# ALPS: A Framework for Implementing Parallel Search Algorithms

Y. Xu
SAS Institute, Inc.

T. K. Ralphs
Lehigh University

L. Ladányi
IBM T. J. Watson Research Center

M. J. Saltzman
Clemson University

Report No. 04T-010

# ALPS: A Framework for Implementing Parallel Search Algorithms

Y. Xu[*], T. K. Ralphs[†], L. Ladányi[‡], and M. J. Saltzman[§]

May 13, 2004

**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.

## 1 Introduction

Tree search algorithms are a general class of algorithms in which the nodes of a directed, acyclic graph are systematically searched with the goal of locating one or more nodes, called *goal nodes*, satisfying a given property. In most cases, the graph to be searched is not known a priori, but is constructed dynamically based on knowledge discovered during the search process. For simplicity, we assume without loss of generality that the graph has a unique *root node* with no incoming arcs, which is the first node to be examined during the search. In this case, the search order uniquely determines a rooted tree that we call 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 knowledge generated during the search process makes such parallelization challenging and scalability difficult to achieve. In [24] and [25], we examined the issues surrounding the 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 implantation of the Abstract Library for Parallel Search (ALPS), which comprises the search handling layer of a proposed hierarchy. Additional layers that implement the data

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

[†]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem PA 18015, `tkralphs@lehigh.edu`

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

[§]Department of Mathematical Sciences, Clemson University, Clemson SC 29634, `mjs@clemson.edu`

handling needed for advanced mathematical programming algorithms are under development, but will not be described here.

A variety of existing software frameworks are based on tree search. For mixed-integer programming, which is the application area we are most interested in, most of these software packages employ a sophisticated variant of branch and bound, a tree search algorithm described in the next section. Among the offerings for solving generic mixed-integer programs are bc-opt [7], FATCOP [5, 6], MIPO [3], PARINO [18], SIP [20], COIN/SBB [10], and bonsaiG [11]. Of this list, FATCOP and PARINO are parallel codes. Generic frameworks that facilitate extensive user customization of the underlying algorithm include SYMPHONY [23], ABACUS [13], COIN/BCP [10], and MINTO [21], of which SYMPHONY, ABACUS and COIN/BCP are parallel codes. Other frameworks for parallel branch and bound include BoB [4], PICO [8], PPBB-Lib [31], and PUBB [28]. With ALPS, we have aimed to develop a framework significantly more general that any of those named here, but without reducing our ability to use the framework to produce customized packages with similar capabilities and performance. ALPS is being developed in association with the Computational Infrastructure for Operations Research (COIN-OR) project [10], which will host the code.

## 1.1 Tree Search Algorithms

In a tree search algorithm, each node of the search graph contains associated data called its *description*. As the search progresses, each node is examined or *processed* to determine (1) whether it satisfies the properties of a goal node and (2) whether it is a leaf node in the search tree or has successors that must be explored. Each node also has an associated *status*, which is one of: `candidate` (available for processing), `active` (currently being processed), `fathomed` (processed and has has no successors), or `processed` (not fathomed, hence has successors). To specify a tree search algorithm, the following main elements are required:

- *Fathoming rule*: determines whether a node has descendants that need to be explored.
- *Branching method*: specifies how to generate the descriptions of a node's successors.
- *Processing method*: creates knowledge useful for determining if a node is a goal node, or fathomed, or is likely to have goal nodes among its descendants.
- *Search strategy*: specifies the processing order of the candidate nodes.

The search consists of choosing a candidate node (initially, the root node), processing it, and then either fathoming or branching.

Variants of tree search algorithms are widely applied in areas such as discrete optimization, artificial intelligence, game playing, theorem proving, and constraint programming. One of the most common variants for discrete optimization is *branch and bound*, originally suggested by Land and Doig [16]. Given a discrete set of feasible solutions and an objective function defined on that set, the discrete optimization problem is to find a feasible solution with minimum objective value. In branch and bound, branching consists of partitioning the feasible set into subsets. Processing consists of solving a relaxation (optimizing over a superset of the current feasible subset). A node can be fathomed if the relaxation solution is in the original feasible set (in which case, the best such solution seen so far is recorded as the *incumbent*). A node can also be fathomed if the objective value of the relaxation solution exceeds the value of the incumbent, as the successors of the current node cannot contain a solution with better value than the incumbent.

## 1.2 Parallelizing Tree Search

In their simplest form, divide and conquer algorithms such as tree search are conceptually easy to parallelize. More sophisticated variants of these algorithms, 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. These inefficiencies highlight the fundamental tradeoff between centralized and decentralized storage and decision-making mechanisms.

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. In the case of ALPS, we further assume that there is exactly one process per processor at all times, though this process might be multi-threaded. The goal of any parallel implementation is to limit overhead as much as possible. Without overhead, a parallel algorithm on $N$ processes would be able to run $N$ times as fast as a corresponding sequential algorithm. The main sources of parallel overhead for tree search algorithms are:

- *Communication Overhead:* time spent transferring knowledge from one process to another.
- *Handshaking:* time spent waiting for knowledge residing on another process to be transferred.
- *Redundant Work:* time spent in 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 useful work for all processes to be kept busy.

The effectiveness of the knowledge sharing mechanism is the main factor affecting overhead and the main issue to be addressed in framework design. This breakdown highlights the tradeoff between centralized and decentralized decision-making. The primary reason for the performance of redundant work is lack of global knowledge about the state of the computation, which results from decentralized storage of knowledge. Any mechanism for centralizing or sharing global knowledge, however, results in increased communication overhead and handshaking. Achieving the proper balance between these sources of overhead is the challenge we must address. For a more detailed treatment of parallel computing concepts, the reader is referred to [15].

Scalability is a measure of 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. Our goal in developing ALPS has been to develop not only a general and scalable framework, but one that is easily customizable and can be used in a wide variety of settings. We have also aimed to develop a framework specifically focused on *data-intensive* algorithms for which large amounts of data are generated and must be shared during the search. Data-intensive algorithms are particularly difficult to implement in a scalable fashion. While some work has been done in this area [8, 28], much remains. In the next section, we describe further details of ALPS.

## 2 Design of ALPS

### 2.1 Knowledge Sharing

In [25], building on ideas of Trienekens and de Bruin from [30], 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 process 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 knowledge pool may field two types of requests on behalf of the pool: (1) A new piece of knowledge to be inserted into the knowledge pool, or (2) a request for relevant pieces of knowledge from the knowledge pool, where "relevant" is defined for each category of knowledge with respect to data provided by the requesting process. A knowledge pool may also choose to "push" certain knowledge to another knowledge pool, even though no specific request has been made. In order to maintain isolate the architecture-dependent parts of the code as much as possible, we do not allow the knowledge pools do not communicate directly with each other. Instead, requests and responses are passed through a knowledge broker. The knowledge broker class contains all architecture-dependent subroutines and information about knowledge pools, such as the location of the destination KB and the communication interface.

### 2.2 Search Handling

The most fundamental knowledge generated as the search progresses is the description of the search graph itself. The node descriptions that comprise the graph are stored in knowledge pools called *node pools*. The node pools collectively contain the list of candidate nodes. The tradeoff between decentralization and centralization of knowledge is most evident in the mechanism for guiding the search and sharing node descriptions among the processors. Because node descriptions represent the work that needs to be performed, it is important that they be distributed evenly among the processors. The search strategy imposes an implicit ordering on the candidates for processing, so not all work to be done has the same priority. Thus, in assessing the distribution of work to the processes, we need to consider not only *quantity*, but also *quality*. The mechanism for distributing work is called *load balancing*.

The simplest approach to load balancing is to store all node descriptions in a single, central node pool to ensure that the work being done is always of the highest priority globally. This is known as a *master-worker* approach, in which the central node pool acts as the master and all other processes are workers. This is the approach we have taken in our previous frameworks, SYMPHONY and COIN/BCP. The approach works well for small numbers of processors, but does not scale well, as the central node pool inevitably becomes a computational bottleneck. Distributing the node pools, however, has its own drawbacks. With data-intensive applications, memory requirements can be minimized by storing the search graph using a differencing scheme in which node descriptions are not stored explicitly, but rather as differences from the predecessor. Decentralizing the storage of node descriptions potentially cripples this idea, which is the reason that many of the schemes for load balancing suggested in the literature do not work well for us. Our approach is to distribute storage of the graph in such a way that the nodes stored together locally constitute connected subtrees of the search tree. Our method for doing this is described in more detail in Section 3.3.

### 2.2.1 The Master-Hub-Worker Paradigm

To overcome the drawbacks of the master-worker approach, ALPS instead 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 processes increases, we simply add more hubs and more clusters of workers. The hubs avoid becoming overburdened by limiting the number of workers they manage. This is similar to a scheme implemented by Eckstein in his PICO framework [8]. 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 in a manner that we describe next. Another advantage of this scheme is that it allows the flexibility to utilize a simple master-slave approach by specifying a single cluster or to utilize just a single worker for sequential computation. Hence, it is appropriate for both large and small numbers of processes.

### 2.2.2 Increased Task Granularity

Another 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. We must also have a way of ensuring that the workers don't autonomously process too many low-priority nodes because of incomplete local knowledge of global priorities. In order to achieve this latter goal, the hub must periodically check its workers and report the status of its cluster to the master. The hub can then decide to ask the worker to abandon work on its current subtree and send the worker a new one. An important point, however, is that this decision is always made asynchronously.

## 3 Implementation

### 3.1 ALPS Base Classes

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 overall hierarchy of ALPS. Each block represents a C++ class, whose name is listed in the block. The lines ending with a triangle represent inheritance relationships. For example, `AlpsSolutionPool`, `AlpsSubtreePool` and `AlpsNodePool` are derived from `AlpsKnowledgePool`. The lines ending with diamonds represent associative relationships. For instance, `AlpsKnowledge` has a data member which is 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 described earlier. These classes 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, as explained below. Two examples of how this is done 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. This packed form is necessary for several reasons. First, the packed form contains only the data from the class in the form of a string. This representation typically takes much less memory than the object itself, hence it is appropriate both for storage of knowledge not currently in use and for communication of knowledge by message-passing. The packed form is also a type-independent representation, which allows ALPS to deal with knowledge types defined by the user. Finally, the packed form provides a convenient way to compare the contents of two knowledge objects to determine if they are duplicates. By hashing objects in their packed form, we can quickly identify duplication. ALPS has the following four built-in knowledge types:

- `AlpsSolution`: A description of the goal state or solution to the problem being solved. This class is virtual and must be derived by the user in order to define the contents of a valid solution.
- `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. This class is virtual and must be derived by the user in order to define these methods.
- `AlpsModel`: Contains the data to describe the original problem. This class is virtual and must be derived by the user in order to define the contexts of the model.
- `AlpsSubTree`: Contains the description of a subtree, which is a hierarchy of `AlpsTreeNode` objects, along with the methods needed for performing a tree search. All methods in this class are generic and independent of the problem being solved.

`AlpsKnowledgePool`. The role of the `AlpsKnowledgePool` is described in Section 2.1. A knowledge pool is a database containing a particular type of knowledge. Such databases store knowledge and service requests from the knowledge broker.

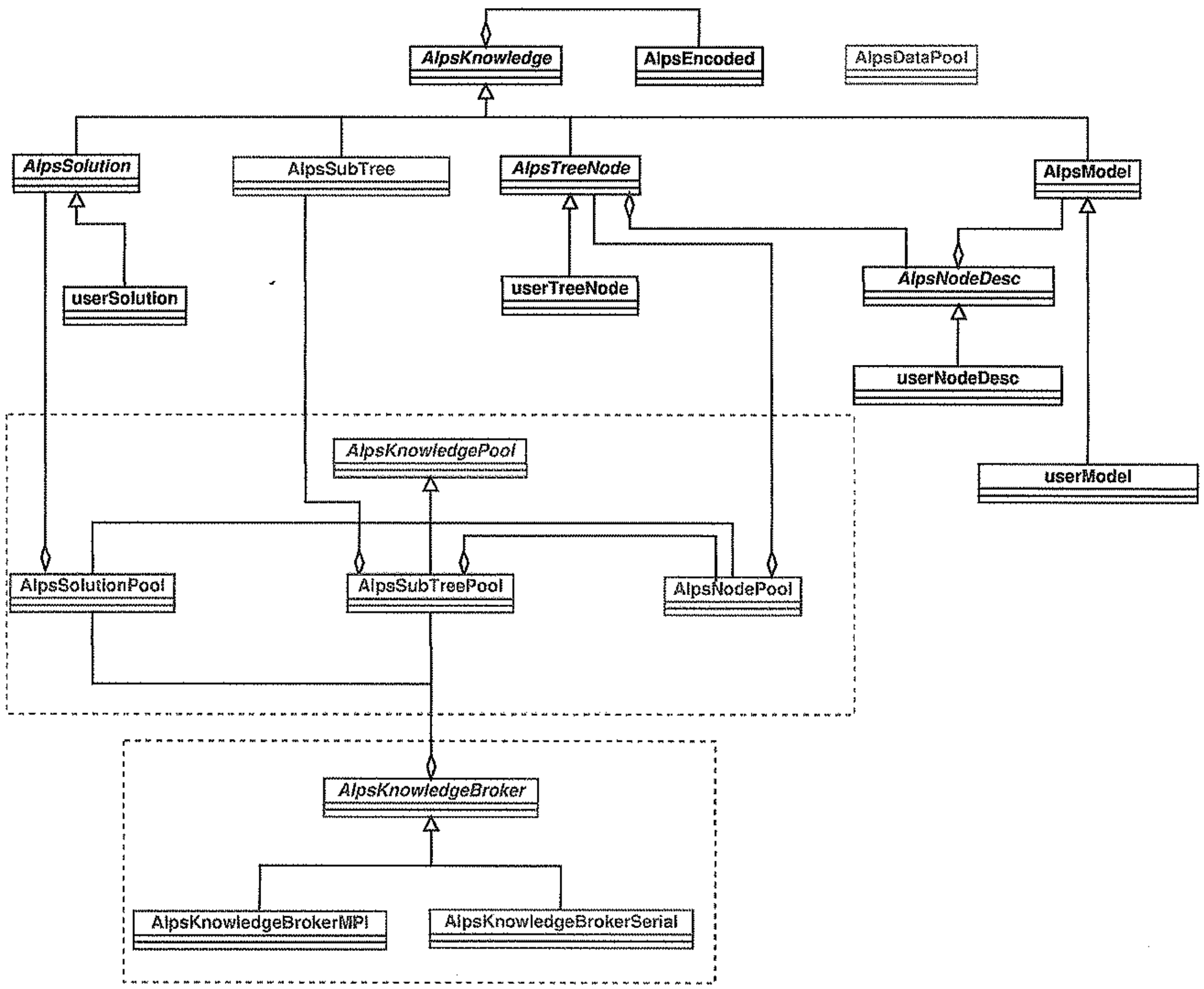

Figure 1: The Class Hierarchy of ALPS

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

**`AlpsKnowledgeBroker`.** The `AlpsKnowledgeBroker` class encapsulates the communication protocol. The knowledge broker is the driver class for each processor and is the class responsible for sending, receiving, and routing all data that resides on that processor. Each knowledge pool must be registered with the broker so that the broker knows where to route a particular type of knowledge when it arrives and where to route requests for that type of knowledge from other knowledge brokers. This is the only class that must be reimplemented in order change the communication protocol. Currently, the protocols supported are a serial layer and MPI.

- `AlpsKnowledgeBrokerMPI`: A communication layer for communication over a network using the message-passing protocol MPI.
- `AlpsKnowledgeBrokerSerial`: A shared-memory communication layer for single-process execution.

## 3.2 Process Clustering

According to the master-hub-worker scheme, ALPS groups processes into *clusters*. Each cluster has a single *hub* and one or more *workers*. By itself, each cluster functions essentially as a master-worker unit. The hub allocates work to the workers and controls the search process, while the workers follow the hub's instructions and perform the work. What makes a cluster different from a standalone master-worker system is that the hub must also function within the larger system and obey instructions from the master process during the search. Cluster size is computed based on the number of hubs and the number of processors, which are set by the user at run time.

## 3.3 Load Balancing

Effective load balancing is a central to reducing overhead. Load balancing methods have been studied extensively in recent years [2, 9, 12, 14, 17, 22, 26, 27, 29]. To balance workload in terms of both quantity and quality, ALPS employs a three-tiered load balancing scheme, consisting of the following:

- *Static load balancing*,
- *Intra-cluster dynamic load balancing*, and
- *Inter-cluster dynamic load balancing*

The search graph consists initially of a single *root node*. The first task is to generate a group of successors of this root node and distribute them to the workers to initialize their local node pools. This step is called *static load balancing* or *mapping*. ALPS uses a *two-level root initialization* scheme, a generalization of the *root initialization* principle [12]. During static load balancing, the master first creates the required number of nodes for hubs (controlled by run time parameter `masterInitNodeNum`). and then distributes them. The hubs in turn create the required number of successors for their workers (controlled by run time parameter `hubInitNodeNum`), after which the workers initialize their subtree pools and begin working.

Time spent performing static load balancing is the main source of ramp-up overhead. This overhead can be significant in cases where the node processing time is large. Two-level root initialization reduces ramp-up time 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. However, because it is difficult to predict a priori the total amount of work required to process a given node and all of its eventual successors, the load usually becomes unbalanced after some time. This necessitates dynamic load balancing, which involves dynamically reallocating the workload during the search.

Inside a cluster, the hub manages dynamic load balancing. Intra-cluster load balancing can be initiated in two ways. First, an individual worker can initiate a load balancing action by reporting 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 many high priority nodes to share their workload with the workers with fewer high priority 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. In a fashion similar to the scheme used in the PICO framework, the master checks the workloads of all clusters and asks the most loaded clusters to donate subtrees to the least loaded clusters. 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 factor of `donorThreshold`. Receivers are the clusters whose workloads are smaller than the average workload by a factor of `receiverThreshold`. Donors and receivers are paired and each donor sends a subtree to its paired receiver.

Our load balancing procedure is unique in that we take into account the tree structure when sharing nodes during the load balancing. We always choose to share a group of nodes that are the leaves of a subtree and we send the entire subtree, rather than just the nodes themselves. This approach results in smaller messages since subtrees are sent in our compact differences format and in a global savings in node storage across all processors. In order to choose a subtree to share, we assign each subtree a priority level, defined as the average priorities of a given number of its best nodes. During load balancing, the donor always chooses the best subtree in its subtree pool and sends it to the receiver. If a donor doesn't 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.

```
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

### 3.4 Task Management

Because each process hosts a knowledge broker and several knowledge pools, 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, meaning that 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 knowledge broker controls the communication to and from the process, it also controls the task scheduling. The KB receives external messages, forwards them to the appropriate local knowledge pool if needed, and forwards all locally generated messages to the appropriate remote knowledge broker. When not listening for messages, the KB schedules the execution of computational tasks by the local knowledge pools. The KB decides when and how long to process each task.

## 4 Applications and Preliminary Computational Results

### 4.1 Developing Applications

Developing application with ALPS comprises mainly of deriving the required and auxiliary problem-specific classes, and writing the `main` function. As described in Section 3.1, the user must derive algorithm-specific classes from the ALPS base classes `AlpsModel`, `AlpsTreeNode`, `AlpsNodeDesc`, and `AlpsSolution`. The user may also want to define problem-specific parameters by deriving a class from `AlpsParameterSet`. For further customization, the user may want to define new types of knowledge. A sample main function is shown in Figure 2.

Table 1: Overall Results on Four Knapsack Instances

| N | **Wallclock** | **Ramp-up** | **Idle** | *Speedup* | *Efficiency* | **nodes** |
|---|---|---|---|---|---|---|
| 1 | 22m15s | – | – | – | – | 254,151 |
| 4 | 4m56s | 0% | 2.9% | 4.5 | 1.13 | 85 m |
| 8 | 2m40s | 0% | 2.6% | 8.3 | 1.04 | 85 m |
| 16 | 1m34s | 0% | 7.8% | 14.2 | 0.89 | 85 m |
| 32 | 53s | 0% | 7.9% | 26.3 | 0.83 | 85 m |

## 4.2 Knapsack Solver

The binary knapsack problem is that of selecting from a set of items a selected subset with the maximum total profit that does not exceed a given total weight. 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. The processing procedure consists 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 as described in [19]. Note that these results are not meant to be representative—complete performance results will be reported in a follow-up paper. Testing was conducted on a Beowulf cluster with 48 dual processor 1G nodes running Red Hat Linux 7.3. Five trials were run for each instance, with two hubs employed when the number of processors was eight or more. The results in Table 1 show the number of processors used ($N$), the speedup, which is the ratio of the sequential and parallel running times, and the parallel efficiency, which is the ratio of the speedup to the number of processors. The efficiency represents the percentage of running time devoted to useful work and should be near one ideally. We used COIN/SBB [10] to produce the sequential running time for comparison. Because our solver does not employ advanced techniques such as dynamic generation of valid inequalities or primal heuristics, we disabled these capabilities with COIN/SBB as well. From Figure 3), we see that the speedup is near linear. Ramp-up time is negligible, but idle time still leaves room for improvement.

## 4.3 Solver for Generic Mixed-integer Linear Programs

For the knapsack solver, node processing times were almost negligible and good feasible solutions were discovered early in the solution process, which makes good speedup relatively easy to achieve. As a more stringent test, we have developed a generic solver for mixed-integer linear programs (MILPs), ALPS Branch and Cut (ABC), employing a straightforward branch and cut algorithm with cuts generated using the COIN-OR Cut Generation Library. ABC consists of the classes

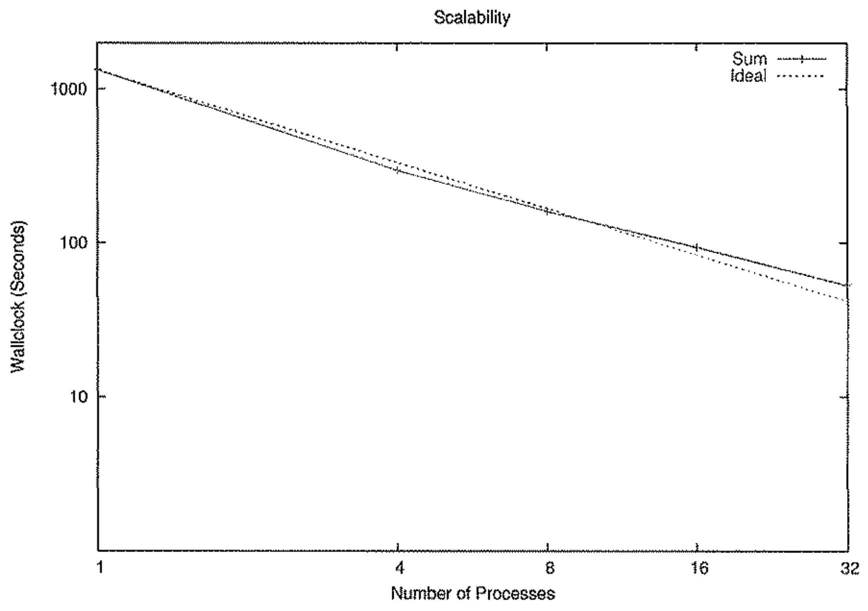

Figure 3: Wallclock Time vs Number of Processes

`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 COIN/SBB rounding heuristic as a primal heuristic.

We tested ABC using four MIPLIB [1] problems: *gesa3, blend2, fixnet6 and cap6000.* As before, these results are not meant to be representative, but just illustrative. 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. This is difficult to deal with, as our experience has shown that attempts to shorten node processing times by branching early to produce successors more quickly usually results in an increase in the size of the tree. Neither of these two case exhibits signs of the performance of redundant work, however, as the number of search nodes decreases as the number of processors increases. This is primarily due to the fact that good feasible solutions are found early in the search process. For *fixnet6* and *cap6000*, the 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. In these cases, good feasible solutions are not found until much later in the search process. These results the challenges that we still face in improving scalability. We discuss prospects for the future in the final section.

Table 2: Computational Results of Sample MILP problems

| **Problem** | **p** | **Wallclock** | **Ramp-up** | **Idle** | *Speedup* | *Efficiency* | **nodes** |
|---|---|---|---|---|---|---|---|
| gesa3 | 1 | 27m6s | – | – | – | – | 403 |
| gesa3 | 4 | 10m14s | 9.8% | 0 | 2.6 | 0.66 | 445 |
| gesa3 | 8 | 4m29s | 35.1% | 0.2% | 6.0 | 0.76 | 337 |
| gesa3 | 16 | 2m41s | 49.1% | 0.1% | 10.1 | 0.63 | 247 |
| blend2 | 1 | 26m5s | – | – | – | – | 2339 |
| blend2 | 4 | 4m18s | 12.8% | 0 | 6.1 | 1.53 | 1019 |
| blend2 | 8 | 3m33s | 14.0% | 0.2% | 7.3 | 0.92 | 717 |
| blend2 | 16 | 2m9s | 34.1% | 0 | 12.1 | 0.76 | 980 |
| fixnet6 | 1 | 45m16s | – | – | – | – | 2729 |
| fixnet6 | 4 | 11m43s | 1.0% | 0 | 3.9 | 0.98 | 3598 |
| fixnet6 | 8 | 10m26s | 3.0% | 0.2% | 4.3 | 0.54 | 4703 |
| fixnet6 | 16 | 6m16s | 4.6% | 0 | 7.2 | 0.45 | 6570 |
| cap6000 | 1 | 71m27s | – | – | – | – | 6129 |
| cap6000 | 4 | 22m24s | 0.2% | 0 | 3.2 | 0.80 | 9551 |
| cap6000 | 8 | 16m52s | 0.3% | 0 | 4.2 | 0.53 | 12363 |
| cap6000 | 16 | 10m40 | 1.2% | 0.2% | 6.7 | 0.42 | 14121 |

## 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 the goals of the project. The preliminary results obtained for ABC highlight very well the two scalability issues we still need to address for MILP—reduction of ramp-up time and elimination of redundant work. Controlling ramp-up time is tricky and we have no ready solutions for this problem. Attempts to branch early in order to produce successors more quickly have failed. Two ideas to explore 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. 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 ALPS performance by refining our methods of reducing parallel overhead as discussed above. Also, we will continue development of the 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 support the implementation of parallel branch and bound algorithms in which the bounds are obtained by some sort of Lagrangian relaxation. Finally, we will built the BiCePS Linear Integer Solver (BLIS) on top of BiCePS. BLIS will be a MILP solver like ABC, but with user customization features akin to SYMPHONY and COIN/BCP. All software discussed in this paper will be maintained in the COIN-OR repository.

## References

[1] MIPLIB 3. `htpp://www.caam.rice.edu/ bixby/miplib/miplib3.html`.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume Proceedings ICM III (1998):645–656, 1998.

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

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

[5] Q. Chen and M. Ferris. Fatcop: A fault tolerant condor-pvm mixed integer programming solver. *SIAM Journal on Optimization*, 11(4):1019–1036, 2001.

[6] Q. Chen, M. Ferris, and J. T. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 103:17–32, 2001.

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

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

[9] C. Fonlupt, P. Marquet, and J. Dekeyser. Data-parallel load balancing strategies. *Parallel Computing*, 24(11):1665–1684, 1998.

[10] Computational Infrastructure for Operations Research. `http://www.coin-or.org`.

[11] L. Hafer. bonsaig: Algorithms and design. Technical Report SFU-CMPTTR 1999-06, Simon Frazer University Computer Science, 1999.

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

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

[14] V. Kumar, A. Y. Grama, and Nageshwara Rao Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

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

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

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

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

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

[20] A. Martin. Integer programs with block structure. Habilitation Thesis, Technical University of Berlin, Berlin, Germany, 1998.

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

[22] A. Osman and H. Ammar. Dynamic load balancing strategies for parallel computers.

[23] T. K. Ralphs and L. Ladányi. *SYMPHONY Version 4.0 User's Manual*, 2004. `http://www.brandandcut.org`.

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

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

[26] P. Sanders. Analysis of random polling dynamic load balancing. Technical Report iratr-1994-12, 1994.

[27] P. Sanders. Tree shaped computations as a model for parallel applications, 1998.

[28] Y. Shinano, K. Harada, and R. Hirabayashi. 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, 1995. IEEE Computer Society Press.

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

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

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