A simulation tool for the performance evaluation of parallel branch and bound algorithms

Arie de Bruin

Alexander H.G. Rinnooy Kan

Harry W.J.M. Trienekens

Erasmus University Rotterdam

ABSTRACT

Parallel computation offers a challenging opportunity to speed up the time consuming enumerative procedures that are necessary to solve hard combinatorial problems. Theoretical analysis of such a parallel branch and bound algorithm is very hard and empirical analysis is not straightforward because the performance of a parallel algorithm cannot be evaluated simply by executing the algorithm on a few parallel systems. Among the difficulties encountered are the noise produced by other users on the system, the limited variation in parallelism (the number of processors in the system is strictly bounded) and the waste of resources involved: most of the time, the outcomes of all computations are already known and the only issue of interest is when these outcomes are produced.
We will describe a way to simulate the execution of parallel branch and bound algorithms on arbitrary parallel systems in such a way that the memory and cpu requirements are very reasonable. The use of simulation has only minor consequences for the formulation of the algorithm.

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

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

1. INTRODUCTION

Operations researchers are beginning to discover that parallel computation is more than just a fad or a fashion. The rapid growth of parallel computing technology, coupled with the existence of an active theoretical research community, provides a natural incentive for operations researchers to consider the implications of these developments for their computational needs and efforts.
The area of *mathematical programming* is one of the first where the impact of parallelization has been felt. The importance of increasingly fast solution methods for mathematical programming problems is evident. In many situations, the limits of what can be achieved on a sequential computer are in sight. Further progress, which is still highly desirable, can only come through the development of appropriate parallel hardware and software.
For the important area of *combinatorial optimization*, theoretical computer scientists have taken the lead in designing clever parallel methods to solve many of the standard problem types (see [Kindervater & Lenstra, 1985, 1986] for a survey). Interestingly enough, virtually all this work has concentrated on speeding up the solution of problems that could already be solved quite efficiently (i.e., in polynomial time) on a

Report 8720/A
Erasmus University Rotterdam, Econometric Institute
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

sequential computer. The practical need for this speed up remains to be demonstrated. Certainly, the need to improve our ability to solve the hard (i.e., NP-complete) combinatorial problems is at least as pressing. Here, parallelism offers a natural opportunity that has hardly been investigated. In the words of R.M. Karp: "Even though you may never be able to go from exponential to polynomial, it's also clear that there is a tremendous scope for parallelism on those hard problems, and parallelism may really help us curb combinatorial explosions."

This paper forms part of an ongoing research project on the design and analysis of *parallel branch and bound methods*. Branch and bound methods form essentially the only tool available to solve hard combinatorial problems to optimality. Their worst case exponential behaviour is a fact of life. At the same time, parallel computation could be of real help in increasing the size of the problems that can be solved to optimality within reasonable time. The ultimate goal of the project is to arrive at more precise insights in the interaction between the algorithm chosen, the parallel machine used and the problem to be solved interact.
In section 2, we describe the options available for parallelizing branch and bound methods. In section 3, we describe the problems encountered while evaluating the performance of a parallel branch and bound algorithm. We argue that testing an algorithm on a few parallel machines will not give sufficient insight in the behaviour of the algorithm and that a more general approach is needed. In section 4 we propose a solution to this problem, namely to use simulation. This section ends with some concluding comments, some computational results and with an indication of future work. Throughout the paper we will use as a running example our experiences with a specific parallel branch and bound algorithm. We describe the problems encountered while testing this algorithm at the University of Colorado at Boulder and the way we solved them using simulation.

2. PARALLEL BRANCH AND BOUND

2.1. The Branch and Bound Algorithm

Branch and Bound algorithms solve discrete optimization problems by partitioning the solution space. Throughout this paper, we will assume that all optimization problems are posed as minimization problems and that solving a problem means finding a feasible solution with minimal value. If there are many such solutions, it does not matter which one is found.
A branch and bound algorithm can be characterized by four rules [Mitten, 1970]: a *branching rule* defining how to split a problem into subproblems, a *bounding rule* defining how to compute a lower bound on the optimal solution of a subproblem, a *selection rule* defining which subproblem to branch from next and an *elimination rule* stating how to recognize and eliminate subproblems which cannot yield the optimal solution to the original problem. With respect to this last rule, three tests can be distinguished. 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 it can be proven 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 possible sequential 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 the following steps are repeatedly executed. Using the selection rule, one of the subproblems in the active set is chosen to branch from. This subproblem is extracted from the active set and decomposed into smaller subproblems using the branching rule. For each of the subproblems thus generated a lower bound is calculated. If during the 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.

2.2. Parallel Branch and Bound Algorithms

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

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

The basic idea behind high level parallelism in branch and bound algorithms is that several iterations of the main loop are performed in parallel. In doing so, there are some decisions to be made which we will now discuss.

2.2.1. Parameters of parallel branch and bound algorithms

It is impossible to get an accurate estimate of the work involved in executing a branch and bound algorithm without carrying out the actual computations. Therefore, the only way to achieve an equitable division of the work among the various processes while executing a parallel branch and bound algorithm 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.
We also have to specify what happens when a process completes its part of the work. There are two extremes: before starting its next job the process can wait for all other processes to complete their part of the work, or, the process can start a new job immediately without waiting for fellow processes to complete their parts.
During execution of a parallel branch and bound algorithm the processes 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 parallel branch and bound algorithms can be classified according to the way in which the available work is divided among the various processes, the unit of work which is chosen as the basic one, the synchronicity of the processes and the way the knowledge obtained by the various processes is exchanged.
In the next sections we will elaborate on these parameters. To arrive at precise insights in the consequences of the various parameter settings is the prime goal of our research.

2.2.1.1. Dividing the work

Dividing the work among the processes is carried out 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. The two extremes are a single central pool accessed by all processes and a private decentral pool for each process. In case a pool dries up, there is the option to refill it from one of its fellow pools.

For example, suppose that we choose our basic unit of work to correspond to the branching from a single subproblem and the computation of the lower bounds on the subproblems thus generated and that we 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 the central 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 provide each process with 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 subproblem in their pool. Apart from that, it is hard to eliminate subproblems by dominance tests because the subproblems are scattered all around.

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

2.2.1.2. Information exchange

Information exchange can only be realized by means of communication. A perfect information exchange can be obtained by letting each process broadcast all knowledge it obtained. This, however, increases the communication complexity of the parallel algorithm, and thus reduces the advantage obtained because there are fewer subproblems to branch from. It is clear that there exists a tradeoff between the number of subproblems eliminated and the amount of communication carried out.

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 to be solved, as well as on the characteristics of the parallel algorithm and the parallel computer system used.

2.2.1.3. Synchronicity

The fact whether or not the processes perform their work in a synchronized way has consequences for the utilization of the various processes as well as for the communication complexity of the resulting parallel algorithm.
For synchronized processes it holds that if the tasks to be performed are not of equal size, computation power is lost while waiting for other processes to complete their task. Synchronization involves communication because one way or another the processes must find out whether the other processes have completed their tasks or not. Therefore synchronization tends to increase the communication complexity of the algorithm. An advantage of synchronizing is that dividing the work among the various processes tends to be easy, because it is known when the processes will be ready to start their next task.
The characteristics of asynchronous processes are exactly the reverse ones.

The synchronicity of the processes has also consequences for the determinism of the resulting parallel branch and bound algorithm. If the processes are not synchronized, the division of the work among the processes and the exchange of the knowledge obtained introduce nondeterminism in the resulting algorithm. In that case, nothing can be said about the exact order in which the extractions, additions and

eliminations in the pools of work are carried out 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 follow a different path, resulting in a different solution. Even if two consecutive executions of the algorithm yield the same solution, the work carried out during these executions might be completely different.
The nondeterminism does not change the properties of the subproblems in the active set. For all these subproblems it still holds that, as long as they are in a pool, they are neither branched from nor eliminated. Therefore the solution yielded by the asynchronous parallel algorithm is always a correct one.

2.2.2. An example of a parallel branch and bound algorithm

For our research, we were amongst others interested in the consequences of the high level parallelism introduced by the branching from several subproblems in parallel. We developed an asynchronous branch and bound algorithm which uses a single central pool for dividing the work among the processes. The basic unit of work is the branching from a single subproblem and the computing of lower bounds to the subproblems thus generated.
The algorithm introduces a master process and several slave processes (c.f. figure 1). The master takes all important decisions and the slaves do as they are told to do by the master. In particular, the master maintains the active set and decides which subproblems to branch from and when. The slaves perform the actual branchings and compute lower bounds to the subproblems thus generated.
In order to supervise everything properly, the master collects all knowledge 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 there are no active subproblems, the slave is temporarily put into a queue of idle slaves until new active subproblems 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.
A slave branches from the subproblem it receives from the master. The subproblems thus generated are sent to the master. After completing the branching, the slave requests new work from the master by notifying the master that it has become idle.
From now on, we will call this parallel branch and bound algorithm the Boulder algorithm. For a complete description of the algorithm we refer to [Trienekens, 1986]. Another example of parallel branch and bound algorithms can be found in [Finkel & Manber, 1987].

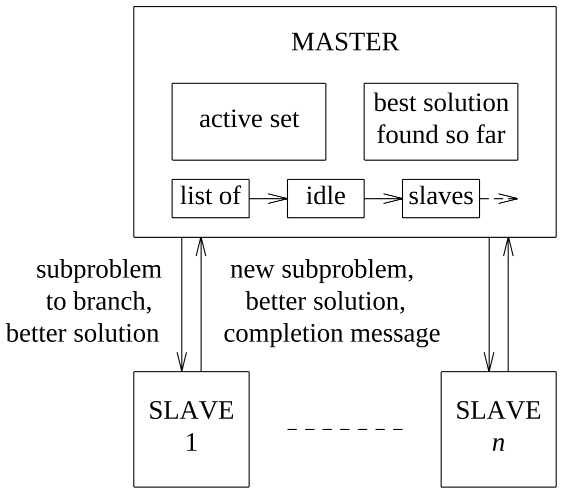
MASTER
active set
best solution found so far
list of
idle
slaves
subproblem to branch, better solution
new subproblem, better solution, completion message
SLAVE 1
SLAVE n

Figure 1. The Boulder algorithm

3. PREDICTING THE PERFORMANCE OF A PARALLEL ALGORITHM

3.1. Theoretical Analysis

In essence the execution of a branch and bound algorithm amounts to collecting knowledge about the problem instance to be solved and the use of this knowledge in solving the problem. The 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 how to continue the process of knowledge collecting, for example the choice of the next subproblem to branch from or the way a particular subproblem is decomposed into smaller ones, are based on the knowledge available at a given moment.

For sequential branch and bound algorithms it holds that the consequences of even a small change in the knowledge collected or the way this knowledge is used in deciding what to do next, need not be clear in advance. For example, the use of a dominance test or the use of another selection rule can generally speaking increase or decrease the time needed for executing the algorithm by an arbitrary factor [Ibaraki, 1976, 1977]. Most of the theoretical analysis done is concerned with the worst case performance of sequential branch and bound algorithms and with preventing anomalies from degrading the performance of the algorithm. An average case analysis is very hard because it is not clear in advance how a branch and bound algorithm will behave depending on the problem instance to be solved. Next to that, it is nearly impossible to state a probability density function over all possible problem instances. So in order to perform a theoretical analysis, many simplifying assumptions have to be made. For an example of such an analysis see [Smith, 1984].

A theoretical analysis of a parallel branch and bound algorithm is even harder because the parallelism introduces additional parameters, for example, the way the work is divided among the various processes or the way the knowledge is collected and used. It is possible that certain knowledge is available within a particular process, whereas other processes do not know of this knowledge because the algorithm does not provide a full knowledge exchange between all processes. As in the sequential case, anomalies and their prevention have been studied ([Lai & Sahni, 1984], [Li & Wah, 1984] and [Trienekens, 1986]). For a theoretical analysis even more simplifying assumptions than in the sequential case have to be made.

3.2. Empirical Analysis

Before we will start a theoretical analysis of parallel branch and bound algorithms we first wanted to get some feeling about the behaviour of parallel branch and bound algorithms and their parameters. Therefore we decided to experiment with some parallel algorithms on some given parallel machines.

In the second half of 1985 we were able to perform extensive computational experiments while visiting the University of Colorado at Boulder, Colorado. The Department of Computer Science of this university 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 they 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].
DPUP enables a process to create a remote process and to establish a communication link with it, to communicate with a remote process created previously and also to kill a remote process started previously thereby discarding the corresponding communication link. Using DPUP it is possible to build a system of parallel executing processes communicating via message passing. The user interface provided by DPUP is independent of the architecture of the actual machine the program is running on. In essence it provides two processes with a way to communicate with each other via message passing. DPUP itself does not specify how this communication is carried out in reality. For a more complete description of DPUP we refer to appendix A.

Our experiments with the DPUP system and the Boulder algorithm showed that for big problems and for a limited number of processors it is possible to get a speed up nearly equal to the number of processors used.
Measuring the performance of the algorithm was one of the most difficult problems we had to cope with. The execution of a parallel program turned out to be heavily influenced by other activities happening in the

system. If a system is executing other processes apart from the processes of the parallel program, it has less time for the parallel program. Also if there is much traffic on the network connecting the various processors, the communications between the processes of the parallel program can be interfered with. The real problem 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 environments mingle with the effects caused by the parallelism. The only way to arrange for 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.
The nondeterminism introduced by the asynchronicity also caused trouble. The division of the work among the processes varied heavily for different runs, and thus the time needed for execution of our parallel program varied as well. In order to get reliable results, we had to make several test runs for each problem instance and average the results.
We also encountered significant waste of computing power. In our experiments we repeatedly solved the same problem instance with the same algorithm, varying only the number or kind of processors. For each choice we had to solve the problem instance several times due to the nondeterminism. The only way to measure the time needed to solve the problem instance was to actually perform the calculations (i.e., branch from subproblems) in real time, so this had to be done over and over again.
For a complete description of the computational results, we again refer to [Trienekens, 1986].

Although the algorithm and the results from the Boulder experiment are interesting in their own right, they cannot serve as more than an indication of the results we are looking for. In our research we try to discover which parameters of parallel computer architectures, of parallel branch and bound algorithms and of problem instances are relevant for the decision what the best algorithm and the best architecture will be for solving the particular problem instance. There is a whole range of values these parameters can take. In Boulder the parameters were confined to very restricted subranges: at most eight processors, two different kind of processors and one basic architecture with a fixed communication speed for each pair of processors.
Repeating this experiment on another architecture will yield more insight, especially if this architecture has parameter values different enough from the ones in Boulder. However, there is the same fundamental drawback in that again a limited subrange of parameter values is investigated. Moreover, we are interested in applying massive parallelism using several hundreds of processors. Today there exists a small number of machines offering that much parallelism, for example the Hypercube and the NCUBE. But again, these machines cover only a small part of the broad spectrum of possible massively parallel architectures. In other words, there are values of parameters that are worth investigating, but for which no hardware has been built yet.

To circumvent all these problems we have decided to use *simulation* as a tool for our investigations. Using simulation we will be able to exert better control on the parameters of interest. Simulation also has the advantage that we avoid the interference from the environment, which we alluded to above.

Another advantage of the decision to use simulation is that we now can combat the waste of computing power. While using simulation, we deal with simulated time instead of real time. We can therefore use a database technique: instead of repeating a calculation, a simulated process can inspect a database containing all subproblems which have already been branched from before, i.e., in the runs up to this particular run. The database will contain the ’solution’ of all these subproblems, and also the time needed for computing them. So if the subproblem is in the database, the simulated process will give back the results found there and pretend that its calculation took the amount of time indicated in the database. Thus, while not satisfied with the fact that due to the limited number of processors available we could not solve really big problems, we now propose to use simulation on only one processor!

4. USING SIMULATION AS A TOOL

4.1. Decisions on How to Simulate

The first decision we made was to construct a simulator which is as general as possible. By this we mean that our simulator should be capable of simulating the execution of all kinds of parallel programs on all kinds of hardware configurations, and not just of branch and bound programs in the style of the Boulder algorithm on the Boulder hardware configuration. A special purpose simulator for the Boulder algorithm might have been easier to write, but incorporates the hidden danger that in the end it will not be able to cope with some interesting new combinations of parameter settings and/or hardware configurations.

The second decision we made concerned the communication primitive(s) to be used in the simulator: message passing, shared memory or a combination of these. For a description of these primitives see [Peterson & Silberschatz, 1985]. From a semantic point of view these primitives are all equivalent. Given a message passing primitive, a shared memory primitive can be implemented and vice versa. Of course the speed and/or efficiency with which the various communications are performed will be influenced, we only state that these primitives are able of performing the same work.
Because all parallel branch and bound programs we have written hitherto use message passing as communication primitive, we decided that the simulator should also use this primitive. However, the design of the simulator should be general enough to realize an efficient implementation of shared memory in a later stage (if needed for our research).
The user interface we have chosen for the message passing primitive is the DPUP user interface. Note that there is nothing special about DPUP itself. We needed an interface offering communication based on message passing and the possibility to dynamically create and kill processes. DPUP is an example of such an interface capturing all the essentials. Notice further that DPUP does not imply anything about the underlying hardware configuration. In fact, the actual hardware configuration to be used is one of the parameters which must be fed to our simulator before it can start to simulate.

A consequence of the choice for the DPUP user interface was that we implemented the same interface (set of routines which can be used in our programs) in our simulator that we had at our disposal in Boulder. While building the simulator we have managed to achieve a high degree of transparency: apart from some minor differences the semantics of the simulator's DPUP routines is the same as the original DPUP semantics. Only a small adaptation of the code we used in Boulder was needed in order to be able to run these programs under our simulator. This means that we could benefit again from the investments in programming effort made in Boulder.

The real DPUP package is written in C to be executed under the Berkeley Unix operating system. As a consequence we had to use C and Unix for our simulator as well. This worked out quite well: the C/Unix combination proved to be a useful system offering good facilities for simulations. In particular the fork/exec mechanism and the pipe facility were useful tools. In the next sections we will elaborate on this aspect.

4.2. An Event Driven Simulation

Using the DPUP system it is possible to create a number of processes, running in general independently of each other. The only way a process can influence another one is through calling a DPUP routine. Therefore the simulator needs to influence the execution of a process only when this process issues a DPUP call. This suggests that such a system can best be simulated using the event driven method [Neelamkavil, 1987], where the events are the DPUP calls.
The event driven method is feasible here because execution of a process *P* can be influenced by other processes only at those points in time where *P* itself has issued a DPUP call, that is where *P* has generated an event and is therefore stopped by the simulator. In other words, *P* cannot be interrupted between events that it generated itself. However, there is one exception to this rule: *P* might be killed by another process. But, if some care is taken, the simulator will be able to roll back the actions that *P* has announced but which it has not been able to realize as yet.

We now give a, rather abstract, description of how this event driven simulation can be set up for the DPUP system and why it works. For more information on simulation, we refer to [Neelamkavil, 1987]. First a few definitions.
We divide an execution of a process in *slices*, where a slice is defined as a fragment of such an execution between two successive DPUP calls issued by the process itself. The *result of a slice* is a triple consisting of an *event*, which is the DPUP action the process is about to undertake at the end of the slice, an *event time*, which is the time at which the slice ended, and the *state* of the process at this point of time. Such a

state consists of the value of the process variables and control information, e.g., the program counter. For every point of time t_0 and for each process in the system we define the t_0-*slice* of this process as the unique slice that starts before t_0 and ends at time t_0 or later.

Suppose the simulation has been performed correctly up to time t_0. Thus all DPUP calls executed before t_0 are known and therefore for all processes the state at the beginning of their t_0-slice can be determined. However, due to the above mentioned exception it is in general not possible to derive from this knowledge the result of the t_0-slice of a process *P* by simulation of *P* alone. This is so because it is possible that another process might kill process *P* at a time *t* later than t_0 but before *P* would have issued the DPUP call terminating its t_0-slice.
On the other hand, there is sufficient knowledge to derive the result of the t_0-slice of each process in the system under the assumption that from t_0 on the process is not killed by another process. Or in other words, taking into account only the effects of DPUP calls that occurred before t_0. Let us call such a result a t_0-*correct result.* Our conclusion is, that if simulation of the system has been performed correctly up to time t_0, that in that case the simulator must be able to build a list containing the t_0-correct results of each process still alive at t_0

The simulator is built as follows. It consists of one main loop, and at the start of each iteration the simulator has built a list containing the *t*-correct results of each process in the system, where *t* is the event time of the oldest result in the list. The oldest result in the list is defined as the result with the smallest event time. The above reflects the fact that the simulation has been performed correctly up to (simulated) time *t*. In each iteration of the loop this list is transformed into a list of t'-correct results, where t' is the time the next DPUP action would occur in the simulated system.
Each iteration starts by extraction of the oldest result from the list. Say, this result is built up from a DPUP call *D*, issued by process *P* at event time *t*. All results in the list are *t*-correct, so the oldest result is *t*-correct as well, which means that in reality the event of this result would also have taken place at time *t*.
It is therefore safe to start simulating the execution of this event, the DPUP call *D*. Notice that it may be possible that this call cannot be simulated to completion, for example, in case *D* being a receive request for which the matching data have not been sent as yet.
Completion of *D* might have its effect on the result list; for example a kill request can remove an entry from this list or a send request can generate an event "receive request can be completed" in the case the receive request was issued earlier. Notice that an event driven simulation is feasible if the simulator is able to simulate the execution of an event only by adjusting the entries in the result list, i.e., only if execution of an event does not influence future events in a too complicated way.
The simulator can determine whether the DPUP call *D* can be completed and if so, what completion of this call *D* will mean for the state of *P*. For example, a receive request will lead to new data for the requesting process. In such a case, it is possible for the simulator to simulate the next slice of *P*, assuming that *P* will not be influenced by future DPUP calls. This yields a new result, which is added to the result list. Suppose the oldest result in this new list has event time t'. The list now has the property that all results occurring in it are t'-correct, that is, the loop invariant has been established again, and the next iteration can start.
As long as the result list is not empty, there are still DPUP calls to be simulated and thus the simulation is not finished. Therefore the main loop of the simulation terminates only if this list is empty.

4.3. The C/Unix Implementation

From the description given above it is clear that in order to simulate a DPUP system in this way, the simulator must be able to manipulate process images: the simulator must be able to stop execution of a process, save the state of this process and restart it later in the same state as it was stopped in.

A simulation language like SIMULA [Birtwistle et al., 1979] provides *coroutines* to realize this. Coroutines are program modules, like Pascal procedures, that can contain *resume* statements. Such a statement, when executed, has the effect that the coroutine in which this statement occurs is stopped and saved by the SIMULA run time system, and that another coroutine, specified in the resume statement and suspended earlier, is restarted from the state in which it was suspended.

In multitasking operating systems like Unix, where many processes compete for a single processor, a similar phenomenon can be observed. At every point in time at most one process can be running. All other processes must be temporarily stopped. The difference with SIMULA is that in SIMULA a process (coroutine) can block itself, while in Unix the operating system takes the decision on when a process will be blocked. However, a process can force the operating system to block it by asking something the operating system cannot do at the moment, for instance by asking for input which is not yet there. Such an indirect resume mechanism is employed in our simulator.

We use the fork/exec mechanism from Unix which enables a process to start up new processes. Communication between these processes can be established using pipes. Pipes are FIFO (First In First Out) queues. The idea is that one process writes on the pipe and another process reads the data from this pipe in the same order the first process has written them. Pipes are unidirectional, so if two processes want to communicate in both directions there are two pipes needed. A process can always write on a pipe, but if a process wants to read something from an empty pipe it is blocked until the other process writes something on the pipe.
To be more specific, during simulation of a system of DPUP processes, each simulated process will be represented by a Unix process. One additional process will be present, the simulator process. Each simulated process is connected to the simulator process via two pipes, one for reading and one for writing.

Therefore simulation of the Boulder algorithm will lead to a configuration of Unix processes as depicted in figure 2. Notice the difference with figure 1. Now there are no direct connections between the simulated processes, every communication (i.e., DPUP call) proceeds via the simulator process.

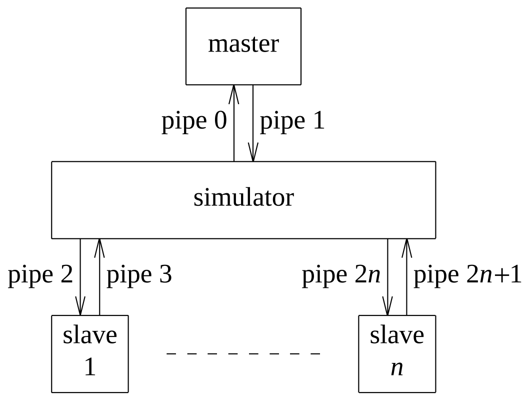

Figure 2. The Boulder algorithm executed by the simulator

We indicated above that our simulations are organized such that at most one simulated process will be active at a time, all other processes will be blocked. It is even possible that all simulated processes will be blocked. In that case the simulator process will be active, e.g., updating internal tables or finding out which simulated process must be restarted next. Another property of our simulations is that at any point of time each blocked simulated process is stopped by the operating system on trying to read from its input pipe connected to the simulator process, which is empty. Therefore the simulator process is able to restart any simulated process simply by writing on their common pipe, thus delivering the message the simulated process is waiting for.
The simulator uses a similar trick to block itself. Because at most one simulated process can be active at any point of time, and because the simulator will be activated only through a DPUP action of this active process, the simulator process can schedule itself by trying to read a new message from the as yet empty pipe connected to the active simulated process.

The main part of the real DPUP system is the set of DPUP routines to be used by the DPUP programs. The body of these routines contain the Unix system calls through which the desired effects will be accomplished. One of our goals is that DPUP programs can be used (nearly) unaltered in our simulation environment as well. To this end, our simulation package consists of the program executed by the

simulator process and a new set of DPUP routines, the bodies of which are rewritten so that in our case the necessary communications, as outlined above, are performed over the pipes to and from the simulator process.
In appendix B we present a more technical description of the implementation of de DPUP calls in the simulator and give an example of its precise workings.

4.4. Some Comments and Ongoing Developments

In the above description we did not specify how simulated time was determined, e.g., the time needed for a process to execute a slice or the time needed for the communication network to transfer a message from one process to another one. In the simulator as it stands now, we let the simulated process determine how much time it took to complete its slice. Communication time is taken to be zero. To this end the simulator sends, on restarting a process, the simulated time at which the process should be restarted according to the event list to this process. When transmitting a new DPUP action to the simulator at the end of a slice, the simulated process sends also the updated simulated time. This new time is determined by adding the amount of (real) time it took the process to execute its slice to the time it received from the simulator.
All kinds of extensions to this scheme are possible. For instance the simulator can manipulate the updated time it received from a process, for example, the simulator can declare one processor to be twice as fast as another one. It is also possible to increase the event time of each communication request, thus simulating a delay caused by communication over the network.

A severe drawback of the simulation as described above is that simulation of massive parallelism is still not possible because of constraints imposed by the Unix operating system. First of all, there is a maximum on the total number of processes in the system and on the number of processes a single user may have at his disposal. Another constraint is that the number of pipes allowed for one process is restricted. This makes the simulator process a bottleneck, because this process must be connected by two pipes to each simulated process.
However, there is an observation that might help us circumvent these constraints. If there are very many processes, most of these processes must be (nearly) identical. Consider for instance the Boulder algorithm again. All slave processes are essentially the same. The idea is now to use only a single Unix process that simulates all slave processes. The picture is again changed; instead of figure 2 we now obtain figure 3.
This is not a straightforward solution, though. The point is that the single Unix process cannot start impersonating another slave process at the end of each slice, but only at a fixed point in its main loop. This is so because after simulation of, say, slave 1 the simulator might decide to restart slave 2. This will be accomplished by sending the "generic slave" a message on its pipe, which will cause this process to execute from the point in the program where slave 2 was stopped. This had better be the same point where slave 1 was stopped as well!
So a difference must be made between points in the program where a slave only generates a DPUP action and points which are swap points as well. A *swap point* is the (only) point in the program where the simulator may decide to run another simulated slave.
This has a few consequences. First of all simulation is no longer invisible to the user. The programmer has to specify which processes can be simulated by generic processes and what the swap point will be in such a generic process. Secondly, correctness of the simulation is not guaranteed that easily any more. For instance, deadlock will occur if a slave wants to read input from another slave which could start its main loop (that is pass its swap point) later in time than the first one does. The first slave would first be simulated by the generic slave, and subsequently blocked in a receive request for which the corresponding send request has not yet occurred. However, in order to generate this send request, the receiver would have to "release" the generic slave, so that the writer slave could get its turn. This is a deadlock: a process possessing a resource (the generic slave process) is waiting for another resource (a message) which is held by another process and can be released only if the other process obtains the first resource as well.
Nevertheless, it seems that for a large class of processes such a solution with generic processes is feasible. In particular this holds for the slave processes in the Boulder algorithm. At the moment we are investigating under what conditions simulation using these options is guaranteed to deliver correct results.

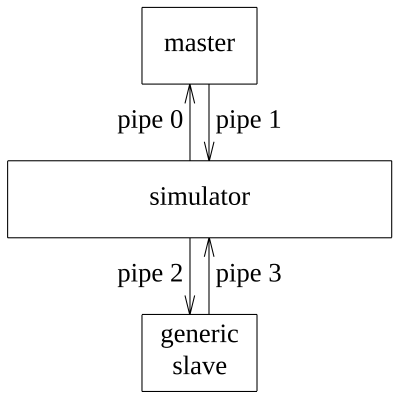

Figure 3. Simulation of the Boulder algorithm using a generic slave:
only 3 processes instead of $n+2$ and only 4 pipes instead of $2n+2$!

While preparing this paper we discovered that a very similar project has been conducted at the University of Wisconsin at Madison, Wisconsin, i.e., the MU project [Livny & Manber, 1987]. Their main goal was to build a user environment in which amongs others the development and debugging of parallel programs would be simplified. To this end they built an event driven simulator based on the primitives offered by the programming language Modula2. In this package dynamic process creation and killing and generic processes have not been included though.

4.5. Some Computational Results

In this section we will give a short example of the use of the simulator in evaluating the performance of the Boulder algorithm. We also will supply some statistics on the time needed for simulating.
We are interested in the consequences of the number of processes the algorithm is executed by and in the consequences of the cost of communication versus the cost of computation for the behaviour of the algorithm. A simple analysis of the algorithm shows that if the number of slave processes or the cost of communication versus the cost of computation increases, a point will be reached at which the master process does not have enough power to provide all slave processes with new work as soon as they become idle. Points of interest are when exactly this will start to happen and what the consequences of this bottleneck will be. Does it only have local consequences (i.e., some slaves are wasting part of their computing power, but the algorithm still runs as fast as before) or does it have global consequences for the algorithm as a whole (i.e., solving the problem will take more time).

machine	setup time in sec	communication time in millisec/byte
zero	0.0	0.0
alfa1	0.256	0.0
alfa2	0.512	0.0
alfa3	1.024	0.0
beta1	0.0	1.6
beta2	0.0	3.2
beta3	0.0	6.4
beta4	0.0	12.8

Figure 4. Machine descriptions

Computational experiments with the Boulder algorithm on the DPUP system of the University of Colorado at Boulder showed no bottlenecks at all. This does not imply that the above mentioned bottleneck will not occur in real life. It only implies that the possible parameter settings of this parallel system are too restricted. Finding a parallel machine not so restricted in its parameter settings is a not so easy

machine	number of slave processes						
	1	2	4	8	16	32	64
zero	324:11	165:22	86:14	47:29	28:29	21:04	20:58
alfa1	345:18	175:59	91:44	50:26	30:18	22:52	25:48
alfa2	366:27	186:51	97:16	53:36	32:55	29:04	35:26
alfa3	408:42	208:49	108:48	61:00	43:53	47:29	55:55
beta1	330:49	168:42	88:00	48:24	29:10	21:42	23:28
beta2	337:26	172:09	89:49	49:35	30:15	25:20	32:54
beta3	350:44	179:00	93:40	52:31	37:59	41:01	49:57
beta4	377:12	193:43	102:36	66:12	63:38	69:34	88:45

Figure 5. Time needed to solve the test problem (in minutes)

machine	number of slave processes						
	1	2	4	8	16	32	64
zero	271	271	271	272	275	289	351
alfa1	271	271	271	272	275	288	329
alfa2	271	271	271	272	276	288	315
alfa3	271	271	271	272	274	287	309
beta1	271	271	271	272	276	288	330
beta2	271	271	271	272	276	288	320
beta3	271	271	271	272	276	287	314
beta4	271	271	271	272	273	286	314

Figure 6. Total number of subproblems branched from

problem. We therefore decided to use the simulator.
The parallel machines to be simulated each contain enough processors to execute the Boulder algorithm with the desired number of slave processes. All pairs of processors can communicate via a point to point connection. We defined three series of simulated parallel machines. The *zero* series consisted of a single machine for which all communication costs were equal to zero. The *alfa* series consisted of three machines for which only the initiation of a communication took a fixed amount of time, whereas all other communication costs were equal to zero, i.e., for which the total cost of a single communication is a constant, independent of the size of the data to be sent/received. Finally, the *beta* series consisted of four machines for which the the cost of communicating is a linear function of the number of bytes to be sent/received. In all series, the communication costs were chosen in such a way that the bottleneck in the Boulder algorithm would become clearly visible with increasing cost and/or increasing number of slave processes. Note that while defining the machines we only have to state the communication costs. The only issue of interest is the ratio between communication and computation costs. Without loss of generality the computation costs can be chosen to be equal to one (i.e., the speed of the machine the simulator is actually running on). We refer to figure 4 for the details of the various machines.
The test problem we used was a 75 city euclidean traveling salesman problem which was created by generating at random points in a two dimensional space (problem e75b from [Trienekens, 1986]). The algorithm we used for solving this problem was based on the minimal spanning 1-tree relaxation described in [Jonker & Volgenant, 1982]

Figure 5 shows the time needed for executing the Boulder algorithm under the assumption that the processors in the parallel machine simulated are all AT&T 3b2/300's (the machine we used for our simulation runs). The database technique described in section 3.2 led to significant savings in time: if during simulation all subproblems could be found in the database, the simulation would take somewhere between 10 and 11 minutes. Note that the time needed for solving the problem on machine zero using only one slave is a lower bound on the total amount of computations to be done. All simulations used a generic slave process.

machine	number of slave processes						
	1	2	4	8	16	32	64
zero	0:08 (324:03)	0:08 (165:13)	0:08 (86:06)	0:09 (47:20)	0:09 (28:20)	0:10 (20:54)	0:13 (20:45)
alfa1	10:46 (334:32)	10:48 (165:11)	10:50 (80:54)	10:50 (39:36)	10:56 (19:22)	10:43 (12:09)	12:36 (13:13)
alfa2	21:25 (345:03)	21:27 (165:24)	21:29 (75:47)	21:12 (32:24)	20:16 (12:38)	19:20 (9:43)	23:31 (11:55)
alfa3	42:41 (366:01)	42:42 (166:06)	42:18 (66:29)	40:30 (20:29)	35:49 (8:04)	38:55 (8:34)	44:22 (11:33)
beta1	6:45 (324:03)	6:47 (161:55)	6:49 (81:10)	6:56 (41:28)	7:14 (21:57)	7:51 (13:51)	10:03 (13:25)
beta2	13:22 (324:04)	13:25 (158:44)	13:30 (76:19)	13:39 (35:56)	14:15 (16:00)	15:42 (9:38)	20:01 (12:53)
beta3	26:36 (324:09)	26:43 (152:17)	26:45 (66:55)	27:17 (25:14)	28:21 (9:38)	32:00 (9:02)	38:02 (11:55)
beta4	53:04 (324:09)	53:16 (140:27)	53:22 (49:14)	54:23 (11:49)	55:59 (7:39)	61:34 (8:00)	77:48 (10:57)

Figure 7. Busy time master (idle time master)

Figure 6 shows the total number of subproblems branched from. Figure 7 shows the time utilization of the master process, i.e., the total time the master was busy (computing and communicating) and the total time the master was idle. If the master is unable to immediately serve a request of a slave, computing power of the slave will be wasted. The ratio of the busy time and the idle time of the master gives an indication of the probability that the master is able to respond immediately to a request of a slave. Figure 8 sheds some more light on this topic by displaying some frequency counts of the number of requests simultaneously received by the master. As soon as the master has completed the handling of all pending requests it checks whether there are any new requests to be served. The master can only serve one request at a time, therefore if there is more than one request pending, the serving of all other request is postponed. As a consequence the corresponding slaves will be idle for some more time and computing power will be wasted.
The figures clearly show that the bottleneck we could not observe in Boulder has now been made visible. When the number of slave processes and communication costs increase the master is unable to keep up with the rate of arrivals of requests from the slaves.

As a final remark we note that even the nondeterminism of an execution on a real parallel system has been preserved during simulation. Due to the fact that the length of a slice is measured by the real cpu time clock of the processor running the simulator, the timings of the various processes are influenced by environmental factors as the real processes would be.

Acknowledgment
The authors would like to thank Hans Kraamer and Wim Pijls for being a stimulating first audience for these simulation plans.

References

G.M. Birtwistle, O.J. Dahl, B. Myhrhaug, K. Nygaard (1979). *Simula Begin.* Studentlitteratur.
R.A. Finkel, U. Manber (1987). DIB - A Distributed Implementation of Backtracking, to appear in *ACM*

machine	number of slaves	number of simultaneously received requests	frequency count
zero	1	1	649
	64	1	957
		2	13
alfa3	1	1	649
	2	1	632
		2	10
	4	1	560
		2	45
		3	2
	8	1	315
		2	114
		3	28
		4	9
		5	1
	16	1	29
		2	21
		3	19
		4	6
		5	9
		6	7
		7	5
		8	0
		9	4
		10	4
		11	7
		12	13
		13	7
		14	1

Figure 8. Frequency count of simultaneously received requests

Transactions on Programming Languages and Systems.

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). Computational Efficiency of Approximate Branch-and-Bound Algorithms. *Mathematics of Operations Research, Vol. 1, No. 3.*

T. Ibaraki (1977). The Power of Dominance Relations in Branch-and-Bound Algorithms. *Journal of the ACM, Vol. 24, No. 2.*

R. Jonker, T. Volgenant (1982). A Branch and Bound Algorithm for the Symmetric Travelling Salesman Problem Based on the 1-Tree Relaxation. *European Journal of Operations Research*.

G.A.P. Kindervater, J.K. Lenstra (1985). Parallel Algorithms. M. O'hEigeartaigh, J.K. Lenstra, A.H.G. Rinnooy Kan (eds.). *Combinatorial Optimization: Annotated Bibliographies*, Wiley, Chichester, Ch. 8.

G.A.P. Kindervater, J.K. Lenstra (1986). *Parallel Computing in Combinatorial Optimization*, Report OS-R8614, Centre for Mathematics and Computer Science, Amsterdam.

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 Algorithms*, School of Electrical Engineering, Purdue University, Report TR_EE 84-6.

M. Livny, U. Manber (1987). *Simulation of Distributed Algorithms for Local Area Networks*, Department of Computer Science, University of Wisconsin, Madison, WI.

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

F. Neelamkavil (1987). *Computer simulation and modelling.* John Wiley & Sons.

J.L. Peterson, A. Silberschatz (1985). *Operating System Concepts.* Addison-Wesley Publishing Company.

D.R. Smith (1984). Random Trees and the Analysis of Branch and Bound Procedures. *Journal of the ACM, Vol. 31, No. 1.*

H.W.J.M. Trienekens (1986). *Parallel Branch and Bound on an MIMD System.* Econometric Institute, Erasmus University Rotterdam, Report 8640/A.

Appendix A

The DPUP User Interface

In this appendix we will describe the user interface of the message passing part of DPUP. For a complete description of DPUP we refer to [Gardner et al., 1986].

Firstly, DPUP lets a process create an other process and establishes a communication link between them.

DP_CREATE_PROCESS(process_name, machine). The parameter "process_name" corresponds to the new process which must be created on the processor described by "machine". This routine returns a *socket*, a descriptor of the communication link, to the newly created process. The effect of execution of this call is that a new process is added to the system.

DP_SETUP(). This routine is the first action a newly created process should undertake. It returns the socket of the communication link to the father of the newly created process, i.e., the process that requested DPUP to create this new process.

Secondly, DPUP offers three communication primitives, one for output and two for input.

DP_WRITE(socket, buffer, buffer_size). This is the output call. The effect is that the message with size "buffer_size" which is stored in "buffer" will be forwarded to the process listening at the communication link described by "socket". Output is asynchronous, that is even if the intended receiver is not willing to communicate with the DP_WRITEr, the call of DP_WRITE terminates. In such a case, the message is stored somewhere by the DPUP communication system until the receiver wants to receive it.

DP_READ(socket, buffer, buffer_size). This is the corresponding input call. If there is a message waiting in the communication link described by "socket", this message is copied to the variable "buffer". If there is no message waiting, i.e., if there is no message in the communication link because the process at the other end of this link has not issued a corresponding DP_WRITE, the process doing the DP_READ is blocked. The call of DP_READ terminates only if there is a message of the given size.

A system in which input can be performed through DP_READs only is not satisfactory. Consider the situation that a process is waiting for messages from several other processes, but doesn't know which message will arrive first. Using DP_READ the process can only indicate that it is interested in a message of a particular other process, because there can only be one sender specified in such a call. Therefore the receiving process will be deaf for messages from the other processes until the message from the specified sender is received.
What is needed here is a mechanism through which the receiving process will be signaled as soon as a message of one of the other processes becomes available. That is what the DPUP routine DP_SELECT is for.

DP_SELECT(candidates, senders). The first parameter is a list of sockets. If a socket is in this list, the DP_SELECTer wants to be signaled as soon as a message will be available on the corresponding communication link. Note that such a message could already have arrived somewhere in the past. The second parameter is a variable which on return will have been set by DPUP to a sublist of the list in "candidates", containing the sockets of all communication links that contain a message to the DP_SELECTer. The DP_SELECT call will block if there are no candidates willing to communicate. In that case it will return later, as soon as a process has DP_WRITten a message to one of the candidate sockets and the corresponding message has arrived.

Finally, DPUP enables a process to kill another process or itself.

DP_KILL_PROC(socket). This is the counterpart of DP_CREATE_PROCESS. The process at the other end of the communication link described by the given socket must be killed.

DP_EXIT(). This routine is used to communicate to DPUP the fact that a process will terminate its execution. This routine is needed, because the system needs to know whether a process is still able to receive messages.

As an example of the use of DPUP, we now will describe the Boulder algorithm in terms of the DPUP routines.
The master starts by issuing a couple of DP_CREATE_PROCESSes, thereby creating a number of slaves, all alike and executing the same code.
Each slave starts off with a DP_SETUP in order to get the communication link to its father (the master). Thereafter each slave enters a loop. Each iteration starts with a DP_READ, using which the slave determines what to do next. If the master sends a better upper bound, the slave stores this bound for later use. If the master sends a subproblem to be branched from, the slave computes all descendants of this subproblem. If a descendant can be solved to optimality, the slave checks the value of this solution against the value of the best solution received hitherto from the master. If the new solution is better, the slave DP_WRITEs it to the master. If a new subproblem is not solved to optimality, the slave checks the lower bound against the value of the best solution it knows of. If this lower bound is better, the slave DP_WRITEs the new subproblem to the master. Finally, if the slave has completed its job, it DP_WRITEs to the master that it is ready to accept new work and it starts a new iteration.
The master, having created all slaves, initializes the list of active subproblems, sends the original problem to one of its slaves, sets up its administration indicating which slave is working and that all other slaves are idle, and then enters its main loop. Each iteration starts with a DP_SELECT on all slaves, whereby the master tries to find out the slaves which have a result to report. If the call of DP_SELECT returns, there is at least one slave with a message. The master selects one of these and does a DP_READ for getting its result. This result can be a better solution, a new subproblem or a completion message. In the first case, the master updates its record of the best solution found so far (a check is needed here!) and removes all active subproblems from the list which can be killed by this new solution through a lower bound test. In the second case, the master checks whether there are idle slaves, and if so, sends the new subproblem to one of these to branch from. If not, the master puts the new subproblem in the list of active subproblems. In the third case, the master takes the most promising subproblem from the list of active subproblems, if this list is not empty, and DP_WRITEs this subproblem to the slave. If the list is empty and the slave was the last one being active, the computation is apparently finished. The master leaves its main loop, reports the answer, DP_KILLS all its slaves and finally exits itself. If there are active slaves left, the master simply adds the sender to the list of idle slaves.

Appendix B

Implementation of the DPUP calls in the Simulator

In order to be able to perform a simulation as described in chapter 4, the simulator process has to maintain several data structures. First of all there is the event list, which is in essence the result list mentioned in the abstract description given earlier, apart from the fact that all low level information on the state of a suspended process has been removed - Unix now handles this on our behalf. There is also a message list which contains all messages that have not yet been accepted by the corresponding destination processes, but for which the DP_WRITE has already been handled completely, that is, this DP_WRITE is no longer on the event list. It might be possible that the DP_READ that will in the end accept the message is already in the event list, but it can also be the case that the corresponding DP_READ event is not yet generated. Finally the simulator will also maintain a process list containing an entry for each simulated process in the DPUP system.
Simulation of an event is done in two stages. At the end of simulation of a slice the simulator process receives the event which ends the slice. This event is placed in the event list together with the event time. This is the only action taken by the simulator in the first stage. No more work is done as yet, because of the possibility that the event will be canceled later during the simulation. If this happens, it is straightforward to remove the event from the list and all traces of it will be gone. For instance, if the event is a DP_WRITE the simulator will not read the message during this stage from the pipe where the sender has put it.
The second stage is started when the event is the earliest event in the event list, so that it is its turn to be simulated. Now it is certain that in reality this event would have occurred as well. Therefore it is safe to update all data structures in the simulator to reflect this event. In the DP_WRITE example, the simulator now reads the message from the pipe and puts it in the message list.

We now give a more detailed description on how the various DPUP routines mentioned in appendix A have been implemented in our simulator.

DP_CREATE_PROCESS. The creator blocks itself, as usual in a call of a DPUP routine, trying to read from the pipe the socket to be used in communicating with the process to be created. As already mentioned, simulation of an event like this proceeds in two stages. In the first stage, the simulator puts the create process request in the event list. This event should cause two actions when it will be rescheduled: a new process should be started and the creator should be restarted by sending it the socket to communicate with the new process. This is done as follows. In the second stage, first a new entry is made in the process table for the new process, then a new event is created (restart creator and send it the socket to communicate with the new process) which is put in the event list with the same event time as the one being handled. After that the simulator starts the new process through the fork/exec mechanism, having set up a pair of pipes between itself and the new process. When simulation of the first slice of this new process has ended, the event which leads to restarting the creator process will be the first one in the event list, so the execution of this process will continue immediately afterwards.

DP_SETUP. The simulator has created the process issuing this call while handling a DP_CREATE_PROCESS. In the fork/exec call the simulator has passed the pipes set up for the new process to communicate with the simulator and the socket to communicate with the creating process as parameters to the new process. DP_SETUP makes the pipes invisible for the user and finally returns the socket.

DP_WRITE. Before blocking itself, trying to receive an acknowledge message from the simulator, the writer sends two pieces of information to the simulator. The first part is an indication that a DP_WRITE is encountered by the process, the second part contains the message itself. As we mentioned before, during the first stage of the handling of this event, only the first part is read from the pipe. The second stage starts with the simulator reading the message itself from the pipe and storing it in the message list.
Then the simulator investigates in its process table the status of the intended receiver of this message. Further action is taken only if the second stage of a read or a select request has been handled on behalf

of the receiver, specifying the sender as a (potential) acceptable candidate, and if at the time of handling there was no message satisfying the read or select request. This situation will occur if in reality the receiver would also have been blocked on this DP_READ or DP_SELECT call because there was no message available at that moment. The action the simulator will take in this case is that an event will be created with the same event time as the DP_WRITE, which will make the receiver runnable again by sending it an indication that a message is available.

DP_READ and *DP_SELECT*. During the second stage of the simulation of this request the simulator searches the message list to find out whether there are acceptable messages which have already been DP_WRITten to this process. If so, the corresponding information (the message or a list of senders respectively) is sent back to the requestor. If there are no adequate messages in the list, the simulator updates the process table entry of the requesting process, indicating the request it has just made. The requesting process can not be restarted, so the simulator starts a new iteration of its main loop.

DP_KILL. The murderer asks the simulator to kill the process at the other end of the given communication link and blocks itself waiting for an acknowledgement. During the first stage the simulator only puts the event in the event list. Notice that no action can be taken as yet, because it is possible that the killer itself will be killed. Therefore the kill is executed only when the second stage starts. The simulator removes the process to be killed from the process table (and also removes all messages sent to but not yet accepted by the process to be killed). Finally the simulator will end the process using the Unix "kill" system call.

DP_EXIT. This is the suicidal counterpart of DP_KILL. It leads to similar actions within the simulator.

Notice that as far as the calling program is concerned, it does not matter that the DP_READ request was handed to and delayed by a simulator process. After the call has been completed, the only thing the caller is aware of is that the message from the master has appeared in its buffer. The caller has no way to find out whether communication with the master process was direct or indirect.

By way of an example, consider a simulation of the Boulder algorithm, and suppose a new iteration of the simulator's main loop, as described earlier, is about to start. Suppose furthermore that in this iteration slave 1 will become active. Process slave 1 was stopped earlier inside a DPUP call, say a DP_WRITE. In that case slave 1 is blocked because it tries to read from pipe 3 an acknowledgement message from the simulator, something that the simulator refused to send up till now.
However, now slave 1 can be restarted, so the simulator writes the acknowledgement on pipe 3 and blocks itself by trying to read a message from pipe 2. Slave 1 is no longer blocked now and starts executing until it again calls a DPUP routine, say a DP_READ which wants to read something from the master process.
The body of the DP_READ procedure sends a read request to the simulator on pipe 2, thus restarting the simulator. Slave 1 subsequently blocks itself trying to read the answer it expects from the simulator on pipe 3. The simulator reads the message from pipe 2, decodes it and updates its internal tables, e.g., the result list. This ends an iteration of the main loop of the simulator.
In one of the next iterations simulated time will have proceeded far enough for slave 1 to be rescheduled again and the simulator sends slave 1 the message from the master it was awaiting on pipe 3. In this way the action "read from pipe 3" executed inside the body of the DP_READ call will finally terminate. In the body of DP_READ the message just received is copied to the buffer specified as a parameter in the DP_READ call and DP_READ returns.