a sequence of at most $d+1$ integers, where d is the maximum level of 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 2 shows an example of how to number the nodes of a search tree.
Path numbers can be ordered lexicographically. Two path numbers $e = e_0 e_1 \cdots e_n$ and $\hat{e} = \hat{e}_0 \hat{e}_1 \cdots \hat{e}_m$ 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 $j = 0, \ldots, n$ and $n < m$.

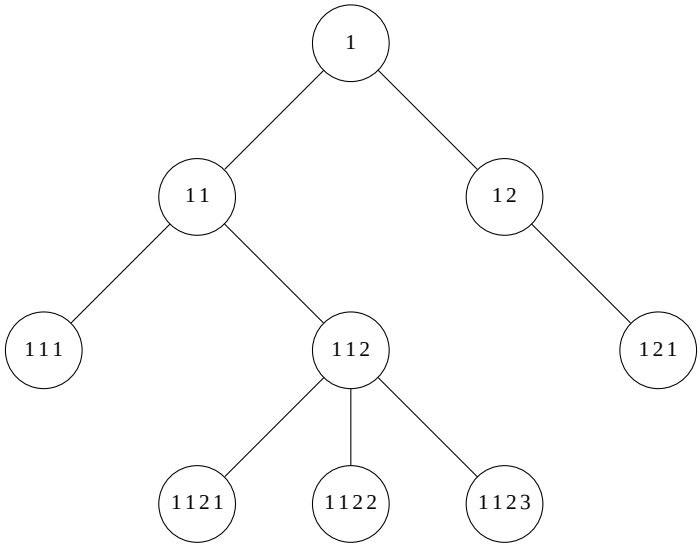

Figure 2.

5.1. Parallel branch and bound with lower bound tests only.

All anomalies are caused by the branching from several subproblems in parallel. Due to these branchings in parallel, the knowledge obtained at a particular point during the execution of the parallel algorithm can differ from the knowledge obtained at the corresponding point during the execution of the corresponding 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 it can be achieved that the knowledge obtained during execution of a parallel branch and bound algorithm is always a superset of the knowledge obtained during execution of the corresponding sequential algorithm. This can be done by letting the order in which the subproblems are branched from by the parallel algorithm resemble the order of the corresponding sequential algorithm as much as possible. To accomplish this, it is necessary that all subproblems can be uniquely identified by their heuristic value.
One way to accomplish a unique identification is by requiring the heuristic function h used by the selection rule to be *injective*, i.e. to assign a unique value to each subproblem:

(5.1) $h(P_i) \neq h(P_j)$ if $P_i \neq P_j$.

This is no real restriction because each non injective heuristic function can easily be transformed into an injective one by adding a unique additional component to each function value, namely the path number of its argument (which corresponds to a node of the search tree). Due to the fact that all path numbers are unique, all heuristic values will be unique.
Note that an injective heuristic function allows for a complete ordering on the subproblems through their heuristic values.

Next to being injective, the heuristic function also must be *non misleading*, i.e. provide no misleading information about the problem instance to be solved. It must be impossible that a subproblem looks more promising than the subproblem it was generated from by decomposition. Otherwise, if the parallel algorithm branches from a subproblem earlier than the corresponding sequential algorithm does, the subproblems thus generated would have a higher priority to be branched from than the subproblems branched from by the corresponding sequential algorithm before branching from this particular subproblem, but which are not yet branched from by the parallel algorithm.
Clearly a heuristic function is non misleading if the heuristic values it assigns do not decrease, i.e.

(5.2) $h(P_i) < h(P_j)$ if P_j is a descendant of P_i.

Now we are ready to consider the case of parallel branch and bound algorithms without dominance tests.
Firstly, we will consider the case of preventing deceleration anomalies from degrading the performance of the parallel algorithm. We start by proving a lemma which yields an upper bound on the work to be done while executing a parallel branch and bound algorithm. This lemma states that if all basic subproblems are either branched from or eliminated, enough knowledge is obtained to solve the problem instance on hand. Then we will prove a lemma which states sufficient conditions to ensure that at each point of time during execution the active basic subproblem with the smallest heuristic value is being branched from. Using these two lemma's we will prove a theorem stating sufficient conditions to prevent deceleration anomalies from degrading the performance. The sufficient conditions will ensure that always at least one of the basic subproblems is being branched from. In the worst case, all basic subproblems are branched from sequentially by the process running on the least powerful processing element.
Secondly, we will consider the case of allowing acceleration anomalies to improve the performance of the parallel algorithm. We will prove a theorem which states necessary conditions for such an improvement to be possible.
Note that the lemma's and theorems are independent of synchronicity or asynchronicity, the way the work is divided among the various processes and the relative speeds of the various processing elements.

Lemma 1:
If during execution of a parallel branch and bound algorithm which uses only lower bound tests and no dominance tests, a point is reached where there are no more active basic subproblems, there is enough knowledge obtained to solve the problem instance on hand. I.e., a feasible solution has been found 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. Therefore branching from these basic subproblems must have yielded a feasible solution which eliminated all non basic subproblems by a lower bound test. For convenience, we call this solution the *sequential solution*.
A parallel branch and bound algorithm starts by branching from the original problem. Therefore, at the beginning of the execution, there is an active basic subproblem. If during execution 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.
If all basic subproblems have been branched from, the sequential solution has been generated and this solution can eliminate all remaining active subproblems by a lower bound test.
If some basic subproblems have been eliminated, the branching from some subproblem must have generated a feasible solution which eliminated these basic subproblems. The basic subproblem whose decomposition yields the sequential solution is either branched from, eliminated or never generated (in case one of its predecessors was eliminated). If this subproblem has been branched from, the sequential solution has been found and this solution can eliminate all active subproblems. If this subproblem has been eliminated or never generated at all, the subproblem itself, or one of its predecessors, must have been eliminated by a lower bound test by another feasible solution found. This other feasible solution must have the same value as the sequential solution and therefore can also eliminate all active subproblems.

Q.E.D.

To be able to prove the next lemma we have to postulate that it is possible to interrupt a process once new knowledge obtained has revealed that the subproblem this process is branching from can be eliminated by a lower bound test.

Lemma 2:
At each point of time during execution of a parallel branch and bound algorithm which uses only lower bound tests, no dominance tests and a heuristic function which is injective and non misleading, some process will be branching from 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 branched from.
The fact that the active basic subproblem with the lowest heuristic values is not being branched from implies that at least one of the processes is branching from a non basic subproblem with a smaller heuristic value.
Let $P_{i_0}, P_{i_1}, ..., P_{i_n}$ be the series of basic subproblems. This series is ordered according to increasing heuristic value. Let P_{i_j} be the active basic subproblem with the smallest heuristic value which is not being branched from at that point of time. Let P_k be the non basic subproblem with smaller heuristic value which is being branched from.
Because P_{i_j} is the active basic subproblem with the lowest heuristic value, the basic subproblems $P_{i_0}, \ldots, P_{i_{j-1}}$ do not exist at that point of time. Furthermore, because the heuristic function used is non misleading, $P_{i_0}, \ldots, P_{i_{j-1}}$ cannot be generated anymore. Hence they must be branched from or eliminated by the parallel algorithm.
The fact that the parallel algorithm is branching from P_k implies that, until then, the algorithm has not obtained enough knowledge to eliminate this subproblem. However, the knowledge obtained by the parallel algorithm is a superset of the knowledge obtained by the corresponding sequential algorithm. Hence the corresponding sequential algorithm too cannot eliminate P_k. Because P_k has a smaller heuristic value, the corresponding sequential algorithm has to branch from P_k before branching from P_{i_j}. Hence P_k is a basic subproblem. This however contradicts the fact that P_k is a non basic subproblem.

Q.E.D.

Let $T_p(I)$ be the time needed for solving problem instance I on a particular parallel computer system with a parallel branch and bound algorithm using only lower bound tests, no dominance tests and a heuristic function which 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 with the corresponding sequential branch and bound algorithm. Let n be the maximum number of processes accessing the same pool of work and let b be the time needed to send a subproblem from a process to a pool (or vice versa).
We assume that adding a subproblem to a pool and pruning all active subproblems or extracting a subproblem from a pool can be done in constant time c and that once we have enough knowledge obtained to solve the problem, the halting of the execution does not take additional time.

Theorem 1*:* $T_p(I) \leq T_s(I) + 2 \,.\, (b + (n-1).c)$

Proof:
Lemma 2 states that at each given point of time the active basic subproblem with the smallest heuristic value will be being branched from. In the worst case, all basic subproblems are branched from sequentially by the process executed on the least powerful processing element.
The term $2.(b+(n-1).c)$ stems from the fact that apart from computations on the branchings, there is also work involved in dividing the subproblems among the processes. Each basic subproblem generated must be added to a pool and (in the worst case) extracted. The time needed for adding (subtracting) a subproblem is the sum of the time needed for sending it to the pool and the waiting time in case of the pool first having to service other requests. Because there are at the most n processes accessing the same pool, there are at the most $n-1$ requests which must be served prior to this request. Serving these queued requests takes at the most $(n-1).c$ time. Because this division of work is not needed by the corresponding sequential algorithm, we have to account for it.

Q.E.D.

Note that theorem 1 does not state anything about the total number of subproblems branched from.

Now for the neccessary conditions for improving the performance by acceleration anomalies.
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.

Theorem 2:
The performance of a parallel branch and bound algorithm which uses only lower bound tests, no dominance tests and a heuristic function which is injective and non misleading cannot be improved by acceleration anomalies unless the heuristic function used is not consistent with the lower bound function used.

Proof (by contradiction):
The only way an acceleration anomaly can improve the performance of a parallel branch and bound algorithm is by eliminating some of the basic subproblems instead of branching from them.
Let $P_0, P_1, \ldots, P_n$ be the series of basic subproblems ordered according to increasing heuristic value. This order is equivalent to the order in which these basic subproblems are branched from.
Suppose the heuristic function used is consistent with the lower bound function used. This implies that 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 was the last subproblem branched from 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.

Q.E.D.

5.2. 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 problems 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, which might involve lots of work.

Because nothing can be said about the order in which the various subproblems are generated by the parallel algorithm, it is impossible to solve the non transitivity problem by ensuring that the tests are always performed in the same order as the corresponding sequential algorithm would perform them.
However, for certain classes of dominance relations and heuristic functions it holds that these tests are transitive. Firstly we will prove a lemma which states that for these classes of dominance relations and heuristic functions no basic subproblems will be eliminated during parallel execution. Then we will prove a lemma which states that once all basic subproblems are branched from, there is enough knowledge obtained to solve the problem instance on hand. Using these two lemma's, a theorem stating sufficient conditions to prevent deceleration anomalies from degrading the performance of a parallel branch and bound algorithm can be proven in a straightforward manner. Again the sufficient conditions will ensure that always at least one of the basic subproblems is being branched from.

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

Lemma 3:
If a parallel branch and bound algorithm uses lower bound tests, a dominance relation which is consistent with the heuristic function and a heuristic function which is injective, non misleading and consistent with the lower bound function, then during execution of this algorithm no basic subproblems are eliminated.

Proof (by contradiction):
First note that the lower bound of a basic subproblem is smaller than the value of the optimal solution because the heuristic function used is non misleading and consistent with the lower bound function.
Now for the proof of the lemma: Suppose there exists a basic subproblem P_i which is eliminated during execution of the parallel branch and bound algorithm. P_i is eliminated either by a lower bound test or by a dominance test.
Suppose P_i was eliminated by a lower bound test. This implies that the lower bound of P_i is greater than

the

value of a feasible solution. However, this contradicts the fact that the lower bound of P_i is smaller than the value of the optimal solution.
Suppose P_i was eliminated by a dominance test by P_j. P_j is either a basic subproblem or not.
Suppose P_j is a basic subproblem. Due to the consistency of the dominance relations with the heuristic function this implies that $h(P_j) < h(P_i)$. Therefore the corresponding sequential algorithm branches from P_j before it branches from P_i. Hence if P_j dominates P_i, P_i would never be branched from by the corresponding sequential algorithm. This however is in contradiction with the fact that P_i is a basic subproblem.
Suppose P_j is a non basic subproblem. Because P_j is non basic, there exists a series of subproblems $P_{j_1}, P_{j_2}, \ldots, P_{j_n}$ such that $P_{j_1}, \ldots, P_{j_{n-1}}$ are non basic subproblems and P_{j_n} is a basic subproblem with the properties that P_{j_k} can eliminate $P_{j_{k-1}}$ $(k = 2, \ldots, n)$ and P_{j_1} can eliminate P_j. This chain of eliminations either consists solely of eliminations by dominance tests or not. If there are only eliminations by dominance tests, P_{j_n} can eliminate P_i directly by a dominance test because dominance relations are transitive. Therefore the above case of P_j being basic applies mutatis mutandis. If one or more of these eliminations is an elimination by lower bound test, the consistency of the dominance relation with the heuristic function and of the heuristic function with the lower bound function imply that the lower bound of P_i is greater than the value of a feasible solution. This however contradicts the fact that the lower bound of P_i is smaller than the value of the feasible solution.

Q.E.D.

If the heuristic function used is injective (assigns unique values), the consistency of the heuristic function with the lower bound function implies that all lower bounds are unique. This however is no real restriction on a lower bound function. Just like in case of a heuristic function, this uniqueness can be easily arranged for by adding as an additional component to the lower bound value yielded the path number of the corresponding node of the search tree.

The conditions stated in Lemma 3 imply only that an optimal solution will be found within a limited effort. Now we have to prove this solution to be optimal.

Lemma 4:
If during execution of a parallel branch and bound algorithm which uses lower bound tests, a dominance relation which is consistent with the heuristic function and a heuristic function which is injective, non misleading and consistent with the lower bound function, a point is reached where there are no more active basic subproblems, there is enough knowledge obtained to solve the problem.

Proof:
Using Lemma 3 it can be concluded that that once there are no more active basic subproblems, an optimal solution has been found. It remains to be proven that there is also enough knowledge obtained to eliminate all remaining active subproblems.
Let P_i be an arbitrary non basic subproblem. Because P_i is non basic, P_i would be either eliminated or never generated at all by the corresponding sequential algorithm. The last case implies that a predecessor of P_i would have been eliminated. Let P_{i_0} be P_i if P_i would have been eliminated by the corresponding sequential algorithm. Otherwise, let P_{i_0} be the predecessor of P_i which would have been eliminated. Because P_{i_0} is non basic, there exists a series of subproblems $P_{i_1}, P_{i_2}, \ldots, P_{i_n}$ such that $P_{i_1}, \ldots, P_{i_{n-1}}$ are non basic subproblems and P_{i_n} is a basic subproblem with the property that P_{i_j} can eliminate $P_{i_{j-1}}$ by a lower bound and/or a dominance test, $j = 1, \ldots, n$. Again this chain of eliminations either consists only of eliminations by dominance tests or not.
If there are only eliminations by dominance tests, P_{i_n} can eliminate P_{i_0} (and therefore P_i) directly by a dominance test because dominance relations are transitive.
If one or more of these eliminations is an elimination by lower bound test, then the sequential solution can eliminate P_{i_0} (and hence P_i) by a lower bound test. Let P_{i_j} be the first subproblem in this series eliminating its predecessor in the series by a lower bound test. Because the dominance relations are consistent with the heuristic function, which in turn is consistent with the lower bound function, it holds that the lower bounds

of P_{i_k}, $k = 0, \ldots, j-1$, are greater than the value of P_{i_j}. This value however is greater than or equal to the value of the sequential solution. Therefore the sequential solution can eliminate P_{i_0} by a lower bound test.
Q.E.D.

Now it is easy to prove that the performance of the above described parallel branch and bound algorithm is not degraded by deceleration anomalies. The proof is analogous to the one given for theorem 1.

6. The Distributed Processing Utilities Package

The Department of Computer Science of the University of Colorado at Boulder, Colorado, United States, owns several Vaxes, Pyramids and Suns. These machines are running the Berkeley Unix operating system and are connected via an Ethernet. On top of Unix Timothy Gardner built the *Distributed Processing Utilities Package* (DPUP), a distributed programming tool box which allows the user to use the machines on the net as a loosely coupled system [Gardner et al. 1986].
This loosely coupled system is an unusual system in that it is possible for a processor to be executing in a time sharing mode concurrently more than one process.
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. 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 can send a message at a time. Therefore all communications have to be handled sequentially. Fortunately an Ethernet has a very high bandwidth, i.e. has a very high communication capacity. So, as long as there are not too many big messages, the chances of two messages colliding or of the Ethernet becoming a bottleneck are small.

7. Our Parallel Branch and Bound Algorithm

For our research we are interested in the consequences of branching from several subproblems in parallel with the best possible use of the knowledge obtained so far. By 'best possible use of the knowledge obtained so far' we mean that during execution of a parallel branch and bound algorithm we try to branch only from those subproblems which the corresponding sequential algorithm would also branch from. The branching from subproblems which would have been eliminated or even never generated by the corresponding sequential algorithm is considered to be superfluous work which unnecessarily slows down the execution.
With this in mind, we developed an asynchronous branch and bound algorithm using a single central pool for dividing the work and with as basic unit of work the branching from a single subproblem and the computing of lower bounds to the subproblems thus generated. We chose asynchronicity because we did not want to waste computing power in case the effort needed for branching from a subproblem or for computing a lower bound varied with 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.
We realize that the best possible use of knowledge obtained so far increases the communication complexity of our algorithm. However, because the lower bound function we intended to use involved lots of computations, we hoped that this increase in communication complexity would be dominated by the decrease in computation time it caused.

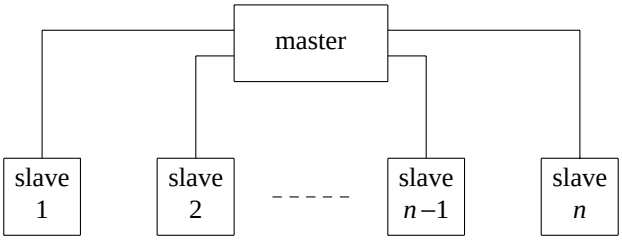

Figure 3.

Our algorithm introduces a master process and several slave processes (c.f. figure 3). The master takes all important decisions and the slaves do as they are told to do by the master. In particular, the master 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 obtained so far by the slaves (the subproblems generated, the feasible solutions found etc.). Interpreting this knowledge, the master maintains the set of active subproblems. Each time the master is aware of a slave becoming idle, it extracts a subproblem from this set and sends this problem to the slave to branch from. If the master runs out of active subproblems, the slave is put into a queue of idle slaves until new subproblems to branch from become available. The execution of the branch and bound algorithm is terminated as soon as all slaves are in this 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 master does not need to know about subproblems eliminated by such a test because those subproblems can never yield the optimal solution. This way the knowledge to be collected and interpreted by the master decreases. Note that the fact that a subproblem is not eliminated by an lower bound test carried out by a slave does not imply that the master will automatically add this subproblem to the active set. This holds because the slave simply might not yet know about a better solution found by another slave in the meantime. The master can even interrupt a slave if new knowledge reveals that the work that that particular slave is performing has become superfluous.
A slave branches from the subproblem it receives from the master. As soon as the computations on one of the subproblems thus generated are finished, this subproblem is sent to the master. This is done to allow the master to take this new knowledge into account as soon as possible. As a consequence, at the start of the execution it is possible to have all the slaves active as soon as possible. After completing the branching, the slave requests new work from the master by notifying the master that it has become idle.
At first sight it may seem strange that a slave sends all the subproblems it generates to the master and than asks the master for a new 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

To execute our 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 it 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 read the first one.
If the system used does not provide the interprocess communication described above, it is still capable of executing the algorithm, although computational power will be lost due to the unnecessary synchronizations.

The master process is not strictly necessary. It is also possible to let the slave processes themselves take all decisions by storing all the knowledge obtained so far in some kind of common memory. This however introduces all kinds of problems if several slaves try to update this information at the same time.

These problems are problems around concurrency control which can be solved in various ways [Andrews & Schneider 1983].

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

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 processor. Running more than one process on a processor introduces all kinds of overhead like process control, swapping etc. Therefore we intended to run all our processes on different processors. 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 assign a processor solely to it. We ran the master process together with one of the slave processes on the same processor. 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 ought to be given a higher priority.

Our algorithm resembles the sequential branch and bound algorithm very much. The only rules which must be added are rules telling how to divide the work among the various processes. A consequence of this is that all the features of a sequential branch and bound algorithm can be easily incorporated into our algorithm.

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 correcting 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 it to give the same order to another slave if it detects that a slave has gone dead. However, a failure in the master process is still lethal.

8. Computational Experiments

We tested our algorithm by solving some random generated instances of the symmetric traveling salesman problem on the DPUP system of the University of Colorado at Boulder.

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. Following 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 $V\{1\}$ 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. The major part of the work involved in executing this algorithm is the sorting of the edges according to their weight. Given a minimal spanning tree, a minimal 1-tree can be constructed by adding the two edges incident to node 1 with minimal weight to the spanning tree.
Obviously a weight transformation $\hat{c}_{i,j} = c_{i,j} + \pi_i + \pi_i$ 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 does 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 found 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 use path numbers to guarantee that all heuristic values would be distinct. Firstly because, due to the use of lagrange multiplier technique, the chances of two lower bounds being equal are very small, secondly, because the use of path numbers would have required additional memory.
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 less forbidden edges incident to a particular node (where m is the number of edges incident to this node), than this subproblem does not have a feasible solution.
Our algorithm does not use dominance relations because we do not know of useful dominance relations for the traveling salesman problem in conjunction with the branching strategy chosen.

The parallel program was written in C, the same language as 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 in turn starts and kills all slave processes.

The problems of debugging a nondeterministic parallel program were partly bypassed by starting to execute our program using only one 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 appeared 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.

While implementing our 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. The 1-tree generated 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 on the various machines, and hence in different 1-trees) Therefore the subproblems generated by decomposition did not only depend on the branching rule but also on the machine the 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 same parallel algorithm or to compare the execution of a 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.

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 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.

A parallel program executed using DPUP is heavily influenced by other activities happening in the net. If a machine is executing other processes apart from the processes of the parallel program, this implies that it has less time for the parallel program. Also 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 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 get all machines stand alone during the night.

8.1. Computational results

We tested our parallel branch and bound 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 always executed a single slave process. Apart from that, the most powerful machine also executed the master process. We started by using two Pyramids and then 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 a couple of 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 consequences of the use of parallelism, and not in solving problems as big as possible as fast as possible. We wanted to study the behaviour of a parallel branch and bound algorithm with increasing number of processes used in parallel. We were especially interested in whether we could keep all processes busy all the time.

As mentioned earlier, asynchronous 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 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.

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)

acchieved efficiency
(nr of subproblems branched)

figure 4

Figure 4 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 increased a little bit due to 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.
As can be easily seen there exists only a limited amount of parallelism within a problem instance. Therefore 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.
Our parallel 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.

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 unreasonable efficiency is by looking at the various tricks a computer uses to speed up 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 goes faster than accessing dynamic variables.

In most cases the addition of a slave process running on a Sun-2, even though increasing the power of the system as a whole, resulted in an increase in the time needed for executing the parallel program. The execution of a program normally ended with all the other processes being idle and waiting for the slave process running on the Sun-2 to complete its last task. Therefore, if there was only a limited amount of parallelism present within the problem instance, the speedup gained at the start was completely destroyed by the waiting in the end. Therefore execution could take a longer time even though the work to be done was completely equal (i.e. identical search trees were generated).
Sometimes even the number of subproblems branched from drastically increased. As mentioned before, this increase was caused by the fact that the least powerful slave process was branching from a basic subproblem at a time that there were no more basic subproblems for branching from by the more powerful processes. In such a case, the other processes started branching from subproblems which would have been eliminated otherwise. As the least powerful process finished branching from its subproblem, the subproblems thus generated were given to the more powerful processes to branch from and the execution left the wrong track.
Figures 5 and 6 show minimum and maximum execution times and the number of subproblems branched from while solving two specific 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 5 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 process running on a Pyramid and the speedup at the start of the execution dominated the waiting at the end. Therefore the increase in execution time caused by fluctuation anomalies is small. Figure 6 displays data about problem e50c, a problem with limited intrinsic parallelism. Here fluctuation anomalies severely increased the time needed for execution.

slave processes running on	fastest time	nr of nodes	slowest time	nr of nodes
2 pyramids	29:39.48	(271)	29:49.46	(271)
3 pyramids	20:21.18	(271)	20:23.92	(271)
4 pyramids	15:49.26	(271)	15:51.26	(271)
5 pyramids	13:55.96	(271)	14:03.40	(271)
2 pyramids + sun	26:47.06	(271)	27:01.80	(271)
3 pyramids + sun	18:54.12	(271)	19:02.46	(271)
4 pyramids + sun	15:00.90	(271)	15:06.34	(271)
5 pyramids + sun	13:18.00	(271)	13:45.38	(271)

figure 5 - problem e75b

slave processes running on	fastest time	nr of nodes	slowest time	nr of nodes
2 pyramids	0:48.78	(12)	0:50.82	(12)
3 pyramids	0:48.10	(15)	0:53.14	(14)
4 pyramids	0:43.12	(20)	0:53.64	(17)
5 pyramids	0:49.02	(24)	0:52.68	(24)
2 pyramids + sun	0:54.80	(13)	1:41.90	(25)
3 pyramids + sun	0:49.32	(17)	1:03.30	(19)
4 pyramids + sun	0:58.40	(21)	1:10.88	(29)
5 pyramids + sun	0:48.42	(26)	1:11.08	(27)

figure 6 - problem e50e

8.2. Branching in parallel from the original problem

As already mentioned, our branch and bound algorithm has one inherent sequential part: a single slave process computes the lower bound to the original problem. Until this slave process has completed this computation, 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 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 our parallel branch and bound algorithm by parallelizing the computing of the lower bound to the original problem. However, parallelizing this lower bound computation, although decreasing the overall computational complexity, increases the communicational 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 run on.

As one of our computational experiments we tried to speed up the execution of our parallel program 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 updates the weights of the edges, sorts the edges according to their new weights and finally 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. The generating of the minimal spanning tree is done by a greedy algorithm and therefore difficult to parallelize [Anderson & Mayr 1984]. Therefore the only part fit for parallelizing is the sorting of the edges. We parallelized the sorting by dividing the edges into sets and giving 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 communicational complexity completely dominated the decrease in computational complexity. Only when we ran our program solely on the Suns and still used floating point arithmetic, it turned out to be faster.

8.3. Robustness

Parallel execution also tends to increases the robustness of a branch and bound algorithm. An branch and bound algorithm is said to be *robust* if it does not spend much effort in branching from subproblems which could have been eliminated if the order in which the subproblems are branched from would have been different. For a sequential branch and bound algorithm it is very easy to spend large amounts of its time decomposing subproblems which can never yield the optimal solution. Because parallel branch and bound algorithms decompose various subproblems at the same time, the chance that all processes are decomposing subproblems which can never yield the optimal solution is smaller. However, as can be easily seen from the fact that deceleration anomalies exist, it cannot be guaranteed that parallel branch and bound algorithms are more robust than sequential branch and bound algorithms.

9. Conclusion

Our experiments showed that it is possible to get high achieved efficiencies while executing asynchronous parallel branch and bound algorithms on a loosely coupled system with one central pool for dividing the subproblems to branch from among the various processes. If the effort needed for branching from a single subproblem is big, the frequency with which the various processes want to access the central pool is low, and accessing the pool is not a bottleneck. Therefore, while executing a parallel branch and bound algorithm using lower bounds which are hard to compute, there is no use for decentral pools.
Each problem instance contains only a limited amount of parallelism. Therefore there exists an upper bound on the number of processes which can be used in parallel successfully to decrease the execution time.
Adding a weak processing element to the system turned out to be dangerous. Most of the time this increased the execution time. The amount of increase was inversely proportional to the intrinsic parallelism of the problem.

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

References

R. Anderson, E. Mayr (1984). *Parallellism 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.*

D.P. Bertsekas (1983). Distributed asynchronous computation of fixed points. *Mathematical Programming 27.*

J.J. Dongarra, I.S. Duff (1985). *Advanced architecture computers.* Mathematics and Computer Science Division, Argonne National Laboratory, Technical Memorandum No. 57.

M.J. Flynn (1966). Very high-speed computing systems. *Proc. IEEE 54.*

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.

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.*

ICL (1981). *DAP: Developing DAP Programs.* Technical Publication 6920, ICL, London.

R. Jonker, T. Volgenant (1982). A branch and bound algorithm for the symmetric travelling salesman problem based on the 1-tree relaxation. *EJOR (1982).*

G.A.P. Kindervater, H.W.J.M. Trienekens (1985). *Experiments with parallel algorithms for combinatorial problems.* Centre for Mathematics and Computer Science, Report OS-R8512.

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.

P.C. Treleaven, D.R. Brownridge, R.P. Hopkins (1982). Data-driven and demand driven computer architecture. *Comput. Surveys 14.*

I. Watson (1984). The dataflow approach - architecture and performance. F.B. Chambers, D.A. Duce, G.P. Jones (eds). *Distributed Computing*, Academic Press, London, Ch. 2.