Towards an Abstract Parallel Branch and Bound Machine

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

Abstract

Many (parallel) branch and bound algorithms look very different from each other at first glance. They exploit, however, the same underlying computational model. This phenomenon can be used to define branch and bound algorithms in terms of a set of basic rules that are applied in a specific (predefined) order.

In the sequential case, the specification of Mitten's rules turns out to be sufficient for the development of branch and bound algorithms. In the parallel case, the situation is a bit more complicated. We have to consider extra parameters such as work distribution and knowledge sharing. Here, the implementation of parallel branch and bound algorithms can be seen as a tuning of the parameters combined with the specification of Mitten's rules.

These observations lead to generic systems, where the user provides the specifications of the problem to be solved, and the system generates a branch and bound algorithm running on a specific architecture. We will discuss some proposals that appeared in the literature.

Next, we raise the question whether the proposed models are flexible enough. We analyze the design decisions to be taken when implementing a parallel branch and bound algorithm. It results in a classification model, which is validated by checking whether it captures existing branch and bound implementations.

Finally, we return to the issue of flexibility of existing systems, and propose to add an abstract machine model to the generic framework. The model defines a virtual parallel branch and bound machine, within which the design decisions can be expressed in terms of the abstract machine. We will outline some ideas on which the machine may be based, and present directions of future work.

Note: This paper has been submitted for publication in 'Solving Combinatorial Optimization Problems in Parallel', LNCS, Springer, Berlin.

1 Introduction

Branch and bound algorithms solve optimization problems by applying a small set of basic rules within a divide-and-conquer-like framework. The framework is about the same in all applications, whereas the specification of the rules is problem dependent.

Report EUR-CS-95-04
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

This observation forms the main motivation for our paper: an automatic generation of the framework, tailored to the problem to be solved.

Roughly speaking, branch and bound methods generate search trees in which each node corresponds to a subset of the feasible solution set. A subproblem associated with a node is either solved directly, or its solution set is split, and for each subset a new node is added to the tree. The process is improved by computing a bound on the solution value a node can produce. If the bound is worse than the value of the best solution found so far, the node cannot produce a better solution, and, hence, it can be excluded from further examination. The order in which the nodes are selected for evaluation may be arbitrary, but a well chosen specific order (e.g., depth first or best bound) will generally reduce the computational effort considerably. We refer to Corrêa & Ferreira [1995] for a more detailed description of branch and bound and the formalism to be used.

In the description of the algorithm, we can clearly identify the four basic rules, as there are: the *branching* rule for the decomposition of nodes, the *bounding* rule for bound computations, the *selection* rule for choosing the next node to be evaluated, and the *elimination* rule for excluding nodes from possible evaluation.

On sequential computers, the specification of the rules makes up most of the work to obtain a useful algorithm. Further, a better structural understanding of the problem to be solved, i.e., a sharper specification of the basic rules, almost immediately leads to shorter computation times, and larger tractable instances.

The advent of parallel computers changed the situation dramatically. Insight into the problem itself is no longer sufficient to obtain an efficient parallel branch and bound algorithm. Implementations highly depend on the target architecture, and even on the search tree (to be) generated. This situation is highly undesirable from a user point of view. Instead of just coping with the problem itself, the user has to take nontrivial decisions concerning external matters.

To relieve the user from the burden of coping with issues that are hardly related to the problem to be solved, several approaches are possible. First, one could alleviate the pain by providing a library with high-level problem independent routines (e.g., for the manipulation of the search tree). The user has complete control over the solution process, since the library routines have to be called explicitly. An example can be found in Cung, Dowaji, Le Cun & Roucairol [1995].

Another approach is to define an architecture independent virtual parallel branch and bound machine. Here, the machine is responsible for the solution process. User routines play a passive role, i.e., they wait to be activated by the machine. The advantage is clear. The user only has to provide the problem specific functions through a well-defined user interface. The interface should incorporate all relevant aspects of branch and bound, but it must not be too general either. In other words, the interface must ensure that all branch and bound algorithms are captured, but it must not prevent the generation of efficient code. In our opinion, it is not opportune to require that a framework developed for branch and bound is also suited for other tree search algorithms, such as the ones stemming from the area of game trees.

The second approach seems to offer more opportunities for our aim to develop a robust and easy to use branch and bound system, that is able to produce efficient code for arbitrary problems on arbitrary architectures. It will form the base of our paper.

In the remainder, we assume that optimization means *minimization*. In this context,

a bound means a *lower* bound, and a feasible solution provides an *upper bound* on the optimal solution value. We will further use the notions of *open list* and *active set* to denote the set of nodes that are still to be evaluated. The organization of the paper is as follows.

Section 2 gives an overview of branch and bound. We focus on models that specify the branch and bound paradigm in a problem independent way. When such a branch and bound model is implemented, we arrive at a so called generic branch and bound system.

Section 3 discusses some proposals for generic branch and bound systems that appeared in the literature. In these systems, a user can plug in basic branch and bound rules for the problem at hand, after which the system will generate a (parallel) branch and bound algorithm running on a specific parallel architecture.

In Section 4, we argue whether the proposed systems are flexible enough. Therefore, we first of all need to investigate the design decisions to be taken by an implementor of parallel branch and bound. This results in a classification model, which is then investigated by checking whether existing parallel branch and bound implementations can find a place in it. The section ends with a stochastic model, within which the behavior of a common type of parallel branch and bound algorithm can be analyzed.

Having investigated the possibilities and pitfalls in parallel branch and bound implementations, we return to the question of whether the proposed models for generic branch and bound are flexible enough. Section 5 is devoted to this discussion. An idea is proposed to add an abstract machine model to the generic branch and bound framework within which the design questions treated in Section 4 can be specified.

In Section 6, we will present the conclusions and indicate future work.

2 Branch and bound

The first branch and bound algorithms appear in papers in the fifties and early sixties, where researchers describe enumerative schemes for solving what we would now call NP-hard problems [Eastman, 1958; Rossman, Twery & Stone, 1958; Land & Doig, 1960; Gilmore, 1962]. Because of the generality of the approach and its reported effectiveness, the method became widely spread. In fact, it is still a major tool for solving hard problems. The first ones to call the method *branch and bound* are Little, Murty, Sweeney & Karel [1963] in their innovating paper on the traveling salesman problem.

The key issue in this paper is the observation that branch and bound algorithms can be described in a problem independent way. Lawler & Wood [1966] recognize this in their survey paper — the first paper that presents a *model* of branch and bound. A branch and bound model tries to capture the general ideas that can be found in all branch and bound algorithms (like subproblem decomposition and elimination by lower bound), separating these from the details that are specific for a particular problem instance or algorithm (like the way a problem is represented, or the specification of the lower bound calculation). An example hereof can be found in Corrêa & Ferreira [1995].

In the next subsection, we will describe some models which have been proposed in the literature. As we will see, the abstraction levels of these models vary. For instance, although each model presents the notion of pruning of subproblems from the open list,

the more concrete models, e.g., the model of Ibaraki [1976], incorporate techniques to be used for this purpose (like the bound test, or the dominance test), whereas the more abstract model in Kumar & Kanal [1983] merely describes pruning as an operator from sets of subproblems to sets of subproblems, together with some properties it should exhibit.

A branch and bound model specifies a high-level abstract algorithm operating on abstract data types like 'subproblem', for which operators like 'compute lower bound' are defined. If in such a model the general branch and bound part can be clearly separated from the problem part, then the model can be made practical. The result is called generic branch and bound.

The idea is to generate an implementation of the abstract algorithm. It results in an incomplete branch and bound algorithm suited for general applications. In order to turn the algorithm into a concrete one, the problem specific part must be added, i.e., the implementation of the abstract data types and the associated high-level operators.

The advantage of this approach is that a user can now concentrate on the problem itself, working towards a good implementation of the abstract branch and bound rules, without having to worry about typical branch and bound implementation details. For instance, complications around managing the open list as a priority list (e.g., what to do if the open list outgrows available memory?) have already been taken care of in the generic part. The advantage becomes especially evident when parallel branch and bound is considered, because there are many more implementation details to take care of in this case.

One has to be careful when choosing a branch and bound model on which a generic branch and bound system is to be based. If the model is too abstract a user has to implement potentially very inefficient operators, e.g., the abstract pruning operator in Kumar & Kanal [1983], alluded to above. If, on the other hand, the model is too specific, it might preclude the exploitation of techniques enhancing the efficiency of the algorithm. For instance, if the model specifies that determining whether a subproblem constitutes a feasible solution must be done using an operator provided especially for this purpose, while for a certain problem domain the result is routinely obtained as a side effect of the decomposition, then obviously more work will be performed than needed.

2.1 Models

In this subsection, we will give a succinct overview of the models of branch and bound appearing in papers by Lawler & Wood [1966], Mitten [1970], Ibaraki [1976, 1977a, 1977b], and Kumar & Kanal [1983] and Nau, Kumar & Kanal [1984]. We will not cover all details, but instead concentrate on the underlying ideas, the data types, and operators that the authors distinguish, and we will give some remarks on the high-level branch and bound algorithms that are presented. We do not discuss more recent models because these are generally slight variants of the older ones presented here.

Lawler & Wood [1966]

The model of Lawler & Wood is based on the observation that many branch and bound algorithms refer to the idea of the relaxation of hard problems to easier ones. The stan-

dard example is integer programming. The relaxation of the constraint that the optimal solution should be an integer valued vector to the admittance of real valued solutions, transforms the problem into the easier linear programming problem.

Suppose a hard problem P is transformed into a relaxation P', and suppose P' has optimal (minimal) solution s'. Now if s' is a solution of P as well, then s' is also a minimal solution of P, because P' is more general than P. If s' is not a solution of P, then the objective function value associated with s' must be a lower bound to the minimal solution of P. In the latter case, the idea — when branching — is to transform P' into a set of new problems $P'_1, \ldots, P'_k$ (for some $k > 1$), by adding constraints to P' that exclude s' as a solution of any P'_i.

Lawler & Wood then investigate necessary properties of the new relaxed subproblems such that solving all of them will eventually yield a minimal solution of P, and sufficient conditions for a subproblem to be eliminated by a lower bound test.

Although branching and bounding are clearly visible in the model, emphasis is more on the mechanism behind branching and bounding: a bound is to be obtained by solving a relaxed problem, and children of a subproblem can only be obtained by adding new constraints. One could say that the working of the branch and bound algorithm is obscured by one particular mechanism to be used to obtain a bound or a decomposition.

Mitten [1970]

The model of Mitten is much more abstract than the previous one, in the sense that it does not emphasize one special technique to obtain a lower bound or a decomposition. This more abstract approach is based on the idea to model subproblems by the set of all their solutions. On the other hand, the idea of relaxation is still visible, because Mitten embeds the set S of feasible solutions in a larger set T of solutions of a more general problem, and models subproblems not by subsets of S but by subsets of T, thereby allowing 'infeasible solutions' in the analysis. The representation of subproblems in terms of T instead of S, however, is not essential (cf. Kumar & Kanal [1983], Nau, Kumar & Kanal [1984], and Corrêa & Ferreira [1995]).

This model is remarkably liberal. For instance, branching is modeled by an operator β, which takes as an argument a set of subproblems (the full open list) and which yields a new set of subproblems (a new open list). Mitten lists properties that β should have, essentially amounting to the condition that each subproblem in the argument (modeled as a subset of T) must be partitioned into one or more subproblems ('smaller' subsets in T) in the result of the function, under the condition that at each invocation some 'progress' should be made. The model, therefore, allows for more than one subproblem to be decomposed at a time.

Pruning is 'implemented' by three operators. First of all, there is a lower bound operator working on subproblems, which should deliver a lower bound to the value of all solutions in T of its argument. Notice that the lower bound is not necessarily a value of a solution in T, as was the case in the model by Lawler and Wood.

Secondly, there is an upper bound operator, taking a set of subproblems (the open list) as an argument, which yields an upper bound to the optimal solution over all subproblems in that set. Again, the upper bound is not necessarily the value of a solution.

In each iteration, all subproblems with lower bound larger than the upper bound on the open list can be discarded.

Mitten also specifies a third operator implementing a feasibility test, by demanding that there exists a collection C containing only subproblems, the elements of which are not in S, i.e., the subproblems contain only nonfeasible solutions. The collection C must not necessarily be exhaustive, but it must contain at least all singleton sets. The feasibility operator must be able to decide for an arbitrary subproblem whether or not it is in C.

Mitten then proceeds by giving a mathematical definition of a function, say *PRUNE*, acting on collections of subsets of T (the open list), and yielding the result after discarding all subproblems detected as infeasible by the feasibility test, as well as all subproblems that are found to be useless by the bound test. The result of an iteration of the abstract branch and bound algorithm starting with open list A is then given by $PRUNE(\beta(A))$.

Ibaraki [1976, 1977a, 1977b]

The contribution of the papers by Ibaraki is threefold. First of all, the subproblems are specified as objects without further structure. This is a useful approach if one wants to derive a generic branch and bound algorithm. The abstract algorithm does not depend on the internals of a problem specification, and it has to manipulate a problem only by applying operators on it. Hence, there is a nice separation of the abstract branch and bound algorithm and the problem dependent part. The generic branch and bound system is responsible for handling the user defined functions in the appropriate way, whereas the user gives structure to the problem by specifying the objects and implementing the operators.

Clearly, the operators should be easily implementable as well. Ibaraki's model specifies an operator O which, when applied to a subproblem P, should yield all its optimal solutions, and an operator f which takes a subproblem P and delivers the value of an optimal solution. Unfortunately, these operators are certainly not easily implementable.

However, a closer inspection of the algorithm reveals that the O- and f-operators are only applied on a specific subset of subproblems, consisting of 'the set of subproblems incidentally solved in the computation of the lower bound'. Although this seems a strange phenomenon at first sight, the idea is in fact plausible in case lower bounds are calculated by a relaxation of the problem to be bounded. So again, we encounter a model where bounding by relaxation is essentially incorporated.

The second contribution of Ibaraki's model is that dominance is modeled, by defining a dominance relation with the property that, if P dominates P', the optimal solution of P is no greater than that of P'. Again, the approach is quite amenable to generic branch and bound, because the user can now specify a relation with the above property which can be efficiently implemented. The abstract algorithm could then take care of the nontrivial job of checking whether newly generated subproblems are dominating or are dominated by older subproblems.

The final contribution of the model is the introduction of the idea of a selection operator. The general operator is difficult to implement because it takes a set of subproblems (the open list) as an argument. Ibaraki proposes to define the selection operator using

a heuristic function h, which assigns to each problem a priority. This operator can be straightforwardly implemented for a specific problem by the user. The selection operator is then defined as 'select the subproblem with the best h-value', which amounts to the fact that the abstract branch and bound algorithm should implement the open list as a priority list.

Kumar & Kanal [1983], Nau, Kumar & Kanal [1984]

The model of Kumar & Kanal [1983] is a variant of Mitten's model in two respects. First of all, it models a subproblem by its set of feasible solutions only. The paper shows that building an abstract branch and bound model using this representation of subproblems is not an essential restriction of Mitten's model.

The second deviation from Mitten's model is a more liberal definition of the *PRUNE* operator. The operator is now specified as a function that takes a set of subproblems and delivers a new set of subproblems with the property that the optimal solution over all subproblems in the argument has the same value as the optimal solution over all subproblems in the result set. A very general definition indeed, the operator does not specify any mechanism such as a lower bound test, a feasibility test, or a dominance relation. Due to its generality, the *PRUNE* operator is not very suitable for generic branch and bound, because the user now has to specify an operator which takes as a parameter a set of subproblems, which is clearly a harder task then specifying a lower bound calculation on one subproblem, or a dominance relation between two subproblems.

In a subsequent paper [Nau, Kumar & Kanal, 1984], a few refinements of the model are presented. First, the generalized model is restricted in several ways by showing how the *PRUNE* operator and the branch operator can be implemented in terms of simpler operators (for the branch operator, this has been discussed already in Ibaraki's model). It leads to several 'more concrete' abstract branch and bound algorithms, e.g., one where pruning is realized only by lower bounds, and another where both lower bounds and dominance are used for pruning.

Furthermore, also along the lines of Ibaraki's model, it is argued that a distinction should be made between the representation of a subproblem and the set of all feasible solutions of a subproblem, i.e., that modeling a subproblem by its feasible solutions is not adequate, since concrete branch and bound algorithms act on representations, and not on sets of solutions.

For instance, it is quite well possible that branching from a subproblem P yields only one new subproblem P' (with the same set of feasible solutions). This means that the lower bound operator cannot be adequately captured by the original model [Kumar & Kanal, 1983], because calculating a lower bound for P might yield another value than for P', whereas in the model they are represented by the same solution set.

The paper, therefore, suggests an adaptation of the model. Each subproblem is now represented by an abstract representation, whereupon the operators should act. The desired properties of the operators are defined in terms of a 'derepresentation' function, which is like the O-function of Ibaraki.

2.2 Comments

The branch and bound model of Lawler & Wood has been quite influential, which is one of the reasons why confusion has arisen about the essentials of branch and bound (as we now appreciate it). An overview of some of the disagreements on whether this or that particular algorithm (e.g., alpha-beta game tree search) can be classified as a branch and bound instance, is given in Kumar & Kanal [1983] and Nau, Kumar & Kanal [1984].

Ibaraki's model can be seen as a formal specification of (a generalization of) the Lawler & Wood model. It fits closely to branch and bound as applied in practice, and provides a good starting point for building a generic system on. Parallelism, however, is not contained in the model, but it can be incorporated in a simple way.

The Mitten and Nau, Kumar & Kanal models are quite general and mathematically very attractive. The iterative process of replacing the open list A by $PRUNE(\beta(A))$ until an optimal solution is found, is a pure formulation of the branch and bound paradigm. However, since the operators involved are applied to full open lists, they are very powerful. As said before, they are generally not easy to implement. Another reason why the operators are not very attractive, is that now the framework has less options for parallelization, which may seem strange at first glance, because the operations allow for the evaluation of several nodes in a single iteration of the algorithm. In this scenario, however, the user becomes responsible for creating sufficient parallelism and the system must be able to detect it, both of which are, of course, undesirable, if not impossible. Without internal knowledge of the user defined operations, the framework can only parallelize by distributing the operations over several processors. A better parallelization can, therefore, be obtained when the user specifies many small operations instead of a few time-consuming ones, as in the mechanism of Ibaraki.

3 Proposals for generic branch and bound

A number of generic branch and bound systems can be found in the literature. Most of them have been developed to obtain efficient parallel implementations of branch and bound algorithms on a specific architecture in an easy way.

There are several reasons why generic branch and bound has not been very popular on sequential computers. One is that after the specification of the basic rules the translation to a sequential algorithm is rather straightforward. The coding of the rules forms most of the work, and the implementation of the method itself is rather simple.

What is more important, is the observation that — generic or not — the most important feature of sequential branch and bound is the fact that almost all of the effort can be put into getting a better structural understanding of the problem. A more subtle formulation of the rules, such as sharper bounds or a more elaborate branching criterion, immediately pays off in shorter computing times.

In this section, we will briefly deal with three approaches to generic parallel branch and bound: the DIB system (a distributed implementation of backtracking) from Wisconsin [Finkel & Manber, 1987], the system developed in Karlsruhe [Kuck, Middendorf & Schmeck, 1993], and the East Anglia system based on the higher-order function approach [McKeown, Rayward-Smith & Turpin, 1991; Rayward-Smith, Rush &

McKeown, 1993]. The approaches have in common that they adopt a parallel version of Ibaraki's model (see Section 2.1), where the parallelism is exploited at the level of the evaluation of nodes. The distribution of the nodes over the processors is restricted to one or sometimes more (East Anglia) fixed strategies.

The DIB system [Finkel & Manber, 1987]

The DIB system is a generic system for backtracking (i.e., it is more general than generic branch and bound), and has been implemented on the Crystal multicomputer, consisting of a number of VAXes connected by a fast token ring network. The user has to define a data type `ProblemType`, and can specify a number of procedures acting on objects of this type, three of which are mandatory: `FirstProb` defining the original problem, the root of the search tree which is an object of type `ProblemType`, `Generate` defining how a (sub)problem can be decomposed, and `PrintAnswer` needed for output.

Branching is implemented piecewise. Each call of `Generate` delivers one child. This operator accepts the parent problem as input parameter. The parent must also be an output parameter because it has to be updated by `Generate`: it must be registered how many children have been generated already. The procedure has an output parameter of type `ProblemType`, which will be the newly generated child.

The user has no means to control the way subproblems are divided over the machines. DIB is quite autonomous in that respect. It generates children by calling `Generate`, and distributes these over the available machines. There is no global list of subproblems, and priority of subproblems cannot be taken into account.

Interestingly enough, the DIB system accepts implementations of problem dependent procedures, in which the user inserts calls to procedures defined by the DIB system itself. In this way, communication can be established. For instance, DIB defines a procedure `ReportResult` that takes a parameter P of type `ProblemType`, which might contain a solution. The effect is that P is sent to the supervising machine, which then automatically applies the user defined operator `PrintAnswer` to P.

In order to be able to implement branch and bound, global information, such as the current upper bound, must be made available to all machines. To this end, DIB offers a broadcast mechanism. The user should define an additional type `InfoType`. Objects of this type can be sent to all other machines in the network simultaneously by calls of the procedure `BroadcastInfo`. Receiving machines can become aware of newly sent information by executing the DIB procedure `UseNewInfo`.

Although DIB offers meager tools to update and interrogate central data, and is, hence, not very well suited to implement branch and bound, the system has been discussed here because it gives an idea on how the interface of a generic branch and bound system might look like. Especially the idea to provide the user with the option to call procedures supplied by the system in the implementation of the problem dependent operators, is interesting. Such a mechanism is not found in the other generic systems discussed below.

The Karlsruhe system [Kuck, Middendorf & Schmeck, 1993]

The Karlsruhe system by Kuck, Middendorf & Schmeck is a rather straightforward generic branch and bound system, implementing branch and bound with elimination of subproblems only by bounding, and selection of subproblems using a heuristic priority function. The user should describe the problem to be solved in terms of two data types, which implement the type subproblem, and a handful of instance functions, implementing the branch and bound operators.

The system accepts the operators `GenerateNodeForInitialProblem`, which should define the problem to be solved, `CreateChildren`, which should implement the branching operator, `CalculateLowerBound` for the lower bound computation, and `CalculatePriority` for the determination of the priority of a subproblem in the open lists. Furthermore, there should be an operator `IsTerminalNode` that tests whether a subproblem cannot be branched from any more. On such terminal subproblems, the operator `CalculateSolution` will be applied to extract the optimal solution. Finally, an optional operator `CalculateHeuristicSolution` can be provided which determines a (possibly suboptimal) solution for a subproblem, which could be used as an upper bound.

An interesting idea is to define a subproblem in terms of two data types instead of one. One data type stores a complete description of the initial problem (e.g., the distance matrix in the case of the traveling salesman problem), while the other serves to describe a subproblem in a short way by giving the 'moves leading to this subproblem', i.e., by giving a representation of the steps needed to transform the initial problem into the subproblem at hand (e.g., a list of required and forbidden edges). Another way to look at the mechanism is to consider the initial problem as a completely unspecified solution from which the optimal solution is built up in a stepwise fashion, by adding more and more components or constraints. For instance, solving a shortest path problem can be done by extending the candidate shortest path with a new edge in every step, starting from the empty path. This technique to represent a subproblem by two data types has a beneficial effect on the efficiency of the resulting algorithm. Subproblems must be stored in the priority list, and if subproblems are small, the list can contain more of them. Moreover, in parallel implementations subproblems will frequently be sent from one processor to another, and smaller descriptions of subproblems will result in faster transmission times.

The system transforms the rules and data given by the user into a parallel branch and bound system running on a transputer network. A sequential implementation is provided as well. The parallel system is built around local open lists. Subproblems with high priority are distributed regularly between neighboring processors. The only parameter the user can influence is the rate according to which node exchange takes place. The system has been tested on the traveling salesman problem and the set covering problem.

The Karlsruhe system implements a rather rigid abstract branch and bound algorithm. A user can influence the resulting algorithm only by tuning the operators, and a little bit by setting the exchange rate of subproblems between adjacent processors. Regrettably, the paper is rather loose in its description of the user interface. The interface should be tested on more branch and bound problems in order to determine its strengths and weaknesses.

The East Anglia higher-order function approach [McKeown, Rayward-Smith & Turpin, 1991; Rayward-Smith, Rush & McKeown, 1993]

The East Anglia system is much more elaborate than the Karlsruhe system, both in the abstract branch and bound operations it can handle and in the run time options with which the user can tune the efficiency of the execution of the resulting algorithm. There is a sequential implementation, as well as a parallel implementation on a Meiko transputer rack. In the sequential case only, the user can use pruning by dominance and even by isomorphism (cf. Ibaraki [1978]).

Subproblems are represented in a similar fashion as in the Karlsruhe system. The user should provide the operators `Bound` and `Priority` with the obvious meaning, and an operator `IsDom`, taking two subproblems and yielding a boolean, which will be true if the first subproblem dominates the second one. Also, an operator `Isomorph` can be specified for checking isomorphism. Checking whether a subproblem corresponds to a feasible solution is done in two stages: first, the operator `Label` determines a solution for the subproblem, and then, the operator `IsFeasible` determines whether the solution is a feasible one. For the correctness of the algorithm, the operator `Label` must deliver a feasible solution only if the feasible solution is an optimal one. The reason behind this mechanism is unclear.

Branching can be defined by an operator `Child`, which takes a subproblem and delivers a list of its children. However, to relieve the user of the burden of maintaining lists, it is also possible to define auxiliary operators, not unlike the one used for decomposition in the DIB system, from which the generic systems can build its own `Child` operator.

Some techniques for diminishing the overhead associated with testing for dominance and isomorphism are available as well. For instance, the user can define an operator which associates a class number with a subproblem. These class numbers should have the property that, if one subproblem dominates another one, both subproblems should have the same class number. The generic system will store subproblems in equivalence classes, and only subproblems in the same class need to be compared, i.e., subjected to the user defined dominance operator.

For the parallel implementation, a few mechanisms can be used to distribute subproblems over the processors. Furthermore, the user can specify whether the parallel branch and bound algorithm will be synchronous or asynchronous, and, in the latter case, how many processors will share a list of active subproblems. Extremes are one global list for the whole system, or one local list for each processor. In this way, many algorithms given in Corrêa & Ferreira [1995] can be specified.

The East Anglia system is by far the most flexible generic branch and bound system of the three. Unfortunately, the papers do not comment on the applicability of the interface, i.e., there is no evidence whether existing branch and bound algorithms can be described easily using this interface, and whether such a description will be transformed by the system into an efficiently executing parallel program, using the tuning options available to the user.

4 Towards generic parallel branch and bound

As argued before, generic branch and bound does not add much value on sequential computers. However, if a generic system is desired, the interfaces of the Karlsruhe and East Anglia systems form a sound basis. Most likely, only minor changes have to be made to meet the user's needs.

In the parallel case, things are different. The user not only has to take care of the basic branch and bound operations, but also needs to take into account a number of design issues as well, e.g., which load distribution strategy should be applied, should there be one global open list or should this list be distributed, and if so, how? The generic systems presented in the previous section may define a useful interface, but on the design issue part they are evidently shortcoming.

Furthermore, there are technical issues to be investigated, related to the architecture of the target parallel machine, things like the precise technicalities of the interprocess communication mechanism, the system calls needed to establish a number of cooperating processes on different machines, and the like.

If we want to make some statements about whether generic branch and bound can be of some help here, and if so, which features should be added to generic branch and bound, we first have to make a little study of parallel branch and bound as such.

The parallelization of branch and bound algorithms is possible at different levels. In the first place, the parallelism inside the basic rules can be exploited. This type of parallelization may give good results, but has the disadvantage that it requires insight into the problem to be solved, which of course is an enormous drawback for a generic system. Second, in general many nodes are available for evaluation. Hence, the evaluation of nodes in parallel also may give good speedups. Here, we do not need to know anything about what is going on inside the evaluation of a node. The approach will, therefore, be the ideal basis for a generic system.

Although the parallel evaluation of nodes seems a simple mechanism, the effort required to achieve an effective implementation should not be underestimated. There is a trade-off between keeping the processors busy doing useful work (i.e., evaluating nodes that might lead to an optimal solution) and the work to be done by the parallel algorithm to reach that goal. There are many solutions to this so called *dynamic load balancing* problem. Depending on the problem type, problem instance, and architecture to be used, a different method may be effective.

In the literature, a diversity of algorithms can be found. It turns out, however, that the algorithms, although very different at first glance, use the same basic computational model, and that they can be classified by means of only a few parameters. In the remainder of this section, we will review a taxonomy that, as far as we know, captures all algorithms proposed so far. We will demonstrate by some examples how parallel branch and bound algorithms indeed fit into the taxonomy. Finally, we will review preliminary work which may help us to take a decision on how to tune important parameters automatically.

4.1 A classification model

As the division of the work among the processors is an important part of a parallel branch and bound algorithm, the storage of the set of active nodes is the first parameter to consider. Another parameter is the access to the set of nodes. Can it be done at arbitrary moments during execution, or do processors have to wait for each other, i.e., is the algorithm synchronized or not? The taxonomy proposed in Corrêa & Ferreira [1995] is based on these parameters. De Bruin & Trienekens [1992] consider additional parameters, such as the unit of work and the possibility of interrupting a processor during its execution.

We will now explain the parameters in more detail, following the above mentioned references closely.

The first parameter concerns the *storage of the active nodes*. Extremes are the *shared data object* model and the *distributed data* model.

In the shared data object model, there is a single database containing the set of active nodes. At any point in time, the processors keep the database consistent with all generated information. The advantage is clear: as the database contains all interesting information, the processors can work on nodes that are of highest priority. The disadvantage of keeping the database consistent is that the processors have to communicate frequently. Not surprisingly, the communication may be a bottleneck in some situations.

In the distributed data model, each processor maintains a database with a set of active nodes. When selecting a node, a processor retrieves the node from the local database, and newly generated nodes are kept locally as well. Measures have to be taken when a local database becomes empty and, hence, a processor may become idle. There are many solutions to overcome this problem of *dynamic workload sharing*. We will give some examples later. The advantage of the model is that the communication can be kept to a minimum, but the certainty that processors always are doing useful work is lost.

The above models can be seen as both ends of a broad spectrum. In between them, there is a variety of others. For instance, the processors can be split into groups, where each group behaves according to the shared data object model, and the groups apply the distributed data model among themselves.

A second parameter to be chosen is the *unit of work*. Up to now, we have implicitly assumed that the database, whether global or local, was updated each time a node had been evaluated. One can, however, also think of letting a processor perform a limited search starting from a given node (which decreases the need for communication), or think of splitting the evaluation of a node with respect to the basic rules and updating the database after the completion of each of the rules (which increases the communication, but at the same time improves the quality of the database).

The third parameter is the *synchronization switch*. When completing its unit of work, a processor has two options: it waits until all other processors have completed their unit of work too, or it continues immediately. Especially in the pure shared data object model, it may be favorable to wait for other processors. Synchronization implies that the processors have full knowledge of the solution process at the time of selecting a new unit of work. Consequently, the processors can choose the 'best' units of work. In the asynchronous case, a processor might not be aware of important information that

is being generated by another processor. Again there is a trade-off between the possibilities. Synchronization will limit the amount of (what turns out to be) useless work to be performed, but also causes (unnecessary) idleness of processors.

As last parameter, we consider the *interruption switch*. To be able to use the information generated by other processors, a processor has to detect the update of the database. There are two basic ways in which a processor can become aware of an update: the processor can poll the database, i.e., check the database at regular time intervals, or the processor can be interrupted whenever the database is updated. Upon an interrupt, the unit of work currently being worked upon is preempted, the processor checks the database, and decides whether to continue with the current unit of work or to perform another task which at present seems more appropriate. In the latter case, the preempted unit of work is stored in its current state in the database.

Tuning the parameters is not always easy. It depends on factors like problem type (i.e., the specification of the basic rules), the problem instance (the size of the set of active nodes may vary heavily), and on the architecture (not every architecture supports arbitrary settings of the parameters, communication protocols may be fast or slow, etc.).

The target architecture also plays an important role in the last phase of the implementation. Processor capabilities and interconnection networks influence the actual implementation in its final stage.

Looking at the implementation of a parallel branch and bound algorithm, the path from the basic rules to implementation is much more complicated than in the sequential case (see Figure 1). Programming turns out to be a precise job, bearing in mind all possibilities.

But suppose an efficient implementation has been obtained for a particular computer, what happens if this computer is replaced by another one? Most likely, a new algorithm has to be developed. The parameters have to be tuned again, and a new implementation has to be made. The situation becomes bizarre in cases where the underlying architecture remains the same, but where the basic rules (for the same problem) are modified or the problem instance to be solved is changed. Even then, a redesign of the implementation may be needed.

4.2 Examples

The taxonomy described in the previous section turns out to be quite powerful. We will illustrate this on some examples. In the algorithms, we do not mention the update of the current best solution: in all cases, a processor which computes a better solution to the problem, sends the solution to all other processors immediately.

Li & Wah [1986]

As the algorithm described by Li & Wah uses the shared data object model, in a synchronized fashion, it can be considered the ‘classical’ parallel branch and bound algorithm. In the algorithm, the processors evaluate a node (i.e., perform the branching, bounding, etc.), update the global database, wait for each other to be ready, and then decide collectively which nodes should be evaluated in the next iteration. Interrupts do not occur.

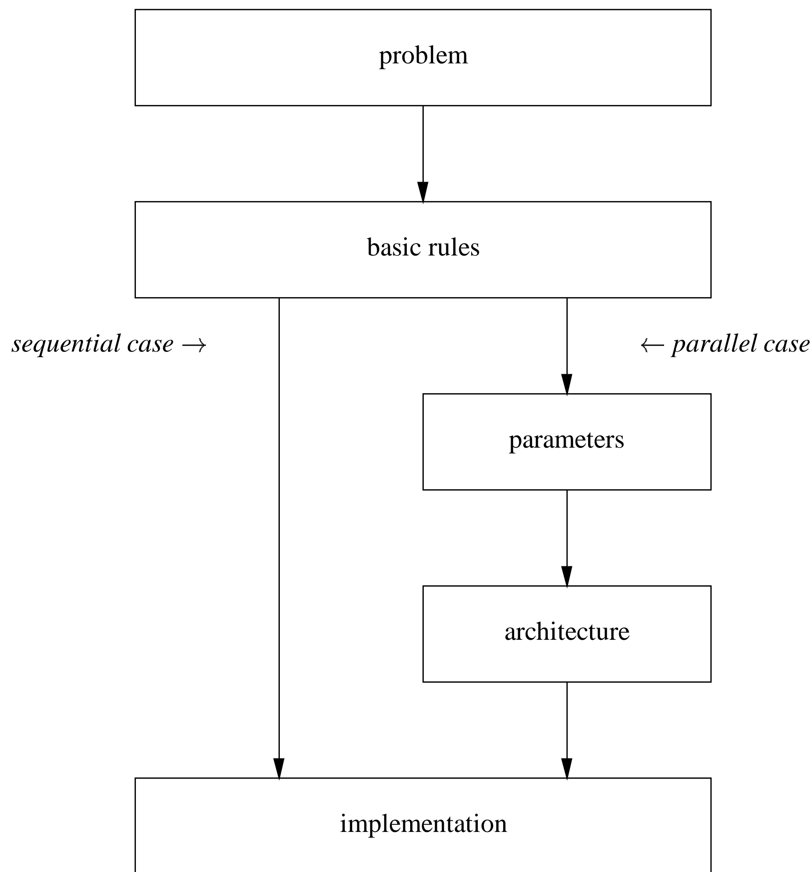

Figure 1: The implementation phases.

Trienekens [1990] and Kindervater [1991]

The algorithms developed by Trienekens and by Kindervater are essentially the same: both are based on the farmer-worker (often also called master-slave) principle. A farmer processor keeps track of the nodes that are to be considered for branching, and the other processors (the workers) evaluate the nodes. An idle worker processor immediately receives a node from the farmer, evaluates the node, and sends the results back to the farmer. The algorithm is about the same as the previous one, except that the database is stored in a single processor, and that there are no synchronizations.

The difference between the two implementations lies in the reported performance. Trienekens obtains quite good speedups, where Kindervater does not. As the underlying problem differs (traveling salesman vs. jobshop), the times needed for the evaluation of a single node are wide apart. In the jobshop problem, the time needed for an evaluation is very small and the farmer becomes a bottleneck, whereas in the traveling salesman

problem this is not the case. A theoretical explanation will be given in the next subsection.

Vornberger [1987] and Clausen & Träff [1991]

Both Vornberger and Clausen & Träff exploit the distributed data model: each processor maintains its own database with active nodes. Vornberger implements his algorithm on a network of transputers, and Clausen & Träff make use of an iPSC hypercube. In both algorithms, the distribution of the workload is done 'on overload'. A processor determines after the evaluation of a node and the corresponding update of its database, whether or not to send nodes to neighboring processors.

In Vornberger's algorithm, a processor decides to send a node to neighboring processors on a time-stamp basis. After a certain amount of time, a processor sends nodes (if available) to neighboring processors. The interval varies for each neighbor, depending on the quality of the last node received from that neighbor.

Clausen & Träff develop a different mechanism. Here, a processor determines whether the size of its database is acceptable. If it contains too many nodes, the superfluous nodes are sent to neighboring processors. The maximum size of the local database is updated dynamically according to some heuristic rules.

Further, both algorithms are completely asynchronous, and processors cannot be interrupted. Incoming messages are dealt with after the evaluation of a node.

Jansen & Sijstermans [1989]

Jansen & Sijstermans also use the distributed data model. However, they work with processes instead of processors. The algorithm employs a variable number of identical processes, each of which examines its own part of the subtree. While evaluating a node, a process can decide to create an additional process that evaluates a child node of the current node and all children thereof. The newly created process performs its work independently of the creating process.

The number of processes that can coexist, and hence the decision whether or not to create a new process, depends on the target architecture and on the problem instance to be solved.

Miller & Pekny [1989]

The algorithm by Miller & Pekny is based on the 'processor shop' model. It uses two different databases, the first one contains nodes that still have to be evaluated completely, and the second one with nodes, whereof the bound has been computed, but that are waiting to be branched.

An idle processor first tries to select a node from the database with nodes that have to be completely evaluated. On success, the processor computes the bound for that node and puts the node, if it cannot be eliminated, in the second database. If the first database is empty the processor selects a node from the second database, decomposes the node, and puts the children generated in the first database. If both databases are empty, the processor waits until work becomes available in either one of the databases

This algorithm also fits in the taxonomy. It can be seen as an asynchronous algorithm within the shared data object model with units of work at the level of the execution of the basic branch and bound rules.

4.3 Analysis

As the examples show, there exists a broad scale of implementations of parallel branch and bound algorithms. In the first place, architectures are very different from each other. Algorithms efficient on one machine may behave poorly on another one. But even given a specific architecture, there may be no ideal implementation. Not only the nature of the problem to be solved, but also the shape of the search tree to be explored influences the performance of an actual implementation.

Parallel branch and bound algorithms encountered in the literature try to capture all possible applications, but most of them only obtain near optimal results for the problem at hand given a specific architecture. To overcome such situations, we would need a prediction of what to expect when actually performing a parallel evaluation of the search tree.

There has not been done much research into the search trees generated by branch and bound algorithms. Smith [1984] randomly generates search trees and analyzes the time and space complexity of their exploration in the sequential case. Evidence is presented that the traveling salesman problem can be solved in polynomial time, i.e., that the generated search tree has a polynomial number of nodes on the average. For an algorithm to be successful in the parallel case, at any time the number of nodes available for evaluation should be high enough to keep processors busy. As Smith does not make any statements in this direction, more research should be done, hopefully leading to interesting results.

As long as a theoretical description of generated search trees does not exist, one could try to give a theoretical explanation of the observed behavior of the algorithm, and extend this to the general case. Among the few things that have been done in this respect is the investigation of the occurrence of *anomalous behavior*, i.e., adding a processor to the computer at hand decreases the running time for a problem instance more than can be expected from just adding computational power to the system, or the addition of a processor causes a slowdown. As the results in this area do not affect our argumentation, we do not go into detail, but refer to Corrêa & Ferreira [1995] for an overview and further references. As another example, we will discuss the *queueing network* model developed by Boxma & Kindervater [1991]. The model can be used to describe the farmer-worker algorithm from Trienekens [1990] and Kindervater [1991] (see the previous subsection), and to analyze the observed different speedups.

Recall that in the farmer-worker model, a farmer keeps track of the generated search tree, and sends a node to a worker as soon as one becomes idle. The workers evaluate the nodes they receive, send the results back to the farmer, and wait for the farmer to send a new node for evaluation.

This gives rise to the queueing model of Figure 2, with P customers, each customer corresponding to one particular worker. This is a well known queueing model, often referred to as the *machine repair* model (the P customers being P machines which after breakdown have to be repaired in repair facility F).

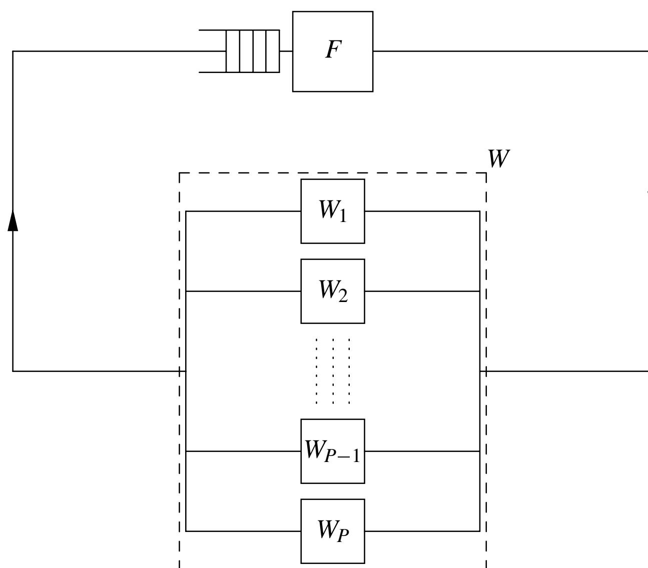

Figure 2: The machine repair model.

The machine repair model has been extensively studied in the queueing literature (see, for example Tijms [1986]). Under the assumption that the service times at the P servers $W_1, \ldots, W_P$ of service station W are independent, identically distributed with mean $1/\alpha$, and the service times at F are independent, negative exponentially distributed stochastic variables, with mean $1/\beta$, the number of busy servers at W is given by:

$$r[1 - \frac{r^P/P!}{\sum_{j=0}^{P} \frac{r^j}{j!}}],$$

with

$$r = \beta/\alpha.$$

Figure 3 displays the fraction of busy servers as a function of $\beta/P\alpha$. The figure shows that, for $P > r$ ($\beta/P\alpha < 1$), the fraction of busy servers decreases rapidly, when $\beta/P\alpha$ decreases.

The results of Trienekens and Kindervater are now easily explained. In the traveling salesman problem the value of $\beta/P\alpha$ is greater than one and good speedups are obtained, whereas in the jobshop this value is small and the observed speedup is poor.

Although the model is a simplification of the real world (e.g., it assumes that there are always enough nodes available for evaluation, the average speed of the farmer is assumed constant, etc.), it can be a useful tool for the development of a parallel branch and bound algorithm, because it gives a good indication whether or not the shared data object model is the right choice.

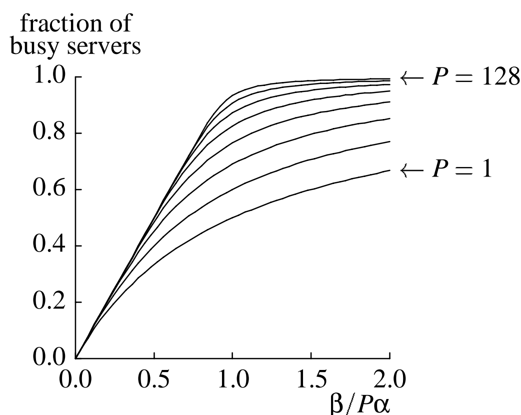

Figure 3: Fraction of busy servers as function of $\beta/P\alpha$ for $P = 1, 2, 4, 8, 16, 32, 64, 128$.

To conclude this section, we would like to stress that more research in this area could lead to a better understanding of parallel branch and bound algorithms.

5 Towards an abstract model for branch and bound

In the preceding sections, we have seen that a parallel branch and bound algorithm can be investigated at three levels. First of all, there is the level of a branch and bound model, which is captured in the interface of a generic branch and bound system. At this level, statements about the relative efficiency of the user provided operators can be formulated (e.g., branching is fast, but bounding is time consuming), and the complexity of the problem in terms of the size of the search tree or its critical subtree can be described (cf. Corrêa & Ferreira [1995]).

The second level is the design level. Relevant issues are discussed in Section 4.1, like synchronicity versus asynchrony, distributed versus centralized open lists, determination of the unit of work, etc.

The third level has been more implicit up till now. It is the level of the target architecture on which the parallel program has to run. Apart from all the technical details that we try to avoid when using generic systems, there are also efficiency parameters at this level. For instance, a transputer rack is optimized towards communication speed, and the ratio of the computation power of the processors versus the communication efficiency favors the communication. On a number of fast work stations connected via ethernet, the ratio is turned around.

Apart from technicalities, the essential decisions on how a branch and bound algorithm is parallelized are taken at the second level. There is, however, an interplay between the different levels. As an example, consider the decision whether or not to use a centralized open list. The decision is taken at the second level, but it is, amongst others, influenced by the time needed for branching and bounding or the shape of the search tree

to be generated (first level), and the interprocessor communication (third level).

In the generic branch and bound systems from the literature, the flexibility at the highest level is abundant. The user is completely free to specify the implementation of the basic rules. At the design level, there is much less liberty. The only system offering some freedom is the East Anglia system, where a few parameters can be set to one of a few predefined values. Once the parameters are set, however, they cannot be dynamically adjusted.

At the bottom level, the generic branch and bound systems proposed in the literature are completely rigid: the target architecture is fixed. An interesting question is how generic systems behave when they are implemented on different architectures. Stated otherwise, one could wonder whether it is always straightforward to translate a branch and bound algorithm, specified in a generic framework, to an arbitrary architecture. We suspect that this is not the case, and that the user should be able to provide some hints as how the translation should be performed.

Our aim is to combine the ease of use of generic branch and bound with the flexibility that different design strategies and different target parallel architectures offer. That is to say, we want to use generic branch and bound, but still be able to obtain an algorithm for our specific problem, based on the right decisions at the design level. One of our problems is to find a way to introduce the desired flexibility.

Consider the discussion of the algorithms by Kindervater and Trienekens in Section 4.2. We would like to be able to express that a better strategy in the jobshop algorithm (Kindervater) is to solve nodes to optimality, whereas for the traveling salesman problem (Trienekens) nodes should be evaluated only one level deep, i.e., it should be possible to indicate the best unit of work. In an ideal situation, the setting of the unit of work is derived from information obtained at the first level (execution of the branch and bound rules is relatively time consuming in the Trienekens case, and relatively efficient in the Kindervater case), and at the third level (the description of the target architecture).

The above discussion suggests that there is a need to formulate decisions at the design level. To this end, we propose to introduce a computational model, or abstract machine. The concept is depicted in Figure 4.

The abstract model is intended to define a virtual parallel architecture. The idea is that it should be possible to formulate design decisions in terms of the abstract machine, whereby the peculiarities of the target architecture remain hidden (the advantage of generic branch and bound), without sacrificing expressivity. For instance, it should be able to specify that there are many workers evaluating nodes and a single farmer maintaining the open list, without making explicit on which machine the open list should reside, or even whether the farmer and a worker should share one processor.

The approach has the following additional advantages. Using the abstract machine, one could specify parameters of the underlying architecture, e.g., adjacent processors can communicate fast, but broadcasts might be expensive. A very promising feature is that the computation can be monitored in terms of the abstract machine. One could envision that the user is presented graphical feedback about the computation going on, such that a better understanding of the behavior of the algorithm can be obtained.

We do not know what language or graphical system should be used to define the abstract machine level. One idea is to base the abstract machine on the farmer-worker model in which the *farmer process* is responsible for delegating work to and accepting

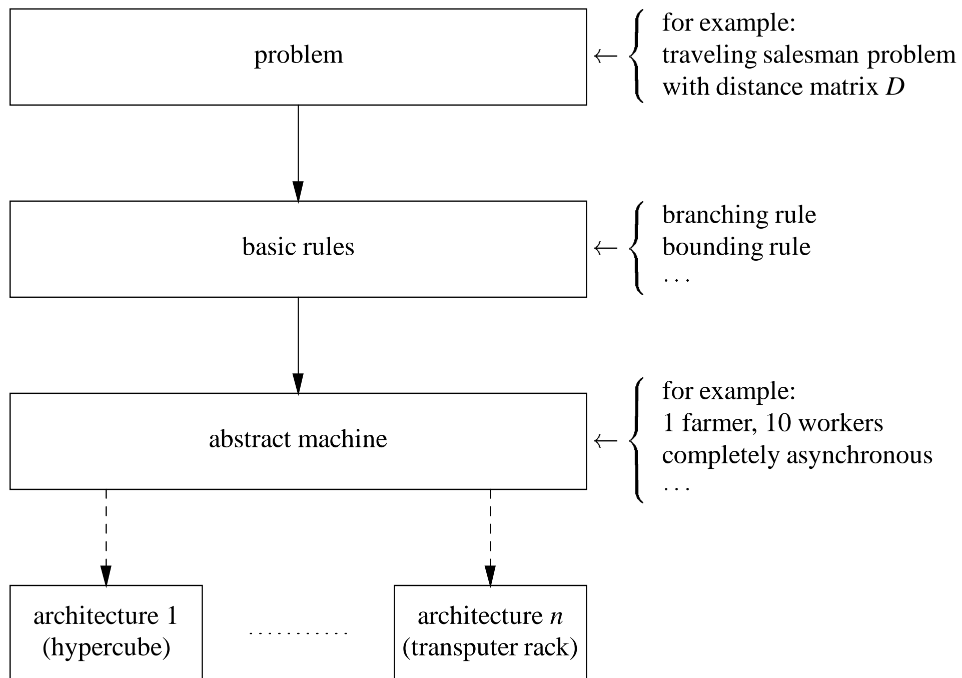

Figure 4: The abstract machine.

results from the workers. That is, the farmer should implement the selection strategy, and do some of the global pruning. Notice that we speak of one farmer, but that it is quite possible to have a distributed implementation of the process. Of course, the decision to have more than one farmer process should be expressible in our model.

The *workers* are then responsible to apply the other branch and bound operations, e.g., branching, bounding and local elimination of subproblems. In this case too, it might be conceptually simpler to talk about one worker, but again the availability of more workers should be expressible in our model.

Apart from farmers and workers, it seems that another type of process is needed, for which we coin the name *supervisor*. This process should handle all meta-information, and monitor the traffic and the idleness of the workers and farmers. The supervisor should also be responsible to interact with the user, providing information about the way the computation is evolving. Finally, one should be able to program the supervisor so that it will execute feedback loops, based on the way the computation proceeds, and according to the options that the supervisor has at its proposal (e.g., switch to another bounding rule if one is available, or change the number of farmer and worker processes).

6 Conclusions and future work

In this paper, we gave an overview of abstract branch and bound models and of proposals for generic branch and bound systems. Branch and bound models vary from very abstract ones, which are difficult to implement, to more concrete ones, which can be turned into an interface of a generic branch and bound system in a rather straightforward way.

A generic branch and bound system should not only specify an interface, given by the basic branch and bound rules, but should also generate an efficient implementation, different for each particular parallel architecture. Closer inspection of existing systems shows that a user is completely free in specifying an implementation of the basic branch and bound operators. There is, however, much less choice at the deeper levels of the implementation: the target architecture is fixed, and the intermediate level, on which design decisions are taken regarding load balancing, data distribution, etc., can only be influenced in one of the proposed systems, and then only in a rather inflexible way.

As a remedy, we propose to add a new layer to a generic branch and bound system, the so called *abstract machine* through which it is possible to generate more flexibility in specifying design options. The machine abstracts from peculiarities of specific concrete parallel architectures, thus allowing the user to concentrate at the higher level design issues.

Another advantage of the approach is that the system can give feedback to the user on how the computation proceeds in terms of the abstract machine, and that it seems possible to specify feedback actions to be taken by the systems in case bottlenecks in the execution of the program are detected.

The ideas presented in this paper are sketchy. It seems that much research is still needed. For instance, it is not at all clear how an abstract machine should be defined, and what should be expressible in it. Furthermore, we need to investigate the flexibility of the interface defined by the generic branch and bound systems proposed in the literature. We plan to recode some existing branch and bound algorithms in terms of these systems, in order to find out whether refinements are needed.

In the end, we hope to arrive at a system that is robust, easy to use, and yet flexible enough to accommodate a broad range of different branch and bound implementations on parallel architectures of different types.

References

[1] O.J. Boxma, G.A.P. Kindervater (1991). A queueing network model for analyzing a class of branch-and-bound algorithms on a master-slave architecture. *Oper. Res. 39*, 1005–1017.

[2] J. Clausen, J.L. Träff (1991). Implementation of parallel branch and bound algorithms - experiences with the graph partitioning problem. *Ann. Oper. Res. 33*, 341–349.

[3] R. Corrêa, A. Ferreira (1995). Parallel best-first branch-and-bound in discrete optimization: a framework. *Solving Combinatorial Optimization Problems in Parallel*, LNCS, Springer, Berlin, to appear.

[4] V.-D. Cung, S. Dowaji, B. Le Cun, C. Roucairol (1995). The outcome of a know-how: a branch-and-bound library. *Solving Combinatorial Optimization Problems in Parallel*, LNCS, Springer, Berlin, to appear.

[5] A. de Bruin, H.W.J.M. Trienekens (1992). *Towards a Taxonomy of Parallel Branch and Bound Algorithms*, Report EUR-CS-92-01, Department of Computer Science, Erasmus University, Rotterdam.

[6] W.L. Eastman (1958). *Linear Programming with Pattern Constraints*, Report BL 20, The Computation Laboratory, Harvard University, Cambridge.

[7] R. Finkel, U. Manber (1987). DIB - a distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst. 9*, 235–256.

[8] P.C. Gilmore (1962). Optimal and suboptimal algorithms for the quadratic assignment problem. *J. Soc. Indust. Apl. Math. 10*, 305–313.

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

[10] T. Ibaraki (1977a). On the computational efficiency of branch-and-bound algorithms. *J. Oper. Res. Soc. Japan 20*, 16–35.

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

[12] T. Ibaraki (1978). Branch-and-bound procedure and state-space representation of combinatorial optimization problems. *Inf. Control 36*, 1–27.

[13] J.M. Jansen, F.W. Sijstermans (1989). Parallel branch-and-bound algorithms. *Future Generations Comput. Syst. 4*, 271–279.

[14] G.A.P. Kindervater (1991). *Exercises in Parallel Combinatorial Computing*, CWI Tract 78, Centre for Mathematics and Computer Science, Amsterdam.

[15] N. Kuck, M. Middendorf, H. Schmeck (1993). Generic branch-and-bound on a network of transputer. R. Grebe et. al. (eds.). *Transputer Applications and Systems '93*, IOS Press, 521–535.

[16] V. Kumar, L.N. Kanal (1983). A general branch and bound formulation for understanding and synthesizing And/Or tree search procedures. *Art. Intelligence 21*, 179–198.

[17] A.H. Land, A.G. Doig (1960). An automatic method for solving discrete programming problems. *Econometrica 28*, 497–520.

[18] E.L. Lawler, D.E. Wood (1966). Branch-and-bound methods: a survey. *Oper. Res. 14*, 699–719.

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

[20] J.D.C. Little, K.G. Murty, D.W. Sweeney, C. Karel (1963). An algorithm for the traveling salesman problem. *Oper. Res. 11*, 972–989.

[21] G.P. McKeown, V.J. Rayward-Smith, H.J. Turpin (1991). Branch-and-bound as a higher-order function. *Ann. Oper. Res. 33*, 379–402.

[22] D.L. Miller, J.F. Pekny (1989). Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. *Oper. Res. Lett. 8*, 129–135.

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

[24] D.S. Nau, V. Kumar, L.N. Kanal (1984). General branch and bound and its relation to A* and AO*. *Art. Intelligence 23*, 29–58.

[25] V.J. Rayward-Smith, S.A. Rush, G.P. McKeown (1993). Efficiency considerations in the implementation of parallel branch-and-bound. *Ann. Oper. Res. 43*, 123–145.

[26] M.J. Rossman, R.J. Twery, F.D. Stone (1958). *A Solution to the Traveling Salesman Problem by Combinatorial Programming*, Peat, Marwick, Mitchell and Co., Chicago (mimeographed).

[27] D.R. Smith (1984). Random trees and the analysis of branch and bound procedures. *J. Assoc. Comput. Mach. 31*, 163–188.

[28] H.C. Tijms (1986). *Stochastic Modeling and Analysis: a Computational Approach*, Wiley, Chichester.

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

[30] O. Vornberger (1987). Load balancing in a network of transputers. *Proc. 2nd Int. Workshop on Distributed Algorithms*, LNCS 312, Springer, Berlin, 116–126.