CHiPPS: A Framework for Implementing Parallel Search Algorithms

Y. Xu*, T. K. Ralphs**

1 Operations R & D, SAS Institute Inc., Cary NC 27513, Yan.Xu@sas.com
2 Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18017, ted@lehigh.edu, coral.ie.lehigh.edu/~ted

Abstract. The COIN-OR High Performance Parallel Search (CHiPPS) framework is a C++ class library for implementing and parallelizing tree search algorithms. It currently consists of a library of base classes known as the Abstract Library for Parallel Search (ALPS) and two other layers useful for implementing algorithms based on mathematical programming paradigms. Because of its general approach, however, ALPS supports the implementation of a wide variety of tree search algorithms and applications by creating derived classes implementing specific algorithmic components. CHiPPS 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 CHiPPS, focusing primarily on the ALPS layer, and indicate how the design addresses these challenges.

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. We assume that the graph to be searched has a unique root node, 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.

During the last two decades, a number of software packages for implementing parallel algorithms based on tree search have been developed. However, to our knowledge, the only other general framework for parallelizing tree search algorithms is the Parallel Implicit Graph Search Library (PIGSeL), which was developed by Peter Sanders [15, 16]. Most other work has been done in specific domains. Platzner and Rinner [10] developed a parallel algorithm for constraint satisfaction based on partitioning the search space of a constraint satisfaction problem (CSP) into independent subspaces. Feeley et al. [5] designed a parallel CSP algorithm to determine the three dimensional structure of nucleic acids. Feldmann et al. [6] developed a distributed algorithm for searching game trees with massively parallel systems and implemented a distributed chess program called ZUGZWANG. Frameworks for parallel branch and bound include BoB [1], PICO [4], PPBB-Lib [19], and PUBB [17]. FATCOP [2, 3], PARINO [8], SYMPHONY [12], ABACUS [7], and COIN/BCP [9] are parallel codes specifically for solving discrete optimization problems.

This abstract describes the COIN-OR High Performance Parallel Search (CHiPPS) framework, a framework whose design goal is to be significantly more general than those named here, but without reducing...
our ability to use the framework to produce customized packages with similar capabilities and performance. In the sections that follow, we briefly summarize the key algorithmic surrounding parallel tree search and describe the design of CHiPPS, emphasizing the Abstract Library for Parallel Search (ALPS), which is the base layer of a hierarchy designed to support a wide range of classes of tree search algorithms.

1.1 Tree Search Algorithms

In a tree search algorithm, each node is examined or processed to determine (1) whether it satisfies the properties of a goal node and (2) whether it is either a leaf node in the search tree or has successors that must also be explored. To track the progress of the algorithm, each node has an associated status, which is one of: candidate (available for processing), active (currently being processed), fathomed (processed and has no successors), or processed (not fathomed, hence has successors). 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 can be fathomed or is likely to have goal nodes among its successors.
- 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. After branching, the successors become candidates and are added to the queue of candidate nodes.

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. A primary contributor to overhead is the performance of redundant work which results from a lack of global knowledge about the state of the computation. Any mechanism for centralizing or sharing global knowledge, however, results in increased communication overhead and handshaking. Achieving the proper balance between the various sources of overhead is the challenge we must address.

2 Overview of CHiPPS

CHiPPS is a C++ class library used for implementing parallel tree search algorithms. Building on ideas of Trienekens and de Bruin from [18], we conceived the design of the CHiPPS based on a tree search methodology driven by the concept of knowledge discovery and sharing. The design and performance of CHiPPS and various specialized solvers implemented with CHiPPS have been described in a number of papers [11, 13, 14, 20, 21]. ALPS is the library of base classes within CHiPPS from which more specialized libraries pertaining to specific problem domains can be derived. We have so far implemented two additional libraries that are focused on support for the implementation of algorithms for discrete optimization. These are the Branch, constrain, and price software (BiCePS) framework, and the BiCePS Linear Integer Solver (BLIS) Framework. These are described below.
2.1 Overview of ALPS

Knowledge Sharing. 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). ALPS is therefore comprised of just three main base classes: AlpsKnowledge, AlpsKnowledgePool, and AlpsKnowledgeBroker. There are also a number of derived and auxiliary classes. Implementing specialized tree search algorithms consists of deriving specialized classes from these base classes. ALPS itself includes the derived classes needed for a basic tree search, as well as several sample applications.

In order to isolate the architecture-dependent parts of the code as much as possible, the knowledge pools do not communicate directly with each other. Instead, requests and responses are passed through the associated 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. 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.

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. The approach works well for small numbers of processors, but does not scale, 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.

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 [4]. 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.

**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.

### 2.2 Overview of 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. BiCePS is the data-handling layer needed in addition to ALPS to support relaxation-based branch-and-bound algorithms. In such algorithms, 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. In the BiCePS library, we have developed compact data structures based on the ideas of *modeling objects* and *differencing*.

### 2.3 Overview of BLIS

The BLIS library provides the functionality needed to solve mixed integer linear programs and consists mainly of classes derived from BiCePS with a few additions. BLIS has classes for storing sharing generated constraints, variables, solutions, and node descriptions, all derived from the classes in BiCePS. Other classes include those for specifying algorithms for constraint generation, primal heuristics, and branching strategies. BLIS can be used as a stand-alone solver for mixed-integer linear programs out of the box or can be customized for specific applications using a well-defined API.

### 3 Conclusions

In this abstract, we have described the main features of the CHiPPS framework. CHiPPS has already been used to implement a number of specialized solvers for applications, mainly in the discrete optimization domain. The performance observed has been extremely good in some domains and not as good in others. This is to be expected. The general design is intended to support a much wider range of applications than we have used it for to date. We are hopeful that through the exposure of this abstract to a broader audience, we will soon see application of this framework in not only the operations research community, but also in the constraint programming and artificial intelligence communities.
References


15. P. Sanders. Parallelizing np-complete problems using tree shaped computations.


