Parallel Branch and Cut for Capacitated Vehicle Routing *

T.K. Ralphs†

December 1, 2002

Abstract

Combinatorial optimization problems arise commonly in logistics applications. The most successful approaches to date for solving such problems involve modeling them as integer programs and then applying some variant of the branch and bound algorithm. Although branch and bound is conceptually easy to parallelize, achieving scalability can be a challenge. In more sophisticated variants, such as *branch and cut*, large amounts of data must be shared among the processors, resulting in increased parallel overhead. In this paper, we review the branch and cut algorithm for solving combinatorial optimization problems and describe the implementation of SYMPHONY, a library for implementing these algorithms in parallel. We then describe a solver for the *vehicle routing problem* that was implemented using SYMPHONY and analyze its parallel performance on a Beowulf cluster.

1 Introduction

A wide variety of problems arising in logistics applications can be viewed as *combinatorial optimization problems* (COPs). An instance of a COP is defined by a pair $(E, \mathcal{F})$ and a *cost vector* $c \in \mathbb{R}^E$, where E is called the ground set and $\mathcal{F} \subseteq 2^E$ is a family of subsets of the ground set whose members are called the *feasible solutions*. Each feasible solution S has an associated cost given by $c(S) = \sum_{e \in S} c_e$. The objective is to find a least cost member of $\mathcal{F}$ (we assume without loss of generality that the problem to be solved is one of *minimization*). Examples of fundamental logistics problems that can be viewed as COPs are routing problems, packing problems, facility location problems, scheduling problems, and inventory problems. More elaborate models, such as the *vehicle routing problem*, combine aspects of these fundamental ones.

Solving large-scale instances of these problems can be extremely difficult in practice because the set $\mathcal{F}$, although often conceptually easy to enumerate, is usually extremely large. The most successful approaches to date have utilized integer programming techniques. To formulate a given COP as an integer program, we associate an incidence vector with each member of $\mathcal{F}$. The feasible solutions are then described as the incidence vectors satisfying

*This research was partially supported by NSF Grant ACI-0102687

†Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18017, tkralphs@lehigh.edu, http://www.lehigh.edu/~tkr2

a given set of linear inequalities. In what follows, we discuss solution techniques for such integer programs and thus, for the remainder of the paper, we interpret the members of the set $\mathcal{F}$ directly as incidence vectors rather than as subsets of E.

Although formulating a COP as an integer program is usually an easy exercise, most interesting COPs are nonetheless in the complexity class NP-hard and it is usually not possible to explicitly enumerate the member of $\mathcal{F}$. The most common solution techniques for these integer programs are instead based on *implicit enumeration*, usually some variant of the *branch and bound* algorithm suggested by Land and Doig [41]. We consider one such variant, known as *branch and cut*, in which the bounds are obtained by the solution of a *linear programming relaxation* augmented by inequalities valid for the set $\mathcal{F}$. Branch and cut uses a divide and conquer strategy and lends itself naturally to parallelization. However, its effectiveness depends on the sharing of large amounts of information among processors, namely the valid inequalities used to augment the LP relaxations, which makes scalability a challenge. This paper discusses our approach to the parallelization of this algorithm and provides an analysis of its scalability.

The literature on parallel computation in general and parallel branch and bound in particular is rich and varied, so we will mention only a few closely related papers here. Kumar and Gupta provide an excellent general introduction to the analysis of parallel scalability in [37]. Good overviews and taxonomies of parallel branch and bound algorithms are provided in both [27] and [55]. Eckstein [22] also provides a good overview of the implementation of parallel branch and bound. A substantial number of papers have been written specifically about the application of parallel branch and bound to integer programming problems. These include such works as [28], [57], [11], [46], and [20].

In the remainder of the paper, we first describe the vehicle routing problem, a prototypical logistics application, and then use it to illustrate some of the principles involved in solving COPs by branch and cut. In Section 4, we discuss the design of SYMPHONY, our framework for implementing parallel branch and cut. After describing the framework, we illustrate its use by discussing the implementation of a basic VRP solver using SYMPHONY. Finally, we present computational results and an analysis of SYMPHONY's scalability.

2 The Vehicle Routing Problem

We consider the classical vehicle routing problem (VRP), introduced by Dantzig and Ramser [21]. In this problem, a quantity d_i of a single commodity must be delivered to each customer $i \in N = [1..n] = \{1, \ldots, n\}$ from a central depot $\{0\}$ using k identical delivery vehicles of capacity C. The objective is to minimize total cost, with $c_{ij} \geq 0$ denoting the cost of transporting goods from i to j, for $0 \leq i, j \leq n$. Note that this cost does not depend on the quantity transported and that the cost structure is assumed *symmetric*, i.e., $c_{ij} = c_{ji}$ and $c_{ii} = 0$.

Conceptually, a solution for this problem consists of a partition $\{R_1, \ldots, R_k\}$ of N into k *routes*, each satisfying $\sum_{j \in R_i} d_j \leq C$, and a corresponding permutation σ_i of each route specifying the service ordering. A number of authors have considered solution techniques for this problem, e.g., [1, 5, 12, 16, 24, 25]. Most of these utilize some form of branch and bound. Numerous classes of valid inequalities, as well as algorithms for generating them in a branch and cut framework, have also been proposed (see, e.g., [42, 43, 4, 14, 19, 44, 7, 6, 12]). An excellent overview of the use of branch and cut algorithms for solving the VRP is contained in [48].

The VRP can be viewed as a combinatorial optimization problem by associating with it the undirected graph consisting of nodes $N \cup \{0\}$ and edges $E \subseteq \{\{i,j\} : i,j \in N, i \neq j\}$. The set E is then the ground set and we associate a cost c_e with each edge $e \in E$. We define a *route* in this graph to be a set of nodes R such that

- $R = \{i_1, i_2, \ldots, i_m\}$,
- $i_j \neq i_l \ \forall j, l \in [1..m], j \neq l$, and
- $E(R) = \{\{i_j, i_{j+1}\} : j \in [1..m]\} \subseteq E$ where i_{m+1} is interpreted to be i_1.

If R is a route, then we call $E(R)$ the edge set of the route. A feasible solution is then any subset of E that is the union of the edge sets of k routes $R_i, i \in [1..k]$ with the following properties:

- $R_i \cap R_j = \{0\}$, $i,j \in [1..k], i \neq j$, and
- $\sum_{j \in R_i} d_j \leq C$, $\forall i \in [1..k]$.

In this solution, each route corresponds to a set of customers serviced by one of the k vehicles. By associating a binary variable with each edge in the graph, we obtain the following *integer programming formulation* of this COP, which has its roots in [43]:

$$
\begin{array}{llcll}
min & \sum_{e \in E} c_e x_e & & & \\
s.t. & \sum_{e=\{0,j\} \in E} x_e & = & 2k & (1) \\
 & \sum_{e=\{i,j\} \in E} x_e & = & 2 \quad \forall i \in N & (2) \\
 & \sum_{\substack{e=\{i,j\} \in E \\ i \in T, j \notin T}} x_e & \geq & 2b(T) \quad \forall T \subset N, \ |T| > 1 & (3) \\
 & 0 \leq x_e & \leq & 1 \quad \forall e = \{i,j\} \in E, \ i,j \neq 0 & (4) \\
 & 0 \leq x_e & \leq & 2 \quad \forall e = \{0,j\} \in E & (5) \\
 & x_e & & integral \quad \forall e \in E. & (6)
\end{array}
$$

For ease of computation, we define $b(T) = \left\lceil \left(\sum_{i \in T} d_i\right)/C \right\rceil$, an obvious lower bound on the number of vehicles needed to service the customers in set T. In this formulation, constraint (1) ensures that there are exactly k vehicles, while constraints (2) ensure that each customer is serviced by exactly one vehicle, as well as ensuring that the solution is the union of edge sets of routes. Constraints (3) ensure that every route includes the depot and that no route has total demand greater than the capacity C.

It is clear from our description that the VRP is closely related to two difficult combinatorial problems. By setting $C = \infty$, we get an instance of the *multiple traveling salesman problem* and by setting $c_e = 0 \ \forall e \in E$, we get a feasibility version of the *bin packing problem* with a fixed number of bins. Because of the combined structure of these two underlying models, instances of the VRP can be extremely difficult to solve in practice. In fact, the largest instance of the VRP solved to date (135 customers) is two orders of magnitude smaller than that of the traveling salesman problem (15112 customers). With the aim of solving some of the extremely difficulty VRP instances that have appeared in the literature, we have developed a parallel branch and cut solver using SYMPHONY. Through parallelism, we have been able to solve several previously unsolved instances from the literature. Some of these are reported on in [53].

3 Background

3.1 Branch and Bound

Branch and bound is a technique for solving optimization problems that uses a divide and conquer strategy to partition the solution space into *subproblems* and then solves each subproblem recursively. In the *processing* or *bounding* phase, we *relax* the problem, admitting solutions that are not in the feasible set $\mathcal{F}$. Solving this relaxation, which is typically much easier than solving the original problem, yields a lower bound on the value of an optimal solution. If the solution to this relaxation is a member of $\mathcal{F}$, then it is optimal and we are done. Otherwise, we identify p subsets of $\mathcal{F}$, $\mathcal{F}_1, \ldots, \mathcal{F}_p$, such that $\cup_{i=1}^{p} \mathcal{F}_i = \mathcal{F}$. Each of these subsets is called a *subproblem*; $\mathcal{F}_1, \ldots, \mathcal{F}_p$ are also sometimes called the *children* of S. We add the children of $\mathcal{F}$ to the list of *candidate subproblems* (those that 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 is empty, in which case we say it is *infeasible* and discard, or *prune* it. Otherwise, we compare the lower bound to the upper bound yielded by the current best solution. If it is greater than or equal to our current upper bound, then we may again prune the subproblem. Finally, if we cannot prune the subproblem, we are forced to branch and add the children of this subproblem to the list of candidates. We continue in this way until the list of candidate subproblems is empty, at which point our current best solution must be the optimal one.

3.2 Branch and Cut

In integer programming, the bounding operation is often accomplished using the tools of linear programming (LP), a technique described in generality, e.g., by Hoffman and Padberg [32]. This general class of algorithms is known as *LP-based branch and bound.* Typically, the integrality constraints (for example, constraints (6)) of an integer programming formulation of the problem are relaxed to obtain an *LP relaxation*, which is then solved to derive a lower bound for the problem. We can improve on this basic idea by using globally valid inequalities (i.e., inequalities satisfied by all members of the feasible set $\mathcal{F}$), to strengthen the LP relaxation. Padberg and Rinaldi called this technique *branch and cut* [49]. An early implementations of the technique is described in [29].

Bounding Operation
Input: A subproblem $S \subseteq \mathcal{F}$, described by a "small" set of inequalities $\mathcal{L}_S$ such that $S = \{x \in \{0,1\}^{|E|} : ax \leq \beta \ \forall (a,\beta) \in \mathcal{L}_S\}$ and α, an upper bound on the global optimal value.
Output: Either (1) an optimal solution $x^* \in S$ to the subproblem, (2) a lower bound on the optimal value of the subproblem and the corresponding relaxed solution $\overline{x}$, or (3) a message `pruned` indicating that the subproblem should not be considered further.
1. If the LP $\min\{cx : ax \leq \beta \ \forall \ (a,\beta) \in \mathcal{L}_S\}$ is infeasible, then $S = \emptyset$. STOP and output `pruned`.
2. Otherwise, consider the LP solution $\overline{x}$. If $c\overline{x} < \alpha$, then continue with Step 3. Otherwise, STOP and output `pruned`—this subproblem cannot produce a solution of value better than α.
3. If $\overline{x} \in S$, then $\overline{x}$ is the optimal solution to this subproblem. STOP and output $\overline{x}$ as x^*.
4. Otherwise, apply separation algorithms and heuristics to $\overline{x}$ to obtain a set of violated inequalities $\mathcal{L}'$. If $\mathcal{L}' = \emptyset$, then $c\overline{x}$ is a lower bound on the value of an optimal element of S. STOP and return $\overline{x}$ and the lower bound $c\overline{x}$.
5. Otherwise, set $\mathcal{L}_S \leftarrow \mathcal{L}_S \cup \mathcal{L}'$ and go to Step 1.

Figure 1: Bounding in the branch and cut algorithm

To see how this general method can be applied to COPs, consider a combinatorial optimization problem $CP = (E, \mathcal{F})$ with *ground set* E, *feasible set* $\mathcal{F}$, and cost function $c \in \mathbb{R}^E$. As before, we view the members of $\mathcal{F}$ as incidence vectors obeying a given set of inequalities. These inequalities are typically the ones we use to form the initial LP relaxation, as described in Figure 3. If we let $\mathcal{P}$ be the polyhedron obtained by taking the convex hull of $\mathcal{F}$, then $\mathcal{F} = \mathcal{P} \cap \{0,1\}^{|E|}$ and furthermore, the member of $\mathcal{F}$ are the extreme points of the polyhedron $\mathcal{P}$. Hence, the minimum of cx over all $x \in \mathcal{F}$ is equal to the minimum of cx over all $x \in \mathcal{P}$.

By Weyl's Theorem (see [36]), there exists a finite set $\mathcal{L}$ of inequalities valid for $\mathcal{P}$ such that

$$\mathcal{P} = \{x \in \mathbb{R}^n : ax \leq \beta \ \ \forall \ (a,\beta) \in \mathcal{L}\} \tag{7}$$

The inequalities in $\mathcal{L}$ are the potential *cutting planes*, or *cuts*, that can be used to strengthen the LP relaxations as needed. For a given $\hat{x} \in \mathcal{P}$, the *separation problem* is to find a member of $\mathcal{L}$ violated by $\hat{x}$. By a result in [30], this problem has the same computational complexity as the original optimization problem. Hence, it is usually difficult, if not impossible, to enumerate all of the inequalities in $\mathcal{L}$. Instead, we use *separation algorithms* and *heuristics* to attempt to generate these inequalities when they are violated. In Figure 1, we describe more precisely how the bounding operation is carried out.

Once we have failed to either prune the current subproblem or separate the current *relaxed solution* from $\mathcal{P}$, we are forced to branch. The branching operation is accomplished by identifying a polychotomy that can be used to divide the current subproblem in such a way that

Branching Operation
Input: A subproblem S and $\hat{x}$, the LP solution yielding the lower bound.
Output: $S_1, \ldots, S_p$ such that $S = \cup_{i=1}^{p} S_i$.
1. Determine sets $\mathcal{L}_1, \ldots, \mathcal{L}_p$ of inequalities such that $S = \cup_{i=1}^{p}\{x \in S : ax \leq \beta \, \forall \, (a,\beta) \in \mathcal{L}_i\}$, $\hat{x} \notin \cup_{i=1}^{n} S_i$, and $S_i \cap S_j = \emptyset \; \forall i,j \in [1..p], i \neq j$.
2. Set $S_i = \{x \in S : ax \leq \beta \;\; \forall \, (a,\beta) \in \mathcal{L}_i \cup \mathcal{L}_S\}$ where $\mathcal{L}_S$ is the set of inequalities used to describe S.

Figure 2: Branching in the branch and cut algorithm

- the current relaxed solution is not feasible for any of the new subproblems and
- the resulting subproblems are disjoint.

This polychotomy is generically called a *branching set* or just a *branching*. We call a branching with the second property listed above *partitive*. A typical method for defining a branching with these two properties is to select a valid inequality $ax \leq \beta$ with integer coefficients (so that the *left hand side value* must be an integer for any integer solution) and an integer right hand side such that the current left hand side value is $f \notin \mathbb{Z}$. Then one can simply branch on the dichotomy

$$ax \leq \lfloor f \rfloor \quad \text{OR} \quad ax \geq \lceil f \rceil. \tag{8}$$

Note that this branching is partitive and that the current relaxed solution is not feasible for either of the generated subproblem. In the case that the valid inequality being branched on is a bound constraint for one of the variables, this procedure reduces to fixing a variable whose current value is fractional to 0 in one branch and 1 in the other. This branching scheme is easy to implement within the context of LP-based branch and bound because we have only to add the appropriate additional constraints to the LP relaxation of the parent in order to create the children. The procedure is described more formally in Figure 2. Figure 3 gives a high level description of the generic branch and cut algorithm for combinatorial optimization.

The primary difference between branch and cut and generic LP-based branch and bound is that the inequalities that are generated at each node of the search tree are globally valid and can hence be utilized during the processing of subsequent search tree nodes. This sharing of cuts is accomplished through *cut pools* that maintain the "most effective" inequalities found so far. This approach has several advantages over standard LP-based branch and bound. Through prudent maintenance of the cut pools, we are able to obtain a global picture of which cuts are the "most important." Since most techniques for generating valid inequalities are heuristic in nature, the cut pool may actually contain inequalities useful in the current search tree node that would not have otherwise been found. Most importantly, however, is that we may be able to greatly reduce or eliminate wasted time spent regenerating cuts that have already been previously generated. A more in-depth treatment of the concept of sharing information (cuts) between nodes in the search tree is contained in [52].

Generic Branch and Cut Algorithm
<u>Input</u>: An initial set $\mathcal{L}_0$ inequalities valid for $\mathcal{F}$.
<u>Output</u>: A global optimal solution x^* to the problem instance.
1. Generate a "good" feasible solution $\hat{x}$ using heuristics. Set $\alpha \leftarrow c\hat{x}$.
2. Form the *root subproblem* $\mathcal{S}_0$ using the set $\mathcal{L}_I$ of initial inequalities. Set $B \leftarrow \{\mathcal{S}_0\}$.
3. If $B = \emptyset$, STOP and output $\hat{x}$ as the global optimum x^*. Otherwise, choose some $\mathcal{S}_i \in B$. Set $B \leftarrow B \setminus \{\mathcal{S}_i\}$. Apply the bounding procedure to $\mathcal{S}_i$ (see Figure 1).
4. If the result of Step 3 is a feasible solution $\overline{x}$, then $c\overline{x} < c\hat{x}$. Set $\hat{x} \leftarrow \overline{x}$ and $\alpha \leftarrow c\overline{x}$ and go to Step 3. If the subproblem was pruned, go to Step 3. Otherwise, go to Step 5.
5. Perform the branching operation. Add the set of subproblems generated to B and go to Step 3.

Figure 3: Description of the generic branch and cut algorithm

Note that, as with cutting planes, the set of variables can also be defined implicitly, yielding a similar technique called *branch and price* in which variables are generated dynamically (see [9]). When both variables and cutting planes are generated dynamically during LP-based branch and bound, the technique is known as *branch, cut, and price* (BCP). Although SYMPHONY supports the implementation of BCP algorithms, we assume henceforth that the set of variables remains fixed. Through a technique called *reduced cost fixing*, it is sometimes possible to prove that the value of a particular variable cannot be nonzero in a given search tree node and all of its descendants. In this case, we delete the variable from the subproblem and consider it *inactive*.

3.3 Related Software

The branch and bound algorithm described above was first suggested by Land and Doig in 1960 [41]. In 1970, Mitten abstracted branch and bound into the theoretical framework we are familiar with today [47]. However, it was another two decades before sophisticated software packages for solving integer programming problems began to be developed. Most of the software packages developed to date implement some version of branch and bound. We divide the available software into two main categories—packages based on general-purpose algorithms for solving mixed integer programs (MIPs) (without the use of special structure) and those facilitating the use of special structure by interfacing with user-supplied, problem-specific subroutines. We call packages in this second category *frameworks*. There have also been numerous special-purpose codes developed for use in solving specific types of combinatorial problems.

Generic MIP solvers such as MINTO [35], MIPO [8], and bc-opt [18] are among the research codes being offered. Commercial MIP solvers include ILOG's CPLEX, IBM's OSL, and Dash's XPRESS. Generally speaking, generic MIP solvers are not capable of solving large instances of difficult combinatorial problems. Such problems require the use of problem-specific subroutines that take advantage of special problem structure. Generic frameworks allow the user to take advantage of special structure. SYMPHONY, COIN/BCP [51] and ABACUS [23, 33, 34] are the most full-featured frameworks available. CONCORDE

[2, 3], a package for solving the traveling salesman problem, also deserves mention as the most sophisticated special-purpose code developed to date.

Numerous software packages implementing parallel branch and bound have also been developed. The previously mentioned SYMPHONY, COIN/BCP, and CONCORDE are all parallel codes that can be run on networks of workstations. Other related software includes frameworks for implementing parallel branch and bound such as PUBB [54], BoB [10], PPBB-Lib [56], and PICO [22]. PARINO [45] and FATCOP [15] are parallel MIP solvers.

4 The SYMPHONY Framework

SYMPHONY is a C library for implementing parallel BCP algorithms developed by the author and Ladányi. The design goal of SYMPHONY is to provide an easy-to-use framework that can be used to implement a solvers for a wide variety of problem settings and across a wide variety of architectures. The vast majority of the subroutines in the branch and cut algorithm—such as those for tree management, LP solution, and cut pool management, as well as inter-process communication—are generic and are internal to the library. The internal library, about which the user need know nothing, interfaces with the user's subroutines through a well-defined Application Program Interface (API) described in [50]. To implement a state-of-the-art parallel solver, the user need only provide a few problem-specific methods we describe below. Full details of the sequential implementation of SYMPHONY are contained in [50] and [40]. Here, we concentrate only on the details most relevant to the parallel implementation.

Implementing a branch and cut solver presents difficult challenges. The primary difficulty is that the number of valid inequalities that may be generated during the algorithm is limitless for all practical purposes. The list of inequalities needed to fully describe the convex hull of feasible solutions for even the simplest of combinatorial problems can easily exceed the combined storage capacity of any modern parallel computer. Fortunately, we can solve reasonably-sized instances of these most problems with only a small number of "important" inequalities, but determining which inequalities are contained in this small subset is difficult at best. We must also deal with the very large search trees that may be generated during the solution process. This involves not only the important question of how to store the descriptions of the individual subproblems, but also how to move them between processors during parallel execution. A final challenge in developing a generic framework is to deal with these issues in a problem-independent way.

Describing a node in the search tree consists of, among other things, listing the cuts (and variables) that are initially *active* in the subproblem. In fact, the vast majority of the methods in branch and cut that depend on the model are related to generating, manipulating, and storing the cuts. From the user's perspective, implementing a branch and cut algorithm using SYMPHONY consists primarily of specifying properties of various classes of cuts, such as how to generate them, how to represent them, and how to realize them within the context of a particular subproblem.

4.1 Data Structures and Storage

There are three basic types of data that must be stored and shared in parallel branch and cut—upper bounds, valid inequalities, and search tree nodes. Only the last two, however, present any real challenge. In addition, we need to able to store the search tree itself. Because this tree can grow extremely large, we have developed compact data structures based on the idea of *differencing*, to be explained below. In the next two sections, we explain how each of these basic data types are represented and stored.

4.1.1 Variables and Constraints

Both the memory required to store the search tree and the time required to process a node are largely dependent on the number of constraints and variables active in each subproblem. Keeping this active set as small as possible is done using ad hoc rules that allow us to judge which cuts and variables are "important" in the subproblem and which are not. These rules and other details regarding the management of the LP relaxations are covered in [40]. Efficient management of the LP relaxation is one of the keys to efficiently implementing branch and cut. For this reason, we need data structures that enhance our ability to efficiently move variables and constraints in and out of the active set.

Because we assume the set of variables is fixed a priori, each variable can be represented simply by a predetermined global index. However, the set of potential constraints is not known at the outset, so cuts are manipulated and stored by means of a user-defined, compact *representation* that contains information about how to add them to a particular LP relaxation. Adding a cut to a given LP relaxation consists of specifying its form with respect to the given set of active variables. This means that the user must provide a method for converting this representation to a row of the constraint matrix. The representation includes the associated bounds (in the case of a variable) or right hand side range (in the case of a constraint) and is also used to store each cut and to pass it from one processor to another as needed. After generation, each cut is assigned a global index by which it is referred to throughout the search tree. Note that this assignment of a global index must be done centrally—this has repercussions for scalability, as we discuss in Section 5.

4.1.2 Search Tree

The description of a search tree node consists primarily of the indices of the cuts and variables that are active in that node. A critical aspect of implementing branch and cut is the maintenance of a complete description of the current *basis* (assuming a simplex-based LP solver) for each node in order to allow a *warm start* to the bound computation (for the definition of a basis and its role in the solution of linear programs, see [17] or [36]). This basis is either inherited from the parent or computed during strong branching (see Section 4.4.3). Along with the set of active cuts and variables, we must also store the identity of the branching set that generated the node.

Because the set of active inequalities and the description of the basis do not tend to change much from parent to child, all of these data are stored as differences with respect to the parent when that description is smaller than the explicit one. This method of storing the entire tree is highly memory-efficient. The list of nodes that are candidates for processing is stored in a heap ordered by a comparison function defined by the search strategy (see 4.3). This allows efficient generation of the next node to be processed.

One way in which we attempt to limit the size of the node descriptions is by allowing the user to specify a problem *core* consisting of a set of variables and constraints that are to be active in every subproblem. The core should consist of a set of variables and constraints that are considered "important" for the given instance, in the sense that there is a high probability that they will be needed to describe an optimal solution. The advantage of specifying a core is that the description of the core can be stored statically at each of the processors and need not be part of the individual node descriptions. This saves both on node set-up costs and communication costs, as well as making storage of the search tree more efficient. An example of how the core is specified for an instance of the VRP is given in Section 4.4.1.

4.2 Parallel Implementation

The solver functions are grouped into four independent modules, the *master module*, the *node processing (NP) modules* (for processing search tree nodes) and two modules that maintain the global data (one for cuts and one for search tree nodes). This modular implementation allows for easy and highly configurable parallelization. The modules can be compiled as either (1) a single sequential code, (2) a multi-threaded shared-memory parallel code, or (3) separate processes running over a distributed network. The modules pass data to each other either through shared memory (in the case of sequential computation or shared-memory parallelism) or through a message-passing protocol defined in a separate communications API (in the case of distributed execution). A schematic overview of the modules is presented in Figure 4. Here, we address only a distributed-memory version of SYMPHONY implemented using the PVM message-passing protocol [26].

4.2.1 The Master Module

The *master module* performs problem initialization and I/O, as well as helping to maintain fault-tolerance. This module is not heavily tasked once the computation has begun, but is kept separated in order to monitor the status of the rest of the processes. The functions performed by the master module include the following tasks:

- Read in the parameters from a data file.
- Read in the data for the problem instance.
- Compute an initial upper bound using heuristics (may also be done in parallel).
- Perform problem preprocessing and determine the problem core.
- Initialize the algorithm by sending data for the *root node* to the *tree manager*.
- Initialize output devices and act as a conduit for output (the output is usually minimal).
- Process requests for problem data.

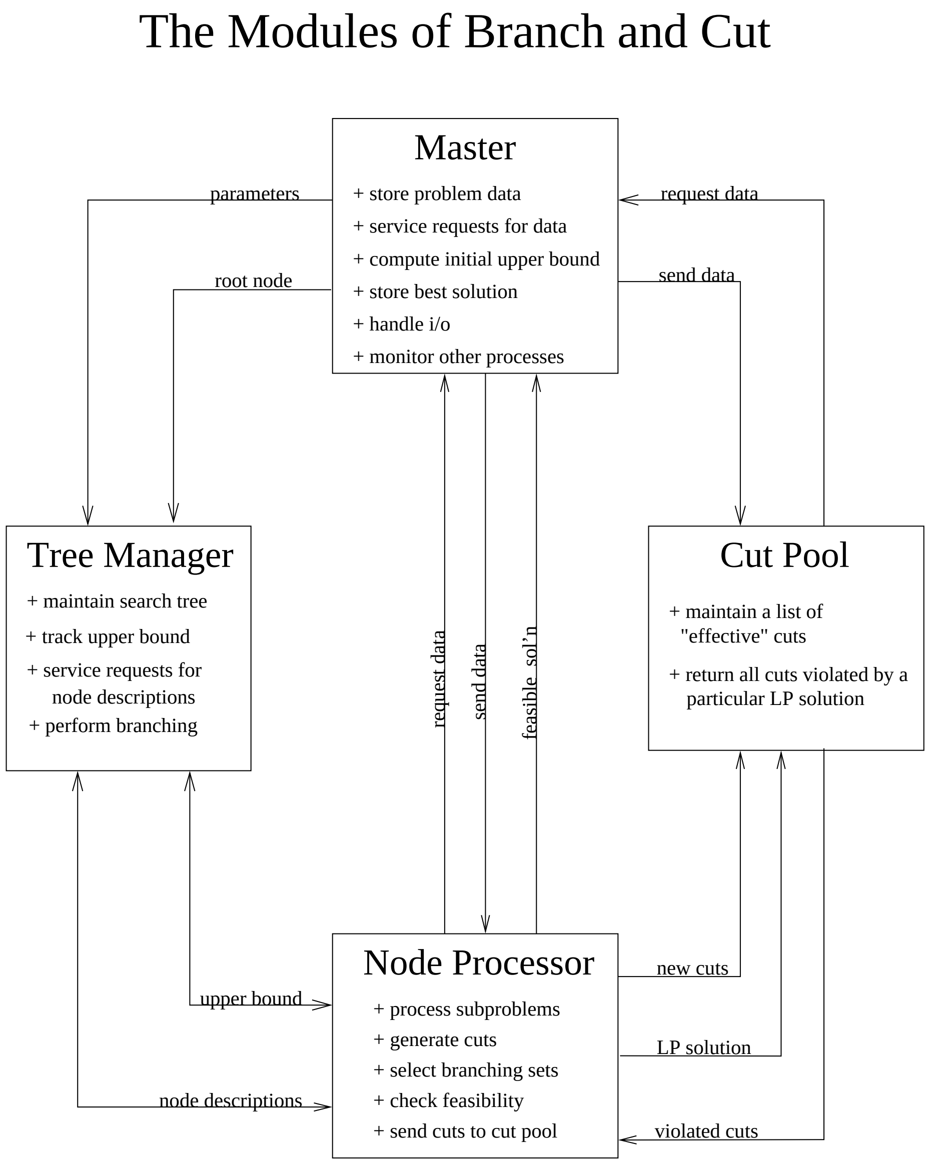

Figure 4: Schematic overview of the branch and cut algorithm

- Receive new solutions and store the best one.
- Receive the message that the algorithm has finished and print out data.
- Ensure that all other modules are still functioning.

4.2.2 The Tree Manager Module

The *tree manager* controls the overall execution of the algorithm by maintaining the search tree and distributing the subproblems to be processed to the NP modules. Because the tree manager's primary job is to maintain the list of candidate subproblems, it can be considered simply as a database for node descriptions. This database can be queried by the NP modules (to be described below) for tasks to be performed. A large part of the work the tree manager performs can be considered overhead since it involves the sharing of data among processors, a task that can be performed at a very low cost in a sequential implementation. Note that in SYMPHONY, there is only one tree manager and hence once list of candidate subproblems. This design decision also has ramifications for scalability, as we discuss in Section 5. Functions performed by the tree manager module are:

- Receive data for the root node and place it on the list of candidates for processing.
- Handle requests from NP modules to release a subproblem for processing.
- Receive branching set information, set up data structures for the children, and add them to the list of candidate subproblems.
- Receive queries from NP modules and decide whether or not they should dive.
- Keep track of the global upper bound and notify all node processing modules when it changes.
- Write current state information out to disk periodically to allow a restart in the event of a system crash.
- Keep track of run data and send it to the master program at termination.

4.2.3 The Node Processing Module

The job of each NP module is to receive and process search tree nodes, using linear programming to perform the bounding and branching operations. These operations are, of course, central to the performance of the algorithm and comprise a large part of the "useful" work done in the parallel algorithm. Search tree nodes are processed in an iterative manner, as described in Figure 1. First, the initial LP relaxation is solved, then cuts are generated based on that solution, then the relaxation is solved again. This generally continues until no more new cuts can be generated, at which point branching occurs. Branching is accomplished by choosing one or more cuts or variables and partitioning the range of allowable values by changing the associated bounds or right hand side range. Functions performed by the NP module are:

- Inform the tree manager when a new subproblem is needed.

- Receive a subproblem and process it in conjunction with the cut pool.
- Decide which cuts should be sent to the global pool to be made available to other NP modules.
- If necessary, choose a branching set and send its description back to the tree manager.
- Perform the fathoming operation.

4.2.4 The Cut Pool Module

As we have already mentioned, the concept of a *cut pool* was first suggested by Padberg and Rinaldi [49], based on the observation that inequalities generated while processing a a given node in the search tree may potentially be useful at other nodes. Since generating these cuts is sometimes a relatively expensive operation, the cut pool maintains a list of the "best" or "strongest" cuts found in the tree thus far for use in processing future subproblems. Hence, the cut pools are essentially databases that can be queried by the NP modules for cuts to be added to the current LP relaxation. Note that although we have only one node pool (candidate list), multiple cut pools can be used. In the case of multiple pools, each one services a separate subtree. More explicitly, the functions of the cut pool module are:

- Receive cuts generated by other modules and store them.
- Receive a solution and return a set of cuts eligible to enter the current LP relaxation.
- Periodically purge "ineffective" and duplicate cuts to control the size of the pool.

4.3 Overview of the Algorithm

As already discussed, SYMPHONY implements a *single-pool* branch and cut algorithm. The term single-pool refers to the fact that there is a single central list of candidate subproblems to be processed, maintained by the tree manager. Most sequential implementations use such a single-pool scheme; however, this has significant ramifications for parallel performance, as we discuss in Section 5. The unit of work in our implementation is a single subproblem, which is processed by a NP module in an iterative manner, as described above.

The master module begins by reading in the parameters and problem data. After initial I/O is completed, subroutines for finding an initial upper bound and constructing the root node are executed. During construction of the root node, the user must designate the core, as well as the initial set of active cuts and variables, after which the data for the root node are sent to the tree manager to initialize the list of candidate nodes. The tree manager in turn sets up the cut pool module(s) and the NP module(s). All node processing modules are marked as idle. The algorithm is now ready for execution.

In the steady state, the tree manager controls the algorithm by maintaining the central list of candidate subproblems and sending them to the node processing modules as they become idle. The NP modules receive nodes from the tree manager, process them, branch (if required), and send back the identity of the chosen branching set to the tree manager, which in turn generates the children and places them on the list of candidates to be processed. The order in which the nodes are processed, determined by the *search strategy*, can have a significant effect on the behavior of the algorithm. This effect is even greater in a parallel implementation, so we describe the basic options in some detail.

The *best-first* strategy always chooses the node with the smallest lower bound to be processed. The advantage of this strategy is that it will, in theory, minimize the overall size of the search tree by preventing the processing of nodes whose lower bound is greater than or equal to the optimal solution value. A *depth-first* approach, on the other hand, always chooses the node that is deepest in the tree to be processed next. This approach has several advantages over best-first. First, it minimizes the size of the candidate list (and hence conserves memory). It also tends to find feasible solutions quickly, which may be important in some applications. Most importantly, however, this strategy retains work generated locally whenever possible and thus can help to reduce communication overhead in a parallel implementation. We call the search strategies that attempt to retain locally generated work when possible *diving strategies*. The main disadvantage of these strategies is that they may result in the processing of nodes whose lower bounds turn out to be above the optimal solution value. These nodes are called *redundant* and their processing is one of the contributors to parallel overhead. This phenomena is described below in Section 5.

A *hybrid strategy* is a compromise between these two extremes. In such a strategy, we continue to dive as long as certain conditions are satisfied. These conditions are chosen to try to maximize the retention of local work, while minimizing the processing of redundant nodes. Typically, we require that the lower bound in one of the children is "close" to the lower bound of the best node otherwise available, where "close" is defined by a chosen parameter. This rule has good performance in practice, but note that the decision about whether to dive or not requires global information and hence can only be made centrally. This also has repercussions for scalability.

4.4 VRP Solver

We now illustrate what is involved in implementing a solver using the SYMPHONY framework by describing some details of the implementation of our VRP solver. The solver described here is the publicly available version distributed with SYMPHONY 3.0. Source code and datasets used in this paper are available for download at `www.branchandcut.org`. Note that this solver does not contain the full complement of separation procedures from the literature. Nonetheless, through the use of parallelism, it was the first to solve several difficult instances from the literature (see [53]). Note again that we describe here only problem-specific methods that must be implemented by the user, with emphasis on those most relevant to parallelizing the algorithm.

4.4.1 Initialization

The first task accomplished by the master process is the determination of the upper bound. In our solver, this can be accomplished using a heuristic procedure that can also be run in parallel. However, to avoid mixing the effects of the scalability of this heuristic solver with that of our branch and cut solver, we took the optimal solution value as the initial upper bound for the results reported below. The next task is to determine the problem core. Regardless of the instance, the core constraints are always taken to be constraints (1) and (2). Note that we cannot include constraints (3) in the core because there are exponentially many of them and they must be generated dynamically. The core variables can be determined either by (1) taking the union of the edge sets of solutions found during the heuristic procedure or by (2) taking the k shortest edges incident to each non-depot node in the graph, plus all the edges incident to the depot. Taking k to be approximately $.1|N|$ yields a set of edges that usually contains the edge set of some optimal solution.

4.4.2 Cut Generation and Representation

For the results shown, we generate only the inequalities (3). A description of the methods used to generate these inequalities is contained in [53]. An inequality of the form (3) can be represented compactly simply as a set T of customer nodes that can be stored in a bit array of length $|N|$. To add such a constraint to a given LP relaxation, it suffices to check, for each active variable, whether the endpoints of the corresponding edge are either both members of T or both members of $V \setminus T$. If not, then the variable has a coefficient of one in the resulting inequality and if so, it has a coefficient of zero. In this way, we are able to store each cut compactly and independent of any particular set of active variables.

4.4.3 Branching

The branching sets consist of the bound constraints corresponding to variables that are fractional in the current relaxed solution, as described in Section 3.2. To select the branching set, we have taken advantage of SYMPHONY's built-in *strong branching* capability. Using this technique, several candidates for branching are "pre-solved" by performing a limited number of iterations of the simplex algorithm and the final branching variable is chosen based on the resulting estimate of the bound in each of the resulting children. This scheme has proven extremely effective for the VRP and typically results in a more than 90% reduction in running time when employed. For the computational results reported here, we chose to evaluate seven branching candidates.

5 Performance and Scalability

To study the behavior of our basic approach, we have tested the performance of our solver on a Beowulf cluster with 48 dual-processor nodes (1 GHz PIII). The operating system was Red Hat Linux 7.2 with OSL 3.0 used to solve the linear programming relaxations. In the next two sections, we describe how we measured performance and present the computational results.

5.1 Performance Measures

Scalability is defined generally as the ability of a parallel system (a parallel algorithm plus a parallel architecture) to take advantage of increased computing resources (in this case, additional processors). We are therefore interested in comparing the running time of the best sequential algorithm, denoted T_0, with that of the parallel algorithm running on p processors, denoted T_p. We define the *speedup* (S_p) as the ratio T_0/T_p and the *parallel efficiency* (E_p) as the ratio S_p/p of speedup to number of processors. If an algorithm has an efficiency of one or more for a given number of processors, we say it has achieved *linear* *speedup*. In theory, it is not possible to achieve an efficiency of more than one, but this can happen in practice (see [13] for an examination of this phenomena).

When an algorithm has an efficiency less than one, we can compute the *parallel overhead* as $O_p = pT_p - T_0$. The parallel overhead has three basic components:

- **Idle Time**: Time spent idle while waiting for information requested from another module.
- **Performance of Redundant Work**: Time spent performing work that would not have been performed in the sequential algorithm.
- **Communication Overhead**: Time spent performing work related to sending and receiving information from other modules.

To achieve high efficiency, we must limit the occurrence of this overhead. In the analysis below, we assess the impact and root causes of each of these sources through direct (when possible) or indirect performance measures.

5.2 Performance Analysis

To assess performance, we explicitly measured the idle time and the time spent performing tasks associated with communication. We also tracked the number of nodes in the search tree as an indirect measure of redundant work. In parallel branch and cut, the size of the search tree is subject to a good deal more random fluctuation than would be expected in parallel branch and bound, due to the added randomness inherent in heuristic cut generation. To eliminate the effect of these inevitable random fluctuations on the analysis, we have done two things. First, we performed three identical runs of each experiment. The numbers that appear in the tables of results are averages over these three runs. Second, we consider the timing information on a "per node" basis. This results in a much more consistent view of the trends as the number of processors increases.

We report here on two sets of experiments. We first tested the code "out of the box" with the default settings used when running sequentially. These results are shown in Table 1. We then changed the settings to try to improve scalability (the settings that were changed will be described below). These results are reported in Table 2. Both tables are organized according to the number of NP modules used, since these modules perform the vast majority of the useful work (the other modules are simply storing shared data). Full results are shown for the runs with 4 NP modules and only summary results for other runs. Note that the scale of these experiments is relatively small, but we can nonetheless draw important conclusions from them.

We have already described generally the main sources of overhead in parallel branch and bound. The results in Tables 1 and 2 break these sources down further into specific identifiable components that were significant to performance. In the tables, *Tree Size* is the number of nodes in the search tree; *Ramp-up* and *Ramp-down* are as described below; *Idle (Nodes)* is the idle time spent by each NP module waiting for a new node description to be sent from the tree manager; *Idle (Cuts)* is the idle time spent by each NP module waiting for cuts to be sent by the cut pool; *CPU sec* is the total CPU time of all modules; and *Wallclock* is the actual running time. Examining the computational results, we can be more specific about the causes of parallel overhead.

Instance	Tree Size	Ramp– up	Ramp– down	Idle (Nodes)	Idle (Cuts)	CPU sec	Wallclock
$A-n37-k6$	10463	2.83	1.45	9.42	24.84	790.67	211.53
$A-n39-k5$	466	2.55	0.04	0.39	1.41	60.02	16.44
$A-n39-k6$	434	1.70	0.03	0.28	0.45	21.36	6.17
$A-n44-k6$	2646	3.03	0.39	2.62	7.92	328.89	87.29
$A-n45-k6$	1028	3.13	0.09	0.78	2.25	150.02	39.97
$A-n46-k7$	53	3.65	0.07	0.10	0.06	6.83	2.76
$A-n48-k7$	3768	5.10	0.60	3.94	11.16	529.27	139.39
$A-n53-k7$	3858	4.47	0.58	4.07	14.67	511.85	136.04
$A-n55-k9$	5527	4.61	1.06	6.30	10.70	809.72	211.53
$A-n65-k9$	18871	12.11	6.13	26.00	105.88	5646.64	1463.16
$B-n45-k6$	877	2.17	0.13	0.74	1.04	76.81	20.85
$B-n51-k7$	461	2.13	0.08	0.38	0.30	39.44	11.06
$B-n57-k7$	3346	5.54	0.47	3.12	4.27	325.36	87.64
$B-n64-k9$	99	3.89	0.15	0.12	0.05	11.22	3.99
$B-n67-k10$	16853	10.19	3.07	18.82	77.83	2192.39	586.56
4 NP's	68753	67.10	14.33	77.08	262.82	11500.49	3024.37
Per Node		0.0010	0.0002	0.0011	0.0038	0.1673	0.1760
8 NP's	69616	189.34	16.50	79.72	372.15	11532.37	1551.67
Per Node		0.0027	0.0002	0.0011	0.0053	0.1657	0.1783
16 NP's	76294	481.83	22.39	88.11	817.67	12921.04	913.80
Per Node		0.0063	0.0003	0.0012	0.0107	0.1694	0.1916
32 NP's	75123	1119.90	49.91	92.32	3171.67	12487.33	538.50
Per Node		0.0149	0.0007	0.0012	0.0422	0.1662	0.2294

Table 1: Computational Results with Default Settings

Ramp-Up and Ramp-down. *Ramp-up time* is the total idle time occurring at the beginning of the algorithm when there are not enough units of work available to employ all processors. Ramp-up time can be a serious problem when either (1) the time to process a node is substantial, or (2) the number of children generated by branching is small. Because of the iterative bounding scheme we employ in branch and cut, the time to process a node can be significant and our branching scheme generates only two children. From Table 1, it is evident that ramp-up time is one of the most serious scalability problems we face, as the idle time *per processor* due to ramp-up is increasing. In [31], Henrich studies this problem and offers some alternatives to what is presented here.

To control ramp-up time, we have implemented a "quick branching" strategy in which branching occurs after a fixed number of iterations in the NP module, regardless of whether or not new cuts have been generated. This continues until all processors have useful work to do, after which the usual algorithm is resumed. The results of applying this scheme with an iteration limit of five are shown in Table 2. Although the ramp-up time did decrease, as expected, the effect was negated by a corresponding increase in the total number of search

Instance	Tree Size	Ramp–up	Ramp–down	Idle (Nodes)	Idle (Cuts)	CPU sec	Wallclock
A − n37 − k6	14305	1.70	2.02	12.31	40.06	1067.49	286.37
A − n39 − k5	483	0.81	0.05	0.35	1.30	54.17	14.49
A − n39 − k6	739	0.90	0.06	0.45	1.10	37.45	10.25
A − n44 − k6	3733	1.58	0.55	3.62	11.64	453.45	119.35
A − n45 − k6	493	0.59	0.05	0.42	1.06	65.09	17.10
A − n46 − k7	176	0.96	0.01	0.15	0.79	25.69	7.02
A − n48 − k7	4243	1.14	0.77	4.31	15.54	593.36	155.05
A − n53 − k7	2808	1.32	0.48	2.95	9.44	385.68	100.98
A − n55 − k9	6960	2.07	1.46	8.12	15.31	913.35	237.30
A − n65 − k9	18165	1.41	5.83	25.89	105.84	5190.83	1335.60
B − n45 − k6	1635	0.72	0.21	1.39	2.09	131.13	34.92
B − n51 − k7	348	0.36	0.03	0.32	0.37	25.35	6.88
B − n57 − k7	4036	0.76	0.39	3.21	5.52	494.13	131.87
B − n64 − k9	100	0.58	0.01	0.08	0.19	15.49	4.22
B − n67 − k10	16224	2.95	2.54	17.85	64.88	2351.30	618.73
4 NP′s	74451	17.87	14.45	81.42	275.11	11803.97	3080.12
Per Node		0.0002	0.0002	0.0011	0.0037	0.1585	0.1655
8 NP′s	82488	67.12	17.07	89.54	370.96	11834.68	1569.27
Per Node		0.0008	0.0002	0.0011	0.0045	0.1435	0.1522
16 NP′s	97078	203.54	41.19	110.36	1045.95	12881.44	908.68
Per Node		0.0021	0.0004	0.0011	0.0108	0.1327	0.1498
32 NP′s	98991	640.74	49.09	135.74	3320.88	13044.33	545.73
Per Node		0.0065	0.0005	0.0014	0.0335	0.1318	0.1764

Table 2: Computational Results with Settings to Minimize Overhead

tree nodes processed. This is due to the fact that the LP relaxations were weaker at the time strong branching occurred and hence the branching decisions were less effective. Note that using an iteration limit less than five results only magnified this effect.

Ramp-down time is the idle time occurring at the end of the algorithm when there is not enough useful work for all processors to be employed. From Table 1, we can see that ramp-down time is not a very serious problem. In theory, ramp-down could be controlled in much the same way as ramp-up—by reverting to "quick branching" scheme whenever the number of nodes in the candidate list falls below a certain threshold. However, we have not yet implemented this strategy.

We would like to briefly mention that during the preparation of this paper, we also experimented with solving the traveling salesman problem, which we have already observed is closely related to the VRP. One big difference between the TSP and the VRP from a computational standpoint is that a great deal more is known about solving the separation problem for the TSP. Hence, although the TSP is a much easier problem to solve in practice, the time to process a single search tree node is usually much longer. In our preliminary experiments, we found that the ramp-up time was therefore a much more serious problem for the TSP. In fact, even for small- to medium-sized instances, we could not achieve linear speedup. Any attempt to eliminate this ramp-up time using the quick branching scheme we have described resulted in much larger search trees. Our tentative conclusion is that achieving parallel efficiency is actually *more difficult* when more is known about the structure of the convex hull of feasible solutions. Eliminating ramp-up time in these cases without introducing additional overhead is a challenging problem that requires significant further study.

Idle Time (Handshaking). The second major contributor to parallel overhead is time spent by the NP modules waiting for requests to be serviced by another module (either the tree manager or the cut pool).

- Tree Manager (Node Pool): Requests to the tree manager can be of three types: (1) request for a new node description, (2) a request for a decision about whether to dive or not, or (3) a request for a list of indices to be assigned to newly generated cuts. Note that because of the differencing scheme we use to store the search tree, reconstructing the node descriptions needed to service requests of type (1) can involve some computational effort. Because there is only one tree manager module to service all of these requests, this is obviously a scalability issue when large numbers of processors are used. However, for up to 32 processors, we have found that the tree manager does not become a bottleneck. The time waiting for information of types (2) and (3) is insignificant and not shown in the tables of results, while time spent waiting for node descriptions (seen in the column *Idle (Nodes)*), when pro-rated per processor remains relatively constant. However, it should be pointed out that this would obviously not be the case when running on very large numbers of processors.

- Cut Pool: Requests to the cut pool can involve significant computational effort, since a large list of cuts must be checked for violation and the pools can grow quite large. This can cause the pool to become a significant bottleneck. In addition, we currently send each violated cut back in a separate message in order to allow the NP module to receive the cut as quickly as possible. This seems to increase latency significantly and is probably responsible for a large part of the waiting time shown in the *Idle (Cuts)* column. This idle time can be dealt with through the use of multiple cut pools servicing different parts of the search tree. We have implemented this scheme and the results are shown in Table 2. There is a decrease in the idle time, but the idle time per node still goes up significantly as the number of processors increases.

Communication overhead. Time spent packing and unpacking messages, is of course also an issue. However, our results have shown that this type of communication overhead is insignificant compared to the first two sources.

Redundant work. Finally, we must concern ourselves with whether or not we are performing redundant work in the parallel version of the algorithm. Based on our experiments, the number of nodes in the search tree remains relatively constant as the number of processors is increased. Furthermore, after removing the effect of the previously described sources of parallel overhead, the total processing time of the algorithm also remains constant as the number of nodes increases. Both of these are solid indicators that little or no redundant work is being performed. However, it should be emphasized that this is possible primarily because of the single-pool approach, which allows a complete global picture of the available work.

6 Conclusions

In this paper, we described an algorithmic framework and an associated software library called SYMPHONY that can be used for the solution of a wide variety of combinatorial optimization problems arising in logistics applications. In computational experiments with the software, we found that the occurrence of parallel overhead is due mainly to two sources: (1) the need for the processors to frequently query the node pool and the cut pools and (2) the ramp-up/ramp-down time needed to employ all processors. For relatively small numbers of processors, we can control these sources of overhead effectively. However, it is important to note that this is due primarily to two important factors. First, we described solution of a problem for which separation is quick and node processing times are relatively short. In such cases, ramp-up time, although still an issue, is significantly reduced over problems where the node processing times are much larger. Currently, we do not know how to control ramp-up time for problems with large node processing times. Second, the single-pool nature of our algorithm allowed us to successfully control the performance of redundant work often occurring in parallel branch and bound. However, it is clear that for very large numbers of processors, the single-pool nature of our algorithm will be a scalability issue.

To address these and other issues, we have already begun designing a next generation framework for branch, cut, and price algorithms that will allow massively parallel computation. The design of this framework is described in [38] and [39] and will be available for download from the Common Optimization Interface for Operations Research (COIN-OR) repository site (`www.coin-or.org`).

References

[1] Y. Agarwal, K. Mathur, and H.M. Salkin, *Set Partitioning Approach to Vehicle Routing*, Networks **7** (1989), 731.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *On the solution of traveling salesman problems*, Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians (1998), 645.

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *CONCORDE TSP Solver*, available at `www.keck.caam.rice.edu/concorde.html`.

[4] J.R. Araque, L. Hall, and T. Magnanti, *Capacitated Trees, Capacitated Routing and Associated Polyhedra*, Discussion paper 9061, CORE, Louvain La Nueve (1990).

[5] J.R. Araque, G. Kudva, T.L. Morin, and J.F. Pekny, *A Branch-and-Cut Algorithm for Vehicle Routing Problems*, Annals of Operations Research **50** (1994), 37.

[6] P. Augerat, J.M. Belenguer, E. Benavent, A. Corberán, D. Naddef, *Separating Capacity Constraints in the CVRP Using Tabu Search*, European Journal of Operations Research, **106** (1998), 546.

[7] P. Augerat, J.M. Belenguer, E. Benavent, A. Corberán, D. Naddef, G. Rinaldi, *Computational Results with a Branch and Cut Code for the Capacitated Vehicle Routing Problem*, Research Report 949-M, Universite Joseph Fourier, Grenoble, France.

[8] E. Balas, S. Ceria, and G. Cornuéjols, *Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework*, Management Science **42** (1996), 9.

[9] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, P. H. Vance, *Branch-and-Price: Column Generation for Huge Integer Programs*, Operations Research **46** (1998) , 316.

[10] M. Benchouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol, *Building a Parallel Branch and Bound Library*, in Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science **1054**, Springer, Berlin (1996), 221.

[11] R. Bixby, W. Cook, A. Cox, and E. K. Lee, *Parallel Mixed Integer Programming*, Rice University Center for Research on Parallel Computation Research Monograph CRPC-TR95554 (1995).

[12] U. Blasum and W. Hochstättler, *Application of the Branch and Cut Method to the Vehicle Routing Problem*, Zentrum für Angewandte Informatik Köln Technical Report zpr2000-386 (2000).

[13] A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens, *Asynchronous Parallel Branch and Bound and Anomalies*, Report EUR-CS-95-05, Department of Computer Science, Erasmus University, Rotterdam (1995).

[14] V. Campos, A. Corberán, and E. Mota, *Polyhedral Results for a Vehicle Routing Problem*, European Journal of Operations Research **52** (1991), 75.

[15] Q. Chen, and M. C. Ferris, *FATCOP: A Fault Tolerant Condor-PVM Mixed Integer Programming Solver*, University of Wisconsin CS Department Technical Report 99-05, Madison, WI (1999).

[16] N. Christofides, A. Mingozzi and P. Toth, *Exact Algorithms for Solving the Vehicle Routing Problem Based on Spanning Trees and Shortest Path Relaxations*, Mathematical Programming **20** (1981), 255.

[17] V. Chvátal, *Linear Programming*, W.H. Freeman and Company (1983).

[18] C. Cordier, H. Marchand, R. Laundy, and L. A. Wolsey, *bc-opt: A Branch-and-Cut Code for Mixed Integer Programs*, Mathematical Programming **86** (1999), 335.

[19] G. Cornuéjols and F. Harche, *Polyhedral Study of the Capacitated Vehicle Routing Problem*, Mathematical Programming **60** (1993), 21.

[20] R. Correa and A. Ferreira, Parallel Best-first Branch and Bound in Discrete Optimization: A Framework, Center for Discrete Mathematics and Theoretical Computer Science Technical Report 95-03.

[21] G.B. Dantzig and R.H. Ramser, *The Truck Dispatching Problem*, Management Science **6** (1959), 80.

[22] J. Eckstein, C. A. Phillips, and W. E. Hart, *PICO: An Object-Oriented Framework for Parallel Branch and Bound*, RUTCOR Research Report 40-2000, Rutgers University, Piscataway, NJ (2000).

[23] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi, *Branch-and-Cut Algorithms for Combinatorial Optimization and Their Implementation in ABACUS*, in Computational Combinatorial Optimization, D. Naddef and M. Jünger, eds., Springer, Berlin (2001), 157.

[24] M.L. Fisher, *Optimal Solution of Vehicle Routine Problems Using Minimum k-Trees*, Operations Research **42** (1988), 141.

[25] B.A. Foster and D.M. Ryan, *An Integer Programming Approach to the Vehicle Scheduling Problem*, Operational Research Quarterly **27** (1976), 367.

[26] A. Geist et al, *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT University Press, Cambridge, MA (1994).

[27] B. Gendron and T. G. Crainic, *Parallel Branch and Bound Algorithms: Survey and Synthesis*, Operations Research **42** (1994), 1042.

[28] A. Grama and V. Kumar, *Parallel Search Algorithms for Discrete Optimization Problems*, ORSA Journal on Computing **7** (1995), 365.

[29] M. Grötschel, M. Jünger, and G. Reinelt, *A Cutting Plane Algorithm for the Linear Ordering Problem*, Operations Research **32** (1984), 1155.

[30] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin (1981).

[31] D. Henrich, *Initialization of Parallel Branch-and-bound Algorithms*, Proceedings of the Second International Workshop on Parallel Procesing and Artificial Intelligence, Chamberry, France (1993).

[32] K. Hoffman and M. Padberg, *LP-Based Combinatorial Problem Solving*, Annals of Operations Research **4** (1985/86), 145.

[33] M. Jünger and S. Thienel, *The ABACUS System for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization*, Software Practice and Experience **30** (2000), 1325.

[34] M. Jünger and S. Thienel, *Introduction to ABACUS—a branch-and-cut system*, Operations Research Letters **22** (1998), 83.

[35] G. L. Nemhauser, M. W. P. Savelsbergh, and G. S. Sigismondi, *MINTO, a Mixed INTeger Optimizer*, Operations Research Letters **15** (1994), 47.

[36] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, Inc. (1988).

[37] V. Kumar and A. Gupta, *Analyzing Scalability of Parallel Algorithms and Architectures*, Journal of Parallel and Dsitributed Computing **22** (1994), 379.

[38] L. Ladányi, T.K. Ralphs, and M. Saltzman, *Implementing Scalable Parallel Search Algorithms for Data-intensive Applications*, Proceedings of the International Conference on Computational Science (2002), Volume I, 592.

[39] L. Ladányi, T.K. Ralphs, and M. Saltzman, *A Library Hierarchy for Implementing Scalable Parallel Search Algorithms*, Lehigh University Industrial and Systems Engineering Technical Report 01T-010 (2001).

[40] L. Ladányi, T.K. Ralphs, and L. E. Trotter, *Branch, Cut, and Price: Sequential and Parallel*, in Computational Combinatorial Optimization, D. Naddef and M. Jünger, eds., Springer, Berlin (2001), 223.

[41] A. H. Land and A. G. Doig, *An Automatic Method for Solving Discrete Programming Problems*, Econometrica **28** (1960), 497.

[42] G. Laporte and Y. Nobert, *Comb Inequalities for the Vehicle Routing Problem*, Methods of Operations Research **51** (1981), 271.

[43] G. Laporte, Y. Nobert and M. Desrouchers, *Optimal Routing with Capacity and Distance Restrictions*, Operations Research **33** (1985), 1050.

[44] A.N. Letchford, R.W. Eglese, and J. Lysgaard, *Multi-star Inequalities for the Vehicle Routing Problem*, Technical Report available at `http://www.lancs.ac.uk/staff/letchfoa/pubs.htm`.

[45] J. Linderoth, *Topics in Parallel Integer Optimization*, Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA (1998).

[46] G. Mitra, I. Hai, and M.T. Hajian, *A Distributed Processing Algorithm for Solving Integer Programs Using a Cluster of Workstations*, Parallel Computing **23** (1997), 733.

[47] L.G. Mitten, Branch-and-bound Methods: General Formulation and Properties, Operations Research **18** (1970), 24.

[48] D. Naddef and G. Rinaldi, *Branch and Cut*, in P. Toth and D. Vigo, eds., Vehicle Routing, SIAM (2000).

[49] M. Padberg and G. Rinaldi, *A Branch-and-Cut Algorithm for the Resolution of Large-Scale Traveling Salesman Problems*, SIAM Review **33** (1991), 60.

[50] T. K. Ralphs, *SYMPHONY Version 3.0 User's Guide*, available at `www.branchandcut.org/SYMPHONY`.

[51] T. K. Ralphs and L. Ladányi, *COIN/BCP User's Guide*, available at `www.coin-or.org`.

[52] T. K. Ralphs, L. Ladányi, and M. Saltzman, *Parallel Branch, Cut, and Price for Large-scale Discrete Optimization*, to appear in Mathematical Programming, per-print available from `http://www.lehigh.edu/ tkr2/research/pubs.html`.

[53] T.K. Ralphs, L. Kopman, W.R. Pulleyblank, and L.E. Trotter Jr., *On the Capacitated Vehicle Routing Problem*, Mathematical Programming Series B **94** (2003), 343.

[54] Y. Shinano, M. Higaki, and R. Hirabayashi, *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 (1995), 392.

[55] H. W. J. M. Trienekens and A. de Bruin, *Towards a Taxonomy of Parallel Branch and Bound Algorithms*, Report EUR-CS-92-01, Department of Computer Science, Erasmus University Rotterdam (1992).

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

[57] S. J. Wright, *Solving Optimization Problems on Computational Grids*, Optima **65** (2001).