ALPS: A FRAMEWORK FOR IMPLEMENTING PARALLEL TREE SEARCH ALGORITHMS

Yan Xu
Operations R & D, SAS Institute Inc., Cary NC 27513
Yan.Xu@sas.com

Ted K. Ralphs
Department of Industrial and Systems Engineering, Lehigh University, Bethlehem PA 18015
tkralphs@lehigh.edu

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

Matthew J. Saltzman
Department of Mathematical Sciences, Clemson University, Clemson SC 29634
mjs@clemson.edu

Abstract ALPS is a framework for implementing and parallelizing tree search algorithms. It employs a number of features to improve scalability and is designed specifically to support the implementation of *data intensive* algorithms, in which large amounts of *knowledge* are generated and must be maintained and shared during the search. Implementing such algorithms in a scalable manner is challenging both because of storage requirements and because of communications overhead incurred in the sharing of data. In this abstract, we describe the design of ALPS and how the design addresses these challenges. We present two sample applications built with ALPS and preliminary computational results.

Keywords: Parallel Algorithm, Integer Programming, Software, Branch and Bound

1. Introduction

Tree search algorithms are a general class in which the nodes of a directed, acyclic graph are systematically searched in order to locate one or more *goal nodes*. In most cases, the graph to be searched is not known a priori, but is constructed dynamically based on information discovered during the search process. We assume the graph has a unique *root node* with no incoming arcs, which is the first node to be examined. In this case, the search order uniquely determines a rooted tree called the *search tree*. Although tree search algorithms are easy to parallelize in principle, the absence of a priori knowledge of the shape of the tree and the need to effectively share information generated during the search makes such parallelization challenging and scalability difficult to achieve. In [Ralphs et al., 2003] and [Ralphs et al., 2004], we examined the issues surrounding parallelization of tree search algorithms and presented a high-level description of a class hierarchy for implementing such algorithms. In this abstract, we follow up on those works by presenting further details of the search handling layer of the proposed hierarchy, called the Abstract Library for Parallel Search (ALPS), which will soon have its first public release.

A variety of existing software frameworks are based on tree search. For mixed-integer programming—the application area we are most interested in—most packages employ a sophisticated variant of branch and bound. Among the offerings for solving generic mixed-integer programs are bc-opt [Cordier et al., 1999], FATCOP [Chen and Ferris, 2001], MIPO [Balas et al., 1996], PARINO [Linderoth, 1998], SIP [Martin, 1998], SBB [Forrest, 2004], GLPK [Makhorin, 2004], and bonsaiG [Hafer, 1999]. Of this list, FATCOP and PARINO are parallel codes. Commercial offerings include ILOG's CPLEX, IBM's OSL (soon to be discontinued), and Dash's XPRESS. Generic frameworks that facilitate extensive user customization of the underlying algorithm include SYMPHONY [Ralphs, 2004], ABACUS [Jünger and Thienel, 2001], BCP [Ladányi and Ralphs, 2001], and MINTO [Nemhauser et al., 1994], of which SYMPHONY and BCP are parallel codes. Other frameworks for parallel branch and bound include BoB [Benchouche et al., 1996], PICO [Eckstein et al., 2000], PPBB-Lib [Tschoke and Polzer, 1998], and PUBB [Shinano et al., 1995]. Good overviews and taxonomies of parallel branch and bound are provided in both [Gendron and Crainic, 1994] and [Trienekens and Bruin, 1992]. Eckstein et al. [Eckstein et al., 2000] also provides a good overview of the implementation of parallel branch and bound. A substantial number of papers have been written specifically about the application of parallel branch and bound to discrete optimization problems, including [Bixby et al., 1995; Correa and Ferreira, 1995; Grama and Kumar, 1995; Mitra et al., 1997].

The goal of the ALPS project is to build on the best existing methodologies while addressing their shortcomings to produce a framework that is more general and extensible than any of the current options. As such, we provide support for the implementation of a range of algorithms that existing frameworks are not general enough to handle. Our design is centered around the abstract notion of *knowledge generation and sharing*, which is very general and central to implementing scalable versions of today's most sophisticated tree search algorithms. Such algorithms are inherently *data-intensive*, i.e., they generate large amounts of knowledge as a by-product of the search. This knowledge must be organized, stored, and shared efficiently. ALPS provides explicit support for these procedures and allows for user-defined knowledge types, making it easy to create derivative frameworks for a wide range of specific classes of algorithms. While our own experience is in developing algorithms for solving mixed-integer linear programs, we have in mind to develop a number of additional layers providing support for tree search algorithms in other areas, such as global optimization. Although we present limited computational results, we want to emphasize that this research is ongoing and that the results are intended merely to illustrate the challenges we still face. The main goal of the paper is to describe the framework itself. ALPS is being developed in association with the Computational Infrastructure for Operations Research (COIN-OR) Foundation [Lougee-Heimer, 2003], which will host the code.

1.1 Tree Search Algorithms

In a tree search algorithm, each node in the search graph has associated data, called its *description*, that can be used to determine if it is a goal node, and if it has any successors. To specify such an algorithm, four main elements are required. The *fathoming rule* determines whether a node has successors that need to be explored. The *branching method* specifies how to generate the descriptions of a node's successors. The *processing method* determines whether a node is a goal node and whether it has any successors. The *search strategy* specifies the processing order of the candidate nodes.

Each node has an associated *status*, which is one of: `candidate` (available for processing), `active` (currently being processed), `fathomed` (processed and has no successors), or `processed` (not fathomed, hence has successors). The search consists of repeatedly selecting a candidate node (initially, the root node), processing it, and then either fathoming or branching. The nodes are chosen according to *priorities* assigned during processing.

Variants of tree search algorithms are widely applied in areas such as discrete optimization, global optimization, stochastic programming, artificial intelligence, game playing, theorem proving, and constraint programming. One of the most common variants in discrete optimization is *branch and bound*, originally suggested by Land and Doig [Land and Doig, 1960]. In branch and bound, branching consists of partitioning the feasible set into subsets. Processing consists of computing a bound on the objective function value, usually by solving a relaxation. A node can be fathomed if (1) the solution to the relaxation is in the original feasible set (in which case, the best such solution seen so far is recorded as the *incumbent*), (2) the objective value of the solution to the relaxation exceeds the value of the incumbent, or (3) the subset is proved to be empty.

1.2 Parallelizing Tree Search

In principle, tree search algorithms are easy to parallelize. Sophisticated variants, however, involve the generation and sharing of large amounts of *knowledge*, i.e., information helpful in guiding the search and improving the effectiveness of node processing. Inefficiencies in the mechanisms by which knowledge is maintained and shared result in *parallel overhead*, which is additional work performed in the parallel algorithm that would not have been performed in the sequential one. The goal of any parallel implementation is to limit this overhead as much as possible.

We assume a simple model of parallel computation in which there are N processors with access to their own local memory and complete connectivity with other processors. We further assume that there is exactly one process per processor at all times, though this process might be multi-threaded. The main sources of parallel overhead for tree search algorithms are:

- *Communication Overhead*: time spent actively sending or receiving knowledge.
- *Idle Time*: time spent waiting for knowledge to be transferred from another processor (including *task starvation*, when the processor is waiting for more work to do).
- *Redundant Work*: time spent performing unnecessary work, usually due to a lack of appropriate global knowledge.
- *Ramp-Up/Ramp-Down*: idle time at the beginning/end of the algorithm during which there is not enough work for all processors.

The effectiveness of the knowledge-sharing mechanism is the main factor affecting this overhead. The sources of overhead listed above highlight the tradeoff between centralized storage and decision making, which incurs increased communication and idle time, and decentralized storage and decision making, which increases performance of redundant work. Achieving the proper balance is the challenge we face. *Scalability* is a measure of how well this balance is achieved, i.e., how well an algorithm takes advantage of increased computing resources, primarily additional processors. Our measure of scalability is the rate of increase in overhead as additional processors are made available. A parallel algorithm is considered scalable if this rate is near linear. An excellent general introduction to the analysis of parallel scalability is provided in [Kumar and Gupta, 1994].

2. Implementation

2.1 Knowledge Sharing

In [Ralphs et al., 2004], building on ideas in [Trienekens and Bruin, 1992], we proposed a tree search methodology driven by the concept of knowledge discovery and sharing. We briefly review the concepts from the earlier work here. The design of ALPS is predicated on the idea that all information required to carry out a tree search can be represented as knowledge that is generated dynamically and stored in various local *knowledge pools* (KPs), which share that knowledge when needed. A single processor can host multiple KPs that store different types of knowledge and are managed by a *knowledge broker* (KB). Examples of knowledge generated while solving mixed-integer programs include feasible solutions, search-tree nodes, and valid inequalities.

The KB associated with a KP may field two types of requests on its behalf: (1) new knowledge to be inserted into the KP or (2) a request for relevant knowledge to be extracted from the KP, where "relevant" is defined for each category of knowledge with respect to data provided by the requesting process. A KP may also choose to "push" certain knowledge to another KP, even though no specific request has been made.

The most fundamental knowledge generated during the search is the descriptions of the search-tree nodes themselves. The node descriptions are stored in KPs called *node pools.* The node pools collectively contain the list of candidate nodes. The tradeoff between centralization and decentralization of knowledge is most evident in the mechanism for sharing node descriptions among the processors, known as *load balancing.* Effective load balancing reduces both idle time associated with task starvation and performance of redundant work. Load balancing methods have been studied extensively [Fonlupt et al., 1998; Henrich, 1993; Kumar et al., 1994; Laursen, 1994; Sanders, 1998; Sinha and Kalé, 1993], but many of the suggested schemes are not suited for our framework. The simplest approach is a *master-worker* design that stores all node descriptions in a single, central node pool. This makes work distribution easy, but incurs high communication costs. This is the approach we have taken in our previous frameworks, SYMPHONY and BCP. It works well for small numbers of processors, but does not scale well, as the central node pool inevitably becomes a computational and communications bottleneck.

2.2 The Master-Hub-Worker Paradigm

To overcome the drawbacks of the master-worker approach, ALPS 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, a *cluster* consists of a hub, which is responsible for managing a fixed number of workers. As the number of processes increases, we simply add more hubs and more clusters of workers. This scheme is similar to one implemented by Eckstein et al. in the PICO framework [Eckstein et al., 2000], except that PICO does not have the concept of a master. This decentralized approach maintains many of the advantages of global decision making while reducing overhead and moving some computational burden from the master process to the hubs. This burden is then further shifted from the hubs to the workers by increasing the task granularity, as described below. Cluster size is computed based on the number of hubs and the number of processors, which are set by the user at run time.

The basic unit of work in our design is a *subtree.* Each worker is capable of processing an entire subtree autonomously and has access to all of the methods needed to manage a tree search. Designating a subtree as the fundamental unit of work helps to minimize memory requirements by enabling the use of efficient data structures for storing subtrees using a differencing scheme similar to that used in both SYMPHONY and BCP. In this scheme, node descriptions are not stored explicitly, but rather as differences from their predecessors' descriptions. This increased granularity also reduces idle time due to task starvation, but, without proper load balancing, may increase the performance of redundant work.

2.3 Load Balancing

Recall that each node has an associated priority that can be thought of as indicating the node's "quality," i.e., the probability that the node or one of its successors is a goal node. In assessing the distribution of work to the processors, we need to consider not only *quantity*, but also *quality*. ALPS employs a three-tiered load balancing scheme, consisting of *static*, *intra-cluster dynamic*, and *inter-cluster dynamic* load balancing. Static load balancing, or *mapping*, takes place during the initial phase of the algorithm. The first task is to generate a group of successors of the root node and distribute them to the workers to initialize their local node pools. ALPS uses a *two-level root initialization* scheme, a generalization of the *root initialization* scheme of [Henrich, 1993]. During static load balancing, the master creates and distributes a user-specified number of nodes for hubs. The hubs in turn create a user-specified number of successors for their workers, then the workers initialize their subtree pools and begin.

Time spent performing static load balancing is the main source of ramp-up, which can be significant when node processing times are large. The problem of reducing ramp-up has long been recognized as a challenging one [Gendron and Crainic, 1994; Borbeau et al., 2000; Eckstein et al., 2000]. Two-level root initialization reduces ramp-up by parallelizing the root initialization process itself. Implementation of two-level root initialization is straightforward, but our experience has shown that it can work quite well if the number of nodes distributed to each worker is large enough and node processing times are short.

Inside a cluster, the hub manages dynamic load balancing. Intra-cluster load balancing is initiated when an individual worker reports to the hub that its workload is below a given threshold. Upon receiving the request, the hub asks its most loaded worker to donate a subtree to the requesting worker. In addition, the hub periodically checks the qualities of the workloads of its workers. If it finds that the qualities are unbalanced, the hub asks the workers with the most high priority nodes to share their workload with the workers that have fewer such nodes.

The master is responsible for balancing the workload among hubs, which periodically report their workload information to the master. The master has a roughly accurate global view of the system load and the load of each cluster at all times. If either the quantity or quality of work is unbalanced among the clusters, the master identifies pairs of *donors* and *receivers*. Donors are clusters whose workloads are greater than the average workload of all clusters by a given factor. Receivers are the clusters whose workloads are smaller than the average workload by a given factor. Donors and receivers are paired and each donor sends a subtree to its paired receiver.

A unique aspect of our load balancing scheme is that it takes account of the differencing scheme for storing subtrees. In order to prevent subtrees from becoming too fractured for efficient storage using differencing, we try at all times to ensure that the search-tree nodes are distributed in a way such that the nodes stored together locally constitute connected subtrees of the search tree. This means the tree structure must be taken into account when sharing nodes during the load balancing. Candidate nodes that constitute the leaves of a subtree are grouped, and the entire subtree is shared, rather than just the nodes themselves. To achieve this, each subtree is assigned a priority level, defined as the average priorities of a given number of its best nodes. During load balancing, the donor chooses the best subtree in its subtree pool and sends it to the receiver. If a donor does not have any subtrees in its subtree pool, it splits the subtree that it is currently exploring into two parts and sends one of them to the receiver. In this way, differencing can still be used effectively.

2.4 Task Management

Because each process hosts a KB and several KPs, it is necessary to have a scheme for enabling multi-tasking. In order to maintain maximum portability and to assert control over task scheduling, we have implemented our own simple version of threading. ALPS processes are message driven—each process devotes one thread to listening for and responding to messages at all times. Other threads are devoted to performing computation as scheduled. Because each processor's KB controls the communication to and from the process, it also controls task scheduling. The KB receives external messages, forwards them to the appropriate local KP if needed, and forwards all locally generated messages to the appropriate remote KB. When not listening for messages, the KB schedules the execution of computational tasks by the local KPs. The KB decides when and for how long to process each task.

3. Class Structure

ALPS consists of a library of C++ classes from which can be derived specialized classes that define various tree search algorithms. Figure 1 shows the ALPS class hierarchy. Each block represents a C++ class, whose name is listed in the block. The lines ending with triangles represent inheritance relationships. For example, the `AlpsSolutionPool`, `AlpsSubtreePool` and `AlpsNodePool` classes are derived from the class `AlpsKnowledgePool`. The lines ending with diamonds represent associative relationships. For instance, `AlpsKnowledge` contains as a data member a pointer to an instance of `AlpsEncoded`. ALPS is comprised of just three main base classes and a number of derived and auxiliary classes. These classes support the core concept of knowledge sharing and

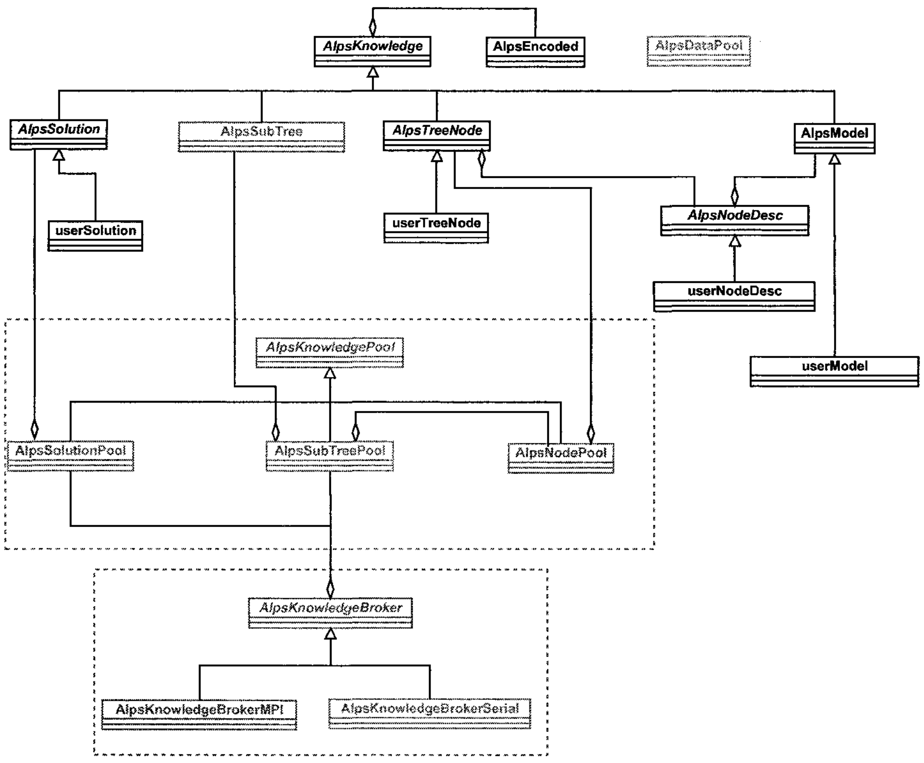

Figure 1. The ALPS class hierarchy.

are described in the paragraphs below. The classes named `UserXXX` in the figure are those that must be defined by the user to develop a new application. Two examples are described in Section 4.

`AlpsKnowledge.` This is the virtual base class for any type of information that must be shared or moved from one process to another. `AlpsEncoded` is an associated class that contains the encoded or packed form of an `AlpsKnowledge` object. The packed form contains the data needed to describe an object of a particular type in the form of a character string. This representation typically takes much less memory than the object itself; hence, it is appropriate both for storage of knowledge and for communication of knowledge between processors. The packed form is also independent of type, which allows ALPS to deal with user-defined knowledge types. Finally, duplicate objects can be quickly identified by hashing their packed forms. ALPS has the following four native knowledge types:

- `AlpsSolution`: A description of the goal state or solution to the problem being solved.
- `AlpsTreeNode`: Contains the data and methods associated with a node in the search graph. Each node contains a description, which is an object of type `AlpsNodeDesc`, as well as the definitions of the process and branch methods.
- `AlpsModel`: Contains the data describing the original problem.
- `AlpsSubTree`: Contains the description of a subtree, which is a hierarchy of `AlpsTreeNode` objects, along with the methods needed for performing a tree search.

The first three of these classes are virtual and must be defined by the user in the context of the problem being solved. The last class is generic and problem-independent.

`AlpsKnowledgePool`. The role of the `AlpsKnowledgePool` is described in Section 2.1. There are several derived classes that define native knowledge types. The user can define additional algorithm-specific knowledge types.

- `AlpsSolutionPool`: The solution pools store `AlpsSolution` objects. These pools exist both at the worker level—for storing solutions discovered locally—and globally at the master level.
- `AlpsSubTreePool`: The subtree pools store `AlpsSubTree` objects. These pools exist at the hub level for storing subtrees that still contain unprocessed nodes.
- `AlpsNodePool`: The node pools store `AlpsTreeNode` objects. These pools contain the queues of candidate nodes associated with the subtrees as they are being searched.

`AlpsKnowledgeBroker`. This class encapsulates the communication protocol. The KB is the driver for each processor and is responsible for sending, receiving, and routing all data that resides on that processor. Each KP must be registered so that the KB knows how to route each specific type of knowledge when it arrives and where to route requests for specific types of knowledge from other KBs. This is the only class whose implementation depends on the communication protocol. Currently, the protocols supported are a serial layer and an MPI [Gropp et al., 1999] layer.

- `AlpsKnowledgeBrokerMPI`: A KB for multiprocessor execution via the MPI message-passing interface.
- `AlpsKnowledgeBrokerSerial`: A KB for uniprocessor execution.

```
#include Alps.h
#include AlpsUser.h    // User-derived classes

int main(int argc, char* argv[])
{
    UserModel model;
    UserParams userPar;

#if defined(SERIAL)
    AlpsKnowledgeBrokerSerial broker(argc, argv, model, userPar);
#elif defined(PARALLEL_MPI)
    AlpsKnowledgeBrokerMPI broker(argc, argv, model, userPar);
#endif
    broker.registerClass("MODEL", new UserModel);
    broker.registerClass("SOLUTION", new UserSolution);
    broker.registerClass("NODE", new UserTreeNode);
    broker.search();
    broker.printResult();
    return 0;
}
```

Figure 2. Sample main function.

4. Applications and Preliminary Results

Developing an application with ALPS consists mainly of implementing derived classes, and writing the `main()` function. As described in Section 3, the user must derive algorithm-specific classes from the base classes `AlpsModel`, `AlpsTreeNode`, `AlpsNodeDesc`, and `AlpsSolution`. The user may also want to define algorithm-specific parameters by deriving a class from `AlpsParameterSet`, or he may even want to define new types of knowledge. A sample code for `main()` is shown in Figure 2.

4.1 Knapsack Solver

The binary knapsack problem is to select from a set of items a subset with the maximum total profit and not exceeding a given total weight. The profit is additive. By deriving classes `KnapModel`, `KnapTreeNode`, `KnapNodeDesc`, `KnapSolution` and `KnapParameterSet`, we have developed a solver for the binary knapsack problem employing a very simple branch and bound algorithm. The nodes of the search tree are described by subproblems obtained by fixing a subset of the items in the global set to be either in or out of the selected subset. The branching procedure consists of selecting an item and requiring it to be in the selected subset in one successor node and not in the other. Processing consists

N	Wall-clock	Ramp-up	Idle	Speedup	Efficiency	Nodes
1	1335	–	–	–	–	254 k
4	296	0%	2.9%	4.5	1.13	85 m
8	160	0%	2.6%	8.3	1.04	85 m
16	94	0%	7.8%	14.2	0.89	85 m
32	53	0%	7.9%	26.3	0.83	85 m

Table 1. Overall results on four knapsack instances.

of solving the knapsack problem without binary constraints (subject to the items that are fixed) to obtain a lower bound, which is then used to determine the node's priority (lower is better). Fathoming occurs when the solution to the relaxation is feasible to the binary problem or the lower bound exceeds the value of the incumbent. The search strategy is to choose the candidate node with the lowest lower bound (best first).

To illustrate the performance of the solver, we randomly generated four difficult knapsack instances using the method described in [Martello and Toth, 1990]. These results are not meant to be comprehensive. Clearly, further testing on a much larger scale is needed and complete performance results will be reported in a full paper to follow. Testing was conducted on a Beowulf cluster with 48 dual processor nodes. Each node has two 1.0-GHz Pentium III processors and 512 megabytes of RAM. The operating system was Red Hat Linux 7.2. The message-passing library used was LAM/MPI. Five trials were run for each instance, with two hubs employed when the number of processors was eight or more. Table 1 shows the number of processors used (N), the wall-clock running time (in seconds), the percentage idle time, the speedup (ratio of the sequential and parallel running times), the parallel efficiency (ratio of the speedup to the number of processors), and the number of nodes enumerated. The efficiency approximates the percentage of running time devoted to useful work and should ideally be near one. Efficiencies significantly below one indicate the presence of overhead. We used SBB [Forrest, 2004] to produce the sequential running times for comparison. Because our solver does not employ advanced techniques such as dynamic cut generation or primal heuristics, we disabled these capabilities with SBB as well. SBB still generated many fewer search-tree nodes due to its use of strong branching. Nonetheless, the comparison provides a useful baseline. From Table 1, we see that the speedup is near linear. Ramp-up time is negligible, but idle time still leaves room for improvement. The number of nodes enumerated is not increasing, which indicates that the performance of redundant work is not a problem.

Problem	N	Wall-clock	Ramp-up	Idle	Speedup	Eff	Nodes
gesa3	1	1626	–	–	–	–	403
gesa3	4	614	9.8%	0	2.6	0.66	445
gesa3	8	269	35.1%	0.2%	6.0	0.76	337
gesa3	16	161	49.1%	0.1%	10.1	0.63	247
blend2	1	1565	–	–	–	–	2339
blend2	4	258	12.8%	0	6.1	1.53	1019
blend2	8	213	14.0%	0.2%	7.3	0.92	717
blend2	16	129	34.1%	0	12.1	0.76	980
fixnet6	1	2716	–	–	–	–	2729
fixnet6	4	703	1.0%	0	3.9	0.98	3598
fixnet6	8	626	3.0%	0.2%	4.3	0.54	4703
fixnet6	16	376	4.6%	0	7.2	0.45	6570
cap6000	1	4287	–	–	–	–	6129
cap6000	4	1344	0.2%	0	3.2	0.80	9551
cap6000	8	1012	0.3%	0	4.2	0.53	12363
cap6000	16	640	1.2%	0.2%	6.7	0.42	14121

Table 2. Computational results of sample MILP problems.

4.2 Mixed-integer Linear Program Solver

For the knapsack solver, node processing times were negligible and good feasible solutions were discovered early in the solution process, which makes scalability relatively easy to achieve. As a more stringent test, we have developed a generic solver for mixed-integer linear programs (MILPs) called ALPS Branch and Cut (ABC), employing a straightforward branch and cut algorithm with cuts generated using the COIN-OR Cut Generation Library [Lougee-Heimer, 2003]. ABC consists of the classes `AbcModel`, `AbcTreeNode`, `AbcNodeDesc`, `AbcSolution`, and `AbcParameterSet`. The search strategy is best first. Strong branching is used to choose the variables to be branched on. ABC also uses the SBB rounding heuristic as a primal heuristic.

We tested ABC using four problems: *gesa3*, *blend2*, *fixnet6*, and *cap6000* from MIPLIB3 [Bixby et al., 1998]. As above, these results are meant to be illustrative, not comprehensive. As with the knapsack example, two hubs were used when the number of processes was eight or more. The results are summarized in Table 2.

From Table 2, we see that for generic MILPs, parallel efficiency is not as easy to achieve. However, the source of overhead is quite problem dependent. For *gesa3* and *blend2*, ramp-up is a major problem, due to large node processing time near the top of the tree. Neither *gesa3* nor *blend2* exhibits signs of the performance of redundant work. Also, as the number of processors increases, the number of search nodes decreases. This is primarily due to the fact that good feasible solutions are found early in the search process. For *fixnet6* and *cap6000*, ramp-up is not a problem, but the number of nodes processed increases when the number of processes increases, indicating the presence of redundant work. For these problems, good feasible solutions are not found until much later in the search process. These results illustrate the challenges that we still face in improving scalability. We discuss prospects for the future in the final section.

5. Summary and Future Work

In this paper, we have described the main features of the ALPS framework. Two applications were developed to test ALPS. The limited computational results highlight the challenges we still face in achieving scalability. The preliminary results obtained for ABC highlight the two most difficult scalability issues to address for MILP—reduction of ramp-up time and elimination of redundant work. Controlling ramp-up time is the most difficult of these. Attempts to branch early in order to produce successors more quickly have thus far been unsuccessful. A number of other ideas have been suggested in the literature. Two that we are currently exploring are (1) using a branching procedure that creates a large number of successors instead of just the current two, and (2) utilizing the processors idle during ramp-up in order to find a good initial feasible solution, thereby helping to eliminate redundant work. The first approach seems unlikely to be successful, but the second one may hold the key. This approach is also being explored by Eckstein et al. in the context of PICO. As for eliminating redundant work, this can be done by fine-tuning our load balancing strategies, which are currently relatively unsophisticated, to ensure a better distribution of high-priority work.

In future work, we will continue to improve the performance of ALPS by refining our methods of reducing parallel overhead as discussed above. Also, we will continue development of the Branch, Constrain, and Price Software (BiCePS) library, the data handling layer for solving mathematical programs that we are building on top of ALPS. BiCePS will introduce dynamically generated cuts and variables as new types of knowledge and support the implementation of parallel branch and bound algorithms in which the bounds are obtained by Lagrangian relaxation. Finally, we will build the BiCePS Linear Integer Solver (BLIS) on top of BiCePS. BLIS will be a LP-based branch, cut, and price solver for MILPS, like ABC, but with user customization features akin to SYMPHONY and BCP.

Acknowledgments. This research was partially supported through NSF grant ACI-0102687 and the IBM Faculty Partnership Program.

References

Balas, E., Ceria, S., and Cornuéjols, G. (1996). Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42:1229–1246.

Benchouche, M., Cung, V.-D., Dowaji, S., Cun, B. L., Mautor, T., and Roucairol, C. (1996). Building a parallel branch and bound library. In *Solving Combinatorial Optimization Problems in Parallel*. Springer, Berlin.

Bixby, R., Ceria, S., McZeal, C., and Savelsbergh, M. (1998). An updated mixed integer programming library: MIPLIB 3. Technical Report TR98-03, Department of Computational and Applied Mathematics, Rice University.

Bixby, R., Cook, W., Cox, A., , and Lee, E. (1995). Parallel mixed integer programming. Research Monograph CRPC-TR95554, Rice University Center for Research on Parallel Computation.

Borbeau, B., Crainic, T., and Gendron, B. (2000). Branch-and-bound parallelization strategies applied to a depot location and container fleet management problem. *Parallel Computing*, 26:27–46.

Chen, Q. and Ferris, M. C. (2001). FATCOP: A fault tolerant Condor-PVM mixed integer program solver. *SIAM Journal on Optimization*, 11:1019–1036.

Cordier, C., Marchand, H., Laundy, R., and Wolsey, L. A. (1999). bc-opt: A branch-and-cut code for mixed integer programs. *Mathematical Programming*, 86:335–353.

Correa, R. and Ferreira, A. (1995). Parallel best-first branch and bound in discrete optimization: A framework. Technical Report 95-03, Center for Discrete Mathematics and Theoretical Computer Science.

Eckstein, J., Phillips, C. A., and Hart, W. E. (2000). Pico: An object-oriented framework for parallel branch and bound. Technical Report RRR 40-2000, Rutgers University.

Fonlupt, C., Marquet, P., and Dekeyser, J. (1998). Data-parallel load balancing strategies. *Parallel Computing*, 24:1665–1684.

Forrest, J. (2004). Simple branch and bound. Available from http://www.coin-or.org.

Gendron, B. and Crainic, T. (1994). Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066.

Grama, A. and Kumar, V. (1995). Parallel search algorithms for discrete optimization problems. *ORSA Journal on Computing*, 7:365–385.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI*. MIT Press, Cambridge, MA, USA, 2nd edition.

Hafer, L. (1999). bonsaiG: Algorithms and design. Technical Report SFU-CMPTTR 1999-06, Simon Frazer University Computer Science.

Henrich, D. (1993). Initialization of parallel branch-and-bound algorithms. In *Second International Workshop on Parallel Processing for Artificial Intelligence(PPAI-93)*.

Jünger, M. and Thienel, S. (2001). The abacus system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience*, 30:1325–1352.

Kumar, V., Grama, A. Y., and Vempaty, N. R. (1994). Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22:60–79.

Kumar, V. and Gupta, A. (1994). Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22:379–391.

Ladányi, L. and Ralphs, T. (2001). *COIN/BCP User's Manual.* Available from http://www.coin-or.org.

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

Laursen, P. S. (May, 1994). Can parallel branch and bound without communication be effective? *SIAM Journal on Optimization*, 4:33–33.

Linderoth, J. (1998). *Topics in Parallel Integer Optimization.* PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA.

Lougee-Heimer, R. (2003). The Common Optimization INterface for Operations Research. *IBM Journal of Research and Development*, 47:57–66.

Makhorin, A. (2004). Introduction to GLPK. Available from http://www.gnu.org/software/glpk/glpk.html.

Martello, S. and Toth, P. (1990). *Knapsack Problems: algorithms and computer implementation.* John Wiley & Sons, Inc., USA, 1st edition.

Martin, A. (1998). Integer programs with block structure. Habilitation Thesis, Technical University of Berlin, Berlin, Germany.

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

Nemhauser, G. L., Savelsbergh, M. W. P., and Sigismondi, G. S. (1994). Minto, a mixed integer optimizer. *Operations Research Letters*, 15:47–58.

Ralphs, T. (2004). *SYMPHONY Version 4.0 User's Manual.* Available from `http://www.branchandcut.org/SYMPHONY`.

Ralphs, T., Ladányi, L., and Saltzman, M. J. (2003). Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98:253–280.

Ralphs, T., Ladányi, L., and Saltzman, M. J. (2004). A library hierarchy for implementing scalable parallel search algorithms. *The Journal of Supercomputing*, 28:215–234.

Sanders, P. (1998). Tree shaped computations as a model for parallel applications. In *ALV'98 Workshop on application based load balancing*, pages 123–132.

Shinano, Y., Harada, K., and Hirabayashi, R. (1995). A generalized utility for parallel branch and bound algorithms. In *Proceedings of the 1995 Seventh Symposium on Parallel and Distributed Processing*, pages 392–401, Los Alamitos, CA. IEEE Computer Society Press.

Sinha, A. and Kalé, L. V. (1993). A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA.

Trienekens, H. W. J. M. and Bruin, A. d. (1992). Towards a taxonomy of parallel branch and bound algorithms. Report EUR-CS-92-01, Erasmus University, Rotterdam.

Tschoke, S. and Polzer, T. (1998). *Portable Parallel Branch and Bound Library User Manual: Library Version 2.0.* Department of Computer Science, University of Paderborn.