The Journal of Supercomputing, 28, 215–234, 2004

A Library Hierarchy for Implementing Scalable Parallel Search Algorithms

T. K. RALPHS* tkralphs@lehigh.edu
Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015

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

M. J. SALTZMAN mjs@clemson.edu
Department of Mathematical Sciences, Clemson University, Clemson, SC 29634

Abstract. This paper 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 challenging both because of data storage requirements and communication overhead. ALPS incorporates a number of new ideas to address this challenge.

The paper also describes the design of two other libraries forming a hierarchy built on top of ALPS. The first is the Branch, Constrain, and Price Software (BiCePS) library, a framework that supports the implementation of parallel branch and bound algorithms in which the bounds are obtained by solving some sort of relaxation, usually Lagrangian. 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, but they pose further difficulties for designing scalable algorithms. The other library is the BiCePS linear integer solver (BLIS), a concretization of BiCePS, in which linear programming is used to obtain bounds in each search tree node.

Keywords: parallel algorithm, parallel search, parallel branch and bound, optimization, integer programming

1. Introduction

This paper describes research in which we are seeking to develop 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. ALPS forms the base layer, which we call the *search handling layer*. Additional layers built on top of ALPS form a hierarchy implementing additional functionality needed for specific applications.

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

*Author for correspondence.

such as planning, scheduling, logistics, telecommunications, bioengineering, robotics, design of intelligent agents, etc. Most DOPs are in the complexity class NP-complete so there is little hope of finding provably efficient algorithms [14]. Nevertheless, sophisticated versions of branch and bound, such as *branch, constrain, and price* (BCP), have been tremendously successful at tackling these difficult problems.

The main feature of such algorithms that makes them difficult to implement is the tremendous amount of information that must be generated and maintained in order to guide the search. We call such information *knowledge*. Knowledge discovered during the search process is stored in *knowledge bases* of various types that communicate with each other through *knowledge brokers* to execute the search. Trienekens and De Bruin [35] suggested that the organization of knowledge bases and the way in which they share knowledge is essentially what differentiates various implementations of parallel branch and bound. This will be our premise as well. Because knowledge discovered during the search process can actually change the way in which the search evolves, the shape of the search tree is not known *a priori*. This makes the process of dividing the work among the processors (called *load balancing*) extremely difficult.

For applications such as BCP, describing the nodes in the search tree (which can themselves be thought of as a special kind of knowledge) requires a vast amount of data. We call such applications *data-intensive*. Fortunately, these data do not tend to vary much from parent to child, so we use data structures based on a unique differencing scheme to store portions of the search tree. This scheme for storing the tree allows us to handle problems much larger than we otherwise could. However, it requires that we pay particular attention to *data handling*.

As such, we have designed a *data handling layer* that is built on top of ALPS. The data handler for BCP is called the Branch, Constrain, and Price Software (BiCePS) library and implements a generic framework for relaxation-based branch and bound. The design of this library is based on the notion that the description of each search tree node consists of a large number of *data objects*, which are themselves knowledge to be shared. In this layer, we make very few assumptions about the nature of the relaxations; in particular, they do not have to be linear programs (LPs), although this is the most typical scenario. A third layer, called the BiCePS linear integer solver (BLIS), implements LP-based branch and bound algorithms.

A number of techniques for developing scalable parallel branch and bound algorithms have been proposed in the literature [6, 11, 12, 16, 17, 32]. 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 needed for efficient storage management impede this fluid movement. Our approach to this difficulty is to divide the search tree into subtrees containing a large number of related search nodes that can be stored together. More complex load balancing schemes are necessary to accomodate this storage strategy.

Parts of this framework are loosely based on previous work in which we developed two object-oriented, generic frameworks for implementing parallel BCP algorithms.

Single- or Multi-Process Optimization over Networks (SYMPHONY) [30] is a framework written in C and COIN/BCP [31] 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 operations research community [29, 31].

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* (defined below) for small numbers of processors, but they employ a master-slave paradigm with a single central node queue. This centralized approach makes load balancing and other decision-making tasks quite easy. However, it limits scalability by creating a bottleneck at the master process. The design of ALPS is intended not only to overcome these limitations, but also to generalize the capabilities of these libraries in order to support a wider variety of applications.

2. Motivation and background

2.1. Developing scalable parallel search algorithms

As we have already stated, the primary design goal of the new framework is to increase scalability. 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 generally 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.

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 or more, 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 Bruin et al. [7] for a treatment of this phenomenon). When an algorithm has an efficiency less than one, we can compute the *parallel overhead* as $O_p = T_p(1 - E_p) = T_p - T_0/p$.

In the remainder of the paper, we take the somewhat idealized view that each processor is the host for one or more knowledge bases that work together in a systematic way to perform the search. These knowledge bases communicate with each other through an interface class called a *knowledge broker* (to be described in Section 3.1.1) that defines how knowledge can be requested and/or shared. Viewed in

terms of this knowledge sharing paradigm, there are four basic components of parallel overhead:

1. *Communication overhead:* Time spent transferring knowledge from one knowledge base to another, i.e., packing the information into the send buffer and unpacking it at the other end.
2. *Idle time (handshaking):* Time spent idle while waiting for information requested from another knowledge base.
3. *Performance of redundant work:* Time spent performing work that would not be performed in the sequential algorithm (primarily due to lack of knowledge that would allow the identification of the work as redundant).
4. *Idle time (ramp-up/ramp-down):* Time at the beginning/end of the algorithm during which there is not enough useful work for all processors to be kept busy. This can be thought of as time spent generating the knowledge needed to initialize the algorithm.

To achieve high efficiency, we must control these four sources of overhead. The first two sources essentially capture the cost of sharing knowledge. On the other hand, the third source of overhead can be thought of as the cost of *not* sharing knowledge. This highlights the fundamental tradeoff inherent in knowledge sharing—it reduces the performance of redundant work, but comes at a cost. The goal is to strike the proper balance in this tradeoff. In a recent study that examined the parallel performance of SYMPHONY [28], we found that for small numbers of processors, centralized knowledge bases that yield a clear global picture can result in near-linear speedup. However, as the number of processors is increased, efficiency is reduced. For large-scale parallelism, some decentralization is clearly needed.

2.2. *Branch and bound*

In order to define some terminology and provide a frame of reference for the way tree search algorithms work, we now review the basic notions of branch and bound. In the next section, we describe the LP-based variant of branch and bound we have already referred to as *branch, cut, and price*. The reader who is already familiar with these methods may skip to Section 3.

A branch and bound algorithm uses a divide and conquer strategy to partition the solution space into *subproblems*, and optimizes individually over each of them. For instance, let S be the set of solutions to a given problem, and let $c : S \to \mathbb{R}$ be a cost function 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" (i.e., low-cost) 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. The relaxed problem is relatively easy to solve, but admits solutions that are not in the feasible set S. Thus, 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.

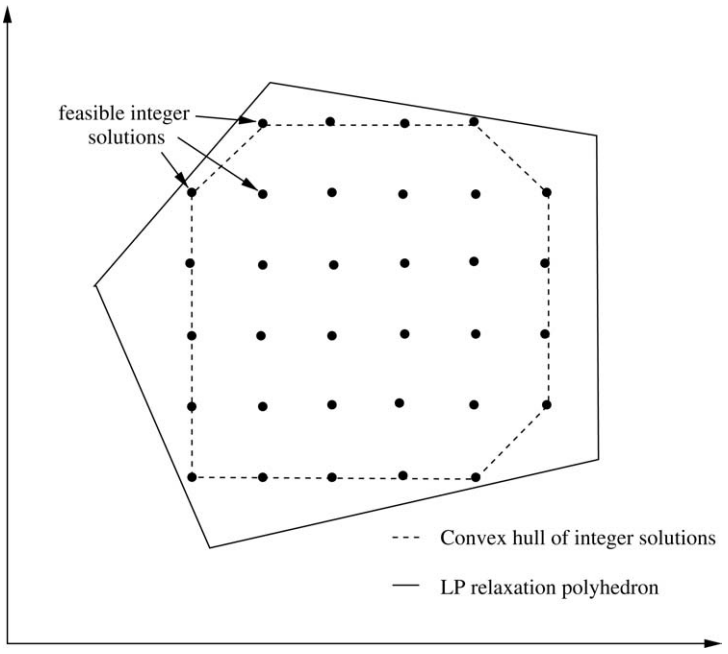

Figure 1. Example of a polyhedron and the convex hull of its integer points.

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* or a *search node*. $S_1, \ldots, S_n$ are also the *children* of S in a tree in which the original solution set S is the root and subproblems derived from a node are the descendants of that node. The children of S are added to the list of *candidate subproblems* (those which need processing). This step is called *branching*.

To continue the algorithm, one of the candidate subproblems is selected, removed from the list, and processed. 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 (*prune* or *fathom*) 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. The algorithm continues in this way until the list of active subproblems is empty, at which point our current best solution is the optimal one.

2.3. *Branch, constrain, and price*

BCP is an implementation of branch and bound used most typically for solving problems that can be formulated as *integer programs* (in this context, the method is

commonly referred to as *branch*, *cut*, *and price*). Early works such as Grötschel et al. [19], Hoffman and Padberg [20], Padberg and Rinaldi [27] laid out the basic framework of BCP and since then, many implementations (including SYMPHONY and COIN/BCP) have built on these preliminary ideas. In a BCP algorithm for solving an integer program, 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 relax the integrality constraints of the 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 $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$, consider the polyhedron $\mathscr{P} = \{x \in \mathbb{R}^n : Ax \geq b\}$ and define $\mathscr{P}_I := \mathscr{P} \cap \mathbb{Z}^n$, as shown in Figure 1. For $c \in \mathbb{R}^n$, the corresponding integer program is:

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

It is well known that (1) is equivalent to $\min\{cx : x \in \text{conv}(\mathscr{P}_I)\}$, and so we consider this problem from now on. By Weyl's Theorem (see Nemhauser and Wolsey [26]), there exists a finite set $\mathscr{L}$ of inequalities valid for $\mathscr{P}_I$ such that

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

The inequalities in $\mathscr{L}$ 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 $\mathscr{L}$ 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. For a more complete description of this methodology, see Nemhauser and Wolsey [26].

In an integer program, we view each component x_i of the solution vector x as a separate variable associated with the i-th 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 $\mathscr{P}_I$, we are forced to branch. The branching operation is accomplished by specifying a set of hyperplanes that 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, 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.

2.4. *Related work*

The branch and bound algorithm as we know it today was first suggested by Land and Doig in 1960 [22]. Soon after, Balas introduced the idea of *implicit enumeration* in suggesting a related algorithm for solving integer programs with variables that can only take on the values 0 or 1 [3]. As early as 1970, Mitten abstracted branch and bound into the theoretical framework we are familiar with today [25]. However, it was not until the 1990s that much of the software employing branch and bound on a large scale originated.

Numerous software packages employing parallel branch and bound have been developed. The previously mentioned SYMPHONY and COIN/BCP are 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 [34], BoB [4], PPBB-Lib [36], and PICO [13]. CONCORDE [1, 2] is a parallel solver for the Traveling Salesman Problem, while PARINO [23] and FATCOP [8] are parallel solvers for general *mixed-integer programs*.

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 [21] provide an excellent general introduction to the analysis of parallel scalability. Good overviews and taxonomies of parallel branch and bound algorithms are provided in both Gendron and Crainic [16] and Trienekens and de Bruin [35]. Eckstein et al. [13] also provide 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 Bixby et al. [5], Correa and Ferreira [9], Grama and Kumar [17], Mitra et al. [24], and Wright [37].

3. Overview of library design

3.1. *The library hierarchy*

To make the framework easy to maintain, easy to use, and as flexible as possible, our design is 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 will be useful in a wide variety of settings. To illustrate, we briefly describe the current hierarchy.

3.1.1. ALPS: The Abstract Library for Parallel Search. As previously described, ALPS is a C++ class library containing the base classes needed to implement 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 alluded to earlier. As differencing is a data-management issue, this scheme is implemented in the data-management layer on top of ALPS. Differencing for optimization problems is implemented in BiCePS. A generic data-management layer that provides subproblem descriptions based on objects and differencing is also provided. Note that this does not preclude the use of ALPS for applications that are not data-intensive.

Knowledge bases. The design of ALPS is predicated on the idea that all information required to carry out a tree search can be represented as knowledge and stored in various knowledge bases. A knowledge base (KB) is a collection of related information about the tree and the problem instance being solved. Basic examples of knowledge bases are collections of solutions, collections of search nodes, or, in the case of BCP, collections of cutting planes (see Section 3.3.4). A KB is managed by a *knowledge-base manager* (KBM). The KBM associated with a KB may field two types of requests: (1) A new piece of knowledge to be inserted in the KB, or (2) a request for relevant pieces of knowledge from the knowledge base, where "relevant" is defined for each category of knowledge with respect to data provided by the requesting process. A KB may also choose to "push" certain knowledge to another KB, even though no specific request has been made. It is even possible for one KB to request that another KB send information to a third KB (this is needed in order to perform load balancing efficiently).

A KBM can function as a repository for existing knowledge, as well as a generator of new knowledge. For example, while processing a subproblem, a worker might generate new subproblems (the children produced after branching), as well as additional cutting planes that may be of use elsewhere. The newly generated knowledge can be stored locally or sent to an appropriate external KB. In order to maintain architecture and protocol independence, KBs do not communicate directly with each other. Instead, requests and responses are passed through a routing class, called a *knowledge broker*. The knowledge broker contains all architecture- and protocol-dependent subroutines, as well as information about other KBs, such as their location.

Algorithm overview. The algorithm is driven by a master process that acts as a KB for solutions, which are defined to be the end product of any search. The master receives data for a problem instance (from an input file or a subroutine that constructs the problem on the fly), and processes it to produce a collection of (one or more) solutions. Upon conclusion of the search process, the solution KB can then be queried by an output routine for the result of the search. The search process consists of constructing the root of the search tree and passing it to one of the workers (the types of workers and their roles is described in Section 3.2.1). In the sequential case, the designated worker carries out the entire search on its own and returns the solutions generated to the solution KB. The parallel case is described more fully below.

In order to determine 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 is usually the lower bound in each search tree node. Default search schemes not requiring a user-defined measure of quality (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 is not appropriate.

It should be emphasized that enforcing the search order as closely as possible can be extremely important to scalability because not doing so can result in the performance of redundant work. In a sequential algorithm or in an implementation employing a single central KB containing descriptions of all search tree nodes available for processing (such as in SYMPHONY and COIN/BCP), it is trivial to enforce the search order because global knowledge regarding the state of the tree is always available. However, when the storage of candidate node descriptions is decentralized, the problem becomes much more difficult. Our approach to this problem is described in Section 3.2.1.

Also associated with a search tree node is its current status. In ALPS, there are 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, i.e., it is a leaf node in the search tree.
- `evaluated`: Indicates the node has been processed, but branching has not been performed. Note that evaluation may change the quality of the node.
- `branched`: Indicates that branching has been performed, i.e., the children of the node have been produced and added to the appropriate KB.
- `fathomed`: Indicates the node has been fathomed. This means that either (1) the quality of the node fell below the quality threshold during processing, or (2) it was determined that the node is *infeasible*, i.e., does not contain any solutions. In most cases, fathomed nodes are deleted immediately.

In terms of these stati, processing a node involves converting its status from `candidate` to `evaluated` or `fathomed`. A node whose processing results in the status `evaluated` is put back into the queue with its new quality to be branched upon when next selected for processing. Within the context of ALPS, branching simply means generating the descendants of the current subproblem and setting their `quality` fields, if appropriate. After branching, the node's status is set to `branched` and the stati of each of the node's descendants is set to `candidate`.

3.1.2. BiCePS: The Branch, Constrain, and Price Software Library

Primal and dual objects. 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. 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, *primal objects* and *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*.

The dual objects represent *constraints*. A constraint in a mathematical programming problem requires that some function of the variables lie between specified upper and lower bounds. The value of a dual object is the value of the function, evaluated with respect to the current values of the variables. The *objective function* is a distinguished constraint with infinite bounds. We assume without loss of generality that the objective function is to be minimized.

It is important to note that the notion of primal and dual objects is a completely general one. We need only assume that there exists a set of variables and constraints with associated values that must lie between certain bounds, as well as an objective function to be optimized. Starting from these concepts, we can easily apply the general theoretical framework of Lagrangian duality.

In Lagrangian 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 its current value and its *reduced cost*. A variable's reduced cost is defined as the marginal increase in the objective function value from increasing the value of that variable. In linear programming, the reduced cost can be easily computed. In the more general case of a differentiable nonlinear 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, both classes can be treated 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. This bound is valid for the original problem as long as either all of the primal objects are present or it is proved that all of the missing objects can be fixed to value zero.

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 can be modified through *bound tightening*. To define object generation, we need the concepts of *primal solution* (the values of variables and slack values of constraints) and *dual solution* (the multipliers of constraints and reduced costs of variables). *Object generation* then consists of producing constraints whose slacks are negative (i.e., whose bounds are violated) and/or variables whose reduced costs are negative, given the current primal or dual solutions, respectively (these are needed to compute the slacks and reduced costs).

Subproblem processing begins with a list of primal and dual objects from which the corresponding relaxation is constructed. The relaxation is solved to obtain an

initial bound for the subproblem. The relaxation is then iteratively tightened by generating constraints from the resulting primal solution. In addition, we may wish to generate variables. Note that the generation of variables "loosens" the formulation and hence must be performed strategically. In Section 3.3.5, we 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. Handling and tracking of objects is described further in Section 3.3.

Branching. In BiCePS, branching is accomplished by defining a *branching set*. A branching set consists of a list of data objects to be added to the relaxation before branching (possibly solely for the purpose of branching) and a list of object bounds to be modified in order to create each of the children, i.e., branching on a single variables, as described in Section 2.3, is accomplished by adding the variable to the branching set and setting its bounds appropriately in each of the child subproblems. Any object can have its bounds modified by branching, but the union of the feasible sets contained in all of the created child subproblems must contain the original feasible set in order for the algorithm to be correct.

3.1.3. BLIS: 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, the relaxation scheme is LP-based. This means that the objective function and all constraints are linear functions of the variables and that the relaxations are linear programs. This allows some of the notions discussed above to be defined more concretely. For instance, we can 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 is explored further in 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 expected by the relaxation solver. In BLIS, both the representation of the relaxation and the solver interface are contained in the open solver interface (OSI) class, a generic API supporting a number of commercial and open-source LP and IP solver libraries [33]. More details on the handling of objects and relaxations is given in Section 3.3.

3.2. *Improved search handling (ALPS)*

Some issues related to scalability have already been addressed in Section 2.1. As pointed out there, design for scalability 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 moving away from our compact data structures. Our new design attempts to reconcile the need for decentralization with our compact storage scheme. Improving the scalability consists essentially of reducing the sources of parallel overhead detailed in Section 2.1. In the following sections, we describe our approach to this issue.

3.2.1. The master-hub-worker paradigm. A major problem with the *master-slave* paradigm employed by SYMPHONY and COIN/BCP is that the tree manager becomes overburdened with requests for knowledge (primarily the descriptions of new search tree nodes). Furthermore, most of these requests are *synchronous*, meaning that the requesting 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 et al. [13] in their PICO framework.

In this scheme, the hubs function as knowledge bases for generating solutions corresponding to a particular set of subtrees, much as the master does for the entire tree. The solutions generated by the hubs are then aggregated in the master process to obtain the final result. Each hub is responsible for balancing the load among its workers. This load balancing consists not only of ensuring that each worker is busy, but also that each worker is busy performing *high-quality* work. This means ensuring that the *high-quality* nodes are evenly distributed among the workers, thereby implicitly enforcing the search order, as discussed earlier. Periodically, load balancing must also be performed between the hubs themselves. 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 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 too high a proportion of high-quality nodes) and *receiver hubs*, which 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. The computational burden to the hubs themselves can be further reduced by increasing the task granularity in a manner that we describe next.

3.2.2. Increased 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 must be 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 needed to manage a tree search, 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 for which it is responsible. The hub dispenses new candidate nodes (leaves of one 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.

Besides increasing the grain size, this scheme has the advantage that implementing a sequential algorithm is simply a matter of starting only a single worker process. Passing the root node of the tree to a worker process and requesting solutions back 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 change allows for increased grain size, as well as more compact storage, it does make some parts of the implementation much more complex. For instance, we must be more careful about how we perform load balancing in order to attempt 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 worker 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 is always made in an asynchronous manner. This feature is described next.

3.2.3. Asynchronous messaging. Another design feature that 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 their subtrees. The hub can then decide at its convenience whether to ask a worker to stop working on its current subtree and begin work on a new one.

An 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 issue will be addressed in the next section.

3.3. Improved data handling (BiCePS)

One of the biggest challenge 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 more detail.

3.3.1. Object representation. We stated in Section 3.1.2 that a BiCePS object can be thought of as either a variable or a constraint. However, we did not really define exactly what the terms variable and 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, i.e., a method of producing the projection of a given valid 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 that determines the appropriate modifications to allow the object to be added to the current relaxation.

Hence, an object's *representation*, i.e., the way it is stored as a stand-alone object, is inherently different from its *realization* in the current relaxation. The representation of an object 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. The only data needed 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 of object within the context of a given relaxation and also defines the method of constructing that realization. When sending an object's description to another process, we need only send the data required to generate the object's realization, 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 that is discussed in Section 3.3.3. Of course, there must also be a corresponding method of decoding as well.

3.3.2. Core and indexed objects. In order to initialize the object list of each search tree node, the user can designate a collection of *core objects* defining 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 the class is called *indexed*. Indexed objects can be treated much more efficiently than other classes because they can be represented simply by their indices. 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 are considering manipulate global objects is, in part, what makes them difficult to implement in parallel. To keep storage requirements at a minimum, it is desirable to know at the time an object is generated whether or not that same object has been generated previously 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, only one copy of each object would be stored centrally and it would simply be copied out whenever it was needed locally. Unfortunately, this creates an unacceptable communications bottleneck.

In practice, we simply ensure that at most one copy of each object is stored locally within each process. We do not try to eliminate duplicate copies that may exist in other locations. This is the limit of what can be done without synchronous messaging. All active objects are stored in a hash table. To generate a hash value for each object, the object is encoded and a hash function applied to the encoded form. Then the encoded form is compared 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 the new copy is deleted and replaced with a pointer to the old copy. This scheme allows efficient tracking and comparison of very large numbers of objects. For additional efficiency, separate hash tables can be maintained for primal and dual objects, as well as for different object subcategories.

This scheme works well in the worker processes because when new objects are generated, they are only active in a single subproblem and it is therefore easy to update the pointer. In the hubs, however, when an entire subtree and its associated objects get sent back from a worker, duplicate objects generated in different subtrees must be identified. Each duplicate object must be replaced by the existing copy and the pointers for the entire subtree updated correctly. This process can be expensive.

There is yet one further level of complexity to the storage of objects. In most cases, objects need only be kept around while they are actually pointed to, i.e., while they are active in some subproblem that is still a candidate for processing. If we don't occasionally "clean house," the object list will continue to grow boundlessly, especially in the hubs. To take care of this problem, we use "smart pointers", which maintain reference counters for the objects they point to. Objects are deleted automatically when the last reference is removed.

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 maintain independent knowledge bases called *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 data about the current relaxation from the worker processes and return relevant objects from the pool, i.e., constraints that are violated by the current relaxed solution and/or variables whose inclusion in the current subproblem would improve the solution. 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 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 how the cut pool is currently implemented in SYMPHONY.

3.3.5. Support for multi-phase algorithms. Because the time required to process a subproblem is largely a function of the number of objects active in that subproblem, we have devised a unique multi-phase approach that processes the search tree with an artificially limited set of variables. In the two-phase version of the method, the algorithm is first run to completion on a specified set of primal objects (variables). Any node that would have been fathomed in the first phase is instead sent to a

knowledge base containing candidates for the second phase. If the set of variables for the first phase is small, but well chosen, this first phase should be quick and result in a near-optimal solution. 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 *reprocessed* with the full set of variables and constraints. During reprocessing, we attempt to prove through a process called pricing that most or all of the variables not included in the first phase are actually unnecessary and that their inclusion in the problem will not reduce the cost of an optimal solution. If all the inactive variables are successfully eliminated in this way, then we have proven that the solution from the first phase was, in fact, optimal. If not, we must go back and reprocess the entire search tree from the first phase with the additional variables present. However, this can be done efficiently by retracing the tree and continuing to process any leaf node in which variables are consequently added to the relaxation.

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 that allows the tree to be reconstructed.

3.3.6. Support for domain decomposition. Our design also supports the concept of *domain decomposition*. Consider a situation in which the set of all 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, each block can be optimized independently and the resulting optimal solutions (defined only on a subset of the primal objects) can be recombined 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, 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 do not go into the details here. However, we plan to provide support for such methods.

3.4. Communications

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 options being implemented are parallel virtual machine (PVM) [15], message passing interface (MPI) [18], a serial layer (for debugging), and the OpenMP protocol for shared-memory architectures [10]. In ALPS, all messages are passed via the knowledge brokers through which shared knowledge is requested and/or provided. Hence, all communication is limited to routines in the knowledge broker class.

4. Conclusion and future plans

In this paper, we have described the main design features of the ALPS class library, a tree search handler, and two layers built on top of ALPS that provide the data handling needed for implementing a data-intensive algorithms such as BCP. This project is being developed as open source under the auspices of the Computational Infrastructure for Operations Research (COIN-OR) initiative. This first version of the library hierarchy is still under development, but source code will be available from the CVS repository at www.coin-or.org.

Acknowledgments

Funding from NSF grant ACI-0102687 and IBM Faculty Partnership Award.

References

1. D. Applegate, R. Bixby, V. Chátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians*, 645, 1998.
2. D. Applegate, R. Bixby, V. Chátal, and W. Cook. CONCORDE TSP Solver, available at www.keck.caam.rice.edu/concorde.html.
3. E. Balas. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13:517–546, 1965.
4. M. Benchouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In *Solving Combinatorial Optimization Problems in Parallel*, Lecture Notes in Computer Science, 1054:221, Springer, Berlin 1996.
5. R. Bixby, W. Cook, A. Cox, and E. K. Lee. Parallel mixed integer programming, Rice University Center for Research on Parallel Computation Research Monograph CRPC-TR95554, 1995.
6. R. L. Boehning, R. M. Butler, and B. E. Gillet. A parallel integer linear programming algorithm. *European Journal of Operations Research*, 34, 1988.
7. A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens. Asynchronous parallel branch and bound and anomalies, Report EUR-CS-95–05, Department of Computer Science, Erasmus University, Rotterdam, 1995.

8. Q. Chen and M. C. Ferris. FATCOP: A fault tolerant condor-PVM mixed integer programming solver. University of Wisconsin CS Department Technical Report 99-05, Madison, WI, 1999.
9. R. Correa and A. Ferreira. Parallel best-first branch and bound in discrete optimization: A framework, Center for Discrete Mathematics and Theoretical Computer Science Technical Report 95-03.
10. L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5:46, 1998.
11. J. Eckstein. Parallel branch and bound algorithms for general mixed integer programming on the CM-5. *SIAM Journal on Optimization*, 4, 1994.
12. J. Eckstein. How much communication does parallel branch and bound need? *INFORMS Journal on Computing*, 9, 1997.
13. J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch and bound, RUTCOR Research Report 40-2000, Rutgers University, Piscataway, NJ, 2000.
14. 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.
15. A. Geist et al. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.
16. B. Gendron and T. G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042, 1994.
17. A. Grama and V. Kumar. Parallel search algorithms for discrete optimization problems. *ORSA Journal on Computing*, 7, 1995.
18. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*, MIT University Press, Cambridge, MA, 1999.
19. M. Grotschel, M. Junger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1155, 1984.
20. K. Hoffman and M. Padberg. LP-based combinatorial problem solving. *Annals of Operations Research*, 4:145, 1985/86.
21. V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22, 1994.
22. A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.
23. J. Linderoth, *Topics in Parallel Integer Optimization*, Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA (1998).
24. G. Mitra, I. Hai, and M. T. Hajian. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing*, 23:733–753, 1997.
25. L. G. Mitten. Branch-and-bound methods: General formulation and properties. *Operations Research*, 18:24, 1970.
26. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*, John Wiley & Sons, Inc., 1988.
27. M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale traveling salesman problems. *SIAM Review*, 33:60, 1991.
28. T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98:253, 2003.
29. T. K. Ralphs. SYMPHONY Version 4.0 user's guide, available at www.branchandcut.org/SYMPHONY.
30. T. K. Ralphs, L. Ladányi, and L. E. Trotter, *Branch, Cut, and Price: Sequential and Parallel*, in Computational Combinatorial Optimization, D. Naddef and M. Jünger, eds., Springer, Berlin, (2001), 223.
31. T. K. Ralphs and L. Ladányi. *COIN/BCP User's Guide*, available at www.coin-or.org
32. R. Rushmeier and G. Nemhauser. Experiments with parallel branch and bound algorithms for the set covering problem. *Operations Research Letters*, 13, 1993.
33. M. J. Saltzman, "COIN-OR: An Open-Source Library for Optimization," in S. Nielsen, ed., *Programming Languages and Systems in Computational Economics and Finance*, Kluwer, Boston, 2002.

34. Y. Shinano, M. Higaki, and R. Hirabayashi. 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, 1995, 392.
35. H. W. J. M. Trienekens and A. de Bruin. *Towards a Taxonomy of Parallel Branch and Bound Algorithms*, Report EUR-CS-92–01, Department of Computer Science, Erasmus University Rotterdam, 1992.
36. S. Tschöke, and T. Polzer. *Portable Parallel Branch and Bound Library User Manual*, Library Version 2.0. Department of Computer Science, University of Paderborn.
37. S. J. Wright. Solving Optimization Problems on Computational Grids. *Optima*, 65, 2001.