Computational Experience with a Software Framework for Parallel Integer Programming

Yan Xu

Operations Research R & D, SAS Institute, Cary, NC 27513, US

Ted K. Ralphs

Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, USA

Laszlo Ladányi

Department of Mathematical Sciences, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA

Matthew J. Saltzman

Department of Mathematical Sciences, Clemson University, Clemson, SC 29634, USA

COR@L Technical Report 09T-006-R2

Computational Experience with a Software Framework for Parallel Integer Programming

YAN XU*[1], TED K. RALPHS†[2], LASZLO LADÁNYI‡[3], AND MATTHEW J. SALTZMAN§[4]

[1]Operations Research R & D, SAS Institute, Cary, NC 27513, US
[2]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, USA
[3]Department of Mathematical Sciences, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
[4]Department of Mathematical Sciences, Clemson University, Clemson, SC 29634, USA

Original Publication: September 26, 2007
Last Revised: May 22, 2009

Abstract

In this paper, we discuss the challenges that arise in parallelizing algorithms for solving generic mixed integer linear programs and introduce a software framework that aims to address these challenges. Although the framework makes few algorithmic assumptions, it was designed specifically with support for implementation of relaxation-based branch-and-bound algorithms in mind. Achieving efficiency for such algorithms is particularly challenging and involves a careful analysis of the tradeoffs inherent in the mechanisms for sharing the large amounts of information that can be generated. We present computational results that illustrate the degree to which various sources of parallel overhead affect scalability and discuss why properties of the problem class itself can have a substantial effect on the efficiency of a particular methodology.

1 Introduction

Recently, trends in the design of microprocessors have undergone dramatic change. The exponential increases in clock speed that marked steady improvements in computer performance over the past several decades appear to have reached a plateau. It now seems inevitable that future improvements will be in the form of increases in the number of processing *cores*—individual processors mounted on a single integrated circuit that can perform independent calculations in parallel but share access to part or all of the memory hierarchy. This has led to a sudden realization that parallelizing

*`Yan.Xu@sas.com`
†`ted@lehigh.edu`
‡`ladanyi@us.ibm.com`
§`mjs@clemson.edu`

algorithms will be the only available avenue for taking advantage of improvements in processor technology for the foreseeable future. This recent change has resulted in an urgent focus on the design and analysis of parallel algorithms and has quickly moved research on such algorithms from its previous position as a scientific novelty to the mainstream.

This paper concerns a framework for parallelization of general tree search algorithms known as the COIN-OR High Performance Parallel Search (CHiPPS) framework. In Ralphs et al. (2003, 2004), we introduced the framework and examined the issues surrounding parallelization of general tree search algorithms. Here, we focus specifically on the performance of the framework's implementation of the branch-and-cut algorithm for solving general mixed integer linear programs (MILPs). The branch-and-cut algorithm is currently the most effective and commonly used approach for solving MILPs, but many MILPs arising in practice remain difficult to solve even with this advanced methodology. As with many computing tasks, the difficulty stems from limitations in both memory and processing power, so a natural approach to overcoming this is to consider the use of so-called "high-throughput" computing platforms, which can deliver a pooled supply of computing power and memory that is virtually limitless.

Branch-and-bound algorithms are a general class employing a divide-and-conquer strategy that partitions the original solution space into a number of smaller subsets and then analyzes the resulting smaller subproblems in a recursive fashion. Such an approach appears easy to parallelize, but this appearance is deceiving. Although it is easy in principle to divide the original solution space into subsets, it is difficult to find a division such that the amount of effort required to solve each of the resulting smaller subproblems is approximately equal. If the work is not divided equally, then some processors will become idle long before the solution process has been completed, resulting in reduced effectiveness. Even if this challenge is overcome, it might still be the case that the total amount of work involved in solving the subproblems far exceeds the amount of work required to solve the original problem on a single processor.

In the remainder of the paper, we describe our approach to meeting the challenges that arise in implementing parallel versions of sophisticated branch-and-bound algorithms, such as branch and cut. The work builds and attempts to improve upon previous efforts, most notably the frameworks SYMPHONY (Ralphs and Güzelsoy, 2008) and COIN/BCP (Ralphs and Ladányi, 2001), both straightforward implementations employing centralized mechanisms for control of task distribution and information sharing. The framework also builds on concepts introduced in the framework now known as PEBBL (Eckstein et al., 2007). The paper is organized as follows. In Section 1.1, we introduce necessary terms and definitions before reviewing previous work in Section 1.2. In Section 2, we briefly describe the implementation details of CHiPPS, a C++ class library developed based on the concepts of knowledge discovery and sharing. In Section 3, we analyze computational results obtained using the framework to solve MILP instances with a wide range of scalability properties. Finally, in Section 4, we conclude and summarize the material presented in the rest of the paper.

1.1 Definitions and Terms

Tree Search and MILP. *Tree search problems* are any of a general class in which the nodes of a directed, acyclic graph must be systematically searched in order to locate one or more *goal nodes*. The order in which the nodes are searched uniquely determines a rooted tree called the *search tree*. Tree search problems can be generally classified as either *feasibility problems* or *optimization problems*. Optimization problems are those in which there is a cost associated with each path from the root node to a goal node and the object is to find a goal node with minimum path cost. The basic components of a tree search algorithm are (1) a *processing method* that determines whether a node is a goal node and whether it has any successors, (2) a *branching method* that specifies how to generate the descriptions of a node's successors, if it has any, and (3) a *search strategy* that specifies the processing order of the candidate nodes.

A *mixed integer linear program* is a problem of the form

$$z_{IP} = \min_{x \in \mathcal{P} \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p})} c^\top x, \tag{1}$$

where $\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ is a polyhedron defined by constraint matrix $A \in \mathbb{Q}^{m \times n}$, right-hand side vector $b \in \mathbb{Q}^m$, and objective function vector $c \in \mathbb{R}^n$. As first noted by Dakin (1965), a MILP can be viewed as a tree search problem in a straightforward way. The well-known branch-and-cut algorithm recursively partitions the feasible region, a process that can be visualized by associating the subsets created with the nodes of a corresponding tree. In this way, the branch-and-cut algorithm can be fit very cleanly into the abstract framework presented in the previous paragraph. This is the basis for the implementation of the MILP solver described in Section 2.3.

Parallel Computing. We consider parallel algorithms targeted for deployment in a modern distributed computing environment. The most typical architecture is the so-called *Beowulf cluster* (Brown, 2004). The important properties of such an architecture are (1) that it is *homogeneous* (all nodes are identical), (2) that it has an "all-to-all" communications network in which any two nodes can communicate directly, (3) that it has no shared access memory, and (4) that it has no global clock by which to synchronize calculations. On such an architecture, all communication and synchronization activities must take place through the explicit passing of "messages" between processors using a *message-passing protocol.* The two most common message-passing protocols are PVM (Geist et al., 1994) and MPI (Gropp et al., 1999).

"High-throughput" computing environments are similar to clusters with respect to most of the above properties. However, processors need not be homogeneous with respect to speed, memory, operating system, or even CPU architecture, though the ability to deploy the program to be run on all nodes (with support for uniform communication protocols) is obviously a prerequisite. Also, the communication network need not be fast, although at least logical direct point-to-point communication is still necessary. In principle, these properties do not preclude deploying distributed programs across such an environment, but algorithmic parameters for such things as task granularity and load balancing strategy would probably be significantly different.

The primary goal of parallel computing is to take advantage of increased processing power in order to either increase the amount of progress that can be made on a given calculation in a fixed amount of time or to perform a fixed calculation faster. The *scalability* of a parallel algorithm (in combination with a specific parallel architecture) is the degree to which it is capable of effectively utilizing increased computing resources (usually processors). The traditional way to measure scalability is to compare the speed with which one can execute a particular algorithm on p processors to the speed of the same algorithm on a smaller number of processors (usually a single processor, though this comparison is not always possible in practice). The aim is for the ratio of the running times with different numbers of processors (called the *speedup*) to be as close as possible to the ratios of the numbers of processors. In other words, if we increase the number of processors by a factor of q, we would ideally like to observe a decrease in running time by the same factor, i.e., from r to r/q.

For a number of reasons, this traditional analysis is not especially well-suited to the assessment of parallel algorithms for solving optimization problems, but we defer discussion of this until Section 3. For now, let us simply say that the ideal speedup is not typically observed for two main reasons. First, the total amount of work to be done during parallel execution typically increases as the number of processors increases. Second, processors may spend some of their time idle when they might be doing useful work. Losses of efficiency stemming from either one of these two main sources are generally termed *parallel overhead* and the goal of parallelization is to reduce such overhead. In analyzing the computational experiments later in the paper, it will be useful to identify the following specific sources of overhead that we consider.

- *Idle time (ramp-up/ramp-down)*: Time spent by processors waiting for their first task to be allocated or waiting for termination at the end of the algorithm.
- *Idle time (synchronization/handshaking)*: Time spent waiting for information requested from another processor or waiting for another processor to complete a task.
- *Communication overhead*: Processing time associated with sending and receiving information, including time spent inserting information into the send buffer and reading it from the receive buffer at the other end.
- *Performance of redundant work*: Processing time spent performing work (other than communication overhead) that would not have been performed in the sequential algorithm.

In Section 3, we discuss further how these various contributors to overhead affect the efficiency of tree search.

Knowledge. We refer to potentially useful information generated during the execution of the algorithm as *knowledge*. Trienekens and de Bruin (1992) formally introduced the notion that the efficiency of a parallel algorithm is inherently dependent on the strategy by which this knowledge is stored and shared among the processors. From this viewpoint, a parallel algorithm can be thought of roughly as a mechanism for coordinating a set of autonomous agents that are either *knowledge generators* (KGs) (responsible for producing new knowledge), *knowledge pools* (KPs) (responsible for storing previously generated knowledge), or both. Specifying such a coordination mechanism consists of specifying what knowledge is to be generated, how it is to be generated, and what is to be done with it after it is generated (stored, shared, discarded, or used for subsequent local computations).

Tasks. Just as KPs can be divided by the type of knowledge they store, KGs may be divided either by the type of knowledge they generate or the method by which they generate it. In other words, processors may be assigned to perform only a particular task or set of tasks. If a single processor is assigned to perform multiple tasks simultaneously, a prioritization and time-sharing mechanism must be implemented to manage the computational efforts of the processor. The *granularity* of an algorithm is the size of the smallest task that can be assigned to a processor. Choosing the proper granularity can be important to efficiency. Too fine a granularity can lead to excessive communication overhead, while too coarse a granularity can lead to excessive idle time and performance of redundant work. We have assumed an asynchronous architecture, in which each processor is responsible for autonomously orchestrating local algorithm execution by prioritizing local computational tasks, managing locally generated knowledge, and communicating with other processors. Because each processor is autonomous, care must be taken to design the entire system so that *deadlocks*, in which a set of processors are all mutually waiting on one another for information, do not occur.

1.2 Previous Work

The branch-and-bound algorithm was first suggested by Land and Doig (1960). Dakin (1965) later described it as a tree search algorithm and Mitten (1970) abstracted the approach into the theoretical framework we are familiar with today. There are a number of software packages that implement parallel branch-and-bound or parallel branch-and-cut algorithms. These can be divided broadly into two categories: black-box solvers and frameworks. Black-box solvers are primarily geared towards out-of-the-box solution of unstructured MILP instances (though they may have hooks for limited forms of customization). Frameworks, on the other hand, allow the user full access to and control over most aspects of the algorithm's implementation in order to implement specialized solvers tuned to take advantage of special structure using customized functionality, such as problem-specific cut generation, branching, or primal heuristic methods, among others. Commercial parallel black-box solvers include ILOG's CPLEX, Dash's XPRESS, and the Gurobi MILP solver, all of which are designed for shared memory architectures. PARINO (Linderoth, 1998), FATCOP (Chen and Ferris, 2001), and PICO (Eckstein et al., 2007) (part of what is now called PEBBL, but used to be called PICO) are noncommercial parallel, black-box MILP solvers, both of which are targeted for distributed environments. ParaLEX (Shinano and Fujie, 2007) is a parallel extension for the CPLEX MILP solver, also specialized for a distributed computing environment.

Among frameworks, SYMPHONY (Ralphs and Güzelsoy, 2008) and CBC (Forrest, 2003) are open source and aimed specifically at implementation of parallel branch and cut. CBC's parallel mode is only for shared-memory architectures. SYMPHONY has implementations for both shared and distributed environments. COIN/BCP (Ralphs and Ladányi, 2001) is an open source framework for the implementation of parallel branch, cut, and price algorithms targeted to distributed memory applications. There are also also a number noncommercial generic frameworks for implementing parallel branch-and-bound algorithms, such as PUBB (Shinano et al., 1995), BoB (Benchouche et al., 1996), PPBB-Lib (Tschoke and Polzer, 2008), PEBBL (Eckstein et al., 2007) (formerly PICO), and OOBB (Gendron et al., 2005).

2 The COIN-OR High Performance Parallel Search Framework

The *COIN-OR High Performance Parallel Search* framework (CHiPPS) is a C++ class library hierarchy for implementing customized versions of parallel tree search. CHiPPS currently consists of three class libraries that combine to provide the functionality required for executing a branch-and-cut algorithm. Below, we describe each of these three libraries.

2.1 ALPS

The central core of CHiPPS is a library of base classes known as the *Abstract Library for Parallel Search* (ALPS). Because of its general approach, ALPS supports implementation of a wide variety of algorithms and applications through creation of application-specific derived classes implementing the algorithmic components required to specify a tree search.

2.1.1 Knowledge Management

Knowledge Objects. ALPS takes an almost completely decentralized approach to knowledge management. The fundamental concepts of knowledge discovery and sharing are the basis for the class structure of ALPS, the base library, shown in Figure 1 (along with the class structure for BiCePS and BLIS, to be described in later sections). In this figure, the boxes represent classes, an arrow-tipped line connecting classes represents an inheritance relationship, and a diamond-tipped line connecting classes indicates an inclusion relationship. A central notion in ALPS is that all information generated during execution of the search is treated as knowledge and is represented by objects of C++ classes derived from the common base class `AlpsKnowledge`. This class is the virtual base class for any type of information that must be shared or stored. `AlpsEncoded` is an associated class that contains the encoded or packed form of an `AlpsKnowledge` object. The packed form consists of a bit array containing the data needed to construct the object. This representation takes less memory than the object itself and is appropriate both for storage and transmission of the object. The packed form is also independent of type, which allows ALPS to deal effectively with user-defined or application-specific knowledge types. To avoid the assignment of global indices, ALPS uses hashing of the packed form as an effective and efficient way to identify duplicate objects.

ALPS has the following four native knowledge types, which are represented using classes derived from `AlpsKnowledge`:

- `AlpsModel`: Contains the data describing the original problem. This data is shared globally during initialization and maintained locally at each processor during the course of the algorithm.
- `AlpsSolution`: Contains the description of a goal state or solution discovered during the search process.
- `AlpsTreeNode`: Contains the data associated with a search tree node, including an associated object of type `AlpsNodeDesc` that contains application-specific descriptive data and internal data maintained by ALPS itself.
- `AlpsSubTree`: Contains the description of a subtree, which is a hierarchy of `AlpsTreeNode` objects along with associated internal data needed for bookkeeping.

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.

Knowledge Pools. The `AlpsKnowledgePool` class is the virtual base class for knowledge pools in ALPS. This base class can be derived to define a KP for a specific type of knowledge or multiple types. The native KP types are:

- `AlpsSolutionPool`: Stores `AlpsSolution` objects. These pools exist both at the local level—for storing solutions discovered locally—and at the global level.

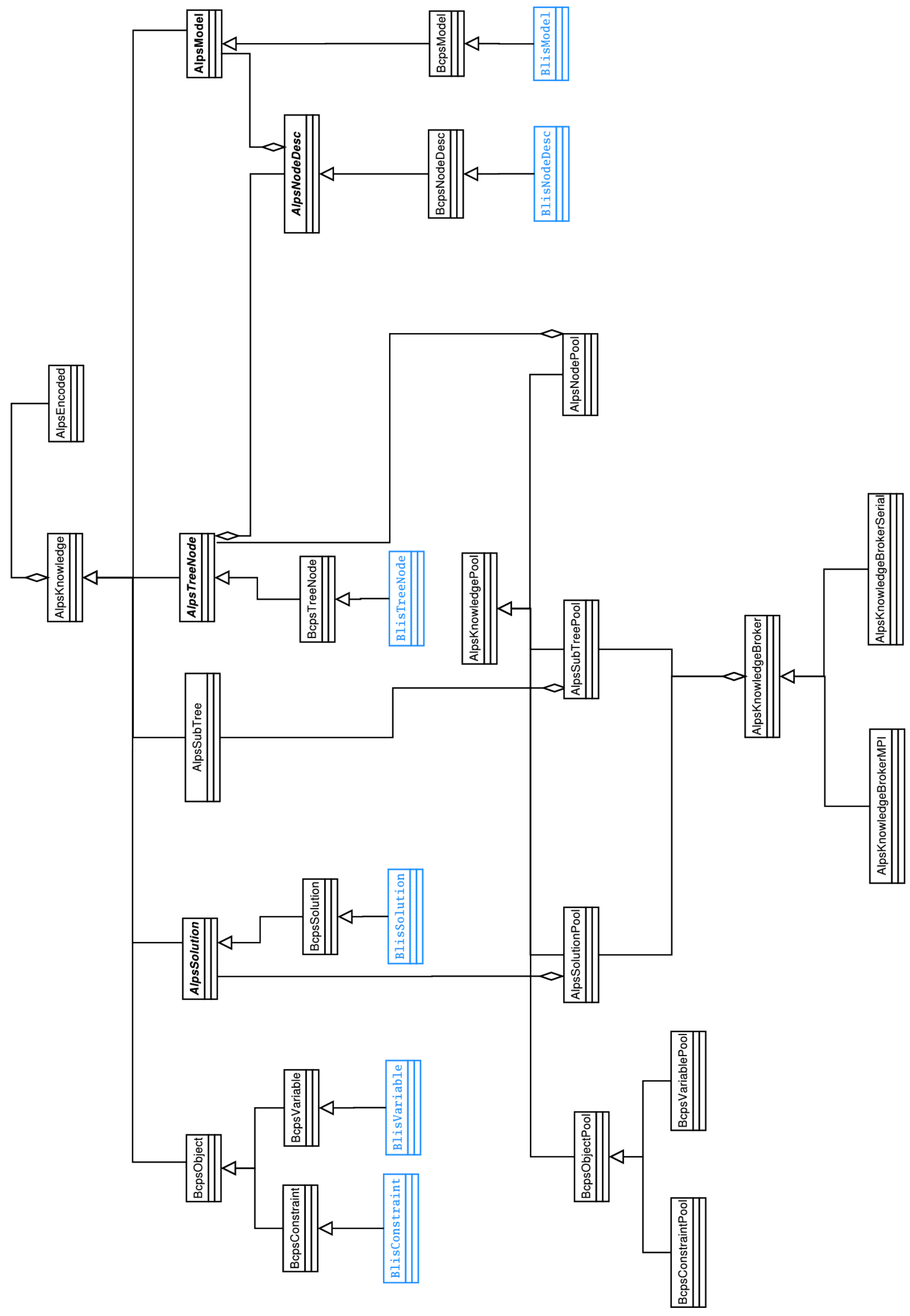

Figure 1: Library hierarchy of CHiPPS

.

- `AlpsSubTreePool`: Stores `AlpsSubTree` objects. These pools are for storing various disjoint subtrees of the search tree that still contain unprocessed nodes in a distributed fashion (see the sections below for more information on how ALPS manages subtrees).
- `AlpsNodePool`: Stores `AlpsTreeNode` objects. Each subtree has a node pool that contains the queue of candidate nodes associated with this subtree.

None of these classes are virtual and their methods are implemented independent of any specific application.

Knowledge Brokers. In ALPS, each processor hosts a single, multitasking executable controlled by a *knowledge broker* (KB). The KB is tasked with routing all knowledge to and from the processor and determining the priority of each task assigned to the processor (see next section). Each specific type of knowledge is represented by a C++ class derived from `AlpsKnowledge` and must be registered at the inception of the algorithm so that the KB knows how to manage it. The KB associated with a particular 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. Derivatives of the `AlpsKnowledgeBroker` class implement the KB and encapsulate the desired communication protocol. Switching from a parallel implementation to a sequential one is simply a matter of constructing a different KB object. Currently, the protocols supported are a serial layer, implemented in `AlpsKnowledgeBrokerSerial` and an MPI layer, implemented in `AlpsKnowledgeBrokerMPI`. In the future, we plan to support shared-memory/multi-core parallel architectures, as well as hybrid shared/distributed architectures that will soon be commonplace.

2.1.2 Task Management

Due to the asynchronous nature of the platforms on which these algorithms are deployed, each single executable must be capable of performing multiple tasks simultaneously. Tasks to be performed may involve management of knowledge pools, processing of candidate nodes, or generation of application-specific knowledge, e.g., valid inequalities. The ability to multi-task is implemented using a crude (but portable) threading mechanism controlled by the KB running on each processor, similar to that used in PEBBL (Eckstein et al., 2000). The KB decides how to allocate effort and prioritize tasks. The principle computational task in ALPS is to process the nodes of a subtree with a given root node until a termination criterion is satisfied, which means that each worker is capable of processing an entire subtree autonomously and has access to all of the methods needed to manage a sequential tree search. The task granularity can therefore be fine-tuned by changing this termination criterion, which, in effect, determines what ALPS considers to be an indivisible "unit of work." The termination criterion can be based on time, number of nodes processed, or any other desired measure. ALPS contains a mechanism for automatically tuning the granularity of the algorithm based on properties of the instance at hand (primarily the average time required to process a node). Fine tuning the task granularity reduces idle time due to task starvation. However, without proper load balancing, increased granularity may increase the performance of redundant work.

Master-Hub-Worker Paradigm. In a typical master-worker approach to parallelism, a single master process maintains a global view of available tasks and controls task distribution to the entire process set directly. This approach is simple and effective for small number of processors, but does not scale well because of the obvious communication bottleneck created by the single master process. To overcome this restriction, 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* and a fixed number of *workers*. The hub manages the workers and supervises load balancing within the cluster, while the master ensures the load is balanced globally. As the number of processors is increased, clusters can be added in order to keep the load of the hubs and workers relatively constant. The workload of the master process can be managed by controlling the frequency of global balancing operations. This scheme is similar to one implemented in the PEBBL framework (Eckstein et al., 2000). The decentralized approach maintains many of the advantages of global decision making while reducing overhead and moving much of the burden for load balancing and search management from the master to the hubs. The burden on the hubs can then be controlled by adjusting the task granularity of the workers.

2.1.3 Load Balancing

The tradeoff between centralization and decentralization of knowledge is most evident in the mechanism for sharing node descriptions among the processors. Because all processes are completely autonomous in ALPS, idle time due to task starvation and during the *ramp-up* and *ramp-down* phases of the algorithm (defined below) are the most significant scalability problems. To combat these issues, ALPS employs a three-tiered load balancing scheme, consisting of *static*, *intra-cluster dynamic*, and *inter-cluster dynamic* load balancing.

Static Load Balancing. Static load balancing, or *mapping*, takes place during the ramp-up phase, which is roughly defined as the time until every KB has been assigned at least one task. The goal is to quickly generate the initial pool of candidate nodes and distribute them to the workers to initialize their local subtree pools. The problem of reducing idle time during the ramp-up phase of branch-and-bound algorithms has long been recognized as a challenging one (Borbeau et al., 2000; Eckstein et al., 2000; Gendron and Crainic, 1994). When the processing time of a single node is large relative to the overall solution time, idle time during the ramp-up phase can be one of the biggest contributors to parallel overhead.

ALPS has two static load balancing schemes: *two-level root initialization* and *spiral initialization*. Based on the properties of the problem to be solved, the user can choose either method. Two-level root initialization is a generalization of the *root initialization* scheme of Henrich (1993). During two-level root initialization, the master first creates and distributes a user-specified number of nodes for each of the hubs. After receiving their allocation, the hubs then create a user-specified number of successors for each of their workers. Spiral initialization is designed for circumstances in which processing a node takes a substantial amount of time or creating a large number of nodes during ramp-up is otherwise difficult. The main difference is that nodes are shared by the master and workers immediately as they are created in order to maximize parallel utilization and ensure all workers are employed as quickly as possible. By sharing nodes immediately, however, the initial distribution of nodes may not actually be as well-balanced at the end of the ramp-up phase as it would be with two-level root initialization, requiring more dynamic load balancing during early phases of the algorithm. It is therefore only recommended when two-level root initialization would result in an exorbitantly long ramp-up phase.

In addition to accelerating the generation of nodes, it is possible to keep processors busy during ramp-up in other ways. For example, processors could be employed by running primal heuristics, performing preprocessing, executing a more intensive version of strong branching in parallel, or even processing multiple trees in parallel. CHiPPS does not yet implement these techniques.

Dynamic Load Balancing. Dynamic load balancing is perhaps the most challenging aspect of implementing parallel tree search and has been studied by a number of authors (Fonlupt et al., 1998; Henrich, 1993; Kumar et al., 1994; Laursen, 1994; Sanders, 1998; Sinha and Kalé, 1993). In ALPS, the hub manages dynamic load balancing within each cluster by periodically receiving reports of each worker's workload. Each node has an associated priority that indicates the node's relative "quality," which is a rough indicator of the likelihood that the node or one of its successors is a goal node. For example, the quality measure typically used in relaxation-based branch-and-bound for optimization problems is value of the bound computed at a given node. In assessing the distribution of work to the processors, we consider both *quantity* and *quality*. If it is found that the qualities are unbalanced (i.e., high-priority nodes are not evenly distributed among workers), the hub asks workers with a surplus of high-priority nodes to share them with workers that have fewer such nodes. Intra-cluster load balancing can also be 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 nodes to the requesting worker. If the hub finds that none of its workers have any work, it asks for more from the master.

The master is responsible for balancing the workload among the clusters, whose managing hubs periodically report workload information to the master. Hence, the master has an approximate 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 nodes to its paired receiver.

The dynamic load balancing scheme in ALPS has one rather unique aspect. In order to allow efficient storage and transmission of groups of search tree nodes, ALPS tries to ensure that the sets of search tree nodes passed between pairs of processors during balancing constitute the leaves of a subtree of the original tree, and are shared as single units, rather than individual nodes. Each candidate 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 the best subtree is too large to be packed into the message buffer (whose size can be controlled by the user), the donor splits it and sends a part of it to the receiver (Xu, 2007). If a donor does not have any subtrees to share, 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 even without centralized storage of the search tree.

2.2 BiCePS

The *Branch, Constrain, and Price Software* (BiCePS) library is the data-handling layer needed in addition to ALPS to support relaxation-based branch-and-bound algorithms. The class names in Figure 1 with prefix `Bcps` are the main classes in BiCePS. BiCePs consists primarily of classes derived from the base classes in ALPS and implements the basic framework of a so-called *Branch, Constrain, and Price* algorithm. In such an algorithm, the processing step consists of solving a relaxation in order to produce a bound. Node descriptions may be quite large and an efficient storage scheme is required to avoid memory issues and increased communication overhead. For this reason, we have developed compact data structures based on the ideas of *modeling objects* and *differencing*. Note that there is no assumption in BiCePS about the structure of the instance to be solved, making it possible to use it as a base for implementing solvers for a wide range of problem classes, including linear, non-linear, and discrete optimization problems.

2.2.1 Knowledge Management

Knowledge Types. In addition to classes `BcpsSolution`, `BcpsTreeNode`, `BcpsNodeDesc`, and `BcpsModel` derived from the corresponding native knowledge types in ALPS, BiCePS introduces a new knowledge type known as a *modeling object* that supports sharing of information about mathematical structure, in the form of new variables and constraints, and enables creation of efficient data structures for describing search tree nodes. A modeling object in BiCePS is an object of the class `BcpsObject` that can be either a *variable* (object of derived class `BcpsVariable`) or a *constraint* (object of derived class `BcpsConstraint`). All such objects, regardless of whether they are variables or constraints, share the basic property of having a current value and a feasible domain. This allows mechanisms for handling of modeling objects, regardless of subtype, to be substantially uniform. The modeling objects are the building blocks used to construct the relaxations required for the bounding operation undertaken during processing of a tree node. Hence, a node description in BiCePS consists of arrays of the variables and constraints that are active in the current relaxation.

Differencing. In many applications, the number of objects describing a tree node can be quite large. However, the set of objects generally does not change much from a parent node to its child nodes. We can therefore store the description of an entire subtree very compactly using the differencing scheme described below. The differencing scheme we use stores only the difference between the descriptions of a child node and its parent. In other words, given a node, we store a *relative* description that has the newly added objects, removed objects and changed objects compared with its parent. The differencing scheme always stores an *explicit* description consisting of all the objects active at the root.

To store a relative description for a node, the differencing scheme identifies the relative difference of this node with its parent. The data structures used to record differences are `BcpsFieldListMod` and `BcpsObjectListMod`. The class `BcpsFieldListMod` stores the modifications in single field, such as the lower bounds of the variables, for example. The class `BcpsObjectListMod`, on the other hand, stores modifications of the overall lists of variables and constraints active in the node. A flag can be set to indicate that explicit storage of the node is more efficient than storing the difference data.

Note that our differencing scheme for storing the search tree also means that we have to spend time simply *recovering* the explicit descriptions of tree nodes when processing them or sharing them among processors. This is done by working down the tree from the first ancestor with an explicit description, applying the differences until an explicit description of the node is obtained. This is the trade-off between saving space and increasing search time. To prevent too much time being spent in recovering node descriptions, the differencing scheme has the option to force storage of an explicit description for all nodes at a certain depth below the last explicitly stored description. We have found that a well-managed differencing policy can have a significant impact on performance—the reduced communication overhead and smaller memory footprint more than compensate for the computation required to reconstruct the instance at the current node.

2.2.2 Task Management

BiCePS assumes that the algorithm employs an iterative bounding scheme. During each iteration, new objects may be generated and used to improve the fidelity of the current relaxation. Because the number of objects generated in each iteration may be quite large, newly generated objects may first be placed in a local pool that acts as a cache of potential candidates for addition. Duplicate and ineffective objects previously added to the relaxation are generally removed aggressively in order to control the size of the relaxation. Those that prove to be particularly effective may be added to a knowledge pool for sharing with other KBs.

2.3 BLIS

The *BiCePS Linear Integer Solver* (BLIS) library provides the functionality needed to solve mixed integer linear programs and consists mainly of classes derived from BiCePS with a few additions. The class names in Figure 1 with prefix `Blis` belong to the BLIS library.

2.3.1 Knowledge Management.

Native knowledge types in BLIS are represented by classes `BlisConstraint`, `BlisVariable`, `BlisSolution`, `BlisTreeNode`, `BlisNodeDesc`, and `BlisModel`, all derived from the classes in BiCePS. Other classes include those for specifying algorithms for constraint generation, primal heuristics, and branching strategies. The main types of knowledge shared by BLIS (in addition to the types discussed above) are *bounds*, *branching information*, and *modeling objects* (variables and constraints).

The bounds that must be shared in branch and cut consist of the single global upper bound and the lower bounds associated with the subproblems that are candidates for processing. Knowledge of lower bounds is important mainly for load balancing purposes and is shared according to the scheme described in Section 2.1.3. Distribution of information regarding upper bounds is mainly important for avoiding performance of redundant work, i.e., processing of nodes whose lower bound is above the optimal value. This information is broadcast immediately in an efficient manner described by Xu (2007).

If a backward-looking branching method, such as one based on pseudo-costs, is used, then the sharing of historical information regarding the effect of branching can be important to the implementation of the branching scheme. The information that needs to be shared and how it is shared depends on the specific scheme used. For an in-depth treatment of the issues surrounding pseudo-cost sharing in parallel, see Eckstein et al. (2000) and Linderoth (1998).

2.3.2 Task Management

In branch and cut, there are a number of distinct tasks to be performed and these tasks can be assigned to processors in a number of ways. The main tasks to be performed are *node processing*, *node partitioning* (branching), *cut generation*, *pool maintenance*, and *load balancing*. The way in which these tasks are grouped and assigned to processors determines, to a large extent, the parallelization scheme of the algorithm and its scalability. In the current implementation of BLIS, node processing, cut generation, and node partitioning are all handled locally during processing of each subtree. Load balancing is handled as previously described.

Branching. The branching scheme of BLIS is similar to that of the COIN-OR Branch-and-Cut framework (CBC) (Forrest, 2003) and comprises three components:

- *Branching Objects*: BiCePS objects that can be branched on, such as integer variables and special ordered sets.
- *Candidate Branching Objects*: Objects that do not lie in the feasible region or objects that will be beneficial to the search if branching on them.
- *Branching Method*: A method to compare candidate objects and choose the best one.

The branching method is a core component of BLIS's branching scheme. When expanding a node, the branching method first identifies a set of initial candidates and then selects the *best* object based on the rules it specifies. The branching methods provided by BLIS include *strong branching* (Applegate et al., 2006), *pseudo-cost branching* (Bénichou et al., 1971), *reliability branching* (Achterberg et al., 2005), and *maximum infeasibility branching.* Users can develop their own branching methods by deriving subclasses from `BcpsBranchStrategy`.

Constraint Generation. BLIS constraint generators are used to generate additional constraints to improve the problem formulation in the usual way. BLIS collects statistics that can be used to manage the generators, allowing the user to specify where to call the generator, how many cuts to generate at most, and when to activate or disable the generator. BLIS stores the generated constraints in local constraint pools. Later, BLIS augments the current subproblem with some or all of the newly generated constraints. The BLIS constraint generator base class also provides an interface between BLIS and the algorithms in the COIN-OR Cut Generator Library (CGL). Therefore, BLIS can use all the constraint generators in CGL, which includes implementations for most of the classes commonly used in modern MILP solvers. A user can easily develop a new constraint generator and add it into BLIS.

Primal Heuristics. The BLIS primal heuristic class declares the methods needed to search heuristically for feasible solutions. The class also provides the ability to specify rules to control the heuristic, such as where to call the heuristic, how often to call the heuristic, and when to activate or disable the heuristic. The methods in the class collect statistics to guide searching and provides a base class from which new heuristic methods can be derived. Currently, BLIS has only a simple rounding heuristic.

3 Computation

Having described the flexibility of the framework in adapting to a wide range of algorithm and problem types, we now discuss our efforts to use the framework "out of the box" to solve generic mixed integer linear programs. It should be emphasized that the challenges involved in developing software that is expected to be used "out of the box" are substantially different than those involved when the algorithm is expected to be used as a "framework" and pre-tuned by the user to take advantage of the known properties of a particular problem class. When properties of a particular problem class are known a priori, improved performance can be expected. With no such foreknowledge, it is crucial that default parameter settings be reasonable and that there be some mechanism by which the parameters can be adjusted automatically as the properties of a particular instance reveal themselves.

The experiments reported here are intended to demonstrate the performance that could be expected by a user employing the framework without the benefit of any foreknowledge. No after-the-fact tuning was done to obtain the results described below. We selected instances that seemed interesting based on the difficulty of their solution using state-of-the-art sequential codes without considering other characteristics, such as the size of the search tree or the amount of time required to process a node. It should be emphasized, however, that the challenges that arise in testing a parallel implementation such as this one are many and difficult to overcome. Because the amount of variability in experimental conditions is such that it is often extremely difficult to ascertain the exact causes for observed behavior, results such as those presented here have always to be taken with a healthy degree of skepticism. The situation is even more difficult when it comes to providing comparisons to other implementations because the scalability properties of individual instances can be very different under different conditions, making it difficult to choose a test set that is simultaneously appropriate for multiple solvers. We have done our best to provide some basis for comparison, but this is ultimately a near-impossible task.

Below, we report on a number of experiments to test performance and identify the major contributors to parallel overhead, as well as the effectiveness of our attempts to reduce it. We ran tests both on instances of the knapsack problem, which exhibit favorable conditions for parallelization and for a wide range of MILP instances without known structure, some of which exhibit very poor scalability properties. A number of different computational platforms were involved, including:

- An IBM Power5 Beowulf cluster at Clemson University with 52 PPC64 dual-core nodes. The clock speed was 1.654 GHz. The two cores share 4 GB RAM, as well as L2 and L3 cache, but have their own L1 cache. The backbone network was a 2 GB/sec Myrinet.

- An Intel/AMD Beowulf cluster at Clemson University with 516 dual-quad-core Intel Xeon 64-bit nodes and 255 dual-quad-core AMD Opteron 64-bit nodes. We used the AMD nodes, which have 4 MB L2 cache, 16 GB RAM and 2.33 GHz clocks. The backbone was a 10 GB/sec Myrinet.

- A Blue Gene system at San Diego Supercomputer Center (SDSC) (Reed, 2003) with three racks with 3072 compute nodes and 384 I/O nodes. Each compute node consists of two PowerPC processors that run at 700 MHz and share 512 MB of memory.

In Section 3.1, we first analyze the overall scalability of CHiPPS with default parameter settings under ideal conditions. The results presented there provide a baseline for comparison to tests with generic MILP instances, which provide a significantly more difficult challenge. In Section 3.2.1, we use a tradition scalability analysis to examine results obtained using the framework to solve a set of relatively easy instances. In Section 3.2.2, we show what can be done with a set of test problems that are much more challenging.

3.1 Knapsack Problems

To demonstrate the scalability of the framework under favorable conditions and on a class of problem with known properties, we first report results of solving knapsack instances with a simple knapsack solver that we refer to below as KNAP, which was built directly on top of ALPS. This solver uses a straightforward branch-and-bound algorithm with no cut generation. These experiments allowed us to look independently at aspects of our framework having to do with the general search strategy, load balancing mechanism, and the task management paradigm. We tested KNAP using a test set composed of difficult instances constructed using the method proposed by Martello and Toth (1990).

The test set consisted of 26 very difficult knapsack instances available from CORAL Laboratory (2009). This experiment was conducted on the SDSC Blue Gene system. Because the instances show a similar pattern, we have aggregated the results, which are shown in Table 1. The column headers have the following interpretations:

- *Total Nodes* is the total number of nodes in the search trees for the 26 instances. Observing the change in the size of the search tree as the number of processors is increased provides a rough measure of the amount of redundant work being done. Ideally, the total number of nodes explored does not grow as the number of processors is increased and may actually decrease in some cases, due to earlier discovery of feasible solutions that enable more effective pruning.

- *Ramp-up* is the percentage of total average wallclock running time time each processor spent idle during the ramp-up phase for solving the 26 instances.

- *Idle* is the percentage of total average wallclock running time each processor spent idle due to work depletion during the primary phase of the algorithm.

- *Ramp-down* is the percentage of total average wallclock running time each processor spent idle during the ramp-down phase, which starts when any of the processors became idle for the rest of the run due to lack of tasks globally.

- *Total Wallclock* is the total wallclock running time (in seconds) for solving the 26 knapsack instances.

- *Eff* is the parallel efficiency and is equal to 64 times the total wallclock running time for solving the 26 instances with 64 processors, divided by p times the total wallclock running time for solving the 26 instances with p processors. Note that the efficiency is being measured here with respect the solution time on 64 processors, rather than one, because of the computer memory and running-time limitation encountered in the single-processor runs.

Efficiency should normally represent approximately the average fraction of time spent by processors doing "useful work," where the amount of useful work to be done, in this case, is measured by the running time with 64 processors. Efficiencies greater than one are not typically observed and when they are observed, they are usually the result of random fluctuations or earlier discovery of good incumbents.

The experiment shows that KNAP scales well, even when using thousands of processors. There are several reasons for this. First, the node evaluation times are generally short and it is easy to generate a large number of nodes quickly during ramp-up. Second, the static load balancing does tend to result in a workload is already quite well balanced at the beginning of the search, perhaps due to the fact that the search trees for these "near-symmetric" instances tend to be well-balanced. Finally, the node descriptions are small and the overhead from any load balancing that is required is relatively low.

As the number of processors is increased to 2048, we observe that the wallclock running time of the main phase of the algorithm becomes shorter and shorter, while the running time of the ramp-up and ramp-down phases inevitably increase, with these phases eventually dominating the overall wallclock running time. When using 2048 processors, the average wallclock running time for solving an instance is 9.8 seconds, among which the ramp-up took 0.9 seconds and ramp-down took 2.2 seconds. This illustrates why the existence of portions of the algorithm with limited parallelizability imposes a theoretical bound on the maximum overall speedup that can be observed for a fixed set of test instances, regardless of the number of processors (Amdahl, 1967). To get a truer picture of how the algorithm scales beyond 2048 processors, we would need to scale the size and difficulty of the instances while scaling the number of processors, as suggested by Kumar and Rao (1987). This analysis was not performed due to resource limitations and the practical difficulty of constructing appropriate test sets.

Table 1: Scalability for solving knapsack instances.

Processors	Total Nodes	Ramp-up	Idle	Ramp-down	Total Wallclock	Eff
64	14733745123	0.69%	4.78%	2.65%	6296.49	1.00
128	14776745744	1.37%	6.57%	5.26%	3290.56	0.95
256	14039728320	2.50%	7.14%	9.97%	1672.85	0.94
512	13533948496	7.38%	4.30%	14.83%	877.54	0.90
1024	13596979694	13.33%	3.41%	16.14%	469.78	0.84
2048	14045428590	9.59%	3.54%	22.00%	256.22	0.77

3.2 Generic MILPs

3.2.1 Moderately Difficult Instances

For our first set of tests on generic MILP instances, we chose instances that were only moderately difficult and employed a traditional scalability analysis. We selected 18 standard MILP instances[1] from Lehigh/CORAL (CORAL Laboratory, 2009), MIPLIB 3.0, MIPLIB 2003 (Achterberg et al., 2006), BCOL (Atamtürk, 2007), and Pfetsch (2005). These instances took at least two minutes but no more than 2 hours to solve with BLIS sequentially. We used four cut generators from the COIN-OR Cut Generation Library (Lougee-Heimer, 2003): `CglGomory`, `CglKnapsackCover`, `CglFlowCover`, and `CglMixedIntegerRounding`. Pseudo-cost branching was used to select branching objects. The local search strategy was a hybrid diving strategy in which one of the children of a given node was retained as long as its bound did not exceed the best available by more than a given threshold. As per default parameter settings, ALPS used two hubs for the runs with 64 processors. For all other runs, one hub was used.

Table 2 shows the results of these computations. Note that in this table, we have added an extra column to capture sources of overhead other than idle time (i.e., communication overhead) because

[1]aflow30a, bell5, bienst1, bienst2, blend2, cls, fiber, gesa3, markshare_4_1_4_30, markshare_4_3_4_30, mas76, misc07, pk1, rout, stein, swath1, swath2, vpm2

Table 2: Scalability for generic MILP.

Processors	Total Nodes	Ramp-up	Idle	Ramp-down	Comm Overhead	Total Wallclock	Average Wallclock	Eff
1	11809956	—	—	—	—	33820.53	—	1.00
4	11069710	0.03%	4.62%	0.02%	16.33%	10698.69	0.00386	0.79
8	11547210	0.11%	4.53%	0.41%	16.95%	5428.47	0.00376	0.78
16	12082266	0.33%	5.61%	1.60%	17.46%	2803.84	0.00371	0.75
32	12411902	1.15%	8.69%	2.95%	21.21%	1591.22	0.00410	0.66
64	14616292	1.33%	11.40%	6.70%	34.57%	1155.31	0.00506	0.46

these become significant here. The effect of these sources of overhead is not measured directly, but was estimated as the time remaining after all directly measured sources had been accounted for. Note also that in these experiments, the master process did not do any exploration of subtrees, so an efficiency of $(p-1)/p$ is the best that can be expected, barring anomalous behavior. To solve these instances, BLIS needed to process a relatively large number of nodes and the node processing time were generally not long. These two properties tend to lead to good scalability, and the results reflect this to a large extent. However, as expected, overhead starts to dominate wallclock running time uniformly as the number of processors is increased. This is again an example of the limitations in scalability for a fixed test set.

Examining each component of overhead in detail, we see that ramp-up, idle, and ramp-down time grow, as a percentage of overall running time, as the number of processors increases. This is to be expected and is in line with the performance seen for the knapsack problems. However, the communication overhead turns out to be the major cause of reduced efficiency. We conjecture that this is primarily for two reasons. First, the addition of cut generation increases the size of the node descriptions and the amount of communication required to do the load balancing. Second, we found that as the number of processors increased, the amount of load balancing necessary went up quite dramatically, in part due to the ineffectiveness of the static load balancing method for these problems.

Table 3 summarizes the aggregated results of load balancing and subtree sharing when solving the 18 instances that were used in the scalability experiment. The column labeled p is the number of processors. The column labeled *Inter* is the number of inter-cluster balancing operations performed (this is zero when only one hub is used). The column labeled *Intra* is the number of intra-cluster balancing operations performed. The column labeled *Starved* is the number of times that workers reported being out of work and proactively requested more. The column labeled *Subtree* is the number of subtrees shared. The column labeled *Split* is the number of subtrees that were too large to be packed into a single message buffer and had to be split when sharing. The column labeled *Whole* is the number of subtrees that did not need to be split. As Table 3 shows, the total number of intra-cluster load balancing operations goes down when the number of processors increases. However, worker starvation increases. Additionally, the number of subtrees needing to be split into multiple message units increases. These combined effects seem to be the cause of the increase in communication overhead.

In order to put these results in context, we have attempted to analyze how they compare to results reported in papers in the literature, of which there are few. Eckstein et al. (2000) tested the scalability of PICO by solving six MILP instances from MIPLIB 3.0. For these tests, PICO was

Table 3: Load balancing and subtree sharing.

Processors	Inter	Intra	Starved	Subtree	Split	Whole
4	—	87083	22126	42098	28017	14081
8	—	37478	25636	41456	31017	10439
16	—	15233	38115	55167	44941	10226
32	—	7318	44782	59573	50495	9078
64	494	3679	54719	69451	60239	9212

used as a pure branch-and-bound code and did not have any constraint generation, so it is difficult to do direct comparison. One would expect that the lack of cut generation would increase the size of the search tree while decreasing the node processing time, thereby improving scalability. Their experiment showed an efficiency 0.73 for 4 processors, 0.83 for 8 processors, 0.69 for 16 processors, 0.65 for 32 processors, 0.46 for 64 processors, and 0.25 for 128 processors. Ralphs (2006) reported that the efficiency of SYMPHONY 5.1 is 0.81 for 5 processors, 0.89 for 9 processors, 0.88 for 17 processors, and 0.73 for 33 processors when solving 22 MILP instances from Lehigh/CORAL, MIPLIB 3.0, and MIPLIB 2003. All in all, our results are similar to those of both PICO and SYMPHONY. However, because we are using a more sophisticated and therefore less scalable algorithm, this can be seen as an improvement. Also, SYMPHONY was not tested beyond 32 processors and it is unlikely that it would have scaled well beyond that.

3.2.2 Difficult Instances

For the second set of experiments, we abandoned the traditional measures of scalability and tried simply to answer the question of whether we could use CHiPPS to accomplish something substantively different with a relatively large number of processors available to us than we could otherwise. We chose 24 test instances primarily from the Lehigh/CORAL test set that we predicted would be difficult or impossible to solve in less than 24 hours on a single processor. Note that because of the difficulty of getting large blocks of computing time, we had to commit to using this test set without knowing much about the instances. This is far from a typical use case—it would be unusual not to have some opportunity to learn about the instances before commencing production runs. Therefore, although these results need to be treated with informed skepticism, we do believe they give a realistic and positive view of the potential of this methodology once larger numbers of processors become more freely available. As technology advances, we will have more opportunities at testing with larger test set and more runs to smooth out the variability.

The instances were solved on up to 255 Opteron nodes on the Clemson Xeon/Opteron cluster running for a full 24 hours. For all tests, we used only one core per processor to eliminate sources of variability due both to the use of shared memory and to inconsistency in the speed of inter-process communications. The results are shown in Table 4. Note that seven instances have been excluded from the reporting because no solution was found during any of the tests. These were simply too difficult for BLIS to make progress on, most likely because of the lack of powerful primal heuristics in the default implementation. In the table, the numbers followed by an "s" are the time in seconds it took to solve the instance to optimality. The number followed by "%" are the gap after 24 hours. "NS" indicates that no solution was found after 24 hours. The lack of an entry indicates the experiment was not performed. The experiments with 255 nodes required reserving the entire Clemson cluster and this could only be done for relatively short periods of time. Thus, we were

Table 4: Scalability of harder MILPs.

Name	1	64	128	255
mcf2	43059s	2091s	1373s	926s
neos-1126860	39856s	2540s	1830s	2184s
neos-1122047	NS	1532s	1125s	1676s
neos-1413153	20980s	2990s	3500s	4230s
neos-1456979	NS	NS	78.06%	
neos-1461051	NS	536s	1082s	396s
neos-1599274	9075s	8108s	1500s	
neos-548047	482%	376.48%	137.29%	
neos-570431	21873s	1308s	1255s	1034s
neos-611838	8005s	886s	956s	712s
neos-612143	4837s	1315s	1716s	565s
neos-693347	NS	1.70%	1.28%	
neos-912015	10674s	275s	438s	538s
neos-933364	11.80%	6.79%	6.67%	
neos-933815	32.85%	8.77%	6.54%	
neos-934184	9.15%	6.76%	6.67%	
neos18	79344s	30.78%	30.78%	

only able to perform these runs for the problems that could solve to optimality relatively quickly. The remaining experiments would have required more than a solid week of computation and this simply was not possible.

A qualitative analysis of the results indicates that for this small test set, we were in fact able to make substantially more progress on a substantial fraction of these problems using parallelism. In a number of cases, running times improved from on the order of a day to on the order of an hour. In some cases, significantly improved gaps were obtained and in others, solutions were found with multiple processors where none were found on a single processor. Of course, the results didn't always improve with additional processing power, but these cases were the exception rather than the rule.

As noted above, efficiency can be a somewhat problematic measure of performance for dynamic search algorithms in general. In addition, it is more appropriate for analysis of parallel algorithms run on dedicated machines, which is what is commonly meant by "high-performance computing". Performance measures for high-throughput computing should more appropriately focus on throughput for a collection of jobs sharing the resources globally across jobs. Processors that are idle during the processing of one problem could be put to work solving other problems, improving overall throughput of a set of jobs. CHiPPS currently does not support such a processing mode directly, but its architecture is such that adding such a mode or integrating it with throughput management systems such as Condor (Thain et al., 2005) could be done.

3.3 The Impact of Problem Properties

As we have mentioned several times, the properties of individual instances can significantly affect scalability, independent of the implementation of the solver itself. Table 5 shows the detailed results

Table 5: Problem properties and scalability.

Instance	P	Nodes	Ramp-up	Idle	Ramp-down	Wallclock	Eff
input150_1	64	75723835	0.44%	3.38%	1.45%	1257.82	1.00
	128	64257131	1.18%	6.90%	2.88%	559.80	1.12
	256	84342537	1.62%	5.53%	7.02%	380.95	0.83
	512	71779511	3.81%	10.26%	10.57%	179.48	0.88
fc_30_50_2	64	3494056	0.15%	31.46%	9.18%	564.20	1.00
	128	3733703	0.22%	33.25%	21.71%	399.60	0.71
	256	6523893	0.23%	29.99%	28.99%	390.12	0.36
	512	13358819	0.27%	23.54%	29.00%	337.85	0.21
pk1	64	2329865	3.97%	12.00%	5.86%	103.55	1.00
	128	2336213	11.66%	12.38%	10.47%	61.31	0.84
	256	2605461	11.55%	13.93%	20.19%	41.04	0.63
	512	3805593	19.14%	9.07%	26.71%	36.43	0.36

of solving three specific MILP instances with BLIS on the SDSC Blue Gene. We use the time for the 64-processor run as the baseline for measuring efficiency.

Instance `input150_1` is a knapsack instance. When using 128 processors, BLIS achieved superlinear speedup mainly due to the decrease in the tree size. BLIS also showed good parallel efficiency as the number processors increased to 256 and 512. Instance `fc_30_50_2` is a fixed-charge network flow instance. It exhibits very significant increases in the size of its search tree (indicating the performance of redundant work) as the number of processors increases, resulting in decreased efficiency. It was found that the optimality gap of instance `fc_30_50_2` improves very slowly during the search, which causes a large number of nodes to be processed. Instance `pk1` is a small integer program with 86 variables and 45 constraints. It is relatively easy to solve. Although the efficiency is reasonable good when using 128 processors, it is eventually wiped out by significant increases in ramp-up and ramp-down overhead as the number of processors increases.

The results in Table 5 show that problem properties can have a tremendous impact on scalability. For instances that have large tree size and short node processing time, it is relatively to achieve good speedup. For instances that are particularly easy to solve or for which the upper bounds are difficult to improve, however, it can be difficult to achieve good speedup regardless of the effectiveness of the implementation.

4 Conclusions and Future Work

It has now become clear that we have no choice but to pursue parallelization of the most fundamental algorithms if we want to continue to enjoy the advantages of improvements in processing power. Parallelizing algorithms such as branch and cut in a scalable fashion is a challenge that involves balancing the costs and benefits of synchronization and the sharing of knowledge during the algorithm. Because this analysis can yield different results for different problem classes, it is difficult in general to develop a single ideal approach that will be effective across all problem classes. However, we have shown the approach taken by CHiPPS to be effective across a fairly wide range of instances. It would be premature to say that parallelism is a panacea or that one could expect uniformly good results out of the box. However, the results in Section 3.2.2 indicate that even in conditions much worse than what would typically be encountered in practice, there is hope. In a more typical use case, solving a large number of instances with similar properties and with the benefit of tuning over the long term, much better speedups should be possible.

Of course, many challenges remain. In the near term, we plan to continue improving CHiPPS by refining the load balancing schemes and knowledge sharing methods. Reducing the length of the ramp-up and ramp-down phases will be a particular focus, as effort during these phases remains a significant scalability issue. We also plan to generalize the implementation of BLIS in order to better support column generation and related algorithms, such as branch and price. In the longer term, we plan to develop modules for solution of additional problem classes, such as non-linear programs. We also hope to improve fault tolerance and develop a knowledge broker appropriate for deployment on computational grids.

References

Achterberg, T., T. Koch, A. Martin. 2005. Branching rules revisited. *Operations Research Letters* **33** 42–54.

Achterberg, T., T. Koch, A. Martin. 2006. MIPLIB 2003. *Operations Research Letters* **34** 1–12.

Amdahl, G. M. 1967. Validity of the single-processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings*. AFIPS Press, 483–485.

Applegate, D., R. Bixby, V. Chvátal, W. Cook. 2006. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, USA.

Atamtürk, A. 2007. Berkeley computational optimization lab data sets. `http://ieor.berkeley.edu/~atamturk/data`.

Benchouche, M., V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, C. Roucairol. 1996. Building a parallel branch and bound library. *Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science* **1054**. Springer, Berlin, 201–231.

Bénichou, M., J. M. Gautheier, P. Girodet, G. Hentges, G. Ribière, O. Vincent. 1971. Integer linear programming. *Mathematical Programming* **1** 76–94.

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

Brown, R. G. 2004. Engineering a beowulf-style compute cluster. `http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book/`.

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

CORAL Laboratory. 2009. Data sets. `http://coral.ie.lehigh.edu/instances.html`.

Dakin, R.J. 1965. A tree-search algorithm for mixed integer programming problems. *The Computer Journal* **8** 250–255.

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

Eckstein, J., C. A. Phillips, W. E. Hart. 2007. PEBBL 1.0 user guide.

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

Forrest, J. J. 2003. Cbc: Coin-or Branch-and-Cut. `https://projects.coin-or.org/Cbc`.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. S. Sunderam. 1994. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA.

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

Gendron, B., T. G. Crainic, A. Frangioni, F. Guertin. 2005. Oobb: Object-oriented tools for parallel branch-and-bound.

Gropp, W., E. Lusk, A. Skjellum. 1999. *Using MPI*. 2nd ed. MIT Press, Cambridge, MA, USA.

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

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

Kumar, V., V. N. Rao. 1987. Parallel depth-first search, part ii: Analysis. *International Journal of Parallel Programming* **16** 501–519.

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

Laursen, P. S. 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. Ph.D. 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.

Martello, S., P. Toth. 1990. *Knapsack Problems: Algorithms and Computer Implementation*. 1st ed. John Wiley & Sons, Inc., USA.

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

Pfetsch, Marc. 2005. Markshare instances. `http://miplib.zib.de/contrib/Markshare`.

Ralphs, T. K. 2006. Parallel branch and cut. E. Talbi, ed., *Parallel Combinatorial Optimization*. Wiley, USA, 53–102.

Ralphs, T. K., M. Güzelsoy. 2008. *SYMPHONY Version 5.1 User's Manual*. `http://www.brandandcut.org`.

Ralphs, T. K., L. Ladányi. 2001. *COIN/BCP User's Manual*. `http://www.coin-or.org/Presentations/bcp-man.pdf`.

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

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

Reed, D. A. 2003. Grids, the teragrid, and beyond. *Computer* **36** 62–68.

Sanders, P. 1998. Tree shaped computations as a model for parallel applications. In ALV'98 Workshop on application based load balancing. SFB 342, TU Munchen, Germany, March 1998. `http://www.mpi-sb.mpg.de/~sanders/papers/alv.ps.gz`.

Shinano, Y., T. Fujie. 2007. ParaLEX: A parallel extension for the CPLEX mixed integer optimizer. F. Cappello, T. Herault, J. Dongarra, eds., *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 97–106.

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

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

Thain, Douglas, Todd Tannenbaum, Miron Livny. 2005. Distributed computing in practice: the condor experience. *Concurrency—Practice and Experience* **17** 323–356.

Trienekens, H. W. J. M., A. de Bruin. 1992. Towards a taxonomy of parallel branch and bound algorithms. Tech. Rep. EUR-CS-92-01, Department of Computer Science, Erasmus University.

Tschoke, S., T. Polzer. 2008. Portable parallel branch and bound library. `http://www.cs.uni-paderborn.de/cs/ag-monien/SOFTWARE/PPBB/ppbblib.html`.

Xu, Y. 2007. Scalable algorithms for parallel tree search. Ph.D. thesis, Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, USA.