Implementing Scalable Parallel Search Algorithms for Data-intensive Applications

L. Ladányi[1], T. K. Ralphs*[2], and M. J. Saltzman[3]

[1] Department of Mathematical Sciences, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, ladanyi@us.ibm.com
[2] Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18017, tkralphs@lehigh.edu, www.lehigh.edu/~tkr2
[3] Department of Mathematical Sciences, Clemson University, Clemson, SC 29634, mjs@clemson.edu, www.math.clemson.edu/~mjs

Abstract. Scalability is a critical issue in the design of parallel software for large-scale search problems. Previous research has not addressed this issue for *data-intensive* applications. We describe the design of a library for parallel search that focuses on efficient data and search tree management for such applications in distributed computing environments.

1 Introduction

This paper describes research in which we are seeking to develop highly scalable algorithms for performing large-scale parallel search in distributed-memory computing environments. This project builds on previous work in which we developed two object-oriented, generic frameworks for implementing parallel algorithms for large-scale discrete optimization problems (DOPs). SYMPHONY (Single- or Multi-Process Optimization over Networks) [12, 10] is a framework written in C and COIN/BCP [13] is a framework written in the same spirit in C++. Because of their generic, object-oriented designs, both are extremely flexible and can be used to solve a wide variety of discrete optimization problems. However, these frameworks have somewhat limited scalability. The goal of this research is to address these scalability issues by designing a more general framework called the Abstract Library for Parallel Search (ALPS).

ALPS is a C++ class library upon which a user can build a wide variety of parallel algorithms for performing tree search. Although the framework is more general than SYMPHONY and COIN/BCP, we will initially be interested in using ALPS to implement algorithms for solving large-scale DOPs. DOPs arise in many important applications, but most are in the complexity class $\mathcal{NP}$-complete so there is little hope of finding provably efficient algorithms [3]. Nevertheless, intelligent search algorithms, such as *branch, constrain, and price* (BCP), have been tremendously successful at tackling these difficult problems.

To support the implementation of parallel BCP algorithms, we have designed two additional C++ class libraries built on top of ALPS. The first, called the

* Funding from NSF grant ACR-0102687 and the IBM Faculty Partnership Program.

Branch, Constrain, and Price Software (BiCePS) library, implements a generic framework for relaxation-based branch and bound. In this library, we make very few assumptions about the nature of the relaxations, i.e., they do not have to be linear programs. The second library, called the BiCePS Linear Integer Solver (BLIS), implements LP-based branch and bound algorithms, including BCP.

The applications we are interested in are extremely *data-intensive*, meaning that they require the maintenance of a vast amount of information about each node in the search tree. However, this data usually does not vary much from parent to child, so we use data structures based on a unique differencing scheme that is memory efficient. This scheme for storing the tree allows us to handle problems much larger than we could otherwise.

A number of techniques for developing scalable parallel branch and bound algorithms have been proposed in the literature [1, 2, 4, 5, 15]. However, we know of no previous work specifically addressing the development of scalable algorithms for data-intensive applications. Standard techniques for parallel branch and bound break down in this setting, primarily because they all depend on the ability to easily shuttle search tree nodes between processors. The data structures we need in order to create efficient storage do not allow this fluid movement. Our design overcomes this difficulty by dividing the search tree into subtrees containing a large number of related search nodes that can be stored together. This requires the design of more sophisticated load balancing schemes that accommodate this storage constraint.

2 Motivation

2.1 Scalability Issues for Parallel Search Algorithms

We will address two primary design issues that are fundamental to the scalability of parallel search algorithms. *Control mechanisms* are the methods by which decisions are made regarding the overall direction of the search, i.e., in what order the search tree nodes should be processed. The design of control mechanisms is closely tied to *load balancing*, the method by which we ensure that all processors have useful work to do. In general, *centralized* control mechanisms allow better decision making, but limit scalability by creating decision-making bottlenecks. Decentralized control mechanisms alleviate these bottlenecks, but reduce the ability to make decision based on accurate global information.

A more important issue for data-intensive algorithms is efficient *data handling*. In these algorithms, there are a huge number of *data objects* (see Section 3.2 for a description of these) that are global in nature and must be efficiently generated, manipulated, and stored in order to solve these problems efficiently. The speed with which we can process each node in the search tree depends largely on the number of objects that are *active* in the subproblem. Thus, we attempt to limit the set of active objects in each subproblem to only those that are necessary for the completion of the current calculation. However, this approach requires careful bookkeeping. This bookkeeping becomes more difficult as the number of processors increases.

2.2 Branch and Bound

In order to illustrate the above principles and define some terminology, we first describe the basics of the branch and bound algorithm for optimization. Branch and bound uses a divide and conquer strategy to partition the solution space into *subproblems* and then optimizes individually over each of them. In the *processing* or *bounding* phase, we relax the problem, thereby admitting solutions that are not in the feasible set S. Solving this relaxation yields a lower bound on the value of an optimal solution.[1] If the solution to this relaxation is a member of S, then it is optimal and we are done. Otherwise, we identify n subsets of S, $S_1, \ldots, S_n$, such that $\cup_{i=1}^{n} S_i = S$. Each of these subsets is called a *subproblem*; $S_1, \ldots, S_n$ are also sometimes called the *children* of S. We add the children of S to the list of *candidate subproblems* (those which need processing). This is called *branching*.

To continue the algorithm, we select one of the candidate subproblems, remove it from the list, and process it. There are four possible results. If we find a feasible solution better than the current best, then we replace the current best by the new solution and continue. We may also find that the subproblem has no solutions, in which case we discard, or *fathom* it. Otherwise, we compare the lower bound to upper bound yielded by the current best solution. If it is greater than or equal to our current upper bound, then we may again fathom the subproblem. Finally, if we cannot fathom the subproblem, we are forced to branch and add the children of this subproblem to the list of active candidates. We continue in this way until the list of active subproblems is empty, at which point our current best solution is the optimal one. The order in which the subproblems are processed can have a dramatic effect on the size of the tree. Processing the nodes in *best-first* order, i.e., always choosing the candidate node with the lowest lower bound, minimizes the size of the search tree. However, without a centralized control mechanism, it may be impossible to implement such a strategy.

2.3 Branch, Constrain, and Price

Branch, constrain, and price is a specific implementation of branch and bound that can be used for solving integer programming problems. Early works [6, 7, 11, 16] laid out the basic framework of BCP. Since then, many implementations (including ours) have built on these preliminary ideas. In a typical implementation of BCP, the bounding operation is performed using linear programming. We relax the integrality constraints to obtain a *linear programming (LP) relaxation*. This formulation is augmented with additional, dynamically generated constraints valid for the convex hull of solutions to the original problem. By approximating the convex hull of solutions, we hope to obtain an integer solution. If the number of columns in the constraint matrix is large, these can also be generated dynamically, in a step called *pricing*. When both constraints and variables are generated dynamically throughout the search tree, we obtain the

[1] We assume without loss of generality that we wish to minimize the objective function

algorithm known as branch, constrain, and price. Branching is accomplished by imposing additional constraints that subdivide the feasible region. Note that the data objects we referred to in Sect. 2.1 are the constraints and the variables. In large-scale BCP, the number of these can be extremely large.

3 The Library Hierarchy

To make the code easy to maintain, easy to use, and as flexible as possible, we have developed a multi-layered class library, in which the only assumptions made in each layer about the algorithm being implemented are those needed for implementing specific functionality efficiently. By limiting the set of assumptions in this way, we ensure that the libraries we are developing will be useful in a wide variety of settings. To illustrate, we briefly describe the current hierarchy.

3.1 The Abstract Library for Parallel Search

The ALPS layer is a C++ class library containing the base classes needed to implement the parallel search handling, including basic search tree management and load balancing. In the ALPS base classes, there are almost no assumptions made about the algorithm that the user wishes to implement, except that it is based on a tree search. Since we are primarily interested in *data-intensive* applications, the class structure is designed with the implicit assumption that efficient storage of the search tree is paramount and that this storage is accomplished through the compact differencing scheme we alluded to earlier. This means that the implementation must define methods for taking the difference of two nodes and for producing an explicit representation of a node from a compact one. Note that this does not preclude the use of ALPS for applications that are not data-intensive. In that case, the differencing scheme need not be used.

In order to define a search order, ALPS assumes that there is a numerical *quality* associated with each subproblem that is used to order the priority queue of candidates for processing. For instance, the quality measure for branch and bound would most likely be the lower bound in each search tree node. In addition, ALPS has the notion of a *quality threshold*. Any node whose quality falls below this threshold is fathomed. This allows the notion of fathoming to be defined without making any assumption about the underlying algorithm. Again, this quality threshold does not have to be used if it doesn't make sense.

Also associated with a search tree node is its current status. In ALPS, there are only four possible stati indicating whether the subproblem has been processed and what the result was. The possible stati are: `candidate` (available for further processing), `processed` (processed, but not branched or fathomed), `branched`, and `fathomed`. In terms of these stati, processing a node involves computing its quality and converting its status from `candidate` to `processed`. Following processing, the node is either `branched`, producing children, or `fathomed`.

3.2 The Branch, Constrain, and Price Software Library

Primal and Dual Objects. BiCePS is a C++ class library built on top of ALPS, which implements the data handling layer appropriate for a wide variety of relaxation-based branch and bound algorithms. In this layer, we introduce the concept of a `BcpsObject`, which is the basic building block of a subproblem. The set of these objects is divided into two main types, the *primal objects* and the *dual objects*. Each of the primal objects has a value associated with it that must lie within an interval defined by a given upper and lower bound. One can think of the primal objects as being the *variables*.

In order to sensibly talk about a relaxation-based optimization algorithm, we need the notion of an objective function. The objective function is defined simply as a function of the values of the primal objects. Given the concept of an objective function, we can similarly define the set of dual objects as functions of the values of the primal objects. As with the values of the primal objects, these function values are constrained to lie within certain given bounds. Hence, one can think of the dual object as the *constraints*.

Note that the notion of primal and dual objects is a completely general one. With this definition of variables and constraints, we can easily apply the general theoretical framework of Lagrangian duality. In Lagrangian duality, each constraint has both a *slack* value and the value of a *dual multiplier* or *dual variable* associated with it. Similarly, we can associate a pair of values with each variable, which are the *reduced cost* (the marginal reduction in the objective function value or the partial derivative of the objective function with respect to the variable in question) and the value of the primal variable itself.

Processing. Although the objects separate naturally into primal and dual classes, our goal is to treat these two classes as symmetrically as possible. In fact, in almost all cases, we can treat these two classes using exactly the same methods. In this spirit, we define a subproblem to be comprised of a set of objects, both primal and dual. These objects are global in nature, which means that the same object may be active in multiple subproblems. The set of objects that are active in the subproblem define the current relaxation that can be solved to obtain a bound. Note that this bound is not valid for the original problem unless either all of the primal objects are present or we have proven that all of the missing primal objects can be fixed to value zero.

We assume that processing a subproblem is an iterative procedure in which the list of active objects can be changed in each iteration by *generation* and *deletion*, and individual objects modified through *bound tightening*. To define object generation, we need the concepts of *primal solution* and *dual solution*, each consisting of a list of values associated with the corresponding objects. Object generation then consists of producing variables whose reduced costs are negative and/or constraints whose slacks are negative, given the current primal or dual solutions (these are needed to compute the slacks and reduced costs).

With all this machinery defined, we have the basic framework needed for processing a subproblem. In overview, we begin with a list of primal and dual objects

from which we construct the corresponding relaxation, which can be solved to obtain an initial bound for the subproblem. We then begin to iteratively tighten the bound by generating constraints from the resulting primal solution. In addition, we may wish to generate variables. Note that the generation of these objects "loosens" the formulation and hence must be performed strategically. The design also provides for multi-phase approaches in which variable generation is systematically delayed. During each iteration, we may tighten the object bounds and use other logic-based methods to further improve the relaxation.

Branching. To perform branching, we choose a *branching object* consisting of both a list of data objects to be added to the relaxation (possibly only for the purpose of branching) and a list of object bounds to be modified. Any object can have its bounds modified by branching, but the union of the feasible sets contained in all child subproblems must contain the original feasible set in order for the algorithm to be correct.

3.3 The BiCePS Linear Integer Solver Library

BLIS is a concretization of the BiCePS library in which we implement an LP-based relaxation scheme. This simply means that we assume the objective function and all constraints are linear functions of the variables and that the relaxations are linear programs. This allows us to define some of the notions discussed above more concretely. For instance, we can now say that a variable corresponds to a column in an LP relaxation, while a constraint corresponds to a row. Note that the form a variable or a constraint takes in a particular LP relaxation depends on the set of objects that are present. In order to generate the column corresponding to a variable, we must have a list of the active constraints. Conversely, in order to generate the row corresponding to a constraint, we must be given the list of active variables. This distinction between the *representation* and *realization* of an object will be explored further is Section 5.

4 Improving Scalability

One of the primary goals of this project is to increase scalability significantly from that of SYMPHONY and COIN/BCP. We have already discussed some issues related to scalability in Section 2. As pointed out there, this involves some degree of decentralization. However, the schemes that have appeared in the literature are inadequate for data-intensive applications, or at the very least, would require abandoning our compact data structures. Our new design attempts to reconcile the need for decentralization with our compact storage scheme, as we will describe in the next few sections.

4.1 The Master-Hub-Worker Paradigm

One of the main difficulties with the master-slave paradigm employed in SYMPHONY and COIN/BCP is that the tree manager becomes overburdened with

requests for information. Furthermore, most of these requests are *synchronous*, meaning that the sending process is idle while waiting for a reply. Our differencing scheme for storing the search tree also means that the tree manager may have to spend significant time simply *reconstructing* subproblems. This is done by working back up the tree undoing the differencing until an explicit description of the node is obtained.

Our new design employs a master-hub-worker paradigm, in which a layer of "middle management" is inserted between the master process and the worker processes. In this scheme, each hub is responsible for managing a cluster of workers whose size is fixed. As the number of processors increases, we simply add more hubs and more clusters of workers. However, no hub will become overburdened because the number of workers requesting information from it is limited. This scheme, which maintains many of the advantages of global decision-making while moving some of the computational burden from the master process to the hubs, is similar to a scheme implemented by Eckstein in his PICO framework [1].

Each hub is responsible for balancing the load among its workers. Periodically, we must also perform load balancing between the hubs themselves. Note that this load-balancing must be done not only with respect to the *quantity* of work available to each hub, but also the *quality*, i.e., nodes of high-quality must be evenly distributed among the hubs. This is done by maintaining skeleton information about the full search tree in the master process. This skeleton information includes only what is necessary to make load balancing decisions—primarily the quality of each of the subproblems available for processing. With this information, the master is able to match *donor hubs* (those with too many nodes or too high a proportion of high-quality nodes) and *receiver hubs*, who then exchange work appropriately.

4.2 Task Granularity

The most straightforward approach to improving scalability is to increase the task granularity and thereby reduce the number of decisions that need to be made centrally, as well as the amount of data that has to 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 used by the tree manager to manage the tree, 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 that it is responsible for. The hub dispenses new candidate nodes (leaves of one of the subtrees it is responsible for) to the workers as needed and tracks their progress. When a worker receives a new node, it treats this node as the root of a subtree and begins processing that subtree, stopping only when the work is completed or the hub instructs it to stop. Periodically, the worker informs the hub of its progress.

The implications of changing the basic unit of work from a subproblem to a subtree are vast. Although this allows for increased grain size, as well as more

compact storage, it does make some parts of the implementation much more difficult. For instance, we must be much more careful about how we perform load balancing in order to try to keep subtrees together. We must also have a way of ensuring that the workers don't go too far down on an unproductive path. In order to achieve this latter goal, each workers must periodically check in with the hub and report the status of the subtree it is working on. The hub can then decide to ask the worker to abandon work on that subtree and send it a new one. An important point, however, is that this decision making is always done in an asynchronous manner. This feature is described next.

4.3 Asynchronous messaging

Another design feature that increases scalability is the elimination of synchronous requests for information. This means that every process must be capable of working completely autonomously until interrupted with a request to perform an action by either its associated hub (if it is a worker), or the master process (if it is a hub). In particular, this means that each worker must be capable of acting as an independent sequential solver, as described above. To ensure that the workers are doing useful work, they periodically send an *asynchronous* message to the hub with information about the current state of its subtree. The hub can then decide at its convenience whether to ask the worker to stop working on the subtree and begin work on a new one.

Another important implication of this design is that the workers are not able to assign global identifiers to newly generated objects. In the next section we will explore the consequences of this decision.

5 Data Handling

Besides effective control mechanisms and load balancing procedures, the biggest challenge we face in implementing search algorithms for data-intensive applications is keeping track of the objects that make up the subproblems. Before describing what the issues are, we describe the concept of objects in a little more detail.

5.1 Object Representation

We stated in Section 3.2 that an object can be thought of as either a variable or a constraint. However, we did not define exactly what the terms variable or constraint mean as *abstract* concepts. For the moment, consider an LP-based branch and bound algorithm. In this setting, a "constraint" can be thought of as a method for producing a valid row in the current LP relaxation, or, in other words, a method of producing the projection of a given inequality into the domain of the current set of variables. Similarly, a "variable" can be thought of as a method for producing a valid column to be added to the current LP relaxation. This concept can be generalized to other settings by requiring each

object to have an associated method that performs the modifications appropriate to allow that object to be added to the current relaxation.

Hence, we have the concept that an object's *representation*, which is the way it is stored as a stand-alone object, is inherently different from its *realization* in the current relaxation. An object's representation must be defined with respect to both the problem being solved and the form of the relaxation. Within BiCePS, new categories of objects can be defined by deriving a child class from the `BcpsObject` class. This derived class holds the data the user needs to realize that type of object within the context of a given relaxation and also defines the method of performing that realization. When sending an object's description to another process, we need to send only the data itself, and even that should be sent in as compact a form as possible. Therefore the user must define for each category of variable and constraint a method for *encoding* the data compactly into a character array. This form of the object is then used whenever the object is sent between processes or when it has to be stored for later use. This encoding is also used for another important purpose that is discussed in Section 5.2. Of course, there must also be a corresponding method of decoding as well.

5.2 Tracking Objects Globally

The fact that the algorithms we consider have a notion of global objects is, in some sense, what makes them so difficult to implement in parallel. In order to take advantage of the economies of scale that come from having global objects, we must have global information and central storage. But this is at odds with our goal of decentralization and asynchronous messaging.

To keep storage requirements at a minimum, we would like to know at the time we generate an object whether we have previously generated that same object somewhere else in the tree. If so, we would like to refer to the original copy of that object and delete the new one. Ideally, we would only need to store one copy of each object centrally and simply copy it out whenever it is needed locally.

Instead, we simply ensure that at most one copy of each object is stored locally within each process. Objects within a process are stored in a hash table, with the hash value computed from the encoded form. If an object is generated that already exists, the new object is deleted and replaced with a pointer to the existing one. Note that we can have a separate hash table for primal and dual objects, as well as for each object category. The same hashing mechanism is implemented in the hubs with the additional complexity that when an entire subtree is sent back from a worker, the pointers to the hashed objects in the search tree nodes must be reconciled. This is the limit of what can be done without synchronous messaging.

There is yet one further level of complexity to the storage of objects. In most cases, we need only keep objects around while they are actually pointed to, i.e., are active in some subproblem that is still a candidate for processing. If we don't occasionally "clean house," then the object list will continue to grow boundlessly, especially in the hubs. To take care of this, we use smart pointers

that are capable of tracking the number of references to an object and deleting the object automatically after there are no more pointers to it.

5.3 Object Pools

Of course, the ideal situation would be to avoid generating the same object twice in the first place. For this purpose, we provide *object pools*. These pools contain sets of the "most effective" objects found in the tree so far so that they may be utilized in other subproblems without having to be regenerated. These objects are stored using a scheme similar to that for storing the active objects, but their inclusion in or deletion from the list is not necessarily tied to whether they are active in any particular subproblem. Instead, it is tied to a rolling average of a quality measure that can be defined by the user. By default, the measure is simply the slack or reduced cost calculated when the object is checked against a given solution to determine the desirability of including that object in the associated subproblem.

5.4 Conclusions

We have described the main design features of a library hierarchy for implementing large-scale parallel search for data-intensive applications. This project is being developed as open source under the auspices of the Common Optimization Interface for Operations Research (COIN-OR) initiative. Source code and documentation will be available from the CVS repository at `www.coin-or.org`.

References

1. Eckstein, J.: Parallel Branch and Bound Algorithms for General Mixed Integer Programming on the CM-5. SIAM Journal on Optimization **4**, 794–814, 1994.
2. Eckstein, J.: How Much Communication Does Parallel Branch and Bound Need? INFORMS Journal on Computing **9**, 15–29, 1997.
3. Garey, M.R., and Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co., San Francisco, 1979.
4. Gendron, B., and Crainic, T.G.: Parallel Branch and Bound Algorithms: Survey and Synthesis. Operations Research **42**, 1042–1066, 1994.
5. Grama, A., and Kumar, V.: Parallel Search Algorithms for Discrete Optimization Problems. ORSA Journal on Computing **7**, 365–385, 1995.
6. Grötschel, M., Jünger, M., and Reinelt, G.: A Cutting Plane Algorithm for the Linear Ordering Problem. Operations Research **32**, 1195–1220, 1984.
7. Hoffman, K., and Padberg, M.: LP-Based Combinatorial Problem Solving. Annals of Operations Research **4**, 145–194, 1985.
8. Kumar, V., and Rao, V.N.: Parallel Depth-first Search. Part II. Analysis. International Journal of Parallel Programming **16**, 501–519, 1987.
9. Kumar, V., and Gupta, A.: Analyzing Scalability of Parallel Algorithms and Architectures. Journal of Parallel and Distributed Computing **22**, 379–391, 1994.

10. Ladányi, L., Ralphs, T.K., and Trotter, L.E.: Branch, Cut, and Price: Sequential and Parallel. In Computational Combinatorial Optimization, D. Naddef and M. Jünger, eds., Springer, Berlin, 229–267, 2001.
11. Padberg, M., and Rinaldi, G.: A Branch-and-Cut Algorithm for the Resolution of Large-Scale Traveling Salesman Problems. SIAM Review **33**, 60–100, 1991.
12. Ralphs, T.K., SYMPHONY Version 2.8 User's Guide. Lehigh University Industrial and Systems Engineering Technical Report 01T-011. Available at `www.branchandcut.org/SYMPHONY`.
13. Ralphs, T.K. and Ladányi, L.: COIN/BCP User's Guide, 2001. Available at `www.coin-or.org`.
14. Rao, V.N., and Kumar, V.: Parallel Depth-first Search. Part I. Implementation. International Journal of Parallel Programming **16**, 479–499, 1987.
15. Rushmeier, R., and Nemhauser, G.L.: Experiments with Parallel Branch and Bound Algorithms for the Set Covering Problem. Operations Research Letters **13**, 277–285, 1993.
16. Savelsbergh, M.W.P.: A Branch-and-Price Algorithm for the Generalized Assignment Problem. Operations Research **45**, 831–841, 1997.