Parallel Branch, Cut, and Price for Large-Scale Discrete Optimization

T. K. Ralphs* L. Ladányi† M. J. Saltzman‡

April 4, 2003

Abstract

In discrete optimization, most exact solution approaches are based on branch and bound, which is conceptually easy to parallelize in its simplest forms. More sophisticated variants, such as the so-called *branch, cut, and price* algorithms, are more difficult to parallelize because of the need to share large amounts of knowledge discovered during the search process.

In the first part of the paper, we survey the issues involved in parallelizing such algorithms. We then review the implementation of SYMPHONY and COIN/BCP, two existing frameworks for implementing parallel branch, cut, and price. These frameworks have limited scalability, but are effective on small numbers of processors. Finally, we briefly describe our next-generation framework, which improves scalability and further abstracts many of the notions inherent in parallel BCP, making it possible to implement and parallelize more general classes of algorithms.

1 Introduction

Recent computational research has focused on the use of parallelism and the evolving "computational grid" to solve difficult large-scale decision and optimization problems. Tree search, of which branch and bound is an important special case, is a fundamental technique for solving such problems and lends itself well to parallelization. This technique has been applied to a vast array of problems in a wide variety of fields. The literature on these techniques, although widely scattered, is rich with approaches presented under a host of different names. Well-known methods such as backtracking, A*, AND/OR tree search, neighborhood search, and branch and bound are all variations on the same basic theme.

*Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, tkralphs@lehigh.edu, http://www.lehigh.edu/~tkr2, funding from NSF grant ACI-0102687 and IBM Faculty Partnership Award

†Department of Mathematical Sciences, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, ladanyi@us.ibm.com

‡Department of Mathematical Sciences, Clemson University, Clemson, SC 29634 mjs@clemson.edu, http://www.math.clemson.edu/~mjs

Applications of tree search are found in such areas as VLSI routing, theorem proving, game theory, combinatorial optimization, and artificial intelligence. Popular benchmark applications from the literature include the integer knapsack problem, the traveling salesman problem, the Golomb ruler problem, the 15 puzzle, and the n-queens problem.

The tremendous attention that branch and bound has received in the literature gives some indication of its importance in many research areas. Unfortunately, most of the interesting problems for which branch and bound provides a viable solution approach are in the complexity class $\mathcal{NP}$-hard and may require searching a tree of exponential size in the worst case. It is natural to consider parallelizing the search process in order to make solution of large-scale problems more practical.

Although it might appear at first that parallelizing branch and bound is straightforward, this turns out not to be the case for most applications. The primary reason is that the search trees may be highly irregular in shape and this shape is not known a priori. This makes the task of dividing up the work among the various processors difficult at best. Furthermore, knowledge that is generated as the search progresses can change the shape of the tree dynamically. The way in which knowledge is stored and shared among the processors during the solution process can have a profound effect on efficiency. This is a central theme in what follows.

In this paper, we focus primarily on the parallelization of sophisticated versions of branch and bound known as *branch, cut, and price* algorithms. A great deal of what we say, however, applies in more general settings, as we point out in Section 6. The paper is organized as follows. In Section 2, we begin with an overview of related work. Section 3 contains a brief review of the basic sequential branch, cut, and price algorithm. In Section 4, we describe the issues involved in parallelizing branch and bound, including a brief overview of parallel computing concepts. In Section 5, we describe the implementation of two related frameworks for parallel branch, cut, and price. The first is SYMPHONY [42, 44], a generic C library originally developed at Cornell beginning in 1993 and first described in the thesis of Ralphs in 1995 [40] and Ladányi in 1996 [27]. The second is COIN/BCP [43], a C++ class library with basically the same design philosophy developed at IBM beginning in 1997. We illustrate various principles with computational results using the SYMPHONY framework to solve a prototypical combinatorial optimization problems. Finally, in Section 6, we discuss the third-generation framework, a hierarchy of C++ class libraries that abstracts and generalizes many of the notions in SYMPHONY and COIN/BCP to allow the development of more scalable solvers implementing a wider variety of algorithms.

2 Related Work

The branch and bound algorithm as we know it today was first suggested by Land and Doig in 1960 [31]. Soon after, Balas introduced the idea of *implicit enumeration* in suggesting a related algorithm for solving the knapsack problem [5]. As early as 1970, Mitten [36] abstracted branch and bound into the theoretical framework we are familiar with today. However, it was not until the 1990s that much of the software used for discrete optimiza-

tion originated. Almost without exception, these new software packages implemented various versions of branch and bound. The packages fall into two groups: solvers employing general-purpose algorithms for solving mixed integer programs (without exploiting special structure) and those allowing the user to take advantage of special structure by interfacing with problem-specific subroutines. We call this second group of packages *frameworks*. There have also been a large number of special-purpose codes developed for use in particular problem settings.

MIP solvers are the most common offering. Among the many options in this category are MINTO [38], MIPO [6], and bc-opt [13]. Generic frameworks, on the other hand, are less numerous. The two frameworks reviewed in this article—SYMPHONY and COIN/BCP—along with ABACUS [16, 25] (developed independently during the same time period) are the most full-featured packages available. Several others, such as MINTO, also have the capability of utilizing problem-specific subroutines. CONCORDE [3, 4], a software package for solving the traveling salesman problem, is perhaps the most sophisticated special-purpose code developed to date. Commercial packages include ILOG's CPLEX, IBM's OSL, and Dash's XPRESS.

Numerous software packages employing parallel branch and bound have been developed. The previously mentioned SYMPHONY, COIN/BCP, and CONCORDE are all parallel codes that can also be run on networks of workstations. Other related software includes frameworks for implementing parallel branch and bound such as PUBB [46, 47], BoB [8], PPBB-Lib [50], and PICO [15]. PARINO [32] and FATCOP [11, 12] are parallel MIP solvers.

The literature on parallel computation in general, and parallel branch and bound in particular, is rich and varied. We concentrate here on those works most closely related to our own. Kumar and Gupta provide an excellent general introduction to the analysis of parallel scalability in [26]. Good overviews and taxonomies of parallel branch and bound algorithms are provided in both [19] and [49]. Eckstein [15] also provides a good overview of the implementation of parallel branch and bound. A substantial number of papers have been written specifically about the application of parallel branch and bound to discrete optimization problems, including [9, 14, 20, 35, 51].

3 Sequential Branch, Cut, and Price

3.1 Branch and Bound

Branch-and-bound algorithms constitute a broad class of methods that are the primary approach for solving difficult discrete optimization problems (DOPs), especially those that are $\mathcal{NP}$-hard. These algorithms use a divide-and-conquer strategy to partition the solution space into *subproblems* and then recursively solve each subproblem. Let S be the set of solutions to a given DOP, and let $c : S \to \mathbb{R}$ be a cost function on members of S. Suppose we wish to determine a least-cost member of S and we are given $\hat{s} \in S$, a "good" solution determined heuristically. Initially, we examine the entire solution space S. In the *processing*

or *bounding* phase, we relax the problem, that is, we admit solutions in a superset of the feasible set S. Solving this relaxation yields a lower bound on the value of an optimal solution. If the solution to this relaxation is a member of S or has cost equal to that of $\hat{s}$, then we are done—either the new solution or $\hat{s}$, respectively, is optimal. Otherwise, we identify subsets $S_1, \ldots, S_q$ of S, such that $\cup_{i=1}^{q} S_i = S$ using a *branching rule*. The problem of finding a minimum-cost element of S_i is called a *subproblem*; $S_1, \ldots, S_q$ are sometimes called the *children* of S. We replace S with the children of S on the list of *candidate subproblems* (those that await processing). This operation is called *branching* and the candidate list is often called the *node queue*.

To continue the algorithm, we select one of the candidate subproblems to remove from the node queue and process it. There are four possible results: If we find a feasible solution better than $\hat{s}$, then we replace $\hat{s}$ with the new solution and continue. We may also find that the subproblem has no solutions, in which case we discard, or *prune* it. Otherwise, we compare the lower bound to our global upper bound, given by the value of the best feasible solution encountered thus far. If it is greater than or equal to our current upper bound, then we may again prune the subproblem. If we cannot prune the subproblem, we are forced to branch and add the children of this subproblem to the list of candidates. We continue in this way until the list of candidate subproblems is empty, at which point our current best solution is the optimal one. Note that the algorithm we have just described is the most common of many variants on this basic theme. A more thorough discussion is contained in [15].

3.2 LP-based Branch and Bound

In LP-based branch and bound [24], the relaxation at a given node in the search tree is a linear programming (LP) problem that can be described by a specified set of *valid inequalities*, i.e., ordered pairs (a, β) such that $\{x \in \mathbb{R}^n : ax \leq \beta\} \supseteq S$, where S is the feasible set for the subproblem. For pure integer programs, a typical branching rule selects a coefficient vector $a \in \mathbb{Z}^n$ (so that ax must be an integer for any integer solution x) and such that $f = a\hat{x} \notin \mathbb{Z}$, where $\hat{x}$ is an optimal solution to the current subproblem. Then one can simply branch by generating the two children

$$S_1 = \{x \in S : ax \leq \lfloor f \rfloor\} \text{ and} \quad (1)$$

$$S_2 = \{x \in S : ax \geq \lceil f \rceil\}. \quad (2)$$

When a is a unit vector, this branching rule reduces to the classic rule of "branching on variables." This idea can easily be generalized by choosing a collection of coefficient vectors, each of which has associated bounds in each child. This collection is called a *branching set*. A branching rule with the property that

- the current solution is not feasible for any of the generated subproblems and
- the feasible regions of the generated subproblems are disjoint

is called *partitive*. Note that a branching rule need not be partitive, but this is obviously desirable and most branching rules used in practice have this property. In particular, branching based on a branching set insures this property will be satisfied. In *strong branching*, the branching set is chosen from a list of candidates by "pre-solving" the subproblems that would result from branching on each of the candidate branching sets to obtain bound estimates, which can then be used to decide on a final branching set.

3.2.1 Branch and Cut

In LP-based branch and bound, efficiency depends substantially on the tightness of the relaxations, that is, how closely the relaxed sets approximate the feasible regions of the subproblems. The LP relaxations can be tightened by dynamically adding globally valid inequalities, i.e., those that are valid for the original feasible set. This technique is called *branch and cut*. These inequalities form a class of *shared knowledge* that is particularly important for our study of mixed integer programs (see Section 4.2.1). Because they are globally valid, the inequalities can be shared through the use of *cut pools*, which act as repositories for previously discovered inequalities. Early implementations of branch and cut are described in [21] and [39].

3.2.2 Branch and Price

Another method for producing tighter relaxations is to use a reformulation technique based on column generation, such as Dantzig-Wolfe decomposition. This may result in a formulation with an extremely large number of variables. In *branch and price*, each LP relaxation is solved initially with only a small subset of the variables present. Additional variables are then generated as needed throughout the search tree. As with branch and cut, the variables that are generated can be shared throughout the search tree. In principle, this can be done through the use of *variable pools*. For more details on branch and price, see [7, 45].

3.2.3 Branch, Cut, and Price

In theory, the two methods described above can be combined to produce the technique called *branch, cut, and price* (BCP), in which both cut and variable pools can be used. However, due to the dynamic nature of the relaxations, developing efficient data structures for representing the objects becomes much more challenging. In addition, the dynamic generation of cutting planes can destroy the structure of the column generation subproblem, making it difficult to simultaneously generate cuts and variables. Computationally, such methods are very difficult to implement and are therefore rarely seen in the literature. For examples of the technique, see [3, 29, 30].

4 Parallel Branch, Cut, and Price

4.1 Parallel Systems and Scalability

Recently, "cluster computing" has become increasingly pervasive, as the use of networks of workstations and the development of so-called *Beowulf clusters* has allowed users with modest budgets to achieve performance approaching that of dedicated "super-computers." In this paper, we assume a parallel architecture similar to that of a typical cluster computer. The main properties of such an architecture are as follows:

- The processing nodes are comprised of commodity PC hardware and are connected by a dedicated high-speed communication network.
- Each processing node has a single central processing unit (in some architectures, processing nodes may have more than one processor).
- There is no shared access memory. Memory is available to each processor locally, but no assumption is made regarding the local memory hierarchy.
- Inter-processor communication is via a *message-passing* protocol. This means that information can only be passed from one processing node to another as an array of characters with an associated *message tag* denoting what type of information is contained in the message. The two most common message-passing protocols are PVM [18] and MPI [22].

This type of architecture and the algorithms we discuss are *asynchronous* by nature, meaning that the processors do not have a common clock by which to synchronize their calculations. Although there is a difference between a *processing node*, which consists of a processor with associated memory and other hardware, and a *processor*, we generally refer the processing nodes simply as *processors* to avoid confusion with search tree nodes.

Generally speaking, the main goal in parallelizing an algorithm is to achieve *scalability*. The *scalability* of a parallel system, that is, the combination of a parallel algorithm and a parallel architecture, is the degree to which it is capable of utilizing increased computing resources (usually processors). To assess this capability, we compare the speed with which we can solve a particular problem instance in parallel to that with which we could solve it on a single processor. The *sequential running time* (T_0) is usually used as the basis for comparison and is taken to be the running time of the best available sequential algorithm. The *parallel running time* (T_p) is the running time of the parallel algorithm and depends on p, the number of processors available. The *speedup* (S_p) is simply the ratio T_0/T_p and hence also depends on p. Finally, the *parallel efficiency* (E_p) is the ratio S_p/p of speedup to number of processors.

Our aim is to maintain an efficiency close to one as the number of processors is increased. When a parallel algorithm has an efficiency of one, we say that it has achieved *linear speedup*. In theory, a parallel algorithm cannot have an efficiency greater than one (this is

called *superlinear speedup*), but in practice, such a situation can actually occur (see [10, 48] for a treatment of this phenomenon). When an algorithm has an efficiency less than one, we can compute the *parallel overhead* as $O_p = T_p(1 - E_p)$. We consider four basic components of parallel overhead:

- *Communication overhead*: Time spent sending and receiving information, including packing the information into the send buffer and unpacking it at the other end.
- *Idle time (handshaking)*: Time spent waiting for information requested from another processor.
- *Idle time (ramp-up/ramp-down)*: Time at the beginning/end of the algorithm during which there is not enough useful work to keep all processors busy.
- *Performance of redundant work*: Work that would not have been performed in the sequential algorithm.

The first three sources of overhead can be considered the cost incurred in order to share information (called *knowledge*) among the processors, whereas the last one is the cost incurred due to *not* sharing this information. This highlights the fundamental tradeoff we must make. To achieve high efficiency, we must limit the impact of the first three sources of overhead without increasing the performance of redundant work.

4.2 Algorithm Design

4.2.1 Knowledge Sharing

The sophisticated versions of branch and bound we outlined in Section 3.2 all depend on the notion of *knowledge sharing* and *knowledge bases*, introduced by Trienekens and de Bruin in [49]. The concept is that *knowledge* discovered while processing one node of the search tree may be useful during processing of subsequent nodes in other parts of the tree. Useful knowledge can therefore be deposited in a specified *knowledge base* that can be accessed during further processing. These knowledge bases can be thought of as oracles that serve requests for previously generated information of a particular type.

The most basic knowledge that is shared during branch and bound is the value of the current upper bound. This knowledge allows nodes whose lower bounds exceed the current upper bound to be pruned. Decisions about how to branch may also be made based on shared knowledge. In sequential branch and bound, knowledge sharing is straightforward because it can be done through shared memory with very little overhead. When parallelizing these algorithms, however, disseminating knowledge efficiently becomes a central issue. At a high level, the primary difference between various implementations of parallel branch and bound is the way in which knowledge storage and dissemination is handled [49]. The effectiveness of knowledge sharing determines in large part an algorithm's scalability.

We consider three distinct types of knowledge to be shared.

- *Bounds*:
  - *Upper bounds*: The upper bound is the value of the best solution found so far in any part of the search tree (recall that we are minimizing). This bound, in combination with the lower bound for a given search tree node, can be used to prune the node and avoid the performance of redundant work. Dissemination of upper bounds generally incurs low overhead and can be accomplished via broadcasting to ensure that all processors have complete knowledge of the current upper bound.
  - *Lower bounds*: Knowledge of the lower bounds corresponding to nodes that are available for processing but not stored locally, is important in avoiding the performance of redundant work. This information is used to determine if work should continue on nodes that are locally available or if new nodes should be requested from an external source. Making knowledge of lower bounds available is more difficult to accomplish because the number of candidate nodes available globally can be extremely large. Furthermore, it is possible that these bounds may not be stored centrally.
- *Node descriptions*: In order to process a node, it is necessary to have a detailed description of it. This consists of the branching decisions that were applied to obtain the node, as well as any other information needed to "warm start" the processing of the node. Node descriptions are shared through the use of one or more knowledge bases called *node pools*. Because the node descriptions can be extremely large in some cases, it is important to store the node descriptions compactly. This can be accomplished using *differencing*, as described in Section 5.2.2.
- *Global objects*: In some applications, the node descriptions consist essentially of a list of objects that can be shared. In the case of mixed integer programming, these objects are the variables and constraints. The efficient storage and dissemination of objects can also create scalability problems, as the number of objects can be extremely large. Objects can be stored and disseminated through the use of knowledge bases called *object pools*. Because the size of these knowledge bases can grow extremely quickly, it is crucial to purge them regularly, retaining only the most "effective" objects for later use. See Section 5.3.4 for more a further description of object pools.

We say a processor has *perfect knowledge* if it has immediate access to all of the knowledge being generated at other processors. Knowledge sharing highlights the tradeoff that is often encountered in parallel algorithm design between eliminating the performance of redundant work and limiting the overhead due to communication. In theory, we would like to get as close to perfect knowledge as we can. However, in practice, the dissemination of knowledge can incur a large overhead. Striking this balance can be difficult. We describe examples of specific approaches to knowledge sharing in Section 5 and 6.

One of the key issues in developing scalable algorithms is *load balancing*. Load balancing consists of ensuring that each processor has non-redundant work to do throughout the course of the algorithm. Load balancing can be viewed as a special case of knowledge sharing in

which the items to be shared are the descriptions of the tasks to be done. Load balancing in the case of parallel branch and bound has to ensure that tasks are evenly distributed both in terms of quantity *and* quality. The quality of a node is usually determined by its lower bound, and hence, effective load balancing requires global information about the distribution of lower bounds throughout the search tree. As we have already noted, this information is easily available in a sequential algorithm, but can be difficult to obtain in parallel.

4.2.2 Task Granularity

An important consideration in designing any parallel algorithm is the desired *task granularity*. The natural unit of work for parallel branch and bound is the processing and branching of a single subproblem. However, one can easily envision algorithms in which the unit of work is smaller or larger than this. At one extreme, we could consider the unit of work to be the processing of an entire subtree. At the other extreme, we could consider breaking the processing of a single subproblem into further units of work. Generally speaking, using larger units of work results in lower communication costs, but can also result in an increase in the performance of redundant work. Several authors have had success recently with using larger units of work in parallel branch and bound [2, 17]. For most of what follows, we assume that the unit of work is a single subproblem. In Section 6, we discuss a more general approach.

4.2.3 Ramp Time

In parallel branch and bound, ramp-up time can be one of the biggest sources of parallel overhead we face. The severity of this problem depends primarily on two factors: (1) how long it takes to process a node, and (2) how many children are produced when branching. These two factors together determine how long it takes before the number of candidates available for processing exceeds the number of processors. In addition, it may depend on the amount of "inherently sequential" work performed by the master process before entering the parallel part of the code (Amdahl called this the *sequential fraction* and was the first to point out its importance in the analysis of scalability [1]).

In solving mixed integer programs, the time required to process each node is typically quite significant. It is also common for branching to result in only two children. This is in fact a worst-case scenario for ramp-up time. Attempts to control ramp-up time must either (1) reduce the time spent processing each node in the initial phases of the algorithm, (2) use a branching rule that produces a large number of children, or (3) change the task granularity (i.e., parallelize the processing of a single search tree node). We discuss a simplistic approach to this issue in Section 5.4.

Ramp-down time is usually less of an issue, but can also be a problem in isolated cases. Ramp-down normally occurs near the end of algorithm execution when the number of candidate nodes available may again fall below the number of available processing nodes. The remedies for controlling idle time during the ramp-down process are essentially the

same as those for controlling idle time during ramp-up. Once the condition is detected, the algorithm must limit the time spent processing each subsequent node until additional work is produced and/or branch in such a way as to create enough children to provide work for idle processing nodes.

4.2.4 Search Strategy

The search strategy determines the order in which the available candidate nodes will be processed. As in sequential branch and bound, the search strategy can make a marked difference in the efficiency of the algorithm. In fact, even in sequential branch and bound, the performance of so-called *non-critical work*, i.e., the processing of a node whose lower bound turns out to be above the value of the optimal solution, can occur. The search strategy becomes a much more important issue in parallel versions of the algorithm because it can be a factor not only in the performance of redundant work, but also in the amount of communication overhead. We emphasize here the distinction between *redundant work*, which is work done in the parallel algorithm that would not have been performed in the sequential algorithm, and non-critical work, which is performed in both the parallel and sequential algorithms.

There are several important considerations in implementing a search strategy for parallel branch and bound. First, our options are clearly be limited by the availability of knowledge with which to make decisions about what work to perform next. Second, we have to give much stronger consideration to so-called *diving* strategies. In sequential codes, periodic diving (retention one of the children generated during branching for processing) is often used to find feasible solutions quickly. However, this strategy has the additional advantage of retaining work that is generated locally, i.e., nodes generated through branching. Therefore, it decreases communication overhead and is particularly attractive in a parallel setting.

As in sequential branch and bound, the two most basic approaches are *best first* and *depth first*. The best-first approach favors the candidate nodes with the smallest lower bounds. The advantage of this approach is that it guarantees that no non-critical work will be performed and hence minimizes the size of the search tree. In the case of parallel branch and bound, however, this strategy is difficult to implement in its pure form because of the need for accurate global information about the candidates currently available for processing. As already discussed, this information is generally not easily available unless a single central node pool is used. Another significant disadvantage of this strategy is that it does not tend to retain generated nodes locally, as described above. As in the sequential case, it may also require more memory globally and does not tend to find feasible solutions quickly.

On the other hand, a pure depth-first approach favors the candidate nodes at maximum depth in the search tree. This strategy has several advantages over best first. First, it is a diving strategy and hence always retains one of the candidates made available locally during branching, which saves communication and node set-up costs. Second, it tends to minimize the number of candidate nodes available globally and hence saves memory. Third, it tends to find feasible solutions quickly. However, its main disadvantage is that it usually results in the performance of large amounts of non-critical work.

Hybrid schemes attempt to capture the advantages of both best-first and depth-first search by continuing to dive as long as certain conditions are satisfied. One such condition is that the number of variables in the current relaxed solution that have fractional values is "low." This gives an indication that there may be a good chance of finding a feasible integer solution quickly by diving. This rule has the advantage of not requiring any global information. Another possible condition is that the lower bound in one of the children is "close" to the lower bound of the best node otherwise available, where "close" is defined by a chosen parameter. Evaluating this condition requires external information, however. Several authors have reported great success with these strategies in a parallel setting [2, 12, 17, 40]. An in-depth survey of search strategies in a sequential setting is contained in [33].

4.2.5 Branching Strategy

For the purposes of parallel branch and bound, there are two important aspects of the branching strategy. First, we may have a choice as to the number of children that are produced by branching. It may be better to produce more children in the early stages of the algorithm in order to limit ramp-up time. In later stages, however, generating fewer children may result in lower communication overhead. Second, we have to consider whether our chosen branching rule requires global information that must be obtained from a knowledge base. This is the case, for example, in some implementations of branching based on pseudo-costs.

5 SYMPHONY and COIN/BCP

In this section, we describe the implementation of parallel BCP within the SYMPHONY and COIN/BCP frameworks. SYMPHONY and COIN/BCP have a common ancestry and take similar approaches, although SYMPHONY is written in C and COIN/BCP is written in C++. Both frameworks are designed to make it easy for the user to implement parallel BCP solvers for a wide variety of problem settings and across a wide variety of architectures. The vast majority of the code that implements the BCP algorithm is generic and is internal to the library. This part of the code is implemented as a "black box," the implementation of which the user need know nothing about. To implement a state-of-the-art parallel solver, the user has only to write a few problem-specific subroutines. All the usual functions of BCP—tree management, node processing, and pool management—as well as inter-process communication, are handled by the framework. Full details of the sequential implementation are contained in [42, 43, 44]. Here, we concentrate only on the details most relevant to the parallel implementation.

5.1 General Design Approach

The two biggest challenges to be faced in applying BCP to large-scale problems is dealing with the huge numbers of *objects* (a generic term referring to the variables and constraints)

that must be accounted for during the solution process and dealing with the very large search trees that can be generated for difficult problem instances. This involves not only the important question of how to store the data, but also how to move it between processors during parallel execution. In developing a generic framework, we have the added challenge of dealing with these issues in a problem-independent way.

Describing a node in the search tree consists of, among other things, specifying which objects are *active* in the subproblem. In fact, the vast majority of the methods in BCP that depend on the model are related to generating, manipulating, and storing these objects. From the user's perspective, implementing a BCP algorithm using SYMPHONY or COIN/BCP consists primarily of specifying various properties of objects, such as how they are generated, how they are represented, and how they should be realized within the context of a particular subproblem. With this approach, we achieved the "black box" structure by separating these problem-specific functions from the rest of the implementation.

5.2 Data Structures and Storage

There are two basic types of data that need to be stored and shared in parallel BCP—global objects and search tree nodes. In addition, we need to able to store the search tree itself. Because this tree can grow to be extremely large, we have developed compact data structures based on the idea of *differencing*, as explained below. In the next two sections, we explain how each of these basic data types are represented and stored.

5.2.1 Objects

Both the size of each node's description and the time required to process it are largely dependent on the number of objects active in the corresponding subproblem. Keeping this active set as small as possible is one of the keys to efficiently implementing BCP. For this reason, we need data structures that allow us to efficiently move objects into and out of the active set. Allowing sets of constraints and variables to move into and out of the linear programs simultaneously is one of the most significant challenges of BCP. We do this by maintaining a user-defined, compact *representation* of each object that contains information about how to add it to a particular LP relaxation. This representation includes the associated bounds (in the case of a variable) or right-hand-side range (in the case of a constraint). For constraints, the user must provide a method for converting this representation to a row of the constraint matrix for a given LP relaxation and similarly for variables. This compact representation is also used to store the objects and to pass them from one processor to another as needed. After generation, each object is assigned a global index by which it is referred to throughout the search tree. This assignment of a global index must be done centrally, which may have repercussions for scalability.

5.2.2 Search Tree

The description of a search tree node consists primarily of the indices of objects that are active in that node. In addition, a critical aspect of implementing BCP is the maintenance of warm-start information for each node (e.g., the current basis, for simplex-based LP solvers). This information may either be inherited from the parent or computed during strong branching. Along with the set of active objects, we must also store the identity of the branching set that was used to generate the node. The branching operation is described in Section 3.2.

Because the set of active objects and the status of the basis do not tend to change much from parent to child, all of these data are stored as *differences* with respect to the parent when that description is smaller than the explicit one. This method of storing the tree is memory efficient and is what allows us to store the entire tree on a single processor. The list of nodes that are candidates for processing is stored in a heap ordered by a comparison function defined by the search strategy 4.2.4. This technique allows efficient generation of the next node to be processed, but it depends on central storage of the search tree.

One further way in which we attempt to limit the size of the node descriptions is by allowing the user to specify a problem *core* consisting of a set of variables and constraints that are to be active in every subproblem. The core should consist of variables and constraints that are considered "important" for the given instance, in the sense that there is a high probability that they will be needed to describe an optimal solution. One advantage of specifying a core is that the description of the core can be stored statically at each of the processors and need not be part of the node description itself. This saves both on node set-up costs and communication costs, as well as making storage of the search tree more efficient. Another advantage of specifying a core is that because the core variables are never considered for removal, they do not have to be actively managed by the framework. This can result in significant savings during the processing of a search tree node.

5.3 Implementation

The solver functions are grouped into four independent computational modules. This modular implementation not only facilitates code maintenance, but also allows easy and highly configurable parallelization. Depending on the computational setting, SYMPHONY's modules can be compiled as either (1) a single sequential code, (2) a multi-threaded shared-memory parallel code, or (3) separate processes running over a distributed network. COIN/BCP has options (1) and (3). The modules pass data to each other either through shared memory (in the case of sequential computation or shared-memory parallelism) or through a message-passing protocol defined in a separate communications API (in the case of distributed execution). A schematic overview of the modules and their functions is presented in Figure 1. Details of each module are given below.

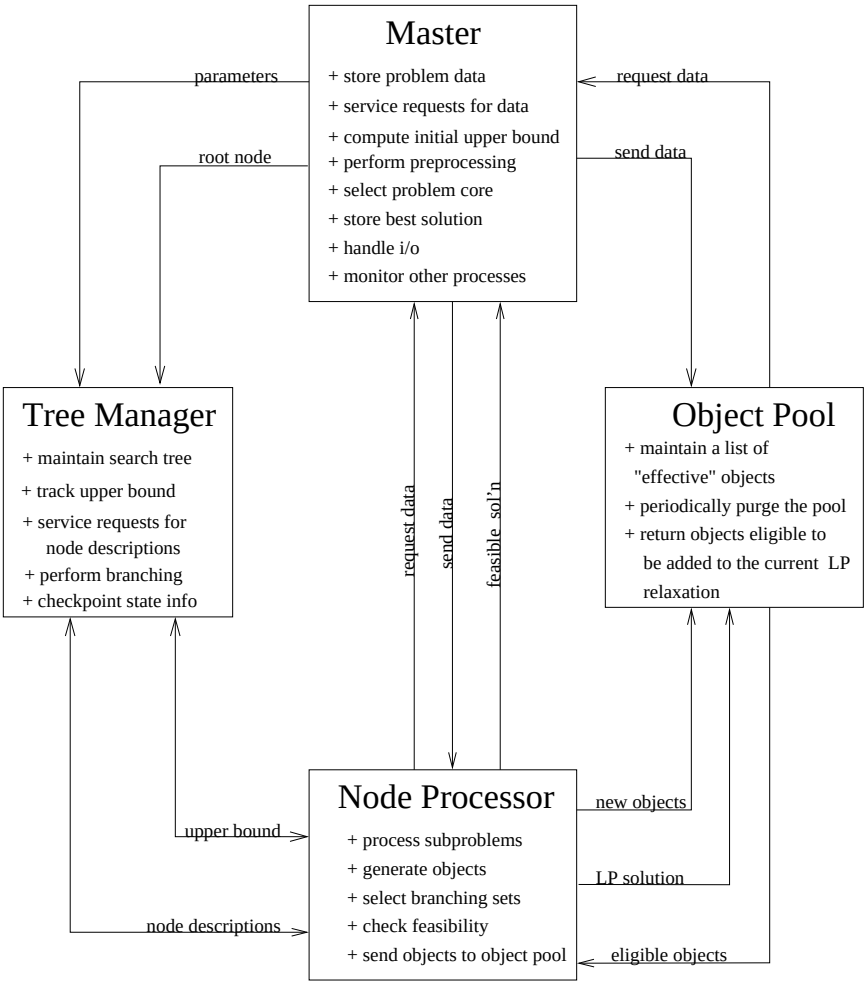

Figure 1: Schematic of the parallel branch, cut, and price algorithm

5.3.1 The Master Module

The *master module* includes functions that perform problem initialization and I/O. The primary reason for a separate master module is fault-tolerance, as this module is not heavily tasked once the computation has begun. If desired, the user may provide a subroutine to calculate an initial upper bound or provide one explicitly. In some cases, the upper bounding may also be done in parallel. A good initial upper bound can decrease the solution time by allowing more variables to be fixed (see Section 5.3.3) and search tree nodes to be pruned earlier. If no upper bounding subroutine is available, then the two-phase algorithm, in which a good upper bound is found quickly using a reduced set of variables, can be used (see Section 5.3.2 for details). During preprocessing, the user must also specify the problem core, as well as the list of extra objects that are to be active in the root node. The importance of selecting a good core is discussed in Section 5.2.2.

5.3.2 The Tree Manager Module

Both SYMPHONY and COIN/BCP implement *single-node-pool* BCP algorithms, in which there is a single central list of candidate subproblems to be processed, maintained by the *tree manager*. Most sequential implementations use such a single-node-pool scheme. However, this has significant ramifications for parallel performance, as we discuss in Section 5.4. The unit of work is a single subproblem, which is processed by the node processing module in an iterative manner, as described in the next section.

The tree manager controls the execution of the algorithm by deciding which candidate node should be chosen as the next to be processed using either one of several built-in rules or a user-defined rule (see Section 4.2.4 for a discussion of search strategies). It also tracks the status of all modules, as well as that of the search tree, and distributes the subproblems to be processed to the node processing module(s). Because the tree manager's primary job is to maintain the list of candidate subproblems, it can be considered a knowledge base for maintaining node descriptions. This knowledge base can be queried by the node processing modules (described below) for tasks to be performed.

The Two-Phase Algorithm. If no heuristic subroutine is available for generating feasible solutions quickly or if generating new variables is prohibitively expensive, then a unique *two-phase algorithm* can also be invoked. In the two-phase method, the algorithm is first run to completion on a specified set of core variables. Any node that would have been pruned in the first phase is instead sent to a pool of candidates for the second phase. If the set of core variables is well chosen, this first phase should execute quickly and result in a near-optimal solution, as well as producing a list of useful cuts.

Using the upper bound and the list of cuts from the first phase, the root node is *repriced*—that is, it is reprocessed with the full set of variables and cuts. The dual solution generated by solving this much stronger LP relaxation can then be used to calculate reduced costs for the variables not in the original core. Any variable that can be fixed using this new reduced cost can be eliminated from the problem globally. If we are successful at fixing all of the

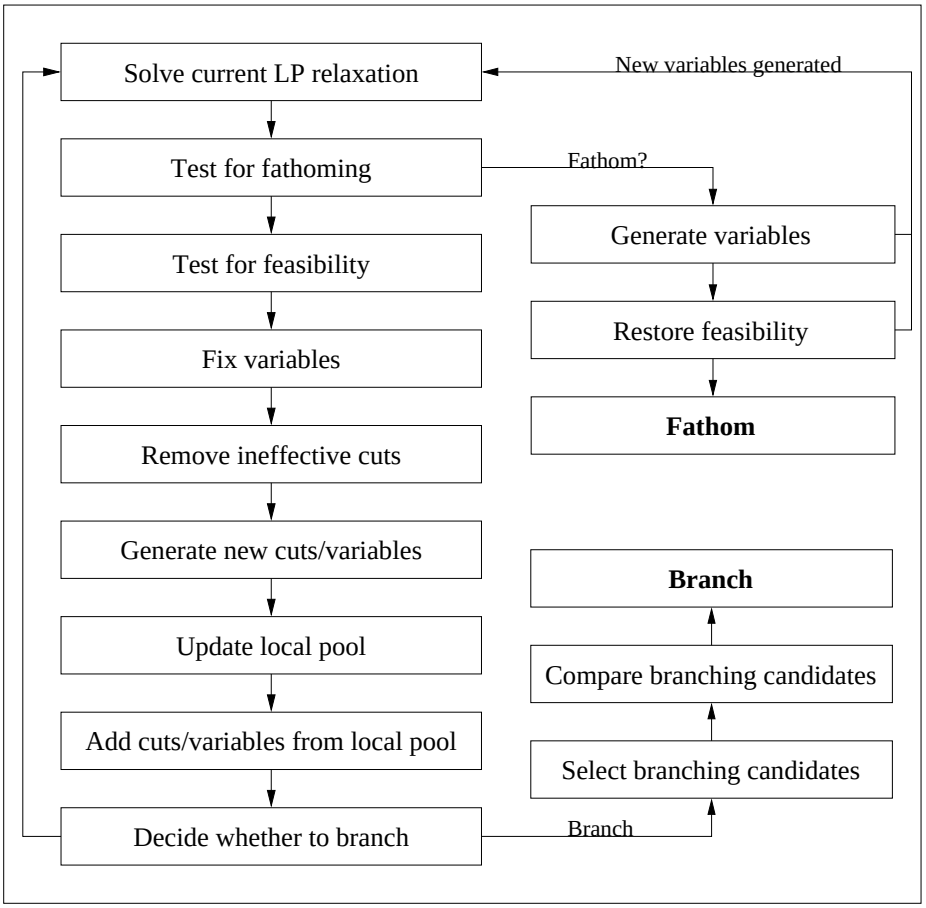

Figure 2: Overview of the NP loop

inactive variables, then we can immediately output the optimal solution. Otherwise, we must go back and price these variables in each leaf of the search tree produced during the first phase and continue processing any node in which there remains an unfixed variable.

In order to avoid pricing variables in every leaf of the tree, we can *trim the tree* before the start of the second phase. Trimming the tree consists of eliminating the children of any node for which each child has lower bound above the current upper bound. We then reprocess the parent node itself. This is typically more efficient, since there is a high probability that, given the new upper bound and cuts, we will be able to prune the parent node and avoid the task of processing each child individually.

5.3.3 The Node Processing Module

The *node processing (NP) module* is central to the algorithm, as it performs the computationally intensive bounding and branching operations on each subproblem. The procedures for bounding and branching are described in more detail in the sections below.

Managing the LP Relaxation. Often, the majority of the computational effort of BCP is spent solving LP relaxations and hence this process should be as efficient as possible. Besides using a good LP engine, the most important factor is to control the size of each relaxation, in terms of number of active objects. The number of objects is controlled through use of a local pool and through periodic purging and fixing. The bounding procedure is iterative and involves the solution of a sequence of LP relaxations, as pictured in Figure 2. First, the initial LP relaxation is solved, then new objects are generated based on the resulting primal and dual solutions, then the relaxation is solved again. This process continues branching occurs (see the next section for a description of when this happens).

When an object is generated, it is first added to a local pool. In each iteration, up to a specified number of the "best" objects (measured by degree of violation or reduced cost) from the local pool are added to the problem. Objects that never prove useful are eventually deleted from the list. In addition, cuts are purged from the LP relaxation itself when they have been deemed ineffective for more than a specified number of iterations, where ineffective is defined as either (1) the corresponding slack variable is positive, (2) the corresponding slack variable is basic, or (3) the dual value corresponding to the row is zero (or very small). Cuts that have remained effective in the LP relaxation for a specified number of iterations are sent to a global pool, where they can be used in later search nodes. Cuts that have been purged from the LP relaxation can be made active again if they later become violated. Note that because ineffective constraints left in the matrix increase the size of the basis unnecessarily, it is usually advisable to adopt an aggressive strategy for their removal.

The number of variables in the relaxation, on the other hand, is controlled both through *reduced cost fixing* and through a user-defined method of *logical fixing* based on problem structure. During each iteration of the NP loop, the reduced cost of each active variable is calculated to see if it can be fixed. If the matrix is *full* at the time of the fixing, meaning that all unfixed variables are active and no more can be generated, then the fixing is permanent for that subtree. Otherwise, it is temporary and only remains in force until the next time that columns are dynamically generated. Column generation can also be handled through the use of the two-phase algorithm described in Section 5.3.2.

Branching. To perform branching in COIN/BCP, a branching set must be selected, as described in Section 3.2, consisting of both a list of objects to be added to the relaxation (possibly only for the purpose of branching) and a list of object bounds to be modified. Any object can have its bounds modified by branching, but the union of the feasible sets contained in all child subproblems must contain the original feasible set. In SYMPHONY, the branching set is selected in a slightly more restrictive manner. In both cases, branching takes place whenever either (1) both cut generation and column generation (if performed) have failed; (2) "tailing off" in the objective function value has been detected; or (3) the user chooses to force branching. To control ramp-up time in the case of parallel execution, we have implemented a simplistic "quick branching" strategy in which branching occurs after a fixed number of iterations in the NP module regardless of whether or not new objects have been generated (see Section 5.4). This process continues until all processors have useful work to do, after which the usual algorithm is resumed. Branching can be fully automated

or fully controlled by the user and the branching rule can result in as many children as the user desires, though two is typical.

Once it is decided that branching will occur, the user must either select a list of candidate sets for *strong branching* (see below for the procedure) or allow the framework to do so automatically by using one of several built-in strategies, such as branching on the variable whose value is farthest from being integral. In SYMPHONY, the number of candidates may depend on the level of the current node in the tree—it is usually advantageous to expend more effort on branching near the top of the tree.

After the list of candidates is selected, each candidate is *pre-solved*, by performing a specified number of iterations of the dual simplex algorithm in each of the resulting subproblems. Based on the objective function values obtained in each of the potential children, the final branching object is selected, again either by the user or by built-in rule. This procedure of using exploratory LP information in this manner to select a branching candidate is commonly referred to as *strong branching* and has its roots in early work by Mitra and others [34]. When the branching set has been selected, the NP module sends a description of that set to the tree manager, which then creates the children and adds them to the list of candidate nodes. It is then up to the tree manager to specify which node the now-idle NP module should process next. If a diving strategy is being employed, the NP module may be instructed to retain one of the children produced during branching. This eliminates the need for the tree manager to construct and send a new node to the node processing module to be processed.

5.3.4 The Object Pool Module

The concept of a *cut pool* was first suggested by Padberg and Rinaldi [39], and is based on the observation that in BCP, the inequalities which are generated while processing a particular node in the search tree are generally globally valid (or can be made so) and are potentially useful at other nodes. Since generating these cuts is sometimes a relatively expensive operation, the cut pool maintains a list of the "best" or "strongest" cuts found in the tree thus far for use in processing future subproblems. Hence, the cut pool functions as an auxiliary cut generator. In theory, we can extend this idea to the more general notion of an *object pool*. SYMPHONY, however, has only the notion of a cut pool, whereas the implementation of pools in COIN/BCP remains unfinished as of this writing. The object pools are knowledge bases that can be queried by the NP modules for objects to be added to the current LP relaxation. Although there is only one node pool, multiple object pools can be used. The implementation of this feature is described below.

Maintaining and Scanning the Pool. As discussed above, our only significant experience with object pools is with SYMPHONY's cut pools, so what we describe here pertains to those. Many of the concepts should be similar for pools of variables, though clearly there are differences. The cut pool's primary job is to receive a solution from an NP module and return cuts from the pool that are violated by it. The cuts are stored along with two pieces of information—the level of the tree on which the cut was generated, known simply as the

level of the cut, and the number of times it has been checked for violation since the last time it was actually found to be violated, known as the number of *touches*. The number of touches can be used as a simplistic measure of its effectiveness. Since the pool can get quite large, the user can choose to scan only cuts whose number of touches is below a specified threshold and/or cuts that were generated on a level at or above the current one in the tree. The idea behind this second criterion is to try to avoid checking cuts that were not generated "nearby" in the tree, as they are less likely to be effective. Any cut generated at a level in the tree below the level of the current node must have been generated in a different part of the tree. Although this is admittedly a naive method, it has proven reasonably effective in practice.

On the other hand, the user may define a specific measure of quality for each cut to be used instead. For example, the degree of violation is an obvious candidate. This measure of quality must be computed by the user, since the cut pool module has no knowledge of the cut data structures. The quality is recomputed every time the user checks the cut for violation and a running average is used as the global quality measure. The cuts in the pool are periodically sorted by this measure and only the highest quality cuts are checked each time. All duplicate cuts, as well as all cuts whose number of touches exceeds or whose quality falls below specified thresholds, are periodically purged from the pool in order to limit computational effort.

Using Multiple Pools. For several reasons, it may be desirable to have multiple pools. The primary purpose of this feature is to increase scalability when employing multiple NP modules by limiting the number of nodes that a cut pool must service. Another important reason may be to limit the size of each pool for more efficient pool management and more uniform distribution of memory requirements. A secondary reason for maintaining multiple cut pools is that it allows us to limit the scanning of cuts to only those that were generated in the same subtree. As described above, this helps focus the search and should increase the efficiency and effectiveness of the search. This idea also allows us to generate locally valid cuts, such as the classical Gomory cuts [37].

With multiple pools, each pool is initially assigned to a particular node in the search tree. After being assigned to that node, the pool services requests for cuts from that node and all of its descendants until such time as one of its descendants is assigned to another cut pool. After that, it continues to serve all the descendants of its assigned node that are not assigned to other pools. Initially, the first pool is assigned to the root node. All other pools are unassigned. During execution, when a new node is selected for processing, the tree manager must determine which pool will service the node. The default is to assign the same pool as that of its parent. However, if there is currently an idle pool (either it has never been assigned to any node or all the descendants of its assigned node have been processed or reassigned), then that cut pool can be assigned to the new node. The new pool is initialized with all the cuts currently in the cut pool of the parent node, after which the two pools operate independently on their respective subtrees. When generating cuts, the NP module sends the new cuts to the cut pool assigned to service the node during whose processing the cuts were generated.

5.4 Performance and Scalability

To study the behavior of our basic approach, we have tested the performance and scalability of our solver using the sample application distributed with SYMPHONY 3.0, which is a pure branch and cut solver for the vehicle routing problem (VRP). Details of the implementation for the VRP solver are contained in [41]. The experiments were run on a Beowulf cluster with 48 dual-processor nodes (1 GHz Pentium III), but only one processor was employed at each node. The operating system was Red Hat Linux 7.2 with OSL 3.0 used to solve the linear programming relaxations. To assess performance, we explicitly measured the idle time and the time spent performing tasks associated with communication. We also tracked the number of nodes in the search tree as a measure of redundant work.

In parallel branch and cut, the size of the search tree is subject to a good deal of random fluctuation due to the unpredictable timing of asynchronous messages. To reduce the effect of these inevitable random fluctuations on the analysis, we have done two things. First, we performed three identical runs of each experiment. The results that appear in the charts are averages over these three runs. Second, we consider some of the timing information on a "per node" basis, that is, the running time is divided by the number of nodes generated. This results in a much more consistent view of some trends as the number of processors increases. The instances, the source code used in these experiments, and the full results in spreadsheet form are available on the World Wide Web at `www.branchandcut.org`.

We report here on two sets of experiments. The results of the experiments are shown in Figures 3–10. We first tested the code "out of the box" with the default settings used when running sequentially. These results are labeled by the term "Default" above the corresponding charts. After noting that the two primary scalability issues were ramp-up time and a bottleneck at the cut pool, we tried to address these issues by (1) implementing a quick branching strategy that we hoped would reduce ramp-up time and (2) scaling the number of cut pools together with the number of NP modules. These attempts at improving scalability met with only modest success; they are summarized in the charts labeled "Production." All charts have along the horizontal axis the number of NP modules used to perform the calculation (4, 8, 16, or 32), as these modules perform the vast majority of the calculation. Along the vertical axis is the relevant time scale, and each dot in the charts represents a problem instance from our test set. The charts thus provide a comprehensive visual summary of our results. The magnitude of these experiments in terms of the number of processors employed is relatively small, but we can nonetheless glean a great deal of information from them.

Section 4.1 describes in general terms the main sources of overhead in parallel branch and bound. The results in Figures 3–10 break these sources down further into the specific identifiable components that were found to be significant to performance. By examining the results in the charts, we can draw some interesting conclusions. Figure 3 shows that the total number of nodes in the search tree, a measure of the amount of redundant work performed, remains constant as the number of processors is increases (a simple statistical test supports this claim). This is a good indication that the single-node-pool approach, which affords us global information about bounds and allows us to process the nodes in

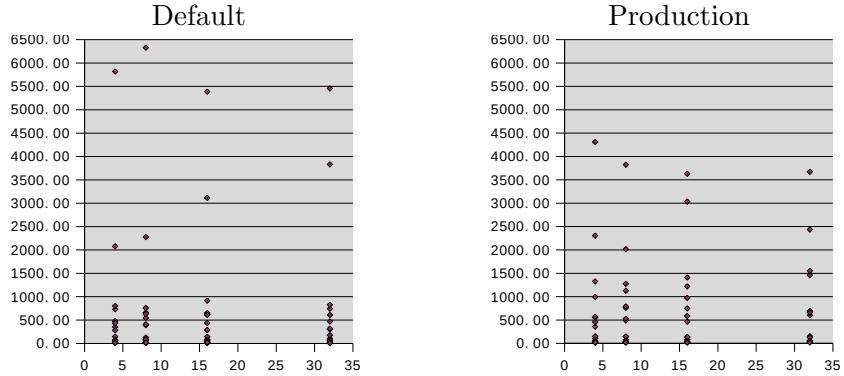

Figure 3: Number of nodes generated for default and production settings.

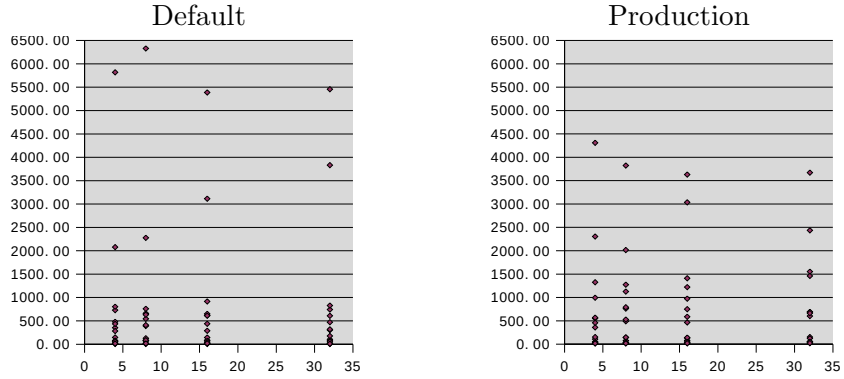

Figure 4: CPU time (processor-seconds) for default and production settings.

approximately the same order as they would be processed in the sequential algorithm, is effective in preventing the performance of redundant work. In addition, Figures 4 (total raw CPU time used for each instance) and 5 (total CPU time used per search tree node) show that the total amount of CPU time required to process the nodes is also remaining constant.

Besides redundant work, the main components of overhead are idle time and time spent performing tasks related to communication. We have found that the time spent in functions related to communication is insignificant compared to the much bigger issue of idle time. The sources of idle time are analyzed in the next two sections.

Ramp-up and Ramp-down. Because of the iterative bounding scheme employed in BCP, the time to process a node can be substantial. Therefore, ramp-up time is one of the most serious scalability problems we face. Ramp-down time, on the other hand, was a much smaller problem. Ramp-up time can be controlled to some degree through the

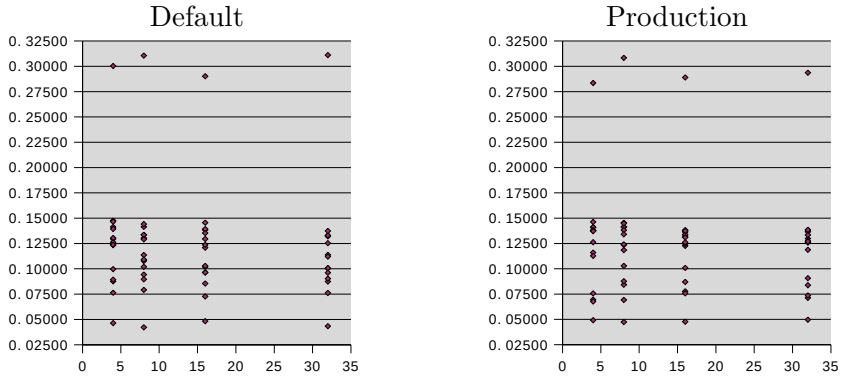

Figure 5: CPU time (processor-seconds per node) for default and production settings.

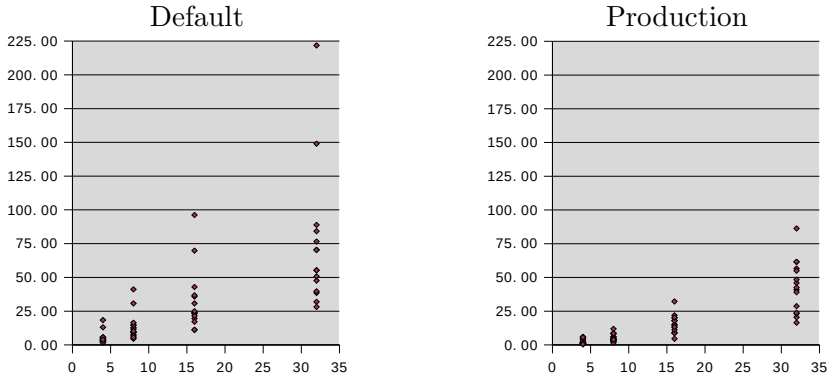

Figure 6: Total ramp time (seconds) for default and production settings.

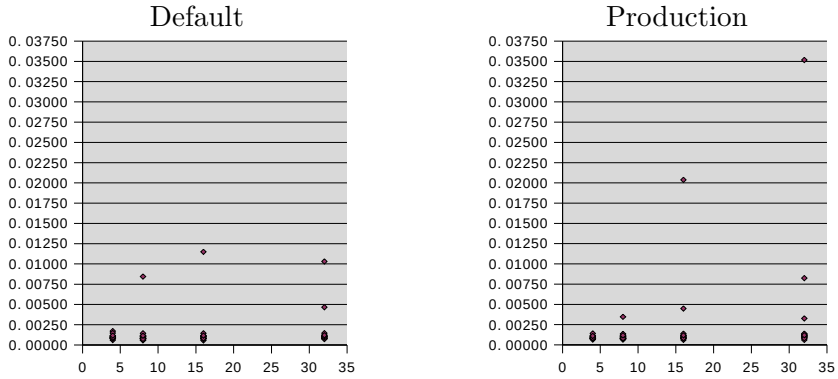

Figure 7: Idle time (seconds per node) waiting for nodes for default and production settings.

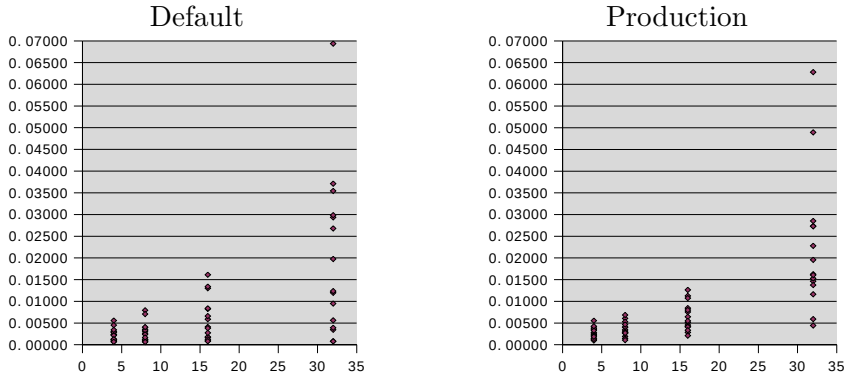

Figure 8: Idle time (seconds per node) waiting for cuts for default and production settings.

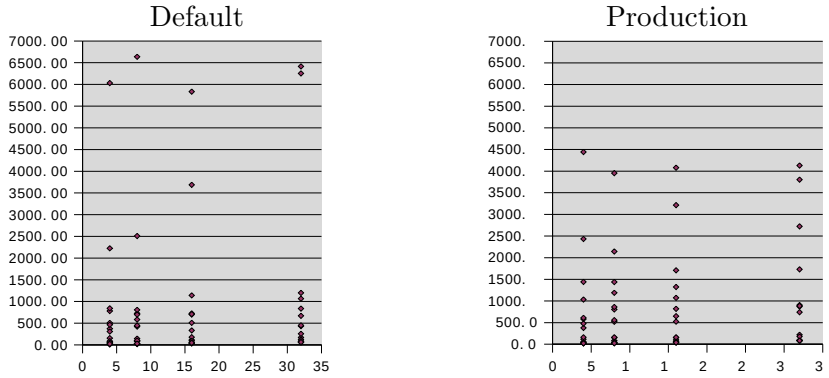

Figure 9: Wall clock time (processor-seconds) for default and production settings.

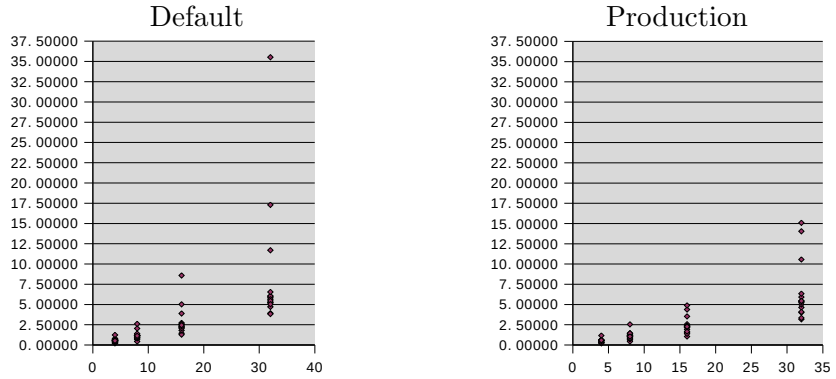

Figure 10: Wall clock time (processor-seconds per node) for default and production settings.

quick branching scheme described earlier. The results of applying this scheme are shown in Figure 6. The first chart is the total ramp time (ramp-up and ramp-down) with default branching, whereas the second chart is the ramp time with quick branching. We first tried branching after only one LP relaxation had been solved, but found that the size of the search tree increased too much with this rule. Forcing branching after 5 LP relaxations had been solved was a better compromise and did result in decreased ramp-up time, without a dramatic increase in the size of the of the search trees (compare the charts in Figure 3). The overall reduction in computational effort, however, was modest. A possible alternative is to introduce a branching rule that results in an increased number of children early in the solution process. In [23], Henrich studies this problem and offers some alternatives that we consider in future work. Obviously, this is an important topic ripe for further study.

Idle Time (Requests for Data). Another major contributor to parallel overhead is the time spent by the node-processing modules waiting for requests to be serviced by a knowledge base. We address the two types of knowledge bases separately.

- *Tree Manager (Node Pool)*: Requests to the tree manager can be of three types: (1) request for a new node description, (2) request for a decision about whether to dive or not, or (3) request for a list of indices to be assigned to newly generated objects. With the differencing scheme we use to store the search tree, reconstructing the node descriptions needed to service requests of type (1) can involve some computational effort. Because there is only one tree manager module to service all of these requests, this is obviously a scalability issue when large numbers of processors are used. Of the three types of requests, requests for node descriptions are the most significant cause of idle time (due to the large message size and the expense of constructing the node description). Surprisingly, however, Figure 7 shows that idle time per node due to requests from the tree manager remain fairly constant in most cases, with a slight increase in the case of 32 NP modules. This upward trend is likely to continue; for large-scale parallelism, the tree manager would become a bottleneck. However, we were pleasantly surprised to see that this did not occur to a significant extent in our experiments.

- *Object Pool*: Requests to the object pools (cut pools in this case) can involve significant computational effort, since the entire pool must be scanned for relevant objects and the pools can grow quite large. This can cause the object pools to become significant bottlenecks as well. However, it is important to realize that the time spent actually scanning the pool is useful work that would also have been performed in the sequential algorithm. It is only the time spent waiting for the cut pool process to receive the message and actually perform the work, as well as the time spent waiting for the cuts to be sent back over the network that can be considered overhead. Currently, we send each cut in a separate message in order to allow the node processing module to begin processing the list of cuts while the cut pool is still working. However, the additional latency that this causes is an issue. We attempted to deal with this through the use of multiple object pools servicing different parts of the search tree. The results

are shown in Figure 8. We can see that this approach did have an effect, but as with ramp-up time, the idle time per node is still increasing significantly.

The overall efficiency can be seen in Figures 9 and 10. Figure 9 shows the total raw wall clock time for each instance, multiplied by the number of processors. With an efficiency of 1, this quantity should stay constant as the number of processors increases. Figure 10 shows the wall clock processing time per search tree node, which should also remain constant with perfect efficiency. Although the efficiency does decrease due to the sources of overhead discussed above, our design is quite adequate for small-scale use and can achieve efficiencies near one. However, this is largely due to the fact that our single-node-pool approach provides essentially perfect information with respect to bounds, which in turn allows us to keep the performance of redundant work in check. To take advantage of large-scale parallelism, we need to change our approach. This is the goal of our current project, which we summarize in the remainder of the paper.

6 The Next-Generation Framework

In order to address some of the scalability issues inherent in the design of SYMPHONY and COIN/BCP, we are currently developing a next-generation framework as a follow-on to these codes. We aim to create a framework with improved scalability, like that in PICO [15] or FATCOP [12], which will allow us to employ true large-scale parallelism. At the same time, we plan to allow even greater flexibility than the currently existing frameworks, SYMPHONY, COIN/BCP, and ABACUS do. To make our new code easy to maintain, easy to use, and as flexible as possible, we have developed a multi-layered class library, in which the only assumptions made in each layer about the algorithm being implemented are those needed for implementing specific functionality efficiently. By limiting the set of assumptions in this way, we ensure that the libraries we are developing will be useful in a wide variety of settings. To illustrate, we briefly describe the current hierarchy design. For additional details, see [28].

6.1 The Library Hierarchy

The Abstract Library for Parallel Search. The ALPS layer is a C++ class library containing the base classes needed to implement parallel search, including basic search tree management and load balancing. In the ALPS base classes, there are almost no assumptions made about the algorithm that the user wishes to implement, except that it is based on a tree search. ALPS performs its tree management in much the same way as most branch and bound frameworks, by prioritizing the search through the use of an application-dependent notion of *node quality* and pruning those nodes whose quality falls below a specified *quality threshold*. However, since we are primarily interested in *data-intensive* applications, in which the description of each search tree node might be very large, the class structure is designed with the implicit assumption that efficient storage of the search tree is paramount and that this storage is accomplished through the compact differencing scheme alluded to

earlier. Also associated with a node is a status that indicates the amount of processing that has been completed and what remains to be done. This is similar to the scheme employed in PICO. A preliminary version of ALPS has already been developed and used to implement a solver for the knapsack problem.

The Branch, Constrain, and Price Software Library. BiCePS is a C++ class library built on top of ALPS that implements the data handling layer appropriate for a wide variety of relaxation-based branch and bound algorithms where the bounding is based on Lagrangian duality. BiCePS abstracts the basic notions of BCP without assuming that the algorithm is LP-based. Such a framework could be use to implement branch and bound algorithms based some form of Lagrangian relaxation (relax and cut for example) or using a nonlinear solver (such as a quadratic programming solver). In BiCePS, we have a more generalized notion of variables (which we call *primal objects*) and constraints (which we call *dual objects*), in which the dual objects are viewed simply as arbitrary functions of the primal objects. Subproblems are composed of lists of primal and dual objects, each with associated lower and upper bounds between which the value of the object must fall. Objects also have an associated marginal cost or *dual value*.

As in BCP, we assume that processing a subproblem involves an iterative bounding procedure, in which the list of active objects (both primal and dual) can be changed in each iteration by *generation* and *deletion*, and individual objects modified through *bound tightening*. The design also provides for multi-phase approaches, as described in Section 5.3.2. To perform branching, we choose a *branching set* consisting of both a list of data objects to be added to the relaxation (possibly only for the purpose of branching) and a list of object bounds to be modified. Any object can have its bounds modified by branching, but the union of the feasible sets contained in all child subproblems must contain the original feasible set in order for the algorithm to be correct.

The BiCePS Linear Integer Solver Library. BLIS is a concretization of the BiCePS library in which we implement an LP-based relaxation scheme, i.e., we assume the objective function and all constraints are linear functions of the variables and that the relaxations are LPs. The primal objects (the variables) correspond to columns in an LP relaxation, while constraints correspond to rows.

6.2 Scalability Features

Task Granularity. The most straightforward approach to improving scalability is to increase the task granularity, thereby reducing the number of decisions that need to be made centrally, as well as the amount of data that has to be sent and received. To achieve this, the basic unit of work in our design is an entire *subtree*. This means that each worker (the worker process is described in the next section) is capable of processing an entire subtree autonomously. The implications of changing the basic unit of work from a subproblem to a subtree are vast. Although this change allows for increased grain size and more compact

storage, it does make some parts of the implementation much more difficult. For instance, because of the memory savings achieved by compactly storing entire subtrees using differencing, we must be much more careful about how we perform load balancing, in order to avoid breaking up subtrees too finely.

The Master-Hub-Worker Paradigm. To avoid the bottleneck created by the classical master-worker paradigm, we have shifted to a variant of the so-called *master-hub-worker* paradigm, which is similar to that proposed and implemented by Eckstein in his PICO framework [15]. This paradigm inserts a layer of "middle managers," called *hubs* between the master process and the worker processes. In this scheme, each hub is responsible for managing a fixed-size cluster of workers. As the number of processors increases, we simply add more hubs and more clusters of workers. In this way, no hub becomes overburdened because the number of workers requesting information from it is limited. In our design, each hub is responsible for tracking a list of subtrees of the current tree that it is responsible for. The hub dispenses new candidate nodes (leaves of the subtrees it is responsible for) to the workers as needed and tracks their progress. When a worker receives a new node, it treats this node as the root of a subtree and begins processing that subtree, stopping only when the work is completed or the hub instructs it to stop. Periodically, the worker informs the hub of its progress and based on that information the hub balances the load among its workers.

Periodically, load balancing must also be performed between the hubs themselves. This load balancing attempts to keep related subtrees together while balancing the *quantity* and the *quality* of nodes yet to be explored among the hubs. This is done by maintaining skeleton information about the full search tree in the master process. This skeleton information includes only what is necessary to make load balancing decisions—primarily the quality of each of the subproblems available for processing. With this information, the master is able to match *donor hubs* (those with too many nodes or too high a proportion of high-quality nodes) and *receiver hubs*, who then exchange work appropriately.

Asynchronous messaging. Another design feature we have endeavored to include is the elimination of synchronous requests for information. This means that every process must be capable of working completely autonomously until interrupted with a request to perform an action by either its associated hub (if it is a worker), or the master process (if it is a hub). In particular, this means that each worker must be capable of acting as an independent sequential solver, as described above. To ensure that a worker is doing useful work, it periodically sends an *asynchronous* message to the hub with information about the current state of its subtree. The hub can then decide at its convenience whether to ask the worker to stop working on the subtree and begin work on a new one. An important implication of this design is that the workers are not able to assign global identifiers to newly generated objects. This has important consequences for data handling, which we explore in the next section.

Data Handling. One of the biggest challenges faced in implementing data-intensive search algorithms is keeping track of the objects that make up the subproblems. The fact that the algorithms we consider have a notion of global objects is, in some sense, what makes them so difficult to implement in parallel. In order to take advantage of the economies of scale that come from having global objects, we must have global information and central storage. But this is at odds with our goal of decentralization and asynchronous messaging. In Section 5.2.1, we have already discussed the difference between an object's *representation* and its *realization* within the context of a particular relaxation.

As in COIN/BCP, new classes of objects can be derived by defining methods, called *encoding* and *decoding*, that convert between an object's compact representation and its realization. The compact form of an object is used not only for efficient storage and transportation of the object, but also as a way of uniquely identifying the object globally. To keep storage requirements at a minimum, we would like to know at the time we generate an object whether we have previously generated the same object (this can easily happen). To determine this, we store object locally within a hash table that allows us to easily determine whether the object already exists. The hash value is generated from an object's encoded form.

Of course, the ideal situation would be to avoid generating the same object twice in the first place. For this purpose, we provide *object pools*, as in SYMPHONY and COIN/BCP. These pools contain sets of the "most effective" objects found in the tree thus far, so that they may be utilized in other subproblems without having to be regenerated.

7 Conclusion

In the first half of this paper, we have provided an overview of the issues involved in implementing sophisticated versions of branch and bound called *branch, cut, and price* algorithms. These algorithms depend on the sharing of large amounts of knowledge in the form of bounds, global objects (cuts and variables), and node descriptions. Implementing this sharing in parallel without introducing undue overhead is difficult at best. In SYMPHONY and COIN/BCP, this is done through centralization of certain knowledge bases, specifically, the node queue. Such a single-node-pool scheme has significant advantages over a decentralized one. Mainly, it is easy to eliminate redundant work and the entire search tree can be stored very compactly using differencing. We have shown that this approach is effective for small-scale parallelism. This centralized approach is not scalable, however, because the central knowledge base will eventually become a bottleneck. We are currently developing a next-generation framework designed to address this and other scalability issues and to generalize many of the notions of parallel BCP. All of the projects mentioned in this paper are being developed as open source as part of the Computational Infrastructure for Operations Research (COIN-OR) Project. Full source code and documentation, as well as a description of the project, is available for free download at `www.coin-or.org`.

References

[1] G. M. Amdahl (1967): Validity of the single-processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings* **30**, Atlantic City, NJ, April 18–20, AFIPS Press.

[2] K. Anstreicher, N. Brixius, J. Goux, J. Linderoth (2002): Solving large quadratic assignment problems on computational grids. *Mathematical Programming, Series B* **91** 563.

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook (1998): On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians*, 645.

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook: *CONCORDE TSP Solver*, available at `www.keck.caam.rice.edu/concorde.html`.

[5] E. Balas (1965): An additive algorithm for solving linear programs with zero-one variables. *Operations Research* **13**, 517–546.

[6] E. Balas, S. Ceria, and G. Cornuéjols (1996): Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science* **42**, 9.

[7] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, P. H. Vance (1998): Branch-and-price: Column generation for huge integer programs. *Operations Research* **46**, 316–329.

[8] M. Benchouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol (1996): Building a parallel branch and bound library. In *Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science* **1054**, Springer, Berlin, 221.

[9] R. Bixby, W. Cook, A. Cox, and E. K. Lee (1995): *Parallel Mixed Integer Programming*, Rice University Center for Research on Parallel Computation Research Monograph CRPC-TR95554.

[10] A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens (1995): *Asynchronous Parallel Branch and Bound and Anomalies*, Report EUR-CS-95-05, Department of Computer Science, Erasmus University, Rotterdam.

[11] Q. Chen, and M. C. Ferris (1999): *FATCOP: A Fault Tolerant Condor-PVM Mixed Integer Programming Solver*, University of Wisconsin CS Department Technical Report 99-05, Madison, WI.

[12] Q. Chen, M. C. Ferris, and J. T. Linderoth (2001): FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research* **103**, 17.

[13] C. Cordier, H. Marchand, R. Laundy, and L. A. Wolsey (1999): bc-opt: A branch-and-cut code for mixed integer programs. *Mathematical Programming* **86**, 335.

[14] R. Correa and A. Ferreira (1995): *Parallel Best-first Branch and Bound in Discrete Optimization: A Framework*, Center for Discrete Mathematics and Theoretical Computer Science Technical Report 95-03.

[15] J. Eckstein, C. A. Phillips, and W. E. Hart (2000): *PICO: An Object-Oriented Framework for Parallel Branch and Bound*, RUTCOR Research Report 40-2000, Rutgers University, Piscataway, NJ.

[16] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi (2001): Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In *Computational Combinatorial Optimization*, D. Naddef and M. Jünger, eds., Springer, Berlin, 157.

[17] M. C. Ferris, G. Pataki, and S. Schmieta (2001): Solving the Seymour problem, *Optima* **66**, 1.

[18] A. Geist et al. (1994): *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT University Press, Cambridge, MA.

[19] B. Gendron and T. G. Crainic (1994): Parallel branch and bound algorithms: Survey and synthesis. *Operations Research* **42**, 1042.

[20] A. Grama and V. Kumar (1995): Parallel search algorithms for discrete optimization problems. *ORSA Journal on Computing* **7**.

[21] M. Grötschel, M. Jünger, and G. Reinelt (1984): A cutting plane algorithm for the linear ordering problem. *Operations Research* **32**, 1155.

[22] W. Gropp, E. Lusk, and A. Skjellum (1999): *Using MPI*, MIT University Press, Cambridge, MA.

[23] D. Henrich (1993): Initialization of parallel branch-and-bound algorithms. *Proceedings of the Second International Workshop on Parallel Processing and Artificial Intelligence*, Chamberry, France.

[24] K. Hoffman and M. Padberg (1985/86): LP-based combinatorial problem solving. *Annals of Operations Research* **4**, 145.

[25] M. Jünger and S. Thienel (2000): The ABACUS system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience* **30**, 1325.

[26] V. Kumar and A. Gupta (1994): Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing* **22**.

[27] L. Ladányi (1996): *Parallel Branch and Cut and Its Application to the Traveling Salesman Problem*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY.

[28] L. Ladányi, T. K. Ralphs, and M. J. Saltzman (2002): Implementing Scalable Parallel Search Algorithms for Data-intensive Applications. *Proceedings of the International Conference on Computational Science*, Amsterdam.

[29] M. Eső, D. L. Jensen, and L. Ladányi (2002): Bid evaluation for FCC auction 31 using column generation. Presentation at the *INFORMS Annual Meeting*, San Jose.

[30] M. Eső, D. L. Jensen, and L. Ladányi (2002): Solving lexicographic multiobjective MIP with column generation. Presentation at the *INFORMS Annual Meeting*, San Jose.

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

[32] J. Linderoth (1998): *Topics in Parallel Integer Optimization*, Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA.

[33] J. Linderoth and M. W. P. Savelsbergh (1999): Computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing* **11**, 173–187.

[34] G. Mitra (1973): Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming* **4**, 155.

[35] G. Mitra, I. Hai, and M.T. Hajian (1997): A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing* **23**, 733–753.

[36] L.G. Mitten (1970): Branch-and-bound methods: General formulation and properties. *Operations Research* **18**, 24–34.

[37] G. L. Nemhauser and L. A. Wolsey (1988): *Integer and Combinatorial Optimization*, John Wiley & Sons, Inc.

[38] G. L. Nemhauser, M. W. P. Savelsbergh, and G. S. Sigismondi (1994): MINTO, a Mixed INTeger Optimizer. *Operations Research Letters* **15**, 47.

[39] M. Padberg and G. Rinaldi (1991): A branch-and-cut algorithm for the resolution of large-scale traveling salesman problems. *SIAM Review* **33**, 60.

[40] T. K. Ralphs (1995): *Parallel Branch and Cut for Vehicle Routing*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY.

[41] T. K. Ralphs (2002): Parallel Branch and Cut for Vehicle Routing. To appear in *Parallel Computing*.

[42] T. K. Ralphs (2002): *SYMPHONY Version 3.0 User's Guide*, available at `www.branchandcut.org/SYMPHONY`.

[43] T. K. Ralphs and L. Ladányi (2001): *COIN/BCP User's Guide*, available at `www.coin-or.org`.

[44] T. K. Ralphs, L. Ladányi, and L. E. Trotter (2001): Branch, cut, and price: Sequential and parallel. In *Computational Combinatorial Optimization*, D. Naddef and M. Jünger, eds., Springer, Berlin, 223.

[45] M. W. P. Savelsbergh (2001): Branch-and-price: Integer programming with column generation. In *Encyclopedia of Optimization*, C. Floudas, P. Pardalos, Eds.

[46] Y. Shinano, M. Higaki, and R. Hirabayashi (1995): Generalized utility for parallel branch and bound algorithms. *Proceedings of the 1995 Seventh Symposium on Parallel and Distributed Processing*, IEEE Computer Society Press, Los Alamitos, CA, 392.

[47] Y. Shinano, K. Harada, and R. Hirabayashi (1997): Control schemes in a generalized utility for parallel branch and bound. *Proceedings of the 1997 Eleventh International Parallel Processing Symposium*, IEEE Computer Society Press, Los Alamitos, CA, 621.

[48] H. W. J. M. Trienekens (1989): *Parallel Branch and Bound and Anomalies*, Report EUR-CS-89-01, Department of Computer Science, Erasmus University, Rotterdam.

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

[50] S. Tschöke, and T. Polzer (1996): *Portable Parallel Branch and Bound Library User Manual*, Library Version 2.0. Department of Computer Science, University of Paderborn.

[51] S. J. Wright (2001): Solving optimization problems on computational grids. *Optima* **65**.