A Library Hierarchy for Implementing Scalable Parallel Search Algorithms

L. Ladanyi
IBM T. J. Watson Research Center

T. K. Ralphs
Lehigh University

M. Saltzman
Clemson University

Report No. 01T-010

A Library Hierarchy for Implementing Scalable Parallel Search Algorithms

L. Ládanyi*, T.K. Ralphs†, and M. Saltzman‡

November 14, 2001

Abstract

This report describes the design of the Abstract Library for Parallel Search (ALPS), a framework for implementing scalable, parallel algorithms based on tree search. ALPS is specifically designed to support *data intensive* algorithms, in which large amounts of data are required to describe each node in the search tree. Implementing such algorithms in a scalable manner is difficult due to data storage requirements. This report also describes the design of two other libraries built on top of ALPS, the first of which is the Branch, Constrain, and Price Software (BiCePS) library, a framework that supports the implementation of parallel branch and bound algorithms in which the bounding is based on some type of relaxation, usually Lagrangean. In this layer, the notion of *global data objects* associated with the variables and constraints is introduced. These global objects provide a connection between the various subproblems in the search tree and present further difficulties in designing scalable algorithms. Finally, we will discuss the BiCePS Linear Integer Solver (BLIS), a concretization of BiCePS, in which linear programming is used to obtain bounds in each search tree node.

1 Introduction

This report describes research in which we are seeking to develop highly scalable algorithms for performing large-scale parallel search in distributed-memory computing environments. To support the development of such algorithms, we have designed the Abstract Library for Parallel Search (ALPS), a C++ class library upon which a user can build a wide variety of parallel algorithms based on tree search.

Initially, we will be interested in using ALPS to implement algorithms for solving large-scale discrete optimization problems (DOPs). DOPs arise in many important applications

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

†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, Clemson University, mjs@clemson.edu

such as planning, scheduling, logistics, telecommunications, bioengineering, robotics, design of intelligent agents, etc. Most DOPs are in the complexity class $\mathcal{NP}$-complete so there is little hope of finding provably efficient algorithms [12]. Nevertheless, intelligent search algorithms, such as *branch, constrain, and price* (BCP), have been tremendously successful at tackling these difficult problems.

To support the implementation of parallel BCP algorithms, we have designed two additional C++ class libraries that are built on top of ALPS. The first, called the Branch, Constrain, and Price Software (BiCePS) library, implements a generic framework for relaxation-based branch and bound. In this library, we make very few assumptions about the nature of the relaxations, i.e., they do not have to be linear programs. The second library, called the BiCePS Linear Integer Solver (BLIS), implements LP-based branch and bound algorithms, including BCP.

The design of these libraries will allow us to develop parallel algorithms that not only have the ability to utilize a very large numbers of processors efficiently, but also have the ability to handle very large problem instances from very difficult problems. The main feature of such algorithms is that they require the maintenance of a vast amount of information about each node in the search tree. However, this data usually does not vary much from parent to child, so we use data structures based on a unique differencing scheme which is memory efficient. This scheme for storing the tree allows us to handle problems much larger than we otherwise could.

A number of techniques for developing scalable parallel branch and bound algorithms have been proposed in the literature [3, 9, 10, 15, 17, 39]. However, we know of no previous work specifically addressing the development of scalable algorithms for data-intensive applications. Standard techniques for parallel branch and bound break down when applied to BCP, primarily because they all depend on the ability to easily shuttle search tree nodes between processors. The data structures we need in order to create efficient storage do not allow this fluid movement. Our design overcomes this difficulty by dividing the search tree into subtrees containing a large number of related search nodes that can be stored together. This requires the design of more sophisticated load balancing schemes that accommodate this storage constraint.

This project builds on previous work in which we developed two object-oriented, generic frameworks for implementing parallel BCP algorithms. SYMPHONY (Single- or Multi-Process Optimization over Networks) [34] is a framework written in C and COIN/BCP [36] is a framework written in the same spirit in C++. Because of their generic, object-oriented designs, both are extremely flexible and can be used to solve a wide variety of discrete optimization problems. Source code and extensive documentation for both frameworks are currently distributed for free to the research community and each is in use at a number universities and research centers world-wide [33].

With respect to sequential computation, these two packages are mature and well refined. They each contain most of the advanced features available in today's optimization codes and occupy the unique position of being the only generic, parallel implementations of BCP we are aware of. Each can achieve linear speedup for small numbers of processors, but

they employ a master-slave paradigm that causes the tree manager (or the cut pool in the case of SYMPHONY) to become a bottleneck when the number of processors is large. Our conclusion is that we are doing *most* things right. We are now in a position to move toward efficient large-scale parallelism.

2 Motivation and Background

2.1 Branch and Bound

In order to define some terminology and have a frame of reference for the way tree search algorithms work, we first describe the basics of branch and bound. After describing branch and bound, we will move on in the next section to describe the LP-based variant of branch and bound we have already referred to called *branch, cut, and price*. The reader who is already familiar with these methods may want to skip to Section 3.

A branch and bound algorithm uses a divide and conquer strategy to partition the solution space into *subproblems* and then optimizes individually over each of them. For instance, let S be the set of solutions to a given problem, and let $c \in \mathbf{R}^S$ be a vector of costs associated with 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. Using branch and bound, we initially examine the entire solution space S. In the *processing* or *bounding* phase, we relax the problem in some fashion. In so doing, we admit solutions that are not in 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 $\hat{s}$, then we are done—either the new solution or $\hat{s}$, respectively, is optimal. Otherwise, we identify n subsets of S, $S_1, \ldots, S_n$, such that $\cup_{i=1}^{n} S_i = S$. Each of these subsets is called a *subproblem*; $S_1, \ldots, S_n$ are also sometimes called the *children* of S. We add the children of S to the list of *candidate subproblems* (those which need processing). This is called *branching*.

To continue the algorithm, we select one of the candidate subproblems, remove it from the list, 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. If it is greater than or equal to our current upper bound, then we may again prune the subproblem. Finally, if we cannot prune the subproblem, we are forced to branch and add the children of this subproblem to the list of active candidates. We continue in this way until the list of active subproblems is empty, at which point our current best solution is the optimal one.

2.2 Branch, Cut, and Price

Branch, cut, and price is a specific implementation of branch and bound used for solving problems that can be formulated as integer programs. Early works such as [18, 20, 30] laid out the basic framework of BCP and since then, many implementations (including

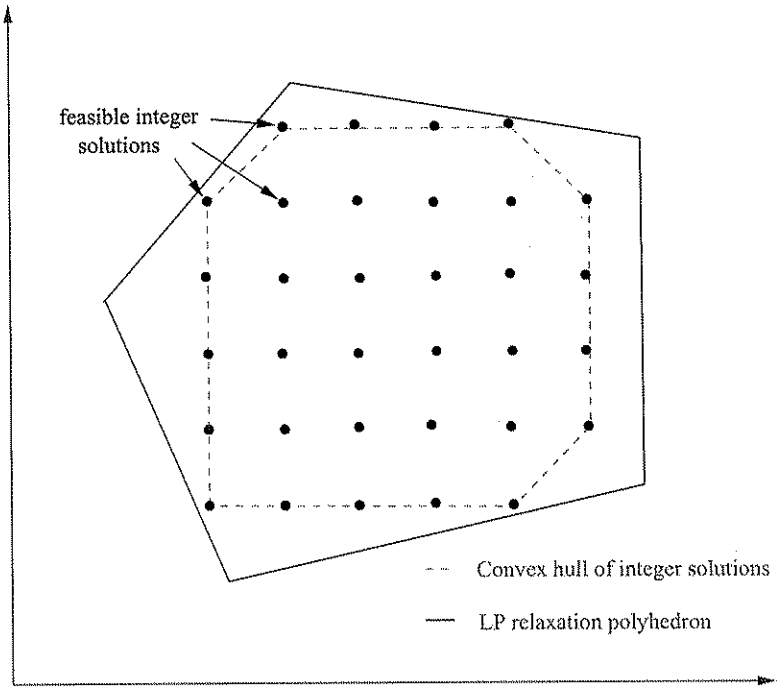

Figure 1: Example of a polyhedron and the convex hull of its integer points

ours) have built on these preliminary ideas. In BCP, the bounding operation is performed using tools from linear programming and polyhedral theory. Since linear programming does not accommodate the designation of integral variables, we typically relax the integrality constraints of an integer program to obtain a *linear programming (LP) relaxation.* This formulation is augmented with additional *constraints* or *cutting planes*, i.e., inequalities valid for the convex hull of solutions to the original problem. In this way, we hope to obtain an integral (and hence feasible) solution.

For example, given $A \in \mathbf{R}^{m \times n}$ and $b \in \mathbf{R}^m$, consider the polyhedron $\mathcal{P} = \{x \in \mathbf{R}^n : Ax \geq b\}$ and define $\mathcal{P}_I := \mathcal{P} \cap \mathbf{Z}^n$, as shown in Figure 2.2. For $c \in \mathbf{R}^n$, the corresponding integer program is:

$$\min\{cx : x \in \mathcal{P}_I\} \tag{1}$$

We have already noted that (1) is equivalent to $\min\{cx : x \in \text{conv}(\mathcal{P}_I)\}$, and so we consider this problem from now on. By Weyl's Theorem (see [29]), we know that there exists a finite set $\mathcal{L}$ of inequalities valid for $\mathcal{P}_I$ such that

$$\mathcal{P}_I = \{x \in \mathbf{R}^n : ax \leq \beta \ \ \forall (a, \beta) \in \mathcal{L}\}. \tag{2}$$

The inequalities in $\mathcal{L}$ are the potential cutting planes to be added to the relaxation as needed. If we knew this list of cutting planes explicitly, we could simply solve the problem using linear programming. Unfortunately, we usually have only an implicit and/or incomplete description of $\mathcal{L}$ and it is difficult, if not impossible, to enumerate all of its members.

Instead, we use *separation algorithms* and *heuristics* to generate them dynamically when they are violated. This is known as *cutting* or *separation.*

In an integer program, we view each component x_i of the solution vector x as a separate *variable* which is associated with the ith column of the matrix A. As with cutting planes, the columns of A can also be defined implicitly if n is large. If column i is not present in the current matrix, then variable x_i is implicitly taken to have value zero. The process of dynamically generating variables is called "pricing" for reasons that derive from the underlying techniques used to perform this operation.

Once we have failed to either prune the current subproblem or separate the current fractional solution from $\mathcal{P}_I$, we are forced to branch. The branching operation is accomplished by specifying a set of hyperplanes which divide the current subproblem (polyhedron) in such a way that the current solution is not feasible for the LP relaxation of any of the new subproblems. For example, in a combinatorial optimization problem, branching could be accomplished simply by fixing a variable whose current value f is fractional to be less than or equal to $\lfloor f \rfloor$ in one branch and greater than or equal to $\lceil f \rceil$ in the other.

2.3 Parallel Scalability

Since one of the primary design goals of our new library is *scalability*, we now define more precisely what we mean by that term. Generally speaking, the scalability of a parallel system (defined as the combination of a parallel algorithm and a parallel architecture) is the degree to which it is capable of efficiently utilizing increased computing resources (usually processors). To assess this capability, we need to 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 used as the basis for comparison and is usually 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 in question and depends on p, the number of processors available. The *speedup* (S_p) is the simply the ratio T_0/T_p and hence also depends on p. Finally, the efficiency (E_p) is the ratio S_p/p of speedup to number of processors.

Amdahl was the first to note that the theoretical maximum speedup for an algorithm is limited by the amount of time it has to spend performing inherently sequential tasks, such as reading in the problem data [1]. Considering a particular problem instance, if the fraction of time spent by the sequential algorithm performing inherently sequential tasks is s (called the *sequential fraction*), then the speedup is limited to $1/s$ since the parallel running time can never drop below $s \cdot T_0$. Since Amdahl's paper, this fact has been cited by critics as a reason why massive parallelism will never work.

It is easy to see that this argument is flawed. It is true that if the problem size is kept constant, efficiency drops as the number of processors increases; however, a number of authors have also pointed out that if the number of processors is kept constant, then efficiency generally *increases* as problem size increases [24, 17, 19]. This is because the sequential fraction is not constant but rather tends to become smaller as problem size increases. This

led Kumar and Rao to suggest a measure of scalability called the *iso-efficiency function* [23], which measures the rate at which the problem size has to be increased with respect to the number of processors in order to maintain a fixed efficiency. This is the measure of scalability we use.

2.4 Scalability Issues for Parallel Search Algorithms

There are several important issues to consider in designing scalable, data-intensive tree search algorithms. First, we must address the fundamental issue of *control mechanisms*, i.e., the methods by which decisions should be made, primarily regarding which search tree nodes should be processed and in what order. Currently, SYMPHONY and BCP use a master-slave paradigm which centralizes all control with the *tree manager*. The design of control mechanisms is closely tied to the issues of *load balancing*, the method that ensures that all processors have useful work to do, and *fault tolerance*.

A somewhat deeper and perhaps more important issue for data-intensive algorithms is *data handling*. In algorithms such BCP, there are a huge number of *data objects* (see Section 3.1.2 for a description of these) that must be efficiently generated, manipulated, and stored in order to solve these problems efficiently. For these algorithms, the speed with which we can process each node in the search tree depends largely on the number of objects that are *active* in the subproblem. Thus, we attempt to limit the set of active objects in each subproblem to only those that are necessary for the completion of the current calculation. However, this approach requires careful bookkeeping. This bookkeeping becomes more difficult as the number of processors increases.

Finally, there is the issue of *ramp up* and *ramp down* time. *Ramp up* time is the time during which processors are idle simply because there hasn't yet been enough work generated to keep all of them busy. *Ramp down* time is the period at the end of the algorithm when there is not enough remaining to keep all of the processors busy. Long periods of ramp up and ramp down can be extremely detrimental to scalability. We must bear this in mind in our design.

2.4.1 Control Mechanisms

In theory, both efficient control mechanisms and efficient data handling favor a centralized approach for several reasons. With regard to control, the advantages are obvious. Decentralized control mechanisms inevitably result in (1) sub-optimal decisions about which subproblems to process due to a lack of global information, (2) sub-optimal load balancing resulting in lower processor utilization, (3) increased communication overhead as load balancing has to be achieved by shuttling data from overloaded processors to idle processors, and (4) decreased fault tolerance.

With regard to data handling, again the advantages are obvious. Since objects are global in nature, any attempt to store information about the tree locally necessarily results in duplication and increased global memory usage. This is underscored by the differencing

scheme we use for storing the search tree.

Despite this, practical issues dictate that a central approach will suffer from limited scalability because the central resource manager will eventually either become computationally overburdened or suffer from a lack of available bandwidth. This results in increased idle time among the processors waiting for replies from the central manager. Memory considerations also limit the degree to which data storage can be centralized.

All this highlights the tradeoff that we are forced to make. We must choose a scheme somewhere along the continuum between complete centralization and complete decentralization that achieves the proper balance for our notion of scalability. In Section 3, we will describe such a scheme.

2.4.2 Load Balancing

One of the biggest problem with decentralized control mechanisms is the resulting difficulty in *load balancing*. In order to maintain high parallel efficiency, it is critical not only to keep each processor busy, but to keep each processor busy with *useful work*. Hence, as in [17], we differentiate between two different notions of load balancing—quantitative load balancing and qualitative load balancing.

Quantitative load balancing consists of ensuring that the amount of work allocated to each processor is approximately equal. In the centralized approach, this is easy. Whenever a processor runs out of work, the central resource manager can simply issue more. However, in a fully decentralized system, we don't have this luxury. A processor that needs work must either sit idle or request work from another processor.

Qualitative load balancing, on the other hand, consists of ensuring not only that each processor has enough work to do, but also that each processor has *high-quality* work to do. A centralized resource manager can easily issue the "best" work that is available at that time, i.e., the work that is most likely to be productive in completing the solution process. Decentralization of control is even more problematic with respect to this measure of load balancing, as it becomes impossible for a processor to recognize whether its work pool contains high quality nodes without some sort of global information.

2.4.3 Data Handling

As we have already mentioned, the biggest challenge in implementing the parallel search algorithms we are most interested in, i.e., BCP algorithms, is that they are extremely data intensive. Furthermore, the data objects (i.e., variables and constraints) that these algorithms deal with are usually *global* in nature. We will discuss global objects in more detail in Section 3.1.2. It is the existence of these global objects connecting subproblems in different parts of the search tree that creates some of the most challenging scalability issues we face. To keep track of these objects, we must have a scheme for identifying them globally in order to minimize duplicate storage while still maintaining decentralized control.

To give the reader some intuition as to the magnitude of this issue, consider the Traveling Salesman Problem (TSP). Given a set of n cities, the TSP is the problem of determining the shortest route beginning and ending at a designated city and going through all other cities. For the Traveling Salesman Problem with 120 cities, which is considered a small instance, the list of inequalities $\mathcal{L}$ necessary to completely describe the corresponding polytope, as described in Section 2.2, is at least 10^{100} times the number of atoms in the known universe. At the same time, the number of variables is more than 14×10^3. Of course, not all these are necessary for the solution of a particular problem instance. Nonetheless, these are daunting statistics to say the least. We will discuss how this is handled in more detail in later sections.

2.4.4 Fault Tolerance

When solving large problems on networks whose processors may fail, fault tolerance is important. However, as the reader may well imagine, designing for fault tolerance gets more difficult as the degree of decentralization goes up. There are two primary ways in which fault tolerance can be maintained—by periodically backing up data to a hard drive; or by duplicating the data elsewhere in the tree so that it can be restored if it is lost. Both of these options are potentially detrimental to efficiency. These issues will be addressed more fully in Section 3.4.

3 Overview of Library Design Features

3.1 The Library Hierarchy

To make the 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 will briefly describe the current hierarchy. Details of the class structure will be given in Section 4.

3.1.1 The Abstract Library for Parallel Search

The ALPS layer is a C++ class library containing the base classes needed to implement the parallel search handling, including basic search tree management and load balancing. In the ALPS base classes, there are almost no assumption made about the algorithm that the user wishes to implement, except that it is based on a tree search. Since we are primarily interested in *data-intensive* applications, 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 we alluded to earlier. This means that the implementation must define methods for taking the difference of two nodes and for producing an explicit representation of a node from a compact one. More details will be

discussed in Section 4.2. Note that this does not preclude the use of ALPS for applications that are not data intensive. In that case, the differencing scheme need not be used.

In order to define a search order, ALPS assumes that there is a numerical *quality* associated with each subproblem which is used to order the priority queue of candidates for processing. For instance, the quality measure for branch and bound would most likely be the lower bound in each search tree node. For algorithms where there might not be a sensible measure of quality, this quality measure does not have to be used. Default search schemes not requiring it (such as depth first search) are also provided, however, most applications require more sophisticated management of the queue than these methods can provide. In addition, ALPS has the notion of a *quality threshold.* Any node whose quality falls below this threshold is fathomed. This allows the notion of fathoming to be defined without making any assumption about the underlying algorithm. Again, this quality threshold does not have to be used if it doesn't make sense.

Also associated with a search tree node is its current status. In ALPS, there are only four possible stati indicating whether the subproblem has been processed and what the result was. The possible stati are:

- `candidate`: Indicates the node is a candidate for processing. This means it is a leaf node in the search tree.
- `active`: Indicates the node is currently being processed.
- `processed`: Indicates the node has been processed, but not fathomed. This means it is a non-leaf node in the search tree.
- `fathomed`: Indicates the node has been fathomed. In most cases, fathomed nodes are deleted immediately, and hence there will not usually be any nodes with this status in the tree.

In terms of these stati, processing a node involves converting its status from `candidate` to either `processed` or `fathomed`. In the case where it gets turned into an internal node, there must be a method of branching. Within the context of ALPS, this simply means a method of generating descendants of the current subproblem and filling out the `quality` field, if appropriate.

3.1.2 The Branch, Constrain, and Price Software Library

Primal and Dual Objects

BiCePS is a C++ class library built on top of ALPS, which implements the data handling layer appropriate for a wide variety of relaxation-based branch and bound algorithms. In this layer, we introduce the concept of a `BcpsObject`, which is the basic building block of a subproblem. The set of these objects is divided into two main types, the *primal objects* and the *dual objects.* Each of the primal objects has a value associated with it that must lie

within an interval defined by a given upper and lower bound. Once can think of the primal objects as being the *variables.*

In order to sensibly talk about a relaxation-based optimization algorithm, we need the notion of an objective function.[1] The objective function is defined simply as a function of the values of the primal objects. Hence, given these values, we can compute the corresponding value of the objective function. Given the concept of an objective function, we can simply define the set of dual objects similarly as functions of the values of the primal objects. As with the values of the primal objects, these function values are constrained to lie within certain given bounds. Hence, one can think of the dual object as the *constraints.*

It is important to note that the notion of primal and dual objects is a completely general one. We need only assume the very general idea that there exists a set of objects with associated values that must lie between certain bounds. In order to have some notion of optimization, one must then be able to sensibly define the notion of an objective function on this set of objects. From the notion of an objective function, the notion of a constraint naturally follows. Once we have constraints, we can easily apply the general theoretical framework of Lagrangean duality.

In Lagrangean duality, each constraint has both a *slack* value and the value of a *dual multiplier* or *dual variable* associated with it. Similarly, we can associate a pair of values with each variable, which are the *reduced cost* and the value of the primal variable itself. Note that in linear programming, the reduced cost is a concretely defined value and can be easily computed. In our more general setting, we can generalize this idea by defining the reduced cost similarly as the marginal reduction in the objective function value from increasing the value of a particular variable. In the case of a differentiable objective function, computing this value involves taking the partial derivative of the objective function with respect to the variable in question.

Subproblems and Bounding

Although the objects separate naturally into primal and dual classes, our goal is to treat these two classes as symmetrically as possible. In fact, in almost all cases, we can treat these two classes using exactly the same methods. In this spirit, we define a subproblem to be comprised of a set of objects, both primal and dual. These objects are global in nature, which means that the same object may be active in multiple subproblems. The set of objects that are active in the subproblem define the current relaxation which can be solved to obtain a bound. Note that this bound is not valid for the original problem unless either all of the primal objects are present or we have proven that all of the missing objects can be fixed to value zero.

We assume that processing a subproblem is an iterative procedure in which the list of active objects can be changed in each iteration by *generation* and *deletion*, and individual objects modified through *bound tightening.* To define object generation, we need the concepts of *primal solution* and *dual solution*, each consisting of the list of values associated with the corresponding objects. *Object generation* then consists of producing variables

[1] We assume without loss of generality that we wish to minimize the objective function

whose reduced costs are negative and/or constraints whose slacks are negative, given the current primal or dual solutions, respectively (these are needed to compute the slacks and reduced costs).

With all this machinery defined, we have the basic framework needed for processing a subproblem. In overview, we begin with a list of primal and dual objects from which we construct the corresponding relaxation, which can be solved to obtain an initial bound for the subproblem. We then begin to iteratively tighten the relaxation by generating constraints from the resulting primal solution. In addition, we may wish to generate variables. Note that the generation of these objects "loosens" the formulation and hence must be performed strategically. In Section 3.5, we will describe multi-phase approaches in which variable generation is systematically delayed. During each iteration, we may apply bound tightening and use other logic-based methods to further tighten the relaxation. More will be said about how objects are handled and tracked in Section 3.3.

Branching

Having described processing, we now also describe how branching is done. A *branching object* consists of both a list of data objects to be added to the relaxation (possibly only for the purpose of branching) and a list of 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.

3.1.3 The BiCePS Linear Integer Solver Library

BLIS is a concretization of the BiCePS library in which we specify the exact form of the relaxation used to derive the bound during the processing phase. In this case, we specify that we are using an LP-based relaxation scheme. This simply means that we assume the objective function and all constraints are linear functions of the variables and that the relaxations are linear programs. This allows us to define some of the notions discussed above more concretely. For instance, we can now say that a variable corresponds to a column in an LP relaxation, while a constraint corresponds to a row. Note that the form a variable or a constraint takes in a particular LP relaxation depends on the set of objects that are present. In order to generate the column corresponding to a variable, we must have a list of the active constraints. Conversely, in order to generate the row corresponding to a constraint, we must be given the list of active variables. This distinction between the *representation* and *realization* of an object will be explored further is Section 3.3.

Implicit in this discussion is the existence of a data structure for representing the relaxation itself. This data structure determines the form of the realization of a constraint or variable. To allow the use of third-party libraries for solving the relaxation, there must also be a solver interface that converts the internal representation of the relaxation into the representation the solver expects. In BLIS, both the representation of the relaxation and the solver interface are contained in the OSI (Open Solver Interface) class. More details on the handling of objects and relaxations will be given in Section 3.3.

3.2 Improved Scalability

One of the primary goals of this project is to increase scalability significantly from that of SYMPHONY and COIN/BCP. We have already discussed some issues related to scalability in Section 2.4. As pointed out then, this involves some degree of decentralization. However, the schemes that have appeared in the literature are inadequate for data-intensive applications, or, at the very least, would require abandoning our compact data structures. Our new design attempts to reconcile the need for decentralization with our compact storage scheme, as we will describe in the next few sections.

3.2.1 The Master-Hub-Worker Paradigm

We have already pointed out that one of the main difficulties with the master-slave paradigm we employ in SYMPHONY and COIN/BCP is that the tree manager becomes overburdened with requests for information. Furthermore, most of these requests are *synchronous*, meaning that the sending process is idle while waiting for a reply. Our differencing scheme for storing the search tree also means that the tree manager may have to spend significant time simply *generating* subproblems. This is done by working back up the tree undoing the differencing until an explicit description of the node is obtained

Our new design employs a master-hub-worker paradigm, in which a layer of "middle management" is inserted between the master process and the worker processes. In this scheme, each hub is responsible for managing a cluster of workers whose size is fixed. As the number of processors increases, we simply add more hubs and more clusters of workers. However, no hub will become overburdened because the number of workers requesting information from it is limited. This is similar to a scheme implemented by Eckstein [9] in his PICO framework.

In this scheme, each hub is responsible for balancing the load among its workers. Periodically, we must also perform load balancing between the hubs themselves. This is done by maintaining skeleton information about the full search tree in the master process. This skeleton information only includes what is necessary to make load balancing decisions—primarily the lower bound in each of the subproblems available for processing. With this information, the master is able to match *donor hubs* (those with too many nodes or else too high a proportion of high-quality nodes) and *receiver hubs*, who then exchange work appropriately.

This decentralized scheme maintains many of the advantages of global decision making while moving some of the computational burden from the master process to the hubs. We can further reduce the computational burden to the hubs themselves by increasing the task granularity in a manner that we will describe next.

3.2.2 Increased Task Granularity

The most straightforward approach to improving scalability is to increase the task granularity and thereby reduce the number of decisions that need to be made centrally, as well as the amount of data that has to sent and received. To achieve this, the basic unit of work in our design is an entire *subtree*. This means that each worker is capable of processing an entire subtree autonomously and has access to all of the methods used by the tree manager to manage the tree, including setting up and maintaining its own priority queue of candidate nodes, tracking and maintaining the objects that are active within its subtree, and performing its own processing, branching, and fathoming. 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 one of the subtrees it is responsible for) to the workers as needed and track 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.

Besides increasing the grain size, this scheme has the advantage that if we wish to have a sequential implementation, it is simply a matter of building the application using only the worker process. Passing the root node of the tree to a worker process is all that is needed to solve the problem sequentially.

The implications of changing the basic unit of work from a subproblem to a subtree are vast. Although this allows for increased grain size, as well as more compact storage, it does make some parts of the implementation much more difficult. For instance, we must be much more careful about how we perform load balancing in order to try to keep subtrees together. We must also have a way of ensuring that the workers don't go too far down an unproductive path. In order to achieve this latter goal, each workers must periodically check in with the hub and report the status of the subtree it is working on. The hub can then decide to ask the worker to abandon work on that subtree and send it a new one. An important point, however, is that this decision making is always done in an asynchronous manner. This feature is described next.

3.2.3 Asynchronous messaging

Another design feature which increases scalability 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 the workers are doing useful work, they periodically send 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.

Another important implication of this design is that the workers are not able to ask the

hub for help in assigning indices to newly generated objects. While this may not seem like an important sacrifice, it actually has a significant effect on the way objects are handled and could have serious repercussions on efficiency. This is one of the questions we will address during future work.

3.3 Improved Data Handling

Besides effective control mechanisms and load balancing procedures, the biggest challenge we face in implementing search algorithms for data-intensive applications is keeping track of the objects that make up the subproblems. Before describing what the issues are, we describe the concept of objects in a little more detail.

3.3.1 Object Representation

We stated in Section 3.1.2 that an object can be thought of as either a variable or a constraint. However, we did not really define exactly what the terms variable or constraint mean as *abstract* concepts. For the moment, consider an LP-based branch and bound algorithm. In this setting, a "constraint" can be thought of as a method for producing a valid row in the current LP relaxation, or, in other words, a method of producing the projection of a given inequality into the domain of the current set of variables. Similarly, a "variable" can be thought of as a method for producing a valid column to be added to the current LP relaxation. This concept can be generalized to other settings by requiring each object to have an associated method which performs the modifications appropriate to allow that object to be added to the current relaxation.

Hence, we have the concept that an object's *representation*, which is the way it is stored as a stand-alone object, is inherently different from its *realization* in the current relaxation. An object's representation must be defined with respect to both the problem being solved and the form of the relaxation. To make this more concrete, consider the subtour elimination constraints from the well-known Traveling Salesman Problem. These constraints involve the variables corresponding to the edges across a cut in a given graph. In this case, the only data we really need to compute the realization of this constraint, given a particular set of active variables, is the set of nodes on one shore of the cut. When we want to add this matrix row to a particular relaxation, we can simply check the edge corresponding to each active variable and see whether it crosses the cut defined by the given set of nodes. If so, then the variable corresponding to that edge gets a coefficient of one in the resulting constraint. Otherwise, it gets a coefficient of zero. Hence, our representation of the subtour elimination constraints could simply be a bit vector specifying the nodes on one shore of the cut. This is a much more compact method of representing the constraint than the alternative of explicitly storing the list of edges that cross the cut.

Within BiCePS, new categories of objects can be defined by deriving a child class from the `BcpsObject` class. This class holds the data the user needs to realize that type object within the context of a given relaxation and also defines the method of performing that

realization. When sending an object's description to another process, we need only send the data itself, and even that should be sent in as compact a form as possible. Therefore the user must define for each category of variable and constraint a method for *encoding* the data compactly into a character array. This form of the object is then used whenever the object is sent between processes or when it has to be stored for later use. This encoding is also used for another important purpose which is discussed in Section 3.3.3. Of course, there must also be a corresponding method of decoding as well. See Section 4.2.1 for more details on the implementation of encoding and decoding.

3.3.2 Core and Indexed Objects

Of course, we cannot add variables or constraints to an empty relaxation, so we must have the idea of *core objects* and a *core relaxation.* The core relaxation consists of objects that are present in every subproblem throughout the tree. Hence, its form does not change and it can simply be stored and used again whenever a new node is created. After creating the core, the additional variables and constraints can be added using the defined method of realization.

If it is possible for the user to uniquely index a particular class of objects so that the object can be realized solely by knowing its index, then this is called an *indexed class.* Indexed objects can be treated much more efficiently than other classes because they can be represented simply by an index. For example, when the variables correspond to the edges of a graph, these variables can simply be indexed using a lexicographic ordering of the edges.

3.3.3 Tracking Objects Globally

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.

To keep storage requirements at a minimum, we would like to know at the time we generate an object whether or not we have previously generated that same object somewhere else in the tree. If so, we would like to somehow refer to the original copy of that object and delete the new one. Ideally, we would only need to store one copy of each object centrally and simply copy it out whenever it is needed locally. Of course, this is not achievable in practice.

Instead, what we do is simply ensure that we have at most one copy of each object stored locally within each process. We do not try to eliminate duplicate copies that may exist in other locations. This is is the limit of what we can do without synchronous messaging. The way that this is done is to store all of the active objects in a hash table. To generate a hash value for each object, we first encode the object and then apply a hash function to the

encoded form. Then we compare the encoded form to existing objects that are already in the corresponding bucket in the table to determine if another copy of the object has already been generated. If so, then we delete the new copy and simply point to the old copy. This scheme allows us to efficiently track and compare very large numbers of objects. Note that we can have a separate hash table for primal and dual objects, as well as for each object category.

This scheme works fine in the worker processes because when new objects are generated, the new copy is only pointed to by one subproblem and it is easy to change the pointer. In the hub, however, when an entire subtree and its associated objects get sent back from a worker, we need to go through the entire list of new objects and ensure that they are not duplicated in the hub. We then must ensure that each object that is duplicated is replaced by the existing copy and the pointers for the entire subtree updated correctly.

There is yet one further level of complexity to the storage of objects. In most cases, we need only keep objects around while they are actually pointed to, i.e., are active in some subproblem that is still a candidate for processing. If we don't occasionally "clean house," then the object list will continue to grow boundlessly, especially in the hubs. To take care of this, we use smart pointers which are capable of tracking the number of times they have been pointed to by some external entity and are deleted automatically after there are no more pointers to that object.

3.3.4 Object Pools

Of course, the ideal situation would be to avoid generating the same object twice in the first place if possible. For this purpose, we provide *object pools*. These pools contain sets of the "most effective" objects found in the tree so far so that they may be utilized in other subproblems without having to be regenerated. These objects are stored using a scheme similar to that for storing the active objects, but their inclusion in or deletion from the list is not necessarily tied to whether they are active in any particular subproblem. Instead, it is tied to a rolling average of a quality measure that can be defined by the user. By default, the measure is simply the slack or reduced cost calculated when the object is checked against a given solution to determine the desirability of including that object in the associated subproblem.

In addition to maintaining this global list, the object pool must receive solutions from the worker processes and return objects from the pool that have a negative reduced cost or negative slack. These objects can then be considered for inclusion in the appropriate subproblem. Clearly, the object pools are another potential bottleneck for the parallel version of the algorithm. In order to minimize this potential, we store only the most relevant objects (as measured by quality), and we check only objects that seem "likely" to be useful in the current relaxation. The objects are stored along with three pieces of information: (1) the aforementioned measure of quality; (2) the level of the tree on which the object was generated, known simply as the *level* of the object; and (3) the number of times it has been checked since the last time it was actually used, known as the number of *touches*. The quality is the primary measure of effectiveness of an object and only

objects of the highest quality are checked. The number of touches an object has can be used as a secondary, if not somewhat simplistic, measure of its effectiveness. The user can choose to scan only objects whose number of touches is below a specified threshold and/or objects that were generated on a level at or above the current one in the tree. There are intuitive reasons why these rules should work well and, in practice, they do a reasonable job of limiting the computational effort of the object pool. This is the way the cut pool is currently implemented in SYMPHONY.

3.4 Fault Tolerance

For a robust implementation, fault tolerance is a requirement. With our decentralized storage scheme, this is a challenging aspect of the implementation. With the current design, if one of the workers dies, then we can simply reassign that work to another worker. At most, what would be lost is the work that was done on the subtree the worker was working on at the time it died. If the processors have the capability of writing to a local disk, then the state of the entire subtree the worker is processing can be written to disk periodically and read back in if it crashes.

Fault tolerance for the hubs is a little more difficult. If the hub has access to a local disk that it can write to, then we can again write out a log file periodically that will allow the hub to be restarted without loss of data. Otherwise, it is unlikely that the hub's data can be recovered very easily. One additional option that we may consider is to partially back up this data at the master process itself. However, this is a potentially expensive operation, so for the first version of the code, we assume that the hubs are running on highly reliable machines and will not die. Of course, this also means that the hubs must be bug free.

3.5 Support for Multi-phase Algorithms

In the two-phase method, we first run the algorithm to completion on a specified set of primal objects (variables). Any node that would have been fathomed in the first phase is sent to a pool of candidates for the second phase instead. If the set of variables is small, but well chosen, this first phase should be quick and result in a near-optimal solution (hence, a good upper bound). In addition, the first phase produces a list of useful constraints. Using the upper bound and the list of constraints from the first phase, the root node is *repriced*—that is, it is reprocessed with the full set of variables and constraints, the hope being that most or all of the variables not included in the first phase will be priced out of the problem in the new root node. Any variable so priced out can be eliminated from the problem globally. If we are successful at pricing out all the inactive variables, we have shown that the solution from the first phase was, in fact, optimal. If not, we must go back and price out the (reduced) set of extra variables by retracing the search tree produced during the first phase in some manner. After retracing the tree, we can then continue processing any node in which variables are consequently added to the relaxation.

In theory, this two-phase method could be extended to multiple phases. In order to

implement such a method, it must be possible to reconstruct the entire search tree after processing in the first phase has ended. In our decentralized storage scheme, this means keeping information in the master which allows the tree to be reconstructed.

3.6 Support for Domain Decomposition

Our design also supports the concept of *domain decomposition*. Consider a situation where the set of primal and dual objects can be partitioned in such a way that the dual objects in each member of the partition are only dependent on the primal objects in that same set. Then feasibility of a given solution can be determined by considering each member of the partition independently. Now suppose that the objective function can also be decomposed in a corresponding way, such that the value of each piece depends only on the value of the primal objects in one of the members of the partition; and that there is a function that can recombine the individual pieces to obtain the objective function value of the full solution represented by the values of the primal objects in each member of the partition. In this case, we can optimize each block independently and then recombine the resulting optimal solutions (defined only on a subset of the primal objects) to obtain the optimal solution to the original problem. For example, this is the case in linear programming when the constraint matrix can be decomposed into blocks. In this case, each block can be optimized independently and the objective function values summed to obtain an optimal solution to the original problem.

This method of decomposing the problem can be thought of as a type of branching scheme in which the children contain not only a subset of the feasible set defined in the parent node, but also, in some sense, a subset of the active objects from the parent node. This type of branching, however is still done in such a way that the bounds and solutions to each of the child subproblems can be recombined to yield a solution to the parent subproblem. Implementing this sort of a branching scheme is challenging at best and we will not go into the details here. However, we plan to provide support for such methods.

3.7 Communications Protocol

The implementation of any parallel algorithm obviously requires a *communications protocol*. However, tying the library to one particular protocol limits portability of the code. Therefore, communication routines are implemented via a separate generic interface to several common protocols. The current options are Parallel Virtual Machine (PVM), Message Passing Interface (MPI), a serial layer (for debugging), and the OpenMP protocol for shared-memory architectures.

4 Overview of the Class Structure

In this section, we give an overview of some of the main classes and their members. A few comments are in order before describing the classes. In what follows, we display the

pseudo-code for the main classes. We have stripped out certain C++ keywords such as `private` for clarity and brevity. However, the data items that begin with an underscore are private. Because of this fact, there are methods defined in each class to get and set the values of these data items. These have also been left out for clarity. The functions are declared using C++ notation. For instance, the keyword `virtual` before a function name means that the function can be redefined in derived classes. An `=0` following a function declaration means that function is declared *pure virtual*, meaning that it cannot be defined in the current class and *must* be defined in a derived class.

4.1 The Abstract Library for Parallel Search

The main classes in ALPS are `AlpsTreeNode` and `AlpsSubTree`. `AlpsTreeNode` contains the data and methods associated with an individual node in the tree, while `AlpsSubTree` contains the data and methods needed to store an entire subtree. In the next two sections, we describe these two classes.

4.1.1 The `AlpsTreeNode` Class

Pseudo-code for the definition of the `AlpsTreeNode` class appears in Figure 2. This class supports the basic notions discussed in Section 3.1.1. The `index` of a node is its global index in the tree. For now, we leave unanswered the question of exactly how this index gets assigned, but it is done in such a way that we ensure each node gets a unique index. The `level` of a node is the level in the tree where it was generated (the root node is at level 0). The `status` of a node is one of `candidate`, `active`, `processed`, `fathomed`, as described in Section 3.1.1. The `quality` is a measure used for determining the order of the priority queue. Pointers to the parent and children of the node are stored in the indicated positions.

The methods that are needed to implement the basic functionality in ALPS are listed below the data members. The `removeChild()` method deletes the specified child from the list, while the `removeDescendants()` removes all of the descendants of a node. Both of these functions are needed to support functionality in the `AlpsSubTree` class. The `createExplicit()` method changes the description of the node from a relative one to an explicit one (in place), while `createDiff()` creates the difference between this node and the node passed as an argument to the function. The `process()` method is, of course, the method that actually processes the node. The last three methods are pure virtual and must be defined in a derived class.

4.1.2 The `AlpsSubTree` Class

Pseudo-code for this class, which pertains to the storage and manipulation of subtrees, appears in Figure 3. The `quality` measure here is the maximum of the quality measures of the leaf nodes of this subtree (assuming that lower quality numbers are better in order

```
class AlpsTreeNode {

   AlpsNodeIndex_t         _index;
   int                     _level;
   AlpsNodeStatus          _status;
   double                  _quality;
   AlpsTreeNode*           _parent;
   int                     _numChildren;
   AlpsTreeNode**          _children;

   void                    removeChild(const AlpsTreeNode*& child);
   void                    removeDescendants();

   virtual AlpsTreeNode*   createExplicit() const = 0;
   virtual AlpsTreeNode*   createDiff(const AlpsTreeNode& explicitNode) const=0;

   virtual void            process() = 0;

};
```

Figure 2: The AlpsTreeNode class

```
class AlpsSubTree {

   double                _quality;
   double                _qualityThreshold;
   AlpsTreeNode*         _root;
   CoinPriorityQueue     _candidateList;

   AlpsTreeNode*         getNextNode();
   void                  removeDeadNodes(AlpsTreeNode*& node);
   void                  replaceNode(AlpsTreeNode* oldNode,
                                     AlpsTreeNode* newNode);

   virtual void          pack(CoinWriteBuffer& buf) = 0;
   virtual void          unpack(CoinReadBuffer& buf) = 0;
};
```

Figure 3: The `AlpsSubTree` class

to match with the assumption that we are solving a minimization problem). The `root` is a pointer to the subproblem that is the root of the tree. The `candidateList` is a priority queue of the leaf nodes of the subtree.

The `getNextNode()` method simply returns the next node in the priority queue ordering. The `removeDeadNodes()` is a method that not only removes the argument node (which should have status `fathomed`), but also recursively calls itself on the parent if the parent has no children left after the removal of the argument. The `replaceNode()` method is used to replace the node description of one node with the node description of another node. This is used when gluing together two subtrees together or when updating the description of a node to reflect the result of processing.

4.2 The Branch, Constrain, and Price Software Library

As discussed in Section 3.1.2, the BiCePS library is built on top of ALPS and hence contains classes derived from those discussed in the previous section. The primary concept that is new at the BiCePS level is that of a `BcpsObject`. Before we describe the `BcpsObject` class, we must describe the concept of an *encodable object*, encapsulated in the `CoinEncoded` and `CoinEncodable` classes.

```
class CoinEncodable {

   CoinPtr<CoinEncoded>    _encoded;

   virtual void            registerClass();
   virtual CoinEncoded*    encode() const = 0;
   virtual CoinEncodable* decode(const CoinEncoded&) const = 0;

   static std::map<const char*, const CoinEncodable*, CoinStrLess>* _decodeMap;
   static const CoinEncodable* decoderObject(const char* name) const;

};
```

Figure 4: The `CoinEncodable` class

4.2.1 The `CoinEncoded` and `CoinEncodable` Class

The `CoinEncodable` class is the base class for all objects that can be encoded into a character array for compact storage or process communication. The reason for this special class is the need to decode these objects at a higher level in the class hierarchy than that at which they were created. For example, the methods defined in BiCePS classes do not know the types of any of the objects that are derived and created in BLIS. For this reason, it is necessary for the objects at a lower level in the hierarchy to be registered with the framework so that their `decode()` method is known to the lower level classes. This is done essentially by creating an empty object of the appropriate type, which can then be used to unpack other objects later on.

The pseudo-code for the `CoinEncodable` class is shown in Figure 4. The `encoded` field is the only data member and is of type `CoinEncoded`, which is pictured in Figure 5. This `CoinEncoded` class is a container for the encoded form, which is stored within the object itself. Figure 6 shows a comparison function for this class.

The `registerClass()` method of `CoinEncodable` is the method that creates the empty object and adds it to the array of unpack methods, called `decodeMap`. The `encode()` and `decode()` methods are the pure virtual methods that do the encoding and decoding of the object. Notice that the `decode()` method is required to return a new object instead of unpacking in place. Finally, the `decoderObject()` method returns a pointer to the empty object containing the `decode` method for type `name`.

```
class CoinEncoded {

   const char*            _type;
   int                    _size;
   const char*            _representation;

};
```

Figure 5: The CoinEncoded class

```
inline bool operator< (const CoinEncoded& obj0,
                       const CoinEncoded& obj1)
{
   return (obj0._size < obj1._size ? true :
           (obj0._size > obj1._size ? false :
            memcmp(obj0._representation, obj1._representation,
                   obj0._size)));
}
```

Figure 6: The comparison function for CoinEncoded class

```
class BcpsObject : public CoinEncodable {

   BcpsIndex_t            _index;
   char                   _intType;
   char                   _status;
   double                 _lbHard;
   double                 _ubHard;
   double                 _lbSoft;
   double                 _ubSoft;

   virtual CoinEncoded*   encode() const = 0;
   virtual CoinEncodable* decode(const CoinEncoded&) const = 0;

};
```

Figure 7: The `BcpsObject` class

4.2.2 The `BcpsObject` Class

The `BcpsObject` class is pictured in Figure 7. The `index` field contains the index for the object, which may not exist. The `intType` field is used to denote whether a variables is discrete, i.e., is constrained to take on only integer values. The `status` field is used to indicate various information we might want to know about the variable, such as whether it has been branched on, whether it can be removed from the problem, whether it is eligible to be sent to the pool, etc. The last four fields have to do with the upper and lower bounds for the object. The *hard* bounds are those given by the user in the original description of the problem. The *soft* bounds are the tightened bounds that may be different in different parts of the search tree. These bounds need to be maintained separately because with column generation, there are times when the soft bounds become invalid and need to be replaced with the hard bounds again. Note that bound changes that occur due to branching are considered changes to the hard bounds since these changes can never be undone at a lower level in the tree.

Although primal and dual objects each have one set of bounds and are represented in the same class structure, the bounds are interpreted differently for the two types of objects. For primal objects, these are bounds on the value of the variable itself. For dual objects, these bounds are on the value of the left-hand side of the constraint, as opposed to the dual variable.

```
class BcpsSolution : public CoinEncodable {

   int                    _size;
   BcpsObject*            _objects;
   double*                _values;

   BcpsSolution* selectNonzeros(const double etol = 1e-6) const;
   BcpsSolution* selectFractional(const double etol = 1e-6) const;

};
```

Figure 8: The `BcpsSolution` class

4.2.3 The `BcpsSolution` Class

The `BcpsSolution` class contains the primal and dual solutions that are generated when solving the relaxations. The solutions are represented simply as an array of the objects with an associated array of values. In most cases, only the objects with nonzero values are present in the array. The class is pictured in Figure 8.

4.2.4 The `BcpsSubProblem` Class

The `BcpsSubproblem` class is pictured in Figure 9 and contains the description of a subproblem. It is important to understand the basic concept of storing these node descriptions using a differencing scheme. Recall that a subproblem is composed of a list of objects along with a few other data items, including the current lower bound. Between a node and its children, two things can change. First, objects could be added and deleted from the list. Second, the bounds on objects could change. Most of the time, the number of such changes is relatively small compared to the size of the overall list. Therefore it makes more sense to store the changes in most cases. The `_numObjectTypes`, `_objectVecs`, `_objectVecStorage`, and `_CoinVecMod` data structures store these differences compactly. Note that each object type is considered separately so that some object types could be stored explicitly while others are differenced if this is the most efficient method. The `createExplicit*()` methods are used to create an explicit node description from a description given as differences. Conversely, `createDiff()` creates a node description expressed in terms of differences from an explicit one. The `_lowerBound` field contains the current lower bound associated with the given subproblem. The `process()` method is, of course, the method for bounding a node. This must be defined in a lower level class. Finally, the `encode()` and `decode()` methods are used for packing a node description to be sent between processes.

```
class BcpsSubproblem : public AlpsTreeNode {
   template <class T> class _CoinVecMod {
    public:
      bool                relative;
      int                 numModify;
      int*                posModify;
      T*                  entries;
   };

   struct _objectVecStorage {
      int                 numRemove;
      int*                posRemove;
      int                 numAdd;
      BcpsObject*         objects;
      CoinVecMod<double>  lbHard;
      CoinVecMod<double>  ubHard;
      CoinVecMod<double>  lbSoft;
      CoinVecMod<double>  ubSoft;
      CoinVecMod<int>     status;
   };

   double                 _lowerBound;
   static int             _numObjectTypes;
   _objectVecStorage*     _objectVecs;

   CoinVec<BcpsObject>*   createExplicitObjects(const int i) const;
   CoinVec<double>*       createExplicitLbOrig(const int i) const;
   CoinVec<double>*       createExplicitUbOrig(const int i) const;
   CoinVec<double>*       createExplicitLbCurrent(const int i) const;
   CoinVec<double>*       createExplicitUbCurrent(const int i) const;
   CoinVec<int>*          createExplicitStatus(const int i) const;

   BcpsObject_p*          createExplicitStorage(const int i) const;

   virtual AlpsTreeNode*  createExplicit() const;
   virtual AlpsTreeNode*  createDiff(const AlpsTreeNode& explicitNode) const;
   virtual void           process() = 0;
   virtual void           encode(CoinWriteBuffer& buf);
   virtual void           decode(CoinReadBuffer& buf);
};
```

Figure 9: The BcpsSubProblem class

5 Conclusion and Future Plans

In this report, we have described the main design features of the library hierarchy and have given an overview of the class structure. This project is being developed as open source under the auspices of the Common Optimization Interface for Operations Research (COIN-OR) initiative. All source code will be available from the CVS repository at `www.coin-or.org`. Current development plans are for an incremental development effort lasting approximately one year. We hope to have a sequential version of this code available by the end of the calendar year 2001, followed by a parallel version with limited capabilities by mid-2002.

References

[1] G.M. AMDAHL, *Validity of the Single-processor Approach to Achieving Large-scale Computing Capabilities*, In *AFIPS Conference Proceedings* **30** (Atlantic City, N.J., April 18–20), AFIPS Press (1967).

[2] E. BALAS AND P. TOTH, *Branch and Bound Methods*, in E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, New York (1985), 361.

[3] R.L. BOEHNING R.M. BUTLER AND B.E. GILLET, *A Parallel Integer Linear Programming Algorithm*, European Journal of Operations Research **34** (1988).

[4] V. CHVÁTAL, *Linear Programming*, W.H. Freeman and Company (1983).

[5] *Using the CPLEX© Callable Library*, CPLEX© Optimization Inc. (1994).

[6] H. CROWDER AND M. PADBERG, *Solving Large Scale Symmetric Traveling Salesman Problems to Optimality*, Management Science **26** (1980), 495.

[7] H. CROWDER, E.L. JOHNSON AND M. PADBERG, *Solving Large-Scale Zero-One Linear Programming Problems*, Operations Research **31** (1983), 803.

[8] V.-D. CUNG, S. DOWAJI, B. LE CUN, T. MAUTHOR AND C. ROUCAIROL, *Concurrent Data Structures and Load Balancing Strategies for Parallel Branch and Bound/A* Algorithms*, DIMACS Series in Discrete Optimization and Theoretical Computer Science **30** (1997).

[9] J. ECKSTEIN, *Parallel Branch and Bound Algorithms for General Mixed Integer Programming on the CM-5*, SIAM Journal on Optimization **4** (1994).

[10] J. ECKSTEIN, *How Much Communication Does Parallel Branch and Bound Need?*, INFORMS Journal on Computing **9** (1997).

[11] M. ESO, *Parallel Branch and Cut for Set Partitioning*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, N.Y., U.S.A. (1999).

[12] M.R. GAREY AND D.S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco (1979).

[13] R.S. GARFINKEL AND G.L. NEMHAUSER, *Integer Programming*, Wiley, New York (1972).

[14] A. GEIST ET AL, *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA (1994).

[15] B. GENDRON AND T.G. CRAINIC, *Parallel Branch and Bound Algorithms: Survey and Synthesis*, Operations Research **42** (1994), 1042.

[16] G.H. GOLUB AND C.F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore (1989).

[17] A. GRAMA AND V. KUMAR, *Parallel Search Algorithms for Discrete Optimization Problems*, ORSA Journal on Computing **7** (1995).

[18] M. GRÖTSCHEL, M. JÜNGER, AND G. REINELT, *A Cutting Plane Algorithm for the Linear Ordering Problem*, Operations Research **32** (1984), 1155.

[19] J.L. GUSTAFSON, *Reevaluating Amdahl's Law*, Communications of the ACM **31** (1988).

[20] K. HOFFMAN AND M. PADBERG, *LP-Based Combinatorial Problem Solving*, Annals of Operations Research **4** (1985/86), 145.

[21] D.C. KOZEN, *The Design and Analysis of Algorithms*, Springer-Verlag, New York (1992).

[22] L. KOPMAN, *A New Generic Separation Algorithm and Its Application to the Vehicle Routing Problem*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, N.Y., U.S.A. (1999).

[23] V. KUMAR AND V.N. RAO, *Parallel Depth-first Search. Part II. Analysis.*, International Journal of Parallel Programming **16** (1987).

[24] V. KUMAR AND A. GUPTA, *Analyzing Scalability of Parallel Algorithms and Architectures*, Journal of Parallel and Distributed Computing **22** (1994).

[25] L. LADÁNYI, *Parallel Branch and Cut and Its Application to the Traveling Salesman Problem*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, N.Y., U.S.A. (1996).

[26] P. LAURSEN, *Can Parallel Branch and Bound without Communication Be Effective?*, **4** (1994).

[27] E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN AND D.B. SHMOYS, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York (1985).

[28] R. MARSTEN, *The Design of The XMP Linear Programming Library*, ACM Transactions on Mathematical Software **7** (1981), 481.

[29] G.L. NEMHAUSER AND L.A. WOLSEY, *Integer and Combinatorial Optimization*, John Wiley & Sons, Inc. (1988).

[30] M. PADBERG AND G. RINALDI, *A Branch-and-Cut Algorithm for the Resolution of Large-Scale Traveling Salesman Problems*, SIAM Review **33** (1991), 60.

[31] T.K. RALPHS, *Parallel Branch and Cut for Vehicle Routing*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, N.Y., U.S.A. (1995).

[32] T.K. RALPHS, L. KOPMAN, W.R. PULLEYBLANK, AND L.E. TROTTER JR., *On the Capacitated Vehicle Routing Problem*, accepted to *Mathematical Programming*.

[33] T.K. RALPHS, *SYMPHONY Version 2.8 User's Guide*, available at `www.branchandcut.org/SYMPHONY`.

[34] T.K. RALPHS, L. LADÁNYI, AND L.E. TROTTER, *Branch, Cut, and Price: Sequential and Parallel*, Lecture Notes on Computational Combinatorial Optimization, D. Naddef and M. Jünger, eds., *to appear*, available at `www.lehigh.edu/~tkr2/research/#pubs`.

[35] T.K. RALPHS AND L. LADÁNYI, *Computational Experience with Branch, Cut, and Price*, in preparation.

[36] T.K. RALPHS AND L. LADÁNYI, *COIN/BCP User's Guide*, available at `www.coin-or.org`.

[37] T.K. RALPHS AND L. LADÁNYI, *SYMPHONY: A Parallel Framework for Branch and Cut*, White paper, Rice University (1999), available at `www.branchandcut.org/SYMPHONY`.

[38] V.N. RAO AND V. KUMAR, *Parallel Depth-first Search. Part I. Implementation.*, International Journal of Parallel Programming **16** (1987).

[39] R. RUSHMEIER AND G. NEMHAUSER, *Experiments with Parallel Branch and Bound Algorithms for the Set Covering Problem*, Operations Research Letters **13** (1993).

[40] M.W.P. SAVELSBERGH, *A Branch-and-Price Algorithm for the Generalized Assignment Problem*, Report COC-9302, Georgia Institute of Technology, Atlanta, Georgia (1993).

[41] A. SCHRIJVER, *Theory of Linear and Integer Programming*, John Wiley & Sons (1986).